DISTRIBUTED ALGORITHMS

Lecture Notes for 6.852

Fall 1992

Nancy A. Lynch
Boaz Patt-Shamir

January 1993

Preface

This report contains the lecture notes used by Nancy Lynch's graduate course in Distributed Algorithms, during fall semester, 1992. The notes were prepared by Nancy Lynch and Teaching Assistant Boaz Patt-Shamir.

The main part of this report is detailed lectures notes for the 25 regular class meetings. We think the reader will find these to be reasonably well polished, and we would very much appreciate comments about any errors, or suggestions on the presentation. Following these notes are the homework assignments. Finally, we included an appendix that contains notes for three additional lectures that were given after the regular term was over. The extra lectures cover material that did not fit into the scheduled number of class hours. These notes are in a very rough shape.

Many thanks go to the students who took the course this semester; they contributed many useful comments and suggestions on the technical content, and also had a very strong influence on the pace and level of the presentation. If these notes are understandable to other audiences, it is largely because of the feedback provided by this first audience. Thanks also go to George Varghese for contributing an excellent lecture on self-stabilizing algorithms.

We would also like to acknowledge the work of Ken Goldman and Isaac Saias in preparing earlier versions (1988 and 1990, respectively) of the notes for this course. We were able to reuse much of their work. The students who took the course during those two years worked hard to prepare those earlier lecture notes, and also deserve a lot of thanks. Jennifer Welch and Mark Tuttle also contributed to earlier versions of the course. Finally, we would like to thank Joanne Talbot for her invaluable help.

Nancy Lynch, `lynch@theory.lcs.mit.edu`
Boaz Patt-Shamir, `boaz@theory.lcs.mit.edu`

January, 1993

Contents

6.852J/18.437J Distributed Algorithms September 10, 1992

Lecturer: Nancy Lynch

Lecture 1

1.1 Introduction to the Course

1.1.1 The Subject Matter

This course is about "distributed algorithms". Distributed algorithms include a wide range of parallel algorithms, which can be classified by a variety of attributes:

- Interprocess Communication (IPC) method: shared memory, message-passing, dataflow.
- Timing Model: synchronous, asynchronous, partially synchronous.
- Failure Model: reliable system, faulty links, faulty processors.
- Problems addressed: resource allocation, communication, agreement, database concurrency control, deadlock detection, and many more.

Some of the major intended application areas of distributed algorithms are

- communication systems,
- shared-memory multiprocessor computation,
- distributed operating systems,
- distributed database systems,
- digital circuits, and
- real-time process-control systems.

Some kinds of parallel algorithms are studied in other courses and will not be covered in this course, e.g., PRAM algorithms and algorithms for fixed-connection networks such as Butterfly. The algorithms to be studied in this course are distinguished by having a higher degree of *uncertainty*, and more *independence of activities*: Some of the types of uncertainty that we will consider are:

- unknown number of processors,
- unknown shape of network,
- independent inputs at different locations,
- several programs executing at once, starting at different times, going at different speeds,
- nondeterministic processors,
- uncertain message delivery times,
- unknown message ordering,
- failures: processor (stopping, transient omission, Byzantine); link (message loss, duplication, reordering)

Because of all this uncertainty, no component of a distributed system "knows" the entire system state. Luckily, not all the algorithms we consider will have to contend with all of these types of uncertainty!

Distributed algorithms can be extremely complex, at least in their details, and can be quite difficult to understand. Even though the actual "code" may be short, the fact that many processors are executing the code in parallel, with steps interleaved in some undetermined way, implies that there can be prohibitively many different executions, even for the same inputs. This implies that it is nearly impossible to understand *everything* about the executions of distributed algorithms. This can be contrasted with other kinds of parallel algorithms such as PRAM algorithms, for which one might hope to understand exactly what the (well-defined) state looks like at each given point in time. Therefore, instead of trying to understand all the details of the execution, one tends to assert certain *properties* of the execution, and just understand and prove these properties.

1.1.2 Style

The general flavor of the work to be studied is as follows.

- Identify problems of major significance in (practical) distributed computing and define abstract versions of the problems for mathematical study.
- Give precise problem statements.
- Describe algorithms precisely.

- Prove rigorously that the algorithms solve the problems.
- Analyze the complexity of the algorithms.
- Prove corresponding impossibility results.

Note the emphasis on rigor; this is important in this area, because of the subtle complications that arise. A rigorous approach seems necessary to be sure that the problems are meaningful, the algorithms are correct, the impossibility results are true and meaningful, and the interfaces are sufficiently well-defined to allow system building.

However, because of the many complications, rigor is hard to achieve. In fact, the development of good formal methods for describing and reasoning about distributed algorithms is the subject of a good deal of recent research. Specifically, there has been much serious work in defining appropriate *formal mathematical models*, both for describing the algorithms and for describing the problems they are supposed to solve; a considerable amount of work has also been devoted to *proof methods*. One difficulty in carrying out a rigorous approach is that, unlike in many other areas of theoretical computer science, there is no any single accepted formal model to cover all the settings and problems that arise in this area. This phenomenon is unavoidable, because there are so many very different settings to be studied (consider the difference between shared memory and message-passing), and each of them has its own suitable formal models.

So, rigor is a goal to be striven for, rather than one that we will achieve entirely in this course. Due to time limitations, and (sometimes) the difficulty of making formal presentations intuitively understandable, the presentation in class will be a mixture of rigorous and intuitive.

1.1.3 Overview of the Course

There are many different orders in which this material could be presented. In this course, we divide it up first according to *timing assumptions*, since that seems to be the most important model distinction. The timing models to be considered are the following.

synchronous: This is the simplest model. We assume that components take steps simultaneously, i.e., the execution proceeds in synchronous rounds.

asynchronous: Here we assume that the separate components take steps in arbitrary order.

partially synchronous (timing-based): This is an "in-between" model — there are some restrictions on relative timing of events, but execution is not completely lock-step.

The next division is by the IPC mechanism: shared memory vs. message-passing. Then we subdivide by the problem studied. And finally, each model and problem can be considered with various failure assumptions.

We now go over the bibliographical list and the tentative schedule (Handouts 2 and 3). The bibliographical list doesn't completely correspond to the order of topics to be covered; the difference is that the material on models will be spread throughout the course as needed.

General references. These include the previous course notes, and some related books. There is not really much in the way of textbooks on this material.

Introduction. The chapter on distributed computing in the handbook on Theoretical Computer Science is a sketchy overview of some of the modeling and algorithmic ideas.

Models and Proof Methods. We shall not study this as an isolated topic in the course – rather, it is distributed through the various units. The basic models used are *automata-theoretic*, starting with a basic state-machine model with little structure. These state machines need not necessarily be finite-state. *Invariant assertions* are often proved about automaton states, by induction. Sometimes we use one automaton to represent the problem being solved, and another to represent the solution; then a correctness proof boils down to establishing a correspondence that preserves the desired *external behavior*. In this case, proving the correspondence is often done using a *mapping* or *simulation* method. Specially tailored state machine models have been designed for some special purposes, e.g., for shared memory models (where the structure consists of processes and shared variables). Another model is the I/O automaton model for *reactive systems*, i.e., systems that interact with an external environment in an ongoing fashion. This model can model systems based on shared variables, but is more appropriate for message-passing systems. One of the key features of this model is that it has good *compositionality* properties, e.g., that the correctness of a compound automaton can be proved using the correctness of its components. *Temporal logic* is an example of a special set of methods (language, logic) mainly designed for proving *liveness* properties (e.g., something eventually happens). *Timed models* are mainly newer research work. Typically, these are specially-tailored models for talking about timing-based systems – e.g., those whose components have access to system clocks, can use timeouts, etc. *Algebraic methods* are an important research subarea (but we will not have time for this). The algebraic methods describe concurrent processes and systems using algebraic expressions, then use equations involving these expressions to prove equivalences and implementation relationships among the processes.

Synchronous Message-Passing. As noted, the material is organized first by timing model. The simplest model (i.e., the one with the least uncertainty) is the *synchronous* model, in which all the processes take steps in synchronous rounds. The shared-memory version of this model is the PRAM. Since it is studied in other courses, we shall skip this subject, and start with synchronous networks.

We spend the first two weeks or so of the course on problems in synchronous networks that are typical of the distributed setting. In the network setting we have the processors at the nodes of a graph G, communicating with their neighbors via messages in the edges. We start with a simple toy example, involving *ring computation.* The problem is to elect a unique leader in a simple network of processors, which are assumed to be identical except for Unique Identifiers (UID's). The uncertainty is that the size of network, and the set of ID's of processors, are unknown (although it is known that the UID's are indeed unique). The main application for this problem is a token ring, where there is a single token circulating, and sometimes it it necessary to regenerate a lost token. We shall see some details of the modeling, and some typical complexity measures will be studied. For example, we shall show upper and lower bounds for the time and the amount of communication (i.e., number of messages) required. We shall also study some other problems in this simple setting.

Next, we'll go through a brief survey of some protocols in more general networks. We shall see some protocols used in unknown synchronous networks of processes to solve some basic problems like finding shortest paths, defining a minimum spanning tree, computing a maximal independent set, etc.

Then we turn to the problem of *reaching consensus.* This refers to the problem of reaching agreement on some abstract fact, where there are initial differences of opinion. The uncertainty here stems not only from different initial opinions, but also from *processor failures.* We consider failures of different types: stopping, where a processor suddenly stops executing its local protocol; omission, where messages may be lost en route; and Byzantine, where a faulty processor is completely unrestricted. This has been an active research area in the past few years, and there are many interesting results, so we shall spend a couple of lectures on this problem. We shall see some interesting bounds on the number of tolerable faults, time, and communication.

Asynchronous Shared Memory. After "warming up" with synchronous algorithms (in which there is only a little uncertainty), we move into the more characteristic (and possibly more interesting) part of the course, on *asynchronous* algorithms. Here, processors are no longer assumed to take steps in lock-step synchrony, but rather can interleave their steps in arbitrary order, with no bound on individual process speeds. Typically, the interactions with the external world (i.e., input/output) are ongoing, rather than just initial input and final output. The results in this setting have quite a different flavor from those for synchronous networks.

The first problem we deal with is *mutual exclusion* This is one of the fundamental (and historically first) problems in this area, and consequently, much work has been dedicated to exploring it. Essentially, the problem involves arbitrating access to a single, indivisible, exclusive-use resource. The uncertainty here is about who is going to request access and when. We'll sketch some of the important algorithms, starting with the original algorithm of Dijkstra. Many important concepts for this field will be illustrated in this context, including progress, fairness, fault-tolerance, and time analysis for asynchronous algorithms. We shall see upper bounds on the amount of shared memory, corresponding lower bounds, and impossibility results. We shall also discuss generalizations of mutual exclusion to more general resource allocation problems. For example, we will consider the *Dining Philosophers* problem – a prototypical resource allocation problem.

Next, we shall study the concept of *atomic registers*: so far, we have been assuming indivisible access to shared memory. But how can one implement this on simpler architectures? We shall look at several algorithms that solve this problem in terms of weaker primitives. An interesting new property that appears here is *wait-freeness*, which means that any operation on the register must complete regardless of the failure of other concurrent operations.

An *atomic snapshot* is a convenient primitive for shared read-write memory. Roughly speaking, the objective is to take an instantaneous snapshot of all the memory locations at once. An atomic snapshot is a useful primitive to have for building more powerful systems. We shall see how to implement it.

A *concurrent timestamp system* is another nice primitive. This is a system that issues, upon request, timestamps that can be used by programs to establish a consistent order among their operations. The twist here is how to implement such systems with bounded-memory. It turns out out such bounded timestamp systems can be built in terms of atomic snapshots. Also, a concurrent timestamp system can be used to build more powerful forms of shared memory, such as multi-writer multi-reader memory.

We shall also reconsider the *consensus* problem in the asynchronous shared memory model, and prove the interesting fact it is impossible to solve in this setting.

A possible new topic this time is *shared memory for multiprocessors*. Much recent research is aimed at identifying different types of memory abstractions that are used, or might be useful, for real systems. Systems architects who develop such types of memory frequently do not give precise statements of what their memory guarantees (especially in the presence of concurrent accesses and failures). So recently, theoreticians have started trying to do this.

Asynchronous Message-Passing Systems. This section deals with algorithms that operate in asynchronous networks. Again, the system is modeled as a graph with processors at nodes, and communication links are represented by the edges, but now the system does not operate in rounds. In particular, messages can arrive at arbitrary times, and the processors can take steps at arbitrary speeds. One might say that we now have "looser coupling" of the components of the system: we have more independence and uncertainty.

Computing in static graphs. The simplest type of setting here is computation in a fixed (unknown) graph in which the inputs arrive at the beginning, and there is a single output to be produced. Some examples are leader election in ring, and minimum spanning tree computation.

Network synchronization. At this point, we could plunge into a study the many special-purpose algorithms designed expressly for asynchronous distributed networks. But instead, we shall first try to impose some structure on such algorithms by considering "algorithm transformations" that can be used to run algorithms designed for a simpler computation model on a a complex asynchronous network.

The first example here arises in the very important paper by Lamport, where he shows a simple method of assigning consistent *logical times* to events in a distributed network. This can be used to allow an asynchronous network to simulate one in which the nodes have access to perfectly synchronized real-time clocks. The second example is Awerbuch's *synchronizer*, which allows an asynchronous network to simulate the lock-step synchronous networks discussed in the first part of the course (at least, those without failures), and to do so efficiently. We shall contrast this simulation result with an interesting lower bound that seems to say that any such simulation must be inefficient (the apparent contradiction turns out to depend on the kind of problem being solved). Third, we shall see that an synchronous network can simulate a centralized (non-distributed) state machine. And fourth, an asynchronous network can be used to simulate asynchronous shared memory. Any of these simulations can be used to run algorithms from simpler models in the general asynchronous network model.

Next, we shall look at some specific problems, such as resource allocation. We shall see how to solve mutual exclusion, dining philosophers etc. in networks.

Detection of stable properties refers to a class of problems with a similar flavor and a common solution. Suppose that there is a separate algorithm running, and we want to design another algorithm to "monitor" the first. To monitors here might mean, for instance, to detect when it terminates or deadlocks, or to take a "consistent snapshot" of its state.

We shall also revisit the *consensus* problem in the context of networks. The problem is easy without faults, but with faults, it is mostly impossible (even for very simple types of faults such as just stopping).

Datalink protocols involve the implementation of a reliable communication link in terms of unreliable underlying channels. We shall see the basic Alternating Bit Protocol (the standard case study for concurrent algorithm verification papers). There have been many new algorithms and impossibility results for this problem.

Special-Purpose Network Building Blocks. Some special problems that arise in communication networks have special solutions that seem able to combine with others. Major examples are the protocols of broadcast-convergecast, reset, end-to-end.

Self-stabilization. Informally, a protocol is said to be self-stabilizing if its specification does not require a certain "initial configuration" to be imposed on the system to ensure correct behavior of the protocol. The idea is that a self-stabilizing protocol is highly resilient: it can recover from any transient error. We shall see some of the basic self-stabilizing protocols, and survey some of the recent results.

Timing-based Systems. These systems lie between synchronous and asynchronous, so they have somewhat less uncertainty than the latter. They are put at the end of the course because they are a subject of newer research, and since they are less well understood. In these systems, processors have some knowledge of time, for example, access to real time, or approximate real time, or some timeout facility. Another possible assumption is that the processor step time, or message delivery time is within some known bounds. Time adds some extra structure and complication to state machine models, but it can be handled in the same general framework. In terms of the various time parameters, one can get upper and lower bounds for various interesting problems, such as mutual exclusion, resource allocation, consensus, and communication.

We may have time to look at some popular clock synchronization algorithms, and perhaps some of their applications.

1.2 Synchronous Network Algorithms

We start with the synchronous network algorithms. Our computation model is defined as follows. We are given a collection of nodes, organized into a directed graph $G = (V, E)$. We shall denote $n = |V|$. We assume the existence of some message alphabet M, and let *null* denote the absence of a message. Each node $i \in V$ has

$states(i)$	:	a set of states
$start(i)$	:	a subset of $states(i)$
$msgs(i)$	:	mapping $states(i) \times out\text{-}nbrs(i)$ to elements of M or *null*

$trans(i)$: mapping $states(i)$ and a vector of messages in $M \cup \{null\}$, one per in-neighbor of i, to $states(i)$

Execution begins with all the nodes in some start states, and all channels empty. Then the nodes, in lock step, repeatedly execute the following procedure called *round.*

1. Apply message generation function to generate messages for out-neighbors.
2. Put them in the channels.
3. Apply transition function to the incoming messages and the state to get the new state.

Remarks. Note the following points. First, the inputs are assumed to be encoded in the start states. Second, the model presented here is deterministic — the transitions and the messages are functions, i.e., single-valued.

1.2.1 Problem Example: Leader Election in a Ring

Consider a graph that is a ring, plus (a technicality for the problem definition) an extra dummy node representing the outside world. The ring is assumed to be unidirectional (messages can be sent only clockwise). The ring is of arbitrary size, unknown to the processors (i.e., size information is not built into their states). The requirement is that eventually, exactly one process outputs a *leader* message on its dummy outgoing channel.

A first easy observation is that if all the nodes are identical then this problem is impossible to solve in this model.

Proposition 1 *If all the nodes in the ring have identical state sets, start states, message functions and transition functions, then they do not solve the leader election problem for* $n > 1$.

Proof: It is straightforward to verify, by induction on the number of rounds, that all the processors are in identical states, after any number of rounds. Therefore, if any processor ever sends a *leader* message, then they all do so at the same time, violating the problem requirement. ∎

As indicated by Proposition 1, the only way to solve the leader election problem is to break the symmetry somehow. A reasonable assumption from practice is that the nodes are identical except for a unique identifier (UID), chosen from some large totally ordered set (e.g., the integers). We are guaranteed that each node's UID is different from each other's in the ring, though there is no constraint on which ID's actually appear in the given ring. (E.g., they don't have to be consecutive integers.) We exploit this assumption in the solutions we present.

1.2.2 Algorithm 1: LeLann, Chang-Roberts

We first describe the algorithm informally. The idea is that each process sends its identifier around the ring; when a node receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it's less than its own, it discards the incoming identifier; and if it's equal to its own, the process declares itself the leader. Intuitively, it seems clear that the process with the largest ID will be the one that outputs a *leader* message. In order to make this intuition precise, we give a more careful description of the system.

The message alphabet M consists of the UID's plus the special message *leader*.

The state set $state(i)$ consists of the following components.

- *own*, of type UID, initially i's UID
- *send*, of type UID or *null*, initially i's UID
- *status*, with values in $\{unknown, chosen, reported\}$, initially *unknown*

The start state $start(i)$ is defined by the initializations above.

The messages function $msgs(i)$ is defined as follows. Send message m, a UID, to clockwise neighbor exactly if m is the current value of *send*. Send *leader* to dummy node exactly if $status = chosen$.

The transition function is defined by the following pseudo-code.

```
Suppose new message is $m$.
$send := null$
If $status = chosen$ then $status := reported$
Case:
$m > own$ then
  $send := m$
$m = own$ then
  $status := chosen$
  $send := null$
else no-op
Endcase
```

Although the description is in a fairly convenient programming-language style, note that it has a direct translation into a state machine (e.g., each state consists of a value for each of the variables). We do not, in general, place restrictions on the amount of computation needed to go from one state to another. Also note the relative scarcity of flow of control statements. This phenomenon is pretty typical for distributed algorithms. The presence of such statements would make the translation from program to state machine less transparent.

How do we go about proving that the algorithm is correct? Correctness means that exactly one process ever does *leader* output. Let i_{max} denote the index (where indices are arranged consecutively around the ring) of the processor with the maximum UID. It suffices to show that (1) i_{max} does a *leader* output at round $n+1$, and (2) no other processor ever does such an output. We prove these properties, respectively, in the following two lemmas.

Lemma 2 *Node i_{max} outputs leader message at round $n+1$.*

Proof: Let v_{max} be the *own* value of i_{max}. Note that *own* values never change (by the code), that they are all distinct (by assumption), and that i_{max} has the largest (by definition of i_{max}). By the code, it suffices to show that after n rounds,

$$status(i_{max}) = chosen \ .$$

This can be proved (as things of this sort are typically proved) by induction on the number of computation steps, here the number of rounds. We need to strengthen the statement to be able to prove it by induction; we make the following assertion.

$$\text{For } 0 \leq r \leq n-1, \text{ after } r \text{ rounds }, \ \ send(i_{max}+r) = v_{max}$$

(Addition is modulo n.) In words, the assertion is that the maximum value appears at the position r away from i_{max}. Now this assertion is straightforward to prove by induction. The key step is that every node other than i_{max} admits the maximum value into its *send* component, since v_{max} is greater than all the other values. ∎

Lemma 3 *No process other than i_{max} ever sends a leader message.*

It suffices to show that all other processes always have $status = unknown$. Again, it helps to have an invariant. Let $[a,b)$ denote the set of positions $\{a, a+1, \ldots, b-1\}$, where addition is modulo n. The following invariant asserts that only the maximum value appear between the maximum and the owner in the ring.

$$\text{If } i \neq i_{max} \text{ and } j \in [i_{max}, i) \ , \text{ then } own(i) \neq send(j) \ .$$

Again, it is straightforward to prove the assertion by induction (the key step is that a non-maximum value does not get past the maximum). Obviously, we use the fact that v_{max} is greater than all the others.

Termination: As written, this algorithm never terminates, in the sense of all the processors reaching some designated final state. We could augment the model to include the concept of a final state. For this algorithm, to have everyone terminate, we could have the elected leader start a report message around the ring, and anyone who receives it can halt.

Note that final states do not play the same role, i.e., that of *accepting state* for distributed algorithms as they do for finite-state automata — normally, no notion of acceptance is used here.

Complexity analysis: The time complexity is $n+1$ rounds, for a ring of size n, until a leader is announced. The communication complexity is $O(n^2)$ messages.

Lecturer: Nancy Lynch

Lecture 2

2.1 Leader Election on a Ring (cont.)

Last time we considered the LeLann/Chang-Roberts algorithm. We showed that its time complexity is $n+1$ time units (rounds), and its communication complexity is $O(n^2)$ messages. The presentation of the algorithm was made fairly formally, and the proofs were sketched. Today we shall use less formal descriptions of three other algorithms.

2.1.1 Algorithm 2: Hirshberg-Sinclair

The time complexity of the LCR algorithm is fine. The number of messages, however, looks excessive. The first algorithm to reduce the worst-case complexity to $O(n \log n)$ was that of Hirshberg-Sinclair. Again, we assume that the ring size is unknown, but now we use *bidirectional* communication. As in LCR, this algorithm elects the process with the maximum UID.

The idea is that every process, instead of sending messages all the way *around* the ring as in the LCR algorithm, will send messages that "turn around" and come back to the originating process. The algorithm proceeds as follows.

> Each process sends out messages (in both directions) that go distances that are successively larger powers of 2 and then return to their origin (see Figure 2.1). When UID v_i is sent out by process i, other processes on v_i's path compare v_i to their own UID. For such a process j whose UID is v_j, there are two possibilities. If $v_i < v_j$, then rather than pass along the original message, j sends back a message to i telling i to stop initiating messages. Otherwise, $v_i > v_j$. In that case, j relays v_i, and since j can deduce that it cannot win, it will not initiate any new messages. Finally, if a process receives its own message before that message has "turned around", then that process is the winner.

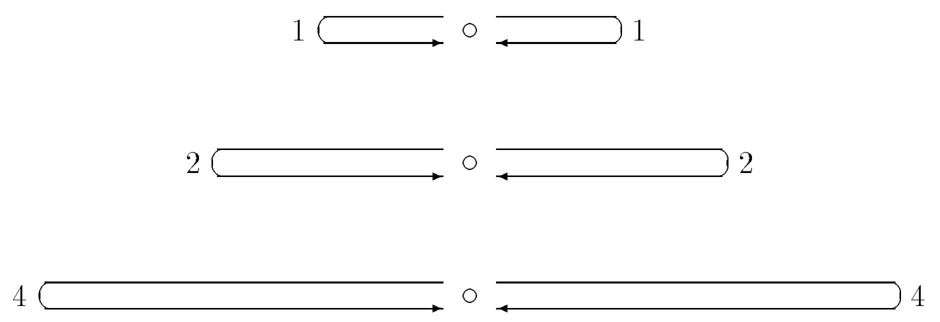


Figure 2.1: Successive message-sends in the Hirshberg-Sinclair algorithm

Analysis. A process initiates a message along a path of length 2^i only if it has not been defeated by another process within distance 2^{i-1} in either direction along the ring. This means that within any group of $2^{i-1}+1$ consecutive processes along the ring, at most one will go on to initiate messages along paths of length 2^i. This can be used to show that at most

$$\left\lfloor \frac{n}{2^{i-1}+1} \right\rfloor$$

in total will initiate messages along paths of length 2^i.

The total number of messages sent out is then bounded by

$$4\Big((1\cdot n)+(2\cdot\left\lfloor\frac{n}{2}\right\rfloor)+(4\cdot\left\lfloor\frac{n}{3}\right\rfloor)+\ldots(2^i\cdot\left\lfloor\frac{n}{2^{i-1}+1}\right\rfloor)+\ldots\Big)\ .$$

The factor of 4 in this expression is derived from the fact that each round of message-sending for a given process occurs in both directions — clockwise and counterclockwise — and that each outgoing message must turn around and return. (For example, in the first round of messages, each process sends out two messages — one in each direction — a distance of one each; and then each outgoing message returns a distance of one, for a net total of four messages sent.) Each term in the large parenthesized expression is the number of messages sent out around the ring at a given pass (counting only messages sent in one direction, and along the outgoing path). Thus, the first term, $(1\cdot n)$, indicates that all n processes send out messages for an outgoing distance of 1.

Each term in the large parenthesized expression is less than or equal to $2n$, and there are at most $1+\lceil\log n\rceil$ terms in the expression, so the total number of messages is $O(n\log n)$, with a constant factor of approximately 8.

The time complexity for this algorithm is just $O(n)$, as can be seen by considering the time taken for the eventual winner. The winning process will send out messages that take time 2, 4, 8, and so forth to go out and return; and it will finish after sending out the $\lceil \log n \rceil$th message. If n is an exact power of 2, then the time taken by the winning process is approximately $3n$, and if not the time taken is at most $5n$.

2.1.2 Counterexample Algorithms

Now we consider the question of whether it is possible to elect a leader with fewer $\Omega(n \log n)$ messages. The answer to this problem, as we shall demonstrate shortly with an impossibility result, is negative. That result, however, is valid only in a restricted model (comparison-based protocols, which are defined in Definition 1 below). For the general case, the following *counterexample algorithms* can be used to show that no such impossibility result can be proved. We use the term "counterexample algorithm" for an algorithm that isn't interesting by itself, e.g., neither practical nor particularly elegant from a mathematical viewpoint. However, it serves to show that an impossibility result cannot be proved.

Algorithm 3: We now suppose that the model is simpler, with only little uncertainty. Specifically, we assume the following.

- n is known to all processes.
- The communication is unidirectional.
- The ID's are positive integers.
- All processes start algorithm at the same time.

In this setting, the following simple algorithm works: [1]

> In the first n rounds, only a message marked with UID "1" can travel around. If a processor with UID "1"does exists, then this message is relayed throughout the ring. Otherwise, the first n rounds are idle. In general, the rounds in the range $kn+1, kn+2, \ldots, (k+1)n$ are reserved for UID $k+1$. Thus, the minimal UID eventually gets all the way around, which causes its originating process to get distinguished.

If it is desired to have the maximum elected instead of the minimum, simply let the minimum send a special message around at the end to determine the maximum.

[1] Actually, algorithm 3 isn't entirely a counterexample algorithm – it has really been used in some systems.

The nice property of the algorithm is that the total number of messages is n. Unfortunately, the time is about $n \cdot M$, where M is the value of the minimum ID; this value is unbounded for a fixed size ring. Note how heavily this algorithm uses the synchrony, quite differently from the first two algorithms.

Algorithm 4: Frederickson-Lynch. We now present an algorithm that achieves the $O(n)$ message bound in the case of unknown ring size (all other assumptions as for algorithm 2). The cost is time complexity which is even worse than algorithm 3 (!), specifically $O(n \cdot 2^M)$, where M is the minimum UID. Clearly, no one would even think of using this algorithm! It serves here just as a counterexample algorithm.

> Each process spawns a message which moves around the ring, carrying the UID of the original process. Identifiers that originate at different processes are transmitted at different rates. In particular, UID v travel at the rate of 1 message transmission every 2^v clock pulses. Any slow identifier that is overtaken by a faster identifier is deleted (since it has a larger identifier). Also, identifier v arriving at process w will be deleted if $w < v$. If an identifier gets back to the originator, the originator is elected.

This strategy guarantees that the process with the smallest identifier gets all the way around before the next smaller gets half-way, etc., and therefore (up to the time of election) would use more messages than all the others combined. Therefore total number of messages (up to the time of election) is less than $2n$. The time complexity, as mentioned above, is $n \cdot 2^M$. Also, note that by the time the minimum gets all the way around, all other transmissions have died out, and thus $2n$ is an upper bound on the number of messages that are *ever* sent by the algorithm (even after the "leader" message is output).

We remark that algorithms 1, 2 and 4 can also be used with the processors waking up at different times. The first two algorithms require no modification, but algorithm 4 needs some work in order to maintain the desired complexities.

2.1.3 Lower Bound on Comparison-Based Protocols

We have seen several algorithms for leader election on a ring. Algorithm 2 was *comparison-based*, and had the complexity of $O(n \log n)$ messages and $O(n)$ time. Algorithms 3 and 4 were non-comparison-based, and had $O(n)$ messages, with huge running time. To gain further understanding of the problem, we now show a lower bound of $\Omega(n \log n)$ messages for comparison-based algorithms [Frederickson-Lynch, Attiya-Snir-Warmuth]. This lower bound holds even if we assume that n is known to the processors.

The result is based on the difficulty of *breaking symmetry.* Recall the impossibility proof from last time, where we showed that because of the symmetry, it is impossible to elect a leader without identifiers. The main idea in the argument that we shall use below is that symmetry is possible with identifiers also — now it is possible to break it, but it must require much communication.

Before we state the main results, we need some definitions.

Definition 1 *We call a distributed protocol* comparison-based algorithm *if the nodes are identical except for UID's, and the only way that UID's can be manipulated is to copy them, and to compare them for* $\{<,>,=\}$ *(the results of the comparisons are used to make choices within the state machine). UIDs can be sent in messages, perhaps combined with other information.*

Intuitively, the decisions made by the state machines depend only on the relative ranks of the UIDs, rather than their value.

The following concept will be central in the proof.

Definition 2 *Let* $X = (x_1, x_2, \ldots, x_r)$ *and* $Y = (y_1, y_2, \ldots, y_r)$*, be two strings of UIDs. We say that* X *is* order equivalent *to* Y *if, for all* $1 \leq i, j \leq r$*, we have* $x_i \leq x_j$ *if and only if* $y_i \leq y_j$*.*

Notice that two strings of UIDs are order equivalent if and only if the corresponding strings of relative ranks of their UIDs are identical.

The following definitions are technical.

Definition 3 *A computation round is called an* active round *if at least one (non-null) message is sent in it.*

Definition 4 *The* k-neighborhood *of process* p *in ring of size* n*, where* $k < \lfloor n/2 \rfloor$*, is defined to consist of the* $2k+1$ *processes at distance at most* k *from* p*.*

The following concept is a key concept in the argument.

Definition 5 *We say that two states* s *and* t correspond *with respect to strings* $X = (x_1, x_2, \ldots, x_r)$ *and* $Y = (y_1, y_2, \ldots, y_r)$*, if all of the UID's in* s *are chosen from* X*, and* t *is identical to* s *except for substituting each occurrence of any* x_i *in* s*, by* y_i *in* t*, for all* $1 \leq i \leq r$*.*
Corresponding messages *are defined analogously.*

We can now start proving our lower bound.

Lemma 1 *Let* p *and* q *be two distinct processes executing a comparison-based algorithm. Suppose that* p *and* q *have order-equivalent* k*-neighborhoods. Then at any point after at most* k *active rounds,* p *and* q *are in corresponding states, with respect to their* k*-neighborhoods.*

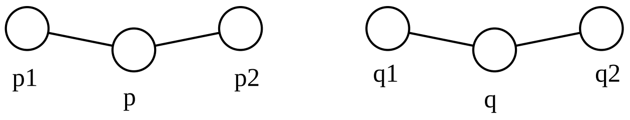


Figure 2.2: scenario in the proof of Lemma 1

Proof: By induction on the number r of rounds in the computation. Notice that possibly $r > k$. For each r, we prove the lemma for all k.

Basis: $r = 0$. By Definition 1, the initial states of p and q are identical except for their UIDs, and hence they are in corresponding initial states, with respect to their 0-neighborhoods (consisting of nothing but their own UID's).

Inductive step: Assume that the lemma holds for all $r' < r$. Let p_1 and p_2 be the respective counterclockwise and clockwise neighbors of p, and similarly q_1 and q_2 for q.

We proceed by case analysis.

1. Suppose that neither p nor q receives a message from either neighbor at round r. Then, by the induction hypothesis (on r, using same k), p and q are in corresponding states before r, and since they have no new input, they make corresponding transitions and end up in corresponding states after r.

2. Suppose now that at round r, p receives a message from p_1 but no message from p_2. Then round r is active. By induction, p_1 and q_1 are in corresponding states with respect to their $(k-1)$-neighborhoods, just before round r. Hence, q_1 also sends a message to q in round r, and it corresponds to the message sent by p_1, (with respect to the $(k-1)$-neighborhoods of p_1 and q_1, and therefore with respect to the k-neighborhoods of p and q). Similarly, p_2 and q_2 are in corresponding states with respect to their $k-1$-neighborhoods, just before round r, and therefore q_2 does not send a message to q at round r. Also, by induction, p and q are in corresponding states with respect to their $k-1$-neighborhoods, and hence with respect to their k-neighborhoods, just before round r. So after they receive the corresponding messages, they remain in corresponding states with respect to their k-neighborhoods.

3. Finally, suppose p receives a message from p_2. We use the same argument as in the previous case to argue that lemma holds in the two subcases (either p_1 send a message at round r or not).

∎

Lemma 1 tells us that many active rounds are necessary to break symmetry, if there are large order-equivalent neighborhoods. We now define particular rings with the special property that they have many order-equivalent neighborhoods of various sizes.

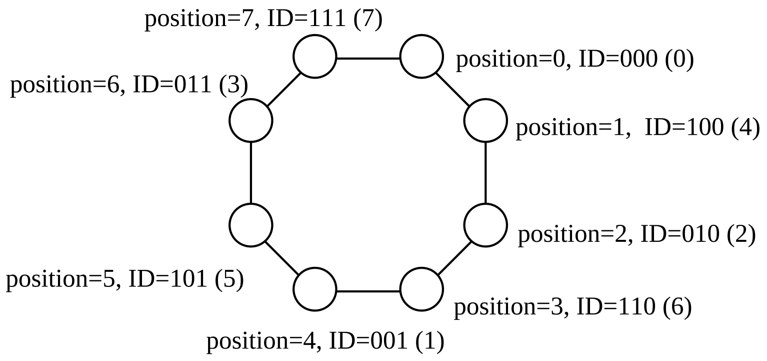


Figure 2.3: Bit-reversal assignment has 1/2-symmetry

Definition 6 *Let $c \leq 1$ be a constant, and let R be a ring of size n. R is said to have* c-symmetry *if for every ℓ, $\sqrt{n} \leq \ell \leq n$, and for every segment S of R of length ℓ, there are at least $\frac{cn}{\ell}$ segments in R that are order-equivalent to S (counting S itself).*

Remark: the square root is a technicality.

Example: For n a power of 2, we shall show that the *bit-reversal* ring of size n has 1/2-symmetry. Specifically, suppose $n = 2^a$. We assign to each process i the integer in the range $[0, n-1]$ whose a-bit binary representation is the *reverse* of the a-bit binary representation of i.

For instance, for $n = 8$, we have $a = 3$, and the assignment is in Figure 2.3.

The bit-reversal ring is highly symmetric, as we now claim.

Claim 2 *The bit-reversal ring is* 1/2*-symmetric.*

Proof: Left as an exercise. ∎

We remark that for the bit-reversal ring, there is even no need for the square root caveat.

For other values of n, non-powers of 2, there is a more complicated construction that can yield c-symmetry for some smaller constant c. The construction is complicated since simply adding a few extra processors could break the symmetry.

Now, suppose we have a ring R of size n with c-symmetry. The following lemma states that this implies many active rounds.

Lemma 3 *Suppose that algorithm A elects a leader in a c-symmetric ring, and let k be such that $\sqrt{n} \leq 2k+1 \leq n$ and $\frac{cn}{2k+1} \geq 2$. Then A has more than k active rounds.*

Proof: By contradiction: suppose A elects a leader, say p, in at most k active rounds. Let S be the k-neighborhood of p (S is a segment of length $2k+1$). Since R is c-symmetric, there must be at least one other segment in R which is order-equivalent to S; let q be the midpoint of that segment. Now, by Lemma 1, p and q remain in equivalent states throughout the execution, up to the election point. We conclude that q also gets elected, a contradiction.
∎

Now we can prove the lower bound.

Theorem 4 *All comparison-based leader election algorithms for rings of size n send $\Omega(n \log n)$ messages before termination.*

Proof: Define

$$K = 1 + \max\left\{k \ : \ \sqrt{n} \le 2k+1 \le n \text{ and } \frac{cn}{2k+1} \ge 2\right\} .$$

Note that $K \approx \frac{cn}{4}$.

By Lemma 3 there are at least K active rounds. Consider the rth active round, where $1 \le r \le K$. Since the round is active, there is some processor p that sends a message in round r. Let S be the $(r-1)$-neighborhood of p. Since R is c-symmetric, there are at least $\frac{cn}{2r-1}$ segments in R that are equivalent to S (provided $\sqrt{n} \le r-1$). By Lemma 1, at the point just before the rth active round, the midpoints of all these segments are in corresponding states, so they *all* send messages.

Thus, the total number of messages is at least

$$\begin{aligned}
\sum_{r=\sqrt{n}+1}^{K} \frac{cn}{2r-1} &= \Omega\left(cn \sum_{r=\sqrt{n}}^{\frac{cn}{4}} \frac{1}{r}\right) \\
&= \Omega\left(cn \left(\ln \frac{cn}{4} - \ln \sqrt{n}\right)\right) \qquad \text{integral approximation of the sum} \\
&= \Omega(n \log n) .
\end{aligned}$$

∎

Remark: The paper by Attiya, Snir and Warmuth contains other related results about limitations of computing power in synchronous rings.

2.2 Leader Election in a General Network

So far, we considered only a very special kind of networks, namely rings. We now turn to consider more general networks, in the same synchronized model. We shall start with leader election.

Let us first state the problem in the general framework. Consider arbitrary, unknown, strongly connected network digraph $G = (V, E)$. We assume that the nodes communicate only over the directed edges.[2] Again, nodes have UIDs, but the set of actual UIDs is unknown. The goal is, as before, to elect a leader.

To solve the problem, we use a generalization of the idea in the LCR algorithm as follows.

[2] In the BFS example below we will remove this restriction and allow two-way communication even when the edges are 1-way.

> Every node maintains a record of the maximum UID it has seen so far (initially its own). At each round, node propagates this maximum on all its outgoing edges.

Clearly, the processor with the maximum UID is the only one that will keep its own UID forever as the maximum. But how does the processor with the maximum "know" that it is the maximum, so it can do an *elected* output? In LCR, it was enough that a processor received its own UID on an incoming link, but that was particular to the ring network topology, and cannot work in general. One possible idea is to wait for sufficiently long time, until the process is guaranteed that it would have received any larger value, if one existed. It can be easily seen that this time is exactly the maximum distance from any node in the graph to any other, i.e., the *diameter*. Hence, this strategy requires some built-in information about the graph. Specifically, every node must have an upper bound on the diameter.[3]

We now specify the above algorithm formally. We describe process i.

```
state(i) consists of components:
    own, of type UID, initially i's UID
    max-id, of type UID, initially i's UID
    status, with values in {unknown, chosen, reported}, initially unknown
    rounds, integer, initially 0.
```

msgs(i): Send message m, a UID, to clockwise neighbor exactly if m is the current value of *max-id*.

Send *leader* to dummy node exactly if *status* = *chosen*.

```
trans(i):
    Suppose new message from each neighbor j is m(j).
    If status = chosen then status := reported
    rounds := rounds + 1
    max-id := max({max-id} ∪ {m(j) : j ∈ in-nbrs(i)})
    If rounds ≥ diam and max-id = own and status = unknown then status := chosen
```

Complexity: it is straightforward to observe that the time complexity is *diam* rounds, and that the number of messages is $diam \cdot |E|$.

[3]It is possible to design a general algorithm for strongly connected digraphs that doesn't use this diameter information; it is left as an exercise.

2.3 Breadth-First Search

The next problem we consider is doing BFS in a network with the same assumptions as above. More precisely, suppose there is a distinguished *source node* i_0. A *breadth-first* spanning tree of a graph is defined to have a *parent* pointer at each node, such that the parent of any node at distance d from i_0 is a node at distance $d-1$ from i_0. The distinguished node i_0 has a *nil* pointer, i.e., i_0 is a root. The goal in computing a BFS tree is that each node will eventually output the name of its parent in a breadth-first spanning tree of the network, rooted at i_0.

The motivation for constructing such a tree comes from the desire to have a convenient communication structure, e.g., for a distributed network graph. The BFS tree minimizes the maximum communication time to the distinguished node.

The basic idea for the algorithm is the same as for the sequential algorithm.

> At any point, there is some set of nodes that is "marked", initially just i_0. Other nodes get marked when they first receive *report* messages. The first round after a node gets marked, it announces its parent to the outside world, and sends a *report* to all its outgoing neighbors, except for all the edges from which the *report* message was received. Whenever an unmarked node receives a *report*, it marks itself, and chooses one node from which the *report* has come as its parent.

Due to the synchrony, this algorithm produces a BFS tree, as can be proved by induction on the number of rounds.

Complexity: the time complexity *diam* rounds (actually, this could be refined a little, to the maximum distance from i_0 to any node). The number of messages is $|E|$ — each edge transmits a report message exactly once.

2.3.1 Extensions

The BFS algorithm is very basic in distributed computing. It can be augmented to execute some useful other tasks. Below, we list two of them. As mentioned earlier, we shall assume that *communication is bidirectional* on all links (even though the graph we compute for might have directed edges).

Message broadcast. Suppose that a processor has a message that it wants to communicate to all the processors in the network. The way to do it is to "piggyback" the message on the *report* messages, when establishing a BFS tree. Another possibility is to first produce a BFS tree, and then use it to broadcast. Notice that for this, the pointers go opposite to the direction of the links. We can augment the algorithm by letting a node that discovers a parent communicate with it so it knows which are its children.

Broadcast-convergecast. Sometimes it is desirable to collect information from throughout the network. For example, let us consider the problem in which each node has some initial input value, and we wish to find to global sum of the inputs. This is easily (and efficiently) done as follows. First, establish a BFS tree, which includes two-way pointers so parents know their children. Then *starting from the leaves*, "fan in" the results: each leaf sends its value to its parent; each parent waits until it gets the values from all its children, adds them (and its input value), and then sends the sum to its own parent. When the root does its sum, the final answer is obtained. The complexity of this scheme is $O(diam)$ time (we can refine as before); the number of messages is $|E|$ to establish the tree, and then an extra $O(n)$ to establish the child pointers and fan in the results.

We remark that funny things happen to the time when the broadcast-convergecast is run asynchronously. We will revisit this as well as the other algorithms later, when we study asynchronous systems.

Lecturer: Nancy Lynch

Lecture 3

3.1 Shortest Paths: Unweighted and Weighted

Today we continue studying synchronous protocols. We begin with a generalization of the BFS problem. Suppose we want to find a *shortest paths* from a distinguished node $i_0 \in V$ (which we sometimes refer to as the *root*), to each node in V. Now we require that each node should not only know its parent on a shortest path, but also the distance.

If all edges are of equal weights, then a trivial modification of the simple BFS tree construction can be made to produce the distance information as well. But what if there are *weights* assigned to edges?

This problem arises naturally in communication networks, since in many cases we have some associated cost, for instance, monetary cost, to traverse the link.

Specifically, for each edge (i, j), let $weight(i, j)$ denote its weight. Assume that every node "knows" the weight of all its incident edges, i.e., the weight of an edge appears in special variables at both its endpoint nodes. As in the case of BFS, we are given a distinguished node i_0 and the goal is to have, at each node, the *weighted* distance from i_0. The way to solve it is as follows.

> Each node keeps track of the *best-dist* it knows from i_0. Initially, *best-dist*$(i_0) = 0$ and *best-dist*$(j) = \infty$ for $j \neq i_0$. At each round, each node sends its *best-dist* to all its neighbors. Each recipient node i updates its *best-dist* by a "relaxation step", where it takes the minimum of its previous *best-dist* value, and of *best-dist*$(j) +$ $weight(j, i)$ for each incoming neighbor j.

It can be proven that, eventually, the *best-dist* values will converge to the correct best distance values. The only subtle point is when this occurs. It is no longer sufficient to wait *diam* time — see, for example, Figure 3.1. However, $n - 1$ time units are sufficient. In fact, the way to argue the correctness of the above algorithm (which is a variant of the well known Bellman-Ford algorithm), is by induction on the number of links in the best paths from i_0.

This seems to imply that a bound on the number of nodes must be known, but this information can easily be computed.

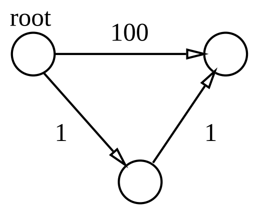


Figure 3.1: the correct shortest paths from the root stabilize only after 2 time units, even though all nodes are reachable in one step.

3.2 Minimum Spanning Tree

We shall now modify our model slightly, and assume that the communication graph is undirected, i.e., if there is an edge between nodes i and j, then i and j can exchange messages in both directions. The edges, as in the previous algorithm, are assumed to have weights which are known at their endpoints. We continue to assume that nodes have UIDs.

Before we define the problem, let us define a few standard graph-theoretic notions. A *spanning tree* for a graph $G = (V, E)$ is a graph $T = (V, E')$, where $E' \subseteq E$, such that T is connected and acyclic; if T is acyclic (but not necessarily connected), T is called a *spanning forest*. The *weight* of a graph is the sum of the weights of its edges.

The *Minimum Spanning Tree* (MST) problem is to find a spanning tree of minimal weight. In the distributed setting, the output is a "marked" set of edges — each node will mark those edges adjacent to it that are in the MST. This problem arises naturally in the context of communication networks, since similarly to the BF tree, the MST is a convenient communication structure. The MST minimizes the cost of broadcast (i.e., when a node wishes to communicate a message to all the nodes in the network).

All known MST algorithms are based on the same simple theory. Notice that any acyclic set of edges (i.e., a forest) can be augmented to a spanning tree. The following lemma indicates which edges can be in a minimum spanning tree.

Lemma 1 *Let $G = (V, E)$ be an undirected graph, and let $\{(V_i, E_i) : 1 \le i \le k\}$ be any spanning forest for G, where each (V_i, E_i) is connected. Fix any i, $1 \le i \le k$. Let e be an edge of lowest cost in in the set*

$$\left\{ e' : e' \in E - \bigcup_j E_j \textit{ and exactly one endpoint of } e' \textit{ is in } V_i \right\} .$$

Then there is a spanning tree for G that includes $\bigcup_j E_j$) and e, and this tree is of as low a cost as any spanning tree for G that includes $\bigcup_j E_j$.

Proof: By contradiction. Suppose the claim is false — i.e., that there exists a spanning tree T that contains $\bigcup_j E_j$, does not contain e, and is of strictly lower cost than any other

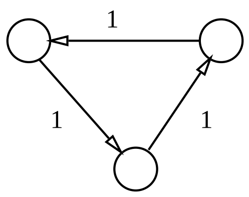


Figure 3.2: the circles represent components. If the components choose MWOE as depicted by the arrows, a cycle would be created.

spanning tree which contains $\{e\} \cup \bigcup_j E_j$. Now, consider the graph T' obtained by adding e to T. Clearly, T' contains a cycle, which has one more edge $e' \neq e$ which is outgoing from V_i.

By the choice of e, we know that $\mathit{weight}(e') \geq \mathit{weight}(e)$. Now, consider T^* constructed by deleting e' from T'. T^* is spanning tree for G, it contains $\{e\} \cup \bigcup_j E_j$, and it has weight no greater than that of T, a contradiction. ▮

The lemma above suggests a simple strategy for constructing a spanning tree: start with the trivial spanning forest that contains no edges; then, for some connected component, add the minimal-weight outgoing edge (MWOE). The claim that we have just proven guarantees that we are safe in adding these edges: there is an MST that contains them. We can repeat this procedure until there is only one connected component, which is an MST. This principle forms the basis for well-known sequential MST algorithms. The Prim-Dijkstra algorithm, for instance, starts with one node and successively adds the smallest-weight outgoing edge from the current (partial) tree until a complete spanning tree has been obtained. The Kruskal algorithm, as another example, starts with all nodes as "fragments" (i.e., connected components of the spanning forest) and successively extends each fragment with the minimun weight outgoing edge, thus combining fragments until there is only one fragment, which is the final tree. More generally, we could use the following basic strategy: start with all nodes as fragments and successively extend an arbitrary fragment with its MWOE, combining fragments where possible.

It is not readily clear whether we can do this in parallel. The answer, in general, is negative. Consider, for example, the graph in Figure 3.2.

However, if all the edges have *distinct* weights, then there is no problem, as implied by following lemma.

Lemma 2 *If all edges of a graph G have distinct weights, then there is exactly one MST for G.*

Proof: Similar to the proof of Lemma 1; left as an exercise. ▮

By Lemma 1, if we have a forest all of whose edges are in the unique MST, then all

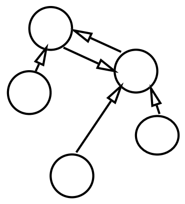


Figure 3.3: arrows represent MWOEs. Exactly one edge is chosen twice in each new component.

MWOE's for all components are also in the unique MST. So we can add them all in at once, in parallel, and there is no danger of creating a cycle. So all we have to worry about is making the edge weights unique: this is easily done using the UIDs. The weight of an edge (i, j) will, for our purposes, be the triplet $(weight, v_i, v_j)$, where $v_i < v_j$, and v_i (resp., v_j) denotes the UID of node i (resp., j). The order relation among the edges is determined by the lexicographical order among the triplets.

Using Lemma 2 and the UIDs as described above, we can now sketch an algorithm for distributed construction of MST. The algorithm builds the component in parallel, in "levels" as follows. It is based on the asynchronous algorithm of Gallager, Humblet and Spira.

> Start with Level 0 components consisting of individual nodes, and no edges. Suppose inductively that we have Level i components, each with its own spanning tree of edges established, and with a known leader within the component. To get the Level $i+1$ components, each Level i component will conduct a search along its spanning tree nodes, for the MWOE of the component. Specifically, the leader broadcasts search requests along tree edges. Each node must find its own MWOE — this is done by broadcasting along all non-tree edges to see whether the other end is in same component or not. (This can be checked by comparing the leader's UID, which is used at all the nodes of a component as the component UID.) Then, the nodes convergecast the responses toward the leader, while taking minimums along the way.
>
> When all components have found their MWOE's, the fragments are merged by adding the MWOE's in — the leader can communicate with the source of the MWOE to tell it to mark the edge. Finally, a new leader must be chosen. This can be done as follows. It is easy to see that for each new connected component, there is a unique new edge that is MWOE of both endpoint components (this is the edge with the least weight among all the new edges). See Figure 3.3 for example. We can let new leader be the larger UID endpoint of this edge.

Note that it is crucial to keep the levels synchronized: so when a node inquires if the other endpoint of a candidate edge is in the same component or not, the other endpoint has up-to-date information. If the UID at the other end is different, we want to be sure that the other end is really in a different component, and not just that it hasn't received the component UID yet.

Therefore, we execute the levels synchronously, one at a time. To be sure we have completed a level before going on to the next, we must allow an overestimate of time (which, unfortunately, can be big). The time for a level can be $\Theta(n)$, in the worst case. (Not $O(diam)$ – the paths on which communication takes place are not necessarily minimum in terms of number of links.) The number of levels is bounded by $\log n$, since number of components is reduced by at least a half at each level.

We conclude that the time complexity is $O(n \log n)$. The message complexity is $O((n + E) \cdot \log n)$, since in each level $O(n)$ messages are sent along all the tree edges, and $O(E)$ additional messages are required to do the exploration for local MWOE's.

Remark: The number of messages can be reduced to $O(n \log n + |E|)$ by being more clever about the local MWOE search strategy. The optimization makes the time increase by at most $O(n)$ time units. The idea is as follows. Each node marks the edges when they are known to be in same component — there is no need to "explore" them again. Also, the edges are explored in order, one at a time, starting from the lightest. This way each edge is either marked or discovered to be the local MWOE. For the message complexity analysis, we note that the number of messages sent over tree edges is, as before, $O(n \log n)$. Let us apportion the cost of explore messages. Each edge gets explored and rejected at most once in each direction, for a total of $O(|E|)$. There can be repeated exploration of edges that get locally accepted (i.e., are found to be outgoing), but there is at most one such exploration per node in each level, summing up for a total of $O(n \log n)$.

3.3 Maximal Independent Set

So far, we saw a collection of basic synchronous network algorithms, motivated by graph theory and practical communications. We shall now see one more example, before going to study models with failures, that of finding a *Maximal Independent Set* of the nodes of a graph.

This problem can be motivated by resource allocation problems in a network. The neighbors in the graph represent processes than cannot simultaneously perform some activity involving a shared resource (such as radio broadcast). It is undesirable to have a process blocked if no neighbors are broadcasting, hence the maximality requirement.

The following algorithm, due to Luby, is particularly interesting because is simple and basic, and because it introduces the use of *randomization.*

Problem Statement. Let $G = (V, E)$ be a undirected graph. A set $I \subseteq V$ of nodes is *independent* if for all nodes $i, j \in I$, $(i, j) \notin E$. An independent set I is *maximal* if any set I' which strictly contains I is not independent. Our goal is to compute a maximal independent set of G. The output is done by each member of I eventually sending a special message *in-I.*

We assume that a bound B on n, the number of nodes, is known to all the nodes. We do not need to assume the existence of UID's — i.e., we assume an "anonymous" network.

The Algorithm. The basic underlying strategy is based on the following iterative scheme. Let $G = (V, E)$ be the initial graph.

```
$I := \phi$
while $V \neq \phi$ do
  choose a set $I' \subseteq V$ that is independent in $G$
  $I := I \cup I'$
  $G :=$ induced subgraph of $G$ on $V - I' - nbrs(I')$
end while
```

This scheme always produces a maximal independent set: by construction, I is independent, and we throw out neighbors of G any element put into I; maximality is also easy, since the only nodes that are removed from consideration are neighbors of nodes inserted into I.

The key question in implementing this general "skeleton" algorithm is how to choose I' at each iteration. We employ a new powerful idea here, namely *randomization.* Specifically, in each iteration, each node i chooses an integer $val(i)$ in the range $\{1, \ldots, B^4\}$ at random, using the uniform distribution. (Here, we require the nodes to know B.) The upper bound B^4 was chosen to be sufficiently large so that with high probability, all nodes choose distinct values. Having chosen these values, we define I' to consist of all the nodes i such that $val(i) > val(j)$ for all nodes j neighbors of i. This obviously yields an independent set, since two neighbors cannot simultaneously be greater than each other.

Formal Modeling. When we come to describe formally this synchronous algorithm, we encounter a problem: it doesn't quite fit the model we have already defined. We need to augment the model with some extra steps for the random choices. The option we adopt here is to introduce a new "function", in addition to the message generation and transition functions, to represent the random choice steps. Formally, we add an *int* ("internal transition") component to the automaton description, where for each state s, $int(s)$ is a probability distribution over the set of states. Each execution step will now proceed by first using *int* to pick a new (random) state, and then apply *msgs* and *trans* as usual.

For the MIS algorithm, we first outline the code in words. The algorithm works in *phases*, where each phase consists of three rounds.

1. In the first round of a phase, the nodes choose their respective *val*'s, and send them to their neighbors. By the end of the first round, when all the values have been received, the winners discover they are in I'.

2. In the second round of a phase, the winners announce they are in I'. By the end of this round, each node learns whether it has a neighbor in I'.

3. In the third round, each node with a neighbor in I' tells all *its* neighbors, and then all the involved nodes: the winners, their neighbors, and the neighbors of winners' neighbors, remove the appropriate nodes and edges from the graph. Specifically, this means the winners and their neighbors discontinue participation after this phase,and the neighbors of neighbors will remove all the edges that are incident on the newly-removed neighbors.

Let us now describe the algorithm formally in our model.

```
State:
  $phase$, integer, initially 1
  $round$: in $\{1, 2, 3\}$, initially 1
  $val$: in $\{1, \dots B^4\}$,
  $nbrs$: set of vertices, initially the neighbors in the original graph $G$
  $awake$: Boolean, initially $true$
  $winner$: Boolean, initially $false$
  $neighbor$: Boolean, initially $false$


$int(i)$: (the global $int$ is described by independent choices at all the nodes)
  if $awake$ and $round = 1$ then $val := rand$ (from uniform distribution)


$msgs(i)$: if $awake = true$ and
  if $round = 1$:
    send $val(i)$ to all nodes in $nbrs$
  if $round = 2$:
    if $winner = true$ then
      send $in\text{-}I$ to dummy node
      send $winner$ to all nodes in $nbrs$
  if $round = 3$:
    if $neighbor = true$ then
      send $neighbor$ to all nodes in $nbrs$
```

Below, we use the notation $m.val$ to denote the particular component of the message.

```
$trans(i)$: if $awake = true$ then
  case:
    $round = 1$:
      if $val > m(j).val$ for all $j \in nbrs$ then $winner := true$
    $round = 2$:
      if some $winner$ message arrives then $neighbor := true$
    $round = 3$:
      if $winner \vee neighbor$ then $awake := false$
      $nbrs := nbrs - \{j :$ a $neighbor$ message arrives from $j\}$
  endcase
  $round := (round \bmod 3) + 1$
```

Analysis. Complete analysis can be found in the original paper by Luby. Here, we just sketch the proof.

We first state, without proof, a technical lemma which we need.

Lemma 3 *Let $G = (V, E)$, and define for an arbitrary node $i \in V$,*

$$sum(i) = \sum_{j \in nbrs(i)} \frac{1}{d(j)} ,$$

where $d(j)$ is the degree of j in G. Let I' be defined as in the algorithm above. Then

$$\Pr[i \in nbrs(I')] \geq \frac{1}{4} \min(sum(i), 1).$$

Informally, the idea in proving Lemma 3 is that since the random choices made by the different nodes are *independent identically distributed* (iid) random variables, the probability of a given node i with $d(i)$ neighbors to be a local maximum is approximately $1/d(i)$. Since we are dealing with overlapping events, the inclusion-exclusion principle is used to obtain the bound in the lemma.

The next lemma is the main lemma in proving an upper bound on the expected number of phases of the algorithm.

Lemma 4 *Let $G = (V, E)$. The expected number of edges removed from G in a single phase of the algorithm is at least $|E|/8$.*

Proof: By the algorithm, all the edges with at least one endpoint in $nbrs(I')$ are removed, and hence, the expected number of edges removed in a phase is at least

$$\frac{1}{2}\sum_{i \in V} d(i) \cdot \Pr[i \in nbrs(I')]$$

This is since each vertex i has the given probability of having a neighbor in I' and if this event occurs, then it is certainly removed, and thereby it causes the deletion of $d(i)$ edges. The $\frac{1}{2}$ is included to take account of possible overcounting, since an edge can have two endpoints that may cause its deletion,

We now plug in the bound from Lemma 3:

$$\frac{1}{8}\sum_{i \in V} d(i) \cdot \min(sum(i), 1)\ .$$

Breaking this up according to which term of the "min" is less, this equals

$$\frac{1}{8}\left(\sum_{sum(i)<1} d(i) \cdot sum(i) + \sum_{sum(i)\geq 1} d(i)\right)$$

Now expand the definition of $sum(i)$ and write $d(i)$ as sum of 1:

$$\frac{1}{8}\left(\sum_{i:sum(i)<1}\ \sum_{j \in nbrs(i)} \frac{d(i)}{d(j)} + \sum_{i:sum(i)\geq 1}\ \sum_{j \in nbrs(i)} 1\right)$$

Next, we make the sum over *edges* – for each edge, we have two endpoints for which the above expression adds in the indicated terms, and so the preceding expression is equal to

$$\frac{1}{8}\left(\sum_{\substack{i:sum(i)<1\\ j:sum(j)<1}} \left(\frac{d(i)}{d(j)} + \frac{d(j)}{d(i)}\right) + \sum_{\substack{i:sum(i)<1\\ j:sum(j)\geq 1}} \left(\frac{d(i)}{d(j)} + 1\right) + \sum_{\substack{i:sum(i)\geq 1\\ sum(j)\geq 1}} (1+1)\right)\ .$$

Now each of the expressions within the summation terms is greater than 1, so the total is at least $\frac{1}{8}|E|$. ∎

Using Lemma 4, it is straightforward to prove a bound on the expected number of phases in Luby's algorithm.

Theorem 5 *The expected number of phases of the MIS algorithm is $O(\log n)$.*

Proof: By Lemma 4, 1/8 of the edges of the current graph are removed in each phase. Therefore, the total expected running time is $O(\log |E|) = O(\log n)$. ∎

Remark. Actually, a slightly stronger result holds: the number of phases in the MIS algorithm can be proven to be $O(\log n)$ *with high probability.*

Discussion. Randomization is a technique that seems frequently very useful in distributed algorithms. It can be used to break symmetry, (e.g., it can be used for the leader election problem; cf. homework problem). However, when designing randomized algorithms, one thing that we must be careful about is that there is some possibility that things will go wrong. We have to make sure that such anomalies do not cause serious problems. In MIS algorithms, for example, probabilistic anomalies may result in outputting a set which is not independent, or not maximal. Fortunately, Luby's algorithm always produces an MIS. The *performance* of the algorithm, however, depends crucially on the random choices — it can take more rounds if we are unlucky and "bad" choices are made. This is generally considered to be acceptable for randomized algorithms. Even so, there is something worse here — it is possible to have equal choices made repeatedly for neighbors, which means that there is a possibility (which occurs with zero probability) that the algorithm never terminates. This is less acceptable.

Lecturer: Nancy Lynch

Lecture 4

Our next topic is the major example we shall study in the synchronous model, namely, *consensus algorithms*. We shall spend some 2-3 lectures on this subject. The *consensus* problem is a basic problem for distributed networks which is of interest in various applications. In studying this problem, we extend our model to include a new source of uncertainty: failures of communication links and processors.

4.1 Link Failures

4.1.1 The Coordinated Attack Problem

The problem we will describe now is an abstraction of a common problem arising in distributed database applications. Informally, the problem is that at the end of processing a given database transaction, all the participants are supposed to agree about whether to *commit* (i.e., accept the results of) the transaction or to *abort* it. We assume that the participants can have different "opinions" initially, as to whether to commit or abort, e.g., due to local problems in their part of the computation.

Let us state the problem more precisely, cast in terms of *generals*; the abstraction is due to Gray.

> Several (sometimes only two) generals are planning to attack (from different directions) against a common objective. It is known that the only way for the attack to succeed, is if *all* the generals attack. If only some of the generals attack, their armies will be destroyed. There may be some reasons why one of the generals cannot attack – e.g., insufficient supplies. The generals are located in different places, and can communicate only via messengers. However, messengers can be caught, and their messages may be lost en route.

Notice that if messages are guaranteed to arrive at their destinations, then all the generals could send messengers to all the others, saying if they are willing to attack (in our previous terminology, all the nodes will broadcast their inputs). After sufficiently many rounds, all the inputs will propagate to all the participants (in the graph setting, this requires knowledge of an upper bound on the diameter). Having propagated all the inputs, virtually any rule can be used: in our case, the rule is "attack if there is no objection". Since all the participants apply the same rule to the same set of input values, they will all decide identically.

In a model with more uncertainty, in which messages may be lost, this simplistic algorithm fails. It turns out that this is not just a problem with this algorithm; we shall prove that there is *no* algorithm that always solves this problem correctly.

Before we go on to prove the impossibility result, we state the problem a bit more carefully. We consider the following model. Consider processes i, for $1 \leq i \leq n$, arranged in an undirected graph G. We assume that all processes start with initial binary values (encoded in their states). We use the same synchronous model we have been working with so far, except that, during the course of execution, any number of messages might be lost. The goal is to reach *consensus* (i.e., identical decisions) in a fixed number, say r, of rounds. We represent the decision by a special value encoded in the processes' states at the end of the last round. (Alternatively, we can require that the decision value will be output using special messages; the difference is simply one more round.)

We impose two conditions on the decisions made by the processes as follows.

Agreement: All processes decide on the same value.

Validity: If all processes start with 0, then the decision value is 0; and if all processes start with 1 and all messages are delivered, then the decision value is 1.

The agreement requirement is natural: we would certainly like to be consistent. The validity requirement is one possible formalization of the natural idea that the value decided upon should be "reasonable". For instance, the trivial protocol that always decides 0 is ruled out by the validity requirement. We comment that the validity condition above is quite weak: if even one message is lost, the protocol is allowed to decide on any value. Nevertheless, it turns out that even this weak requirement is impossible to satisfy in any nontrivial graph (i.e., a graph with more than one node). To see this, it suffices to consider the special case of two nodes connected by one edge (cf. homework problem).

We shall need the following standard definition.

Definition 1 *Let α and β be two executions. We say that α is* indistinguishable *from β with respect to a process i, denoted $\alpha \overset{i}{\sim} \beta$, if in both α and β, i has the same initial state and receives exactly the same messages at the same rounds.*

The basic fact for any distributed protocol, is that if $\alpha \overset{i}{\sim} \beta$, then i behaves the same in α and β. This simple observation is the key in the proof of the following theorem.

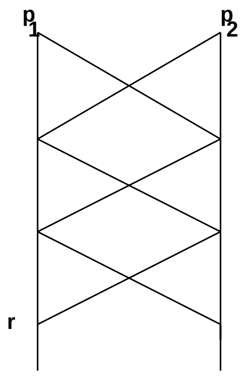


Figure 4.1: Pattern of message exchanges between p_1 and p_2

Theorem 1 *There is no algorithm that solves the coordinated attack problem on a graph with two nodes connected by an edge.*

Proof: By contradiction. Suppose a solution exists, say for r rounds. Without loss of generality, we may assume that the algorithm always sends messages in both directions at every round, since we can always send dummy messages. We shall now construct a series of executions, such that each of them will be indistinguishable from its predecessor with respect to one of the processes, and hence all these executions must have the same decision value.

Specifically, let α_1 be the execution that results when both processes 1 and 2 start with value 1, and all messages are delivered. By validity, both must decide 1 at the end of α_1. Now let α_2 be the execution that is the same as α_1, except that the last message from process 1 to 2 is not delivered. Note that although process 2 may go to a different state at the last round, $\alpha_1 \overset{1}{\sim} \alpha_2$, and therefore, process 1 decides 1 in α_2. By agreement, process 2 must also decide 1 in α_2. Next, let α_3 be the same as α_2 except that the last message from process 2 to 1 is lost. Since $\alpha_2 \overset{2}{\sim} \alpha_3$, process 2 decides 1 in α_3, and by agreement, so does process 1.

Continuing in this way, by alternately removing messages, eventually we get down to an execution α^* in which both processes start with 1 and no messages are delivered. As before, both processes are forced to decide 1 in this case.

But now consider the execution in which process 1 starts with 1 but process 2 starts with 0, and no messages are delivered. This execution is indistinguishable from α^* with respect to process 1, and hence process 1 will still decide 1, and so will process 2, by agreement. But this execution is indistinguishable from the execution in which they both start with 0 with restpect to process 2, in which validity requires that they both decide 0, a contradiction. ∎

Theorem 1 tells us, in simple words, that there is very little that we can do in distributed networks if the communication may be faulty, and the correctness requirements are strict. This result is considered to reflect a fundamental limitation of the power of distributed networks.

However, something close to this problem *must* be solved in real systems. Recall the database commit problem. There, we had even a stronger condition for validity, namely, if anyone starts with 0, then the decision value must be 0. How can we cope with the impossibility indicated by Theorem 1? We must relax our requirements. One possible idea is to use randomization in the algorithm, and allow some probability of violating the agreement and/or validity condition. Another approach is to allow some chance of nontermination. We will return to nonterminating commit protocols in a lecture or two.

4.1.2 Randomized Consensus

In this section we discuss some properties of randomized protocols for consensus. We shall consider, as above, processes in an arbitrary connected graph. Our model will allow randomized processes, as for the MIS (Maximal Independent Set) problem.

In this section we weaken the problem requirement, and we allow for some small probability ϵ of error. Consider a protocol that works in a fixed number r of rounds. Our main results in this section [VargheseL92], are bounds on the "liveness probability" that an algorithm can guarantee in terms of ϵ and r, that is, for the probability that all the participants attack in the case where both start with 1 and all messages are delivered.

We shall give an *upper bound* on the liveness probability. (Note that here an "upper bound" is a negative result: no algorithm will decide to attack in this case with too high probability.) We shall also see an algorithm that (nearly) achieves this probability.

Formal Modeling. We must be careful about the probabilistic statements in this setting, since it is more complicated than for the MIS case. The complication is that the execution that unfolds depends not only on the random choices, but also on the choices of which messages are lost. These are not random. Rather, we want to imagine they are under the control of some "adversary", and we want to evaluate the algorithm by taking the worst case over all adversaries.

Formally, an adversary is (here) an arbitrary choice of

- initial values for all processes, and
- subset of $\{(i,j,k) : (i,j) \text{ is an edge, and } 1 \le k \le r \text{ is a round number}\}$.

The latter represents which messages are delivered: (i,j,k) means that the message crossing (i,j) at round k gets through. Given any particular adversary, the random choices made by the processes induce a probability distribution on the executions. Using this probability space, we can express the probability of events such as agreeing, etc. To emphasize this fact, we shall use the notation $\Pr_A$ for the probability function induced by a given adversary A.

Let us now restate the generals problem in this probabilistic setting. We state it in terms of two given parameters, ϵ and L.

Agreement: For every adversary A,

$$\Pr_A[\text{some process decides 0 and some process decides 1}] \leq \epsilon$$

Validity:

1. If all processes start with 0, then 0 is the only possible decision value.
2. If A is the adversary in which all processes start with 1 and all messages get through, then $\Pr_A[\text{all decide 1}] \geq L$.

We stress that we demand the first validity requirement *unconditionally* — in the database context, for example, one cannot tolerate any probability of error in this respect.

Our goal is, given ϵ, a bound on the probability of disagreement, and r, a bound on the time to termination, we would like to design an algorithm with the largest possible L.

Example Protocol: (two processes). We start with a description of a simple protocol for two processes that will prove insightful in the sequel. The protocol proceeds in r rounds as follows.

In the first round, process 1 chooses an integer *key*, uniformly at random from the range $\{1, \ldots, r\}$. The main idea of the algorithm is that this *key* will serve as a "threshold" for the number of rounds of communication that must be completed successfully, or otherwise the processors will resort to the default value, 0. Specifically, at each round the processes send messages with the following contents.

- their initial values,
- (from process 1 only) the value of *key*, and
- a *color*, green or red.

Intuitively, a message colored red signifies that some message was lost in the past. More precisely, round 1 messages are colored green, and process i's round k message is colored green provided that it has received all the messages at all previous rounds, and they were all colored green; otherwise, it is colored red.

We complete the specification of the protocol by the following *attack rule*:

A process attacks exactly if it "knows" that at least one process started with 1, it "knows" the value of *key*, and it has received green messages in all the first *key* rounds.

Let us make a small digression here to formally define what does it mean for a process to "know" the value of some variable.

Definition 2 *A process i is said to* know *at time k the value of a variable v if either of the following holds.*

- *v is a local variable of i, or*
- *by time $k' \leq k$, i receives a message from a process that knows v at time $k' - 1$.*

The definition is simplified since we consider only variables whose values are set before any message is sent, and never changed afterwards. It is straightforward to extend the definition to the general case.

Another way to view the "knowledge" of a process is to think of the *execution graph* defined as follows. The execution graph is a directed graph, with a node (i, k) for each process i and for each round k, and a directed edge for each delivered message, i.e., $((i, k), (i', k+1))$ is an edge iff a message is sent from i to i' at round k and is delivered by round $k+1$. With this abstraction, Definition 2 says that a process i knows at time k everything about the nodes from which (i, k) is reachable. Notice that in effect we assume that a node tells "all it knows" in every message.

Analysis. Let $\epsilon = \frac{1}{r}$, and let $L = 1$. Validity is easy: a process decides 1 only if there is some 1-input; if all messages are delivered, and all inputs are 1, then the protocol will decide 1. The more interesting part here is agreement. Fix a particular adversary. If all messages are delivered, then the protocol will have all processes decide on the same value. We now focus on the case that some message is not delivered, and let k denote the first round at which a message is lost.

Claim 2 *The only case in which there is disagreement is where $key = k$.*

Proof: We use the fact that if either process receives green messages for at least l rounds, then the other does for at least $l - 1$ rounds.

If $k < key$, then one process fails to receive a green message at some round strictly before the kth round. Consequently, the other fails to do so by round $k + 1 \leq key$, and by the protocol, neither attacks.

If $k > key$ then both get at least *key* green messages. Also, since this means $k > 1$, both know value of *key*, and both know same initial values, and thus they decide on the same value. ∎

Corollary 3 *Let A be any adversary, and let r be the number of rounds. Then*

$$\Pr_A[\textit{disagreement}] \leq \frac{1}{r}$$

An Upper Bound on the Liveness. An obvious question that arises after the 2-processor algorithm is *can we do better?*

Before giving an answer, we observe that the algorithm above has the unpleasant property that if just one first-round message gets lost, then we are very likely not to attack. We might be interested in improving the probability of attacking as a function of the number of messages that get through, aiming at a more refined liveness claim. Another direction of improvement is to extend the algorithm to arbitrary graphs, since the 1-round difference claimed above does not always hold in general graphs. Therefore, the answer to the question above is *yes*, we can give a modification for more general graphs that has provably better liveness bounds for partial message loss. On the other hand, however, an almost-matching upper bound for the liveness we prove below shows we cannot do significantly better.

We start with a definition of a notion that is important the impossibility proof and in the extended algorithm we shall describe later. We define the "information level" of a process at a given time-point. This notion will be used instead of the simple count of the number of consecutive messages received.

Definition 3 *Let $i \in V$ be a process, and let k be a round number. Fix an adversary A. We define $H_A(i,k)$, the set of* heights *of i at round k inductively as follows.*

1. *0 is an element of $H_A(i,k)$.*
2. *1 is an element of $H_A(i,k)$ if at round k, process i knows about some initial value of 1.*
3. *$h > 1$ is an element of $H_A(i,k)$ if at round k, for all processes $j \neq i$, process i knows that $h-1 \in H_A(j,l)$ for some l.*

Finally, define $\textit{level}_A(i,k) = \max H_A(i,k)$.

Intuitively, $\textit{level}_A(i,k) = h$ means that at round k, processor i knows that all processors know that all processors know ... (h times) ... that there exists an input value 1. As it turns out, this level plays a key role in the randomized consensus probability bounds.

We can now prove the upper bound on the liveness.

Theorem 4 *Any r-round protocol for the generals problem with probability of disagreement at most ϵ and probability of attack (when no message is lost) at least L satisfies*

$$L \leq \epsilon(r+1) .$$

We prove Theorem 4 using the following more general lemma.

Lemma 5 *Let A be any adversary, and let $i \in V$ be any node. Then*

$$\Pr_A[i \textit{ decides 1}] \leq \epsilon \cdot \textit{level}_A(i,r)$$

We first show how the lemma implies the theorem.

Proof: (of Theorem 4).
For the liveness condition, we need to consider the adversary A in which all processors have input 1 and no messages are lost. The probability that all processors decide 1 is at most the probability that any of them decides 1, which is, by Lemma 5, at most $\epsilon \cdot \textit{level}_A(i,r)$, and since by the definition of A, $\textit{level}_A(i,r) \leq r+1$, we are done. ∎

We now prove the lemma.

Proof: (of Lemma 5).
First, based on A, we define another adversary A' as follows. A' is the adversary in which the only initial values of 1 and the only messages delivered, are those that i knows about in A. Put in the execution graph language, A' is obtained from A by modifying the execution as follows. We remove all the edges (i.e., messages) which are not on a path leading to any node associated with process i; we also change the inputs in the initial states from 1 to 0 for all the initial states from which there is no path to any node associated with i. The idea in A' is to change all the 1 inputs that are "hidden" from i in A to 0, while preserving the "view" from process i's standpoint.

Now we can prove the lemma by induction on the value of $\textit{level}_A(i,r)$.

Base: Suppose $\textit{level}_A(i,r) = 0$. Then in A, i doesn't know about any initial 1 value, and hence in A', all the initial values are 0. By validity, i *must* decide 0, and since $A \overset{i}{\sim} A'$, we conclude that i decides 0 in A as well.

Inductive step: Suppose $\textit{level}_A(i,r) = l > 0$, and suppose the lemma holds for all levels less than l. Consider the adversary A' defined above. Notice that there must exist some j such that $\textit{level}_{A'}(j,r) \leq l-1$: otherwise, we would have $\textit{level}_A(i,r) > l$, a contradiction. Now, by the inductive hypothesis,

$$\Pr_{A'}[j \text{ decides } 1] \leq \epsilon \cdot \textit{level}_{A'}(j,r)$$

and hence

$$\Pr_{A'}[j \text{ decides } 1] \leq \epsilon(l-1)$$

But by the agreement bound, we must have

$$\Pr_{A'}[i \text{ decides } 1] \leq \Pr_{A'}[j \text{ decides } 1] + \epsilon$$

and thus

$$\Pr_{A'}[i \text{ decides } 1] \leq \epsilon \cdot l$$

and since $A \overset{i}{\sim} A'$, we may conclude that

$$\Pr_{A}[i \text{ decides } 1] \leq \epsilon \cdot l$$

∎

Extended Algorithm. The proof of Theorem 4 actually suggests a modified algorithm, which works in arbitrary connected graphs. Roughly speaking, the processes will compute their information levels explicitly, and will use this information level in place of the number of consecutive message deliveries in a threshold-based algorithm. As in the basic algorithm, process 1 still needs to choose *key* at the first round, to be used as a threshold for the level reached. To get any desired $1-\epsilon$ probability of agreement, we now let *key* be a real chosen uniformly from the interval $[0, 1/\epsilon]$.

We extend the definition of level slightly to incorporate knowledge of the *key*, as follows.

Definition 4 *Let $i \in V$ be a process, and let k be a round number. Fix an adversary A. We define $H'_A(i,k)$, the set of* heights *of i at round k inductively as follows.*

1. *0 is an element of $H'_A(i,k)$.*

2. *1 is an element of $H'_A(i,k)$ if at round k, process i knows about some initial 1-value,* and knows *key.*

3. *$h > 1$ is an element of $H'_A(i,k)$ if at round k, for all processes $j \neq i$, process i knows that $h-1 \in H_A(j,l)$ for some l.*

Define the modified level *by $mlevel_A(i,k) = \max H'_A(i,k)$.*

The difference from Definition 3 is in (2) — we require that the knowledge contain the value of *key* also. It is not hard to show that *mlevel* is always within one of *level* (cf. homework problem). The idea in the protocol is simply to calculate the value of $mlevel_A(i,k)$ explicitly.

> Each process maintains a variable *mlevel*, which is broadcast in every round. The processes update *mlevel* according to Definition 4. After r rounds, a process decides 1 exactly if it knows the value of *key*, and its calculated *mlevel* is at least *key*.

Analysis. For simplicity, we will just carry out the analysis assuming that the graph G is complete. Modifications for other graphs are left as an exercise.

First, consider the validity condition. If all processes start with 0, then *mlevel* remains 0 at all processors throughout the execution, and so they all decide 0.

Lemma 6 *Assume the graph is complete. If all processes start with* 1 *and no message is lost, then*

$$\Pr[\mathit{attack}] \geq \min(1, \epsilon r)$$

Proof: If all processes start with 1 and no message is lost, then by the end of the algorithm $\mathit{mlevel} \geq r$ everywhere, and the probability L of all attacking is at least equal to the probability that $r \geq \mathit{key}$. If $r \geq \frac{1}{\epsilon}$, then $\Pr[\text{attack}] = 1$. If $r < \frac{1}{\epsilon}$, then $\Pr[\text{attack}] = \frac{r}{\frac{1}{\epsilon}} = r\epsilon$. ∎

We remark that it is possible to get a more refined result, for intermediate adversaries: for any given adversary A, define $L(A)$ to be the probability of attack under A, and mlevel_A to be the minimum value of $\mathit{mlevel}_A(i, r)$, i.e., the minimum value of the *mlevel* at the end of the protocol, taken over all the processes. Then $L(A) \geq \min(1, \epsilon \cdot \mathit{mlevel}_A)$. The proof is left as a simple exercise.

We now analyze the agreement achieved by the protocol. (This does not depend on the graph being complete.)

Lemma 7 *Let A be any adversary. Then the probability of disagreement under A is no more than ϵ.*

Proof: By the protocol, if $\mathit{mlevel}_A \geq \mathit{key}$ then all processes will attack, and if $\mathit{mlevel}_A < \mathit{key} - 1$ then all processes will refrain from attacking. This follows from the fact that the final *mlevel* values are within 1 of each other. Hence, disagreement is possible only when $\mathit{mlevel}_A \leq \mathit{key} \leq \mathit{mlevel}_A + 1$. The probability of this event is $\frac{1}{1/\epsilon} = \epsilon$, since mlevel_A is fixed, and *key* is uniformly distributed in $[0, 1/\epsilon]$. ∎

4.2 Faulty Processors

In this section we continue our treatment of the consensus problem, but now we consider another situation, where processors fail, rather than links. We shall investigate two cases: *stopping failures*, where processors may experience "sudden death", and *Byzantine failures*, where a faulty process may exhibit absolutely unconstrained behavior; the former aims at modelling unpredictable crashes, and the latter may model buggy programs.

In both cases, we assume that the number of failures is bounded in advance. Even though this assumption is realistic in the sense that it may be highly unlikely for a larger number of failures to occur, there is a serious drawback to these models: if the number of failures is already quite high, then it is likely in practice that we will have more failures. More precisely, assuming a bound on the number of failures implicitly implies that the failures are *negatively correlated.* It is arguable that failures are independent, or even positively correlated.

4.2.1 Stop Failures

We first consider the simple case where they fail by stopping only. That is, at some point during the algorithm, a processor may stop taking steps altogether. In particular, *a processor can stop in the middle of the message sending step at some round*, i.e., at the round in which the processor stops, only a subset of the messages it "should have" sent may actually be sent. We assume that the links are perfectly reliable — all the messages that are sent are delivered. For simplicity, we assume that the communication graph is complete. As before, we assume that the n processors start with initial values.

Now the problem is as follows. Suppose that at most f processors fail. We want all the *nonfaulty* processors to eventually decide, subject to:

Agreement: all values that are decided upon agree.

Validity: if all processors have initial value v, then the only possible decision value is v.

We shall now describe a simple algorithm that solves the problem. In the algorithm, each processor maintains a labeled tree that represents the "complete knowledge" of the processor as follows. The tree has $f+2$ levels, ranging from 0 (the root), to $f+1$ (the leaves). Each node at level k, $0 \leq k \leq f$, has $n-k$ children. The nodes are labeled by strings as follows. The root is labeled by the empty string λ, and each node with label $i_1 \ldots i_k$ has $n-k$ children with labels $i_1 \ldots i_k j$, where j ranges over all the elements of $\{1, \ldots, n\} - \{i_1, \ldots, i_k\}$. See Figure 4.2 for an illustration.

The processes fill the tree in the course of computation, where the meaning of a value v at a node labeled $i_1 \ldots i_k$ is informally " i_k knows that i_{k-1} knows ... that i_2 knows that the value of i_1 is v". The algorithm now simply involves filling in all the entries in the tree, and after $f+1$ rounds, deciding according to some rule we shall describe later.

More precisely, each process i fills in its own initial value for the root, $value_i(\lambda)$, in the start state. Then in round 1, process i broadcasts this value to all the other processes; for simplicity of presentation, we also pretend that process i sends the value to itself (though in our model, this is simulated by local computation). At each round k, $1 \leq k \leq f+1$, process i broadcasts all the values from all the level $k-1$ nodes in its tree whose labels do not contain i, to all the processes. The recipients record this information in the appropriate

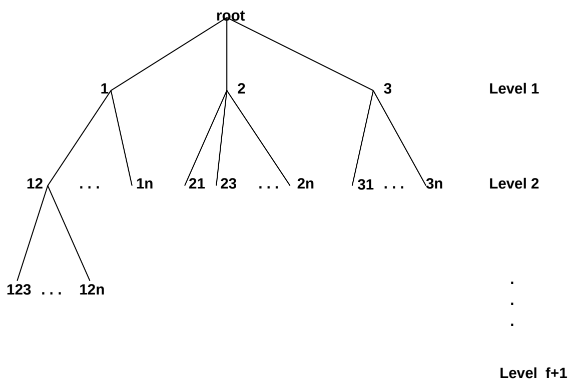


Figure 4.2: the tree used for the stop-failures algorithm

node at level k of their tree: the value sent by process j associated with its node $i_1 i_2 \ldots i_{k-1}$ is associated with the node $i_1 i_2, \ldots i_{k-1} j$. This is $value_i(i_1 i_2 \ldots i_{k-1} j)$. In this way the trees are filled up layer by layer.

Thus, in general, a value v associated with node labeled $i_1 i_2 \ldots i_k$ at a process j means that i_k told j that i_{k-1} told i_k that $\ldots$ i_1 told i_2 that v was i_1's initial value.

Lastly, we specify a decision rule. Each process that has not yet failed defines W to be the set of values that appear anywhere in its tree. If W is a singleton set, then choose the unique element of W; otherwise, choose a pre-specified *default* value.

We shall now argue that the above algorithm satisfies the requirements.

Claim 8 *(Validity) If all the initial values are v, then the only possible decision value is v.*

Proof: Trivial; if all start with v, then the only possible value at any node in anyone's tree is v, and hence, by the decision rule, only v can be decided upon. ∎

Claim 9 *(Agreement) All the values decided upon are identical.*

Proof: Let W_i be the set of values in the tree of processor i. By the decision rule, it suffices to show that $W_i = W_j$ for any two processors i and j that decide (i.e., that complete the algorithm). We prove that if $v \in W_i$ then $v \in W_j$. Suppose $v \in W_i$. Then $v = value_i(p)$ for some label p that does not contain i. (Convince yourself of this.) If $|p| \leq f$, then $|pi| \leq f+1$, so process i relays value v to process j at round $|pi|$, and $v = value_j(pi)$, and thus, $v \in W_j$. On the other hand, if $|p| = f+1$, then there is some nonfaulty process l appearing in the label p, so that $p = qlr$, where q and r are strings. Then at round $|ql|$, processor l succeeds in sending to j, and therefore v is relayed to j (since that's the value relayed further by the processes in r). Thus, $v = value_j(ql)$, and hence $v \in W_j$.

By the same argument, if $v \in W_j$ then $v \in W_i$, and we conclude that $W_i = W_j$. ∎

One can easily see that the number of bits communicated in the execution of the protocol is prohibitively large ($O(n^{f+1})$) and deem this protocol impractical. The important thing we learned from the above algorithm, however, is that the consensus problem is solvable for (this simple kind of) processor failures. This, of course, should be contrasted with Theorem 1 which states that the problem is unsolvable in the case of communication failures. In the next lecture we shall see how can the number of communicated bits be reduced dramatically.

Lecturer: Nancy Lynch

Lecture 5

5.1 Byzantine Failures

Last time, we considered the consensus problem for stopping failures. Now suppose that we allow a worse kind of processor failure (cf. Lamport et al., Dolev-Strong, Bar-Noy et al.): the processors might not only fail by stopping, but could exhibit arbitrary behavior. Such failures are known as "Byzantine" failures, because of the bad reputation of the Byzantine Generals. It must be understood that in this model, a failed process can send *arbitrary* messages and do *arbitrary* state transitions. The only limitation on the behavior of a failed process is that it can only "mess up" the things that it controls locally, namely its own outgoing messages and its own state.

In order for the consensus problem to make sense in this setting, its statement must be modified slightly. As before, we want all the nonfaulty processes to decide. But now the agreement and validity conditions are as follows.

Agreement: All values that are decided upon *by nonfaulty processes* agree.

Validity: If all *nonfaulty processes* begin with v, then the only possible decision value by a *nonfaulty process* is v.

It is intuitively clear the now the situation is somewhat harder than for stopping faults. In fact, we shall show that there is a bound on the number of faults that can be tolerated. Specifically, we will see that $n > 3f$ processes are needed to tolerate f faults. To gain some intuition as to the nature of the problem, let us consider the following example.

What might go wrong with too few processes. Consider three processes, i, j, k, and suppose they are to solve consensus tolerating one Byzantine fault. Let us see why it is impossible for them to decide in two rounds (See Figure 5.1).

Scenario 1: Processes i and k are nonfaulty and start with 0. Process j is faulty. In the first round, processes i and k report their values truthfully, and process j tells both i and k that its value is 1. Consider the viewpoint of process i. In the second round, process k tells

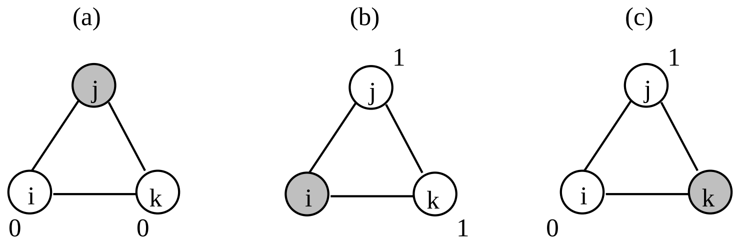


Figure 5.1: possible configurations for three processors. The shaded circles represent faulty processes. Configurations (a), (b) and (c) depict Scenarios 1, 2 and 3 respectively.

i (truthfully) that j said 1, and process j tells i (falsely) that k said 1. In this situation, the problem constraints require that i and k decide 0.

Scenario 2: Dual to Scenario 1. Processes j and k are nonfaulty and start with 1. Process i is faulty. In the first round, processes j and k report their values truthfully, and process i tells both others that its value is 0. Consider the view of process j. In the second round, process k tells j that i said 0 and i tells j that k said 0. In this situation, the problem constraints require that j and k decide 1.

Scenario 3: Now suppose that processes i and j are nonfaulty, and start with 0 and 1, respectively. Process k is faulty, but sends the same messages to i that it does in Scenario 1 and sends the same messages to j that it is in Scenario 2. In this case, i will send the same messages to j as it does in Scenario 2, and j will send the same messages to i as it does in Scenario 1. Then i and j must follow the same behavior they exhibited in Scenarios 1 and 2, namely, decide 0 and 1, respectively, thereby violating agreement!

Note that i can tell that *someone* is faulty in Scenario 3, since k tells i 0, and j tells i that k said 1. But process i doesn't know which of j and k is faulty. We remark that although the protocol could have more rounds, the same kind of argument will carry over. We shall see later a rigorous proof for the necessity of $n > 3f$.

An Algorithm. We shall now describe an algorithm, assuming $n > 3f$ processes. Recall the "full information protocol" described in last lecture for stopping faults. The basic idea in the Byzantine case is based on the same tree and propagation strategy as for the stopping faults case, but now we shall use a different decision rule. It is no longer the case that we want to accept any value for insertion into the set of "good" values, just because it appears *somewhere* in the tree. Now we must beware of liars! We use the following easy fact.

Lemma 1 *If i, j and k are all nonfaulty, then $value_i(p) = value_j(p)$ for every label p ending in k.*

Proof: Follows from the simple fact that since k is nonfaulty, it sends the same thing to everyone. ∎

Now we can specify a decision rule.

> If a value is missing at a node, it is set to some pre-specified default. To determine the decision for a given tree, we work from the leaves up, computing a new value *newvalue* for each node as follows. For the leaves, $newvalue_i(p) := value_i(p)$. For each node, *newvalue* is defined to be the *majority* value of the *newvalue*'s of the children. If no majority value exists, we set $newvalue := default$. The decision is $newvalue_i(root)$.

We now start proving the correctness of the algorithm. First we prove a general lemma.

Lemma 2 *Suppose that p is a label ending with the index of a nonfaulty process. Then there is a value v such that $value_i(p) = newvalue_i(p) = v$ for all nonfaulty processes i.*

Proof: By induction on the tree labels, working from the leaves on up.

Base case: Suppose p is a leaf. Then, by Lemma 1, all nonfaulty processes i have the same $value_i(p)$; which is the desired value v, by the definition of *newvalue* for leaves.

Inductive step: Suppose $|p| = k$, $0 \leq k \leq f$. Then Lemma 1 implies that all nonfaulty processes i have the same value $value_i(p)$; call this value v. Therefore, all the nonfaulty processes l send the same value v for pl to all processes. So for all nonfaulty i and l, we have $value_i(pl) = v$. By the inductive hypothesis, we have that $newvalue_i(pl) = v$ for all nonfaulty processes i and l, for some value v.

We now claim that a majority of the child labels of p end in nonfaulty process indices. This is true because the number of children is $n - k \geq n - f > 2f$, and since at most f may be faulty, we have that there always is a "honest" majority. (Note this is where we use the bound on the number of processes.)

The induction is complete once we observe that since for any nonfaulty i, $v = newvalue_i(pl)$ for a majority of the children pl, $newvalue_i(p) = v$ by the algorithm. ∎

We now argue validity.

Corollary 3 *Suppose all nonfaulty processes begin with the same value v. Then all nonfaulty processes decide v.*

Proof: If all nonfaulty processes begin with v, then all nonfaulty processes send v at the first round, and therefore $value_i(j) = v$ for all nonfaulty processes i, j. Lemma 2 implies that $newvalue_i(j) = v$ for all nonfaulty i and j, and therefore, $newvalue_i(root) = v$ for all nonfaulty i. ∎

Before we show agreement, we need a couple of additional definitions.

Definition 1 *Consider an execution of the algorithm. A tree node p is said to be* common *in the given execution if, at the end of the execution, all the nonfaulty processes have the same $newvalue(p)$.*

Definition 2 *Consider a rooted tree T. A subset of the tree-nodes C is a* path covering *of T if every path from a leaf to the root contains at least one node from C. A path covering is* common *if it consists entirely of common nodes.*

Lemma 4 *There exits a common path covering of the tree.*

Proof: Consider any path. Each of the $f+1$ nodes on the path ends with a distinct index, and therefore there exists a node on the path, say p, with a label that ends in a nonfaulty index. Then Lemma 2 implies that p is common. ▮

Lemma 5 *Let p be a node. If there is a common path covering of the subtree rooted at p, then p is common.*

Proof: By induction on p, starting at the leaves and working up.

Base case: If p is a leaf, there is nothing to prove.

Inductive step: Suppose p is at level k. Assume that there is a common path covering C of p's subtree. If p itself is in C, then p is common and we are done, so suppose $p \notin C$.

Consider *any* child pl of p. C induces a common path covering for the subtree rooted at pl. Hence, by the inductive hypothesis, pl is common. Since pl was an arbitrary child of p, all the children of p are common, which in turn implies that p is common by the definition of *newvalue*. ▮

We can now conclude that the algorithm satisfies the validity requirement.

Corollary 6 *The root is common.*

Proof: By Lemmas 4 and 5. ▮

Authentication. The original paper of Lamport, Pease and Shostak, and a later paper of Dolev and Strong, present algorithms that work in the presence of Byzantine failures, but where the nodes have the extra power of *authentication* based on *digital signatures.* Informally, this means that no one can claim that another process said something unless in fact it did. There is no nice formal characterization given for this model. The interesting fact is that the algorithm is much like the stopping fault algorithm, rather than the full Byzantine algorithm above. In particular, it works for any number of processes, even when $n < 3f$.

5.2 Reducing the Communication Cost

A major drawback of the algorithms presented above is that they require an excessive amount of communication — specifically, the number of messages communicated in the tree algorithms is exponential in f. (Here, we are counting the separate values being sent as separate messages, though in terms of the formal model, many would be packaged into a single message. The bound would be better expressed in terms of the number of *bits* of communication.) In this section we discuss some methods to cut down this cost.

5.2.1 Stopping Faults

We start by reducing the number of messages communicated in the stopping faults algorithm to polynomial in f. Let us examine the algorithm described in the previous lecture more closely. In that algorithm, we have everyone broadcasting everything to everyone. However, in the end, each process only looks to see whether the number of distinct values it has received is exactly 1. Consider a process which has already relayed 2 distinct values to everyone. We claim that this process doesn't need to relay anything else, since every other process already has received at least two values. Thus, the algorithm boils down to following rule.

> Use the "regular" algorithm, but each process prunes the messages it relays so that it sends out only *the first two messages containing distinct values.*

The complexity analysis of the above algorithm is easy: the number of rounds in is the same as before ($f+1$), but the number of messages is only $O(n^2)$, since in total, each process sends at most two messages to each other process.

What is a bit more involved is proving the correctness of the above algorithm. In what follows we only sketch a proof for the correctness of the optimized algorithm. The main interest lies in the proof technique we employ below.

Let O denote this optimized algorithm, and U the unoptimized version. We prove correctness of O by relating it to U. First we need the following property.

Lemma 7 *In either algorithm O or algorithm U, after any number k of rounds, for any process index i and for any value v, the following is true. If v appears in i's tree, then it appears in the tree at some node for which the index i is not in the label.*

Lemma 7 is true because there are only stopping failures, so i can only relay a value, and others claim i relayed it, if i in fact originally had it.

Define $values_O(i,k)$, $0 \leq k \leq f+1$, to be the set of values that appear in levels $\{0,\ldots,k\}$ of process i's tree in O. We define $values_U(i,k)$ analogously. The following lemma is immediate from the specification of algorithm U.

Lemma 8 *In algorithm U, suppose that i sends a round $k+1$ message to j, and j receives and processes it. Then $values_U(i,k) \subseteq values_U(j,k+1)$.*

The key pruning property of O is captured by the following lemma.

Lemma 9 *In algorithm O, suppose that i sends a round $k+1$ message to j, and j receives and processes it.*

1. *If $|values_O(i,k)| \leq 1$, then $values_O(i,k) \subseteq values_O(j,k+1)$.*

2. *If $|values_O(i,k)| \geq 2$, then $|values_O(j,k+1)| \geq 2$.*

Note that Lemma 7 is required to prove Lemma 9.

Now to see that O is correct, imagine running the two algorithms *side-by-side*: O, the new optimized algorithm, and U, the original unoptimized version, with the same initial values, and with failures occurring at the same processes at exactly the same times. (If a process fails at the middle sending in one algorithm, it should fail at the same point in the other algorithm.) We state "invariants" relating the states of the two algorithms.

Lemma 10 *After any number of rounds k, $0 \leq k \leq f+1$: $values_O(i,k) \subseteq values_U(i,k)$.*

Lemma 11 *After any number of rounds k, $0 \leq k \leq f+1$:*

1. *If $|values_U(i,k)| \leq 1$ then $values_O(i,k) = values_B(i,k)$.*

2. *If $|values_U(i,k)| \geq 2$ then $|values_O(i,k)| \geq 2$.*

Proof: By induction. Base case is immediate. Assume now that the lemma holds for k. We show it holds for $k+1$.

1. Suppose $|values_U(i,k+1)| \leq 1$. It follows that $|values_U(i,k)| \leq 1$, so by inductive hypothesis, we have $values_O(i,k) = values_U(i,k)$. By Lemma 10, it suffices to show that $values_U(i,k+1) \subseteq values_O(i,k+1)$.

   Let $v \in values_U(i,k+1)$. If $v \in values_U(i,k)$ then $v \in values_O(i,k) \subseteq values_O(i,k+1)$. So now assume that $v \notin values_U(i,k+1)$. Suppose v arrives at i in a round $k+1$ in a message from some process j, where $v \in values_U(j,k)$. Since $|values_U(i,k+1)| \leq 1$, Lemma 8 implies that $|values_U(j,k)| \leq 1$. By inductive hypothesis, $values_O(j,k) = values_U(j,k)$, and so $|values_O(j,k)| \leq 1$. By Lemma 9, $values_O(j,k) \subseteq values_O(i,k{+}1)$, and hence $v \in values_O(i,k+1)$, as needed.

2. Suppose $|values_U(i,k+1)| \geq 2$. If $|values_U(i,k)| \geq 2$ then by inductive hypothesis, we have $|values_O(i,k)| \geq 2$, which implies the result. So assume that $|values_U(i,k)| \leq 1$. By the inductive hypothesis, we have $values_O(i,k) = values_U(i,k)$. We consider two subcases.

(a) For all j from which i receives a round $k+1$ message, $|values_B(j,k) \leq 1|$.

In this case, for all such j, we have by inductive hypothesis that $values_A(j,k) = values_B(j,k)$. Lemma 9 then implies that for all such j, $values_U(j,k) \subseteq values_O(i,k+1)$. It follows that $values_O(i,k+1) = values_U(i,k+1)$, which is sufficient to prove the inductive step.

(b) There exists j from which i receives a round $k+1$ message such that $|values_U(j,k)| \geq 2$.

By the inductive hypothesis, $|values_O(j,k)| \geq 2$. By Lemma 9, $|values_A(i,k+1)| \geq 2$, as needed.

∎

The truth of Lemma 11 for $k = f+1$, and the fact that $W_i = W_j$ for nonfaulty i and j in U, together imply that i and j decide in the same way in algorithm O: this follows from considering the two cases in which W_i and W_j in U are singletons or have more than one value ($W_i \neq \phi$, since the root always gets a value).

Note: The proof technique of running two algorithms side by side and showing that they have corresponding executions is very important. In particular, in cases where one algorithm is an optimization of another, it is often much easier to show the correspondence than it is to show directly that the optimized algorithm is correct. This is a special case of the *simulation* proof method (cf. Model Section in the bibliographical list).

5.2.2 The Byzantine Case

In the Byzantine model, it does not seem so easy to obtain an optimized, pruned algorithm with a polynomial number of messages. We only summarize results in this section.

Polynomial communication and $2f+k$ rounds (for constant k), assuming $n \geq 3f+1$ processes, was achieved by a fairly complex algorithm by Dolev, Fischer, Fowler, Lynch, and Strong [DolevFFLS82].

Essentially this algorithm was expressed in a more structured fashion by Srikanth and Toueg [SrikanthT87]. Their structure involved substituting an authentication protocol for the assumed signature scheme in an efficient authenticated BA algorithm (used one by Dolev and Strong). Using this authentication protocol requires twice the number of rounds as does the Dolev-Strong protocol, and also needs $3f+1$ processes. The basic idea is that whenever a message was sent in the Dolev-Strong algorithm, Srikanth and Toueg run a protocol to send and accept the message.

An improvement on the $2f$ rounds required by the above algorithms was achieved by Coan [Coan86], who presented a family of algorithms requiring $f+\epsilon f$ rounds, where $0 < \epsilon \leq 1$. The message complexity is polynomial, but as ϵ approaches zero, the degree of the polynomial increases. An important consequence of this result is that no lower bound larger than $f+1$ rounds can be proved for polynomial algorithms, although no fixed degree polynomial algorithm is actually given for $f+1$ rounds. A paper by Bar Noy, Dolev, Dwork, and Strong [BarNoyDDS87] presents similar ideas in a different way. We remark that they introduced the tree idea used above to present the consensus algorithms.

Moses and Waarts achieve $f+1$ rounds with polynomial communication [MosesW88]. The protocol looks pretty complicated, with many involved strategies for pruning the tree. Also, it does not work for $3f+1$ processors as does the exponential communication protocol, but rather for

Berman and Garay use a different approach to solve the same problem as MW. They achieve $4f+1$ processors and $r+1$ rounds. Their algorithm is still fairly complicated.

5.2.3 The Turpin-Coan Algorithm

It is hard to cut down the amount of communication for general Byzantine agreement. We close this topic with an interesting trick that helps somewhat, though it does not reduce the communication from exponential to polynomial. The problem addressed here is how to reduce BA for a multivalued domain to binary-valued BA. In other words, the algorithm we present uses binary BA as a subroutine. If the domain is large, the savings can be substantial.

In the Turpin-Coan algorithm we assume that $n \geq 3f+1$, and that default value is known initially by all non-faulty processes. For each process, a local variable x is initialized to the input value for that process.

Below, we describe the Turpin-Coan algorithm. This time, the style is changed a little bit. Namely, the code for each round is written explicitly. Implicitly, we assume that the code will be executed for rounds $1, 2, \ldots$ in sequence. Of course, in the underlying state machine model, this convention is expanded into manipulation of an explicit *round* variable, which gets read to determine which code to perform, and gets incremented at every round.

Round 1:

1. Broadcast x to all other processes.
2. In the set of messages received, if there are $\geq n-f$ for a particular value, v, then $x := v$, otherwise $x \leftarrow nil$.

Round 2:

1. Broadcast x to all other processes.

2. Let v_{max} be the value, other than *nil*, that occurs most often among those values received, with ties broken in some consistent way. Let *num* be the number of occurrences of v_{max}.
3. if $num \geq n - f$ then $vote = 1$ else $vote = 0$.

After round 2, run the binary Byzantine agreement subroutine using *vote* as the input value. If the bit decided upon is 1, then decide v_{max}, otherwise decide the default value.

Claim 12 *At most one value v is sent in round 2 messages by correct processes.*

Proof: Any process p sending a value v in round 2 must have received at least $n - f$ round 1 messages containing v. Since there are at most f faulty processes, this means that all other processes received at least $n - 2f$ copies of v. Since the total number of messages received is n, no process could have received $n - f$ messages containing a value $v' \neq v$ in round 1. Therefore, the claim holds. ∎

Theorem 13 *The Turpin-Coan algorithm solves multivalued Byzantine agreement when given a Boolean Byzantine agreement algorithm as a subroutine.*

Proof: It is easy to see that this algorithm terminates.

To show validity, we must prove that if all nonfaulty processes start with a value, w, then all nonfaulty processes must decide w. After the first round, all nonfaulty processes will have set $x \leftarrow w$ because at least $n - f$ processes broadcast it reliably. Therefore, in the second round, each nonfaulty process will have $v_{max} = w$ and $num \geq n - f$, and will therefore obtain $vote = 1$. The binary agreement subroutine is therefore required to choose 1, and each nonfaulty process will choose w.

In showing agreement, we consider two cases. If the subroutine decides on $vote = 0$, then the default value is chosen by all nonfaulty processes, so agreement holds. If the subroutine decides on $vote = 1$, then we must argue that the local variable v_{max} is the same for each nonfaulty process. Note that for the subroutine to agree on $vote = 1$, then some nonfaulty process p must have started with $vote = 1$. Therefore, process p must have received at least $n - f$ round 2 messages containing some particular value v. Since there are at most f faulty processes, each other process must have received at least $n - 2f$ round two messages with non-nil values from nonfaulty processes. By Claim 12, *all* of the non-nil messages from nonfaulty processes must have the same value, namely v. Therefore, v must be the value occurring most often, since there are at most f faulty processes and $n - 2f > f$. ∎

The proof method uses a bound on the number of faulty processes to obtain claims about similarity between the views of different processes in a fault-prone system. This is a useful technique.

Lecturer: Nancy Lynch

Lecture 6

We have seen algorithms for distributed agreement, even in the case of Byzantine faults. The algorithm we saw used $n \geq 3f+1$ processes and $f+1$ synchronous rounds of communication. In this lecture we will consider whether these bounds are necessary.

6.1 Number of Processes for Byzantine Agreement

It turns out that it is impossible to beat the $3f+1$ upper bound on the number of processes, i.e., if $n \leq 3f$, then there is no protocol that ensures agreement. The proof is neat. We first prove that three processes cannot tolerate one fault, as suggested in an earlier example (last lecture). We then show why is it impossible to tolerate $f \geq n/3$ faults for any n.

The first proof of this appeared in [PeaseSL80]; the proof given here follows that in [FischerLM86].

Lemma 1 *Three processes cannot solve Byzantine agreement in the presence of one fault.*

Proof: By contradiction. Assume there is a protocol P consisting of three processes, p, q and r, such that P with arbitrary inputs will satisfy the Byzantine agreement conditions even if one process malfunctions. We shall construct a new system S from (copies of) the same processes, and show that S must exhibit contradictory behavior. It follows that P cannot exist.

Take two copies of each process in P, and arrange them into a 6-process system S as shown in Figure 6.1.

When configured in this way, the system appears to every process as if it is configured in the original three-process system P. Notice that the processes are not required to produce any specific output in S; however, S with any particular input assignment does exhibit some well-defined behavior. We shall see that no such well-defined behavior is possible.

Suppose that the processes in S are started with the input values indicated in Figure 6.1. Consider the processes p-q-r-p' in S with the given inputs. To q and r, it appears as if they are running in the three process system P, in which p is faulty (i.e., "pretending" it's running in a 6-processor system). This is an allowable behavior for Byzantine Agreement on three processes, and the correctness conditions for BA imply that q and r must eventually

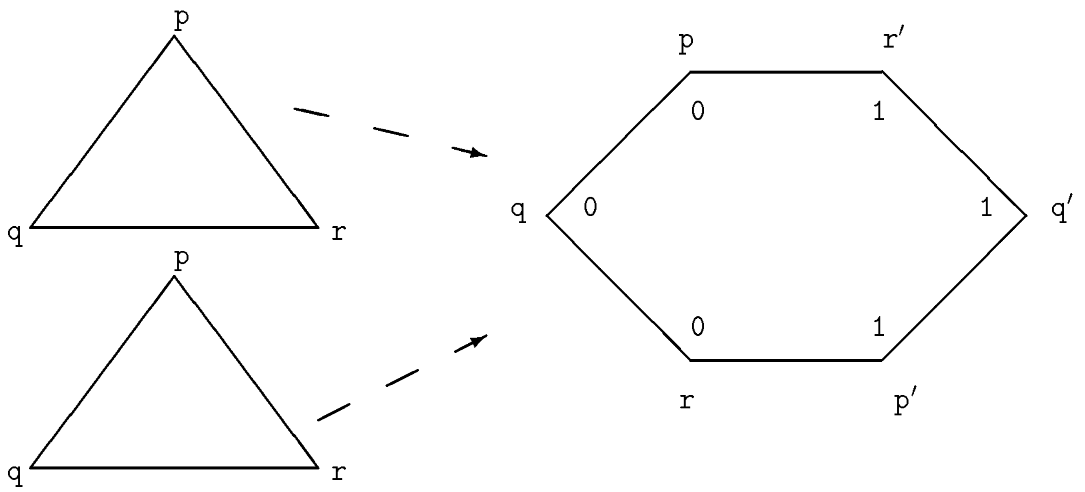


Figure 6.1: The system S: arrangement of processes in the proof of Lemma 1

agree on 0 in the three-process system. Since the six-process system S appears identical to q and r, both will eventually decide on 0 in S as well.

Next consider the processes r-p'-q'-r'. By similar reasoning, p' and q' will eventually agree on 1 in S.

Finally consider the processes q-r-p'-q'. To r and p', it appears as if they are in a three-process system with q faulty. Then r and p' must eventually agree in P (although there is no requirement on what value they agree upon), and so they agree in S. However this is impossible since we just saw that r must decide 0 and p' must decide 1, and we arrived to the desired contradiction. ▮

We now use Lemma 1 to show that Byzantine agreement requires $n \geq 3f + 1$ processes to tolerate f Byzantine faults [LamportPS82]. We will do this by showing how an $n \leq 3f$ process solution that can tolerate f Byzantine failures implies the existence of a 3 process solution which can tolerate a single Byzantine failure. This contradicts the above lemma.

Theorem 2 *There is no solution to the Byzantine agreement problem on n processes in the presence of f Byzantine failures, when $1 < n \leq 3f$.*

Proof: Assume there is a solution for Byzantine agreement with $3 \leq n \leq 3f$. (For $n = 2$, there can be no agreement: if one process starts with 0 and the other with 1, then each must allow for the possibility that the other is lying and decide on its own value. But if neither is lying, they this violates the agreement property.) Construct a three-process system with each new process simulating approximately one-third of the original processes. Specifically, partition the original processes into three nonempty subsets, P_1, P_2, and P_3, each of size at most f. Let the three new processes be p_1, p_2, and p_3, and let each p_i simulate the original processes in P_i as follows. Each process p_i keeps track of the states of all the original processes in P_i, assigns its own initial value to every member of P_i, and simulates the steps of all the processes in P_i as well as the messages between the processes in P_i. Messages from processes in P_i to processes in another subset are sent from p_i to the process simulating that subset. If any simulated process in P_i decides on a value v then p_i can decide on the value v. (If there is more than one such value, then p_i will choose a particular one, chosen according to some default.)

To see that this is a correct 3-process solution we reason as follows. At most one of p_1, p_2, p_3 is allowed to be faulty, and each simulates at most f original processes, so the simulation contains no more than f simulated faults. (Designate the faulty processes in the simulated system to be exactly those that are simulated by a nonfaulty actual process, regardless of whether they actually exhibit a fault.) This is as required by the simulated solution. The n-process simulated solution then guarantees that the simulated processes satisfy agreement, validity, and termination.

Let us argue briefly that the validity, agreement, and termination of the n-process simulation carry over to the 3-process system. Termination is easy. For validity, if all nonfaulty actual processes begin with a value v, then all the nonfaulty simulated processes begin with v. Validity for the simulated system implies that the only simulated decision value for a simulated non-faulty process must be v, so the only actual decision value for an actual non-faulty process is v. For agreement, suppose p_i and p_j are nonfaulty actual processes. They they simulate only nonfaulty processes. Agreement for the simulated system implies that all of these simulated processes agree, so p_i and p_j also agree.

We conclude that given a system that tolerates $f \geq n/3$ faults, we can construct a protocol for 3 processes that tolerates one faulty process, contradicting Lemma 1. ▮

Let us recap what assumptions have we used for this proof.

Locality: A process's actions depend only on messages from its input channels and its initial value.

Faultiness: A faulty process is allowed to exhibit any combination of behaviors on its outgoing channels. (We could strengthen this assumption, and insist that the behavior of each channel can arise in some system in which the process is acting correctly.)

The locality assumption basically states that communication only takes place over the edges of the graph, and thus it is only these inputs and a process' initial value that can affect its behavior. Note that the nodes are allowed to have information about the network in which they are supposed to run (e.g., the three-process network above) built into them, but this information is not changed “magically” if processes are assembled into an unexpected system (e.g., the six-process network above).

The strengthened version of the fault assumption expresses a masquerading capability of faulty processes. We cannot determine if a particular edge leads to a correct process, or to a faulty process simulating the behavior of a correct process over the edge. The fault assumption gives faulty processes the ability to simulate the behaviors of different correct processes over different edges.

6.2 Byzantine Agreement in General Graphs

We have shown that Byzantine agreement can be solved with n processes and f faults, where $n \geq 3f+1$. In proving this result, we assumed that any process could send a message directly to any other process. We now consider the problem of Byzantine agreement in general communication graphs [Dolev82].

Consider a communication graph, G, where the nodes represent processes and an edge exists between two processes if they can communicate. It is easy to see that if G is a tree, we cannot accomplish Byzantine agreement with even one faulty process. Any faulty process that is not a leaf would essentially cut off one section of G from another. The nonfaulty processes in different components would not be able to communicate reliably, much less reach agreement. Similarly, if removing f nodes can disconnect the graph, it should also be impossible to reach agreement with f faulty processes. We formalize this intuition using the following standard graph-theoretic concept.

Definition 1 *The* connectivity *of a graph G, $conn(G)$, is the minimum number of nodes whose removal results in a disconnected graph or a trivial 1-node graph. We say a graph G is k-*connected *if $conn(G) \geq k$.*

For example, a tree has connectivity 1, and a complete graph with n nodes has connectivity $n-1$. Figure 6.2 shows a graph with connectivity 2. If q and s are removed, then we are left with two disconnected pieces, p and r.

We remark that *Menger's Theorem* states that a graph is k-connected if and only if every pair of points is connected by at least k node-disjoint paths. We shall use this alternative characterization of the connectivity in the sequel.

In the following theorem we relate the connectivity of a graph to the possibility of executing a Byzantine agreement protocol over it. The proof uses methods similar to those used in our lower bound proof for the number of faulty processes.

Theorem 3 *Byzantine agreement can be solved on a graph G with n nodes and f faults if and only if*

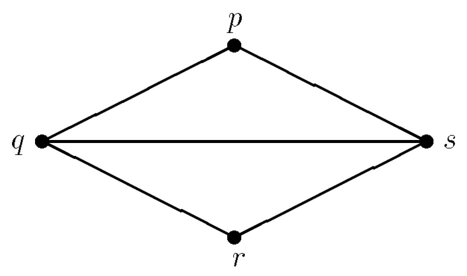


Figure 6.2: A graph G with $\text{conn}(G) = 2$.

1. $n \geq 3f + 1$, *and*

2. $conn(G) \geq 2f + 1$.

Proof: We already know that $n \geq 3f + 1$ processes are required for a fully connected graph. It is easy to see that for an arbitrary communication graph, we still need $n \geq 3f + 1$ (since removing communication links does not "help" the protocols).

We now show the *if* direction, namely, that Byzantine agreement is possible if $n \geq 3f+1$ and $\text{conn}(G) \geq 2f + 1$. Since we are assuming G is $2f + 1$-connected, Menger's Theorem implies that there are at least $2f + 1$ node-disjoint paths between any two nodes. We can simulate a direct connection between these nodes by sending the message along each of the $2f+1$ paths. Since only f processes are faulty, we are guaranteed that the value received in the *majority* of these messages is correct. Therefore, simulation of a fully connected graph can be accomplished, simulating each round of the protocol for a fully-connected graph by at most n rounds of in G. We have already seen an algorithm that solve Byzantine agreement in this situation.

We now prove the *only if* direction of the theorem. The argument that Byzantine agreement is not possible if $\text{conn}(G) \leq 2f$ is a bit more intricate. We will only argue the case for $f = 1$, for simplicity.

Assume there exists a graph, G, with $\text{conn}(G) \leq 2$ in which consensus can be solved in the presence of one fault, using protocol P. Then there are two nodes in G that either disconnect the graph or reduce it to one node. But this latter case means that there are only three nodes in the graph, and we already know that Byzantine agreement cannot be solved in a 3-node graph in the presence of 1 fault. So assume they disconnect the graph.

The picture is then as in Figure 6.2, where p and r are replaced by arbitrary connected graphs (and there may be several edges from each of q and s to each of the two disconnected groups). Again for simplicity, however, we just consider p and r to be single nodes. We construct a system S by "rewiring" two copies of P, as shown in Figure 6.3. Each process in S behaves as if it was the same-named process in P, for the graph in Figure 6.2.

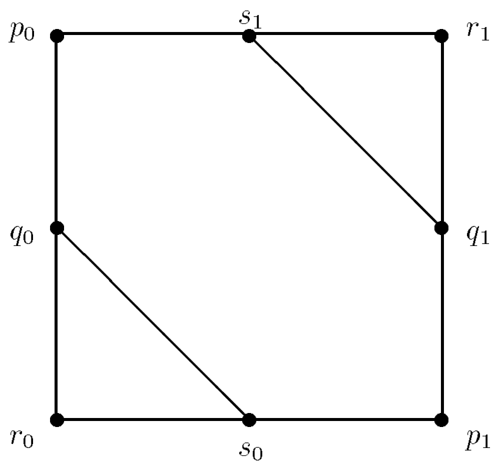


Figure 6.3: System S, made by "rewiring" two copies of P.

Now assume we start up system S with inputs corresponding to the subscripts. Consider the behavior of the processes p_0, q_0 and r_0 of S outlined in Figure 6.4. As before, it appears to all three that they are in P with input 0, as in Figure 6.5, in an execution in which s is a faulty process. By the locality assumption, the outlined processes will behave in the same way in these two cases. By the validity property for P, these processes must all decide 0 in P, and hence in S.

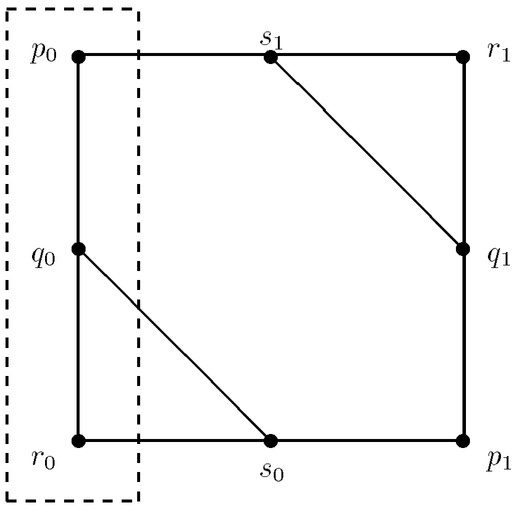


Figure 6.4: A set of processes in S.

Now consider Figure 6.6 and the corresponding 4-node situation of Figure 6.7. By the same argument, all the outlined processes will decide 1 in S.

Finally, consider Figure 6.8 and its corresponding 4-node situation of Figure 6.9. Since

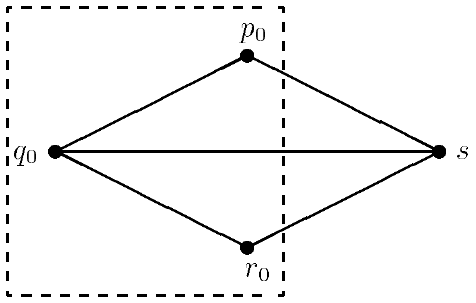


Figure 6.5: A configuration of P (with s faulty) that the outlined processes cannot distinguish from Figure 6.4.

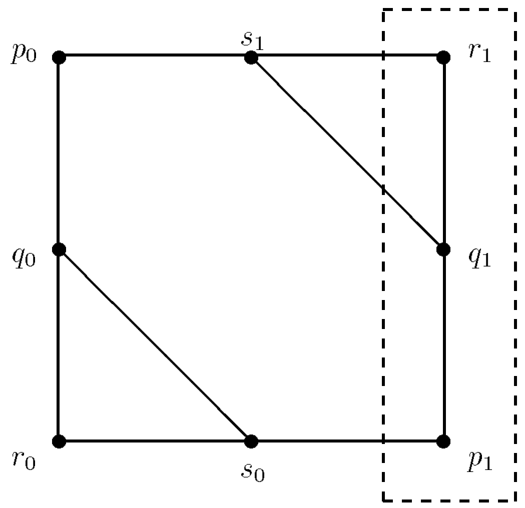


Figure 6.6: Another set of processes in S.

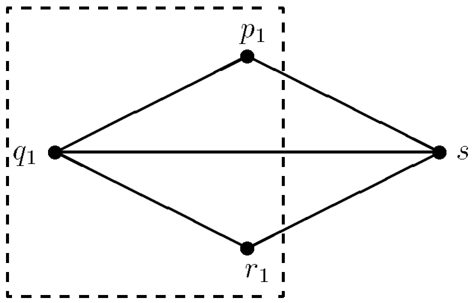


Figure 6.7: A configuration (with s faulty) that the outlined processes cannot distinguish from Figure 6.6.

only q is faulty, the agreement condition requires that the outlined processes decide on the same value in P, and hence in S.

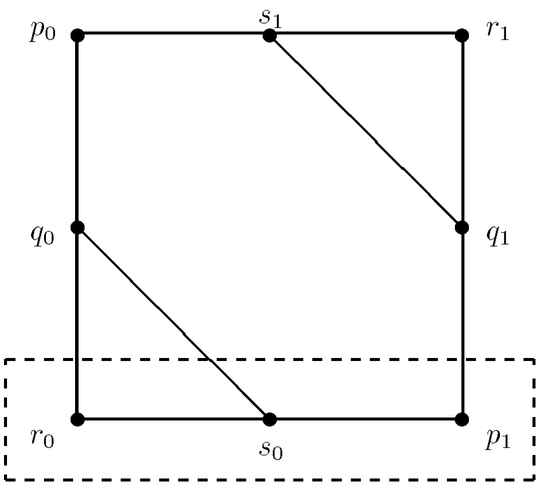


Figure 6.8: Yet another set of processes in S.

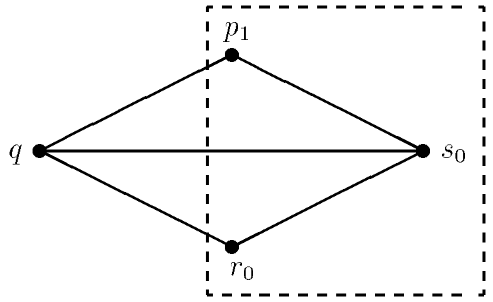


Figure 6.9: The nonfaulty processes must agree, giving us a contradiction.

However, we have already shown that process p_1 must decide 1 and process r_0 must decide 0. Thus, we have reached a contradiction. It follows that we cannot solve Byzantine agreement for $\text{conn}(G) \leq 2$ and $f = 1$.

To generalize the result to $f > 1$, we use the same diagrams, with q and s replaced by graphs of at most f nodes each and p and r by arbitrary graphs. Again, removing q and s disconnects p and r. The edges of Figure 6.2 now represent bundles of all the edges between the different groups of nodes in p, q, r and s.

∎

6.3 Weak Byzantine Agreement

A slightly stronger result that can be obtained from the same proof method. Lamport defined a weaker problem than Byzantine agreement, still in the case of Byzantine faults, by changing the validity requirement.

- **Validity:** If all processes start with value v *and no faults occur*, then v is the only allowable decision value.

Previously we required that even if there were faults, if all nonfaulty processes started with v then all nonfaulty processes must decide v. Now they are only required to decide v in the case of no failures. This weakened restriction is motivated loosely by the database commit problem since we only require commitment in the case of no faults.

Lamport tried to get better algorithms for weak Byzantine agreement than for Byzantine agreement, but failed. Instead he got an impossibility result. Note: In obtaining this result, Lamport did not even need the assumption that the algorithm terminates in a fixed number of rounds, but only that every execution *eventually* leads to termination for all nonfaulty processes. In fact, the impossibility results proved so far in this lecture still work even with this weakening of the termination requirement. For the present result, we will explicitly assume the weaker termination requirement.

Theorem 4 *$n \geq 3f+1$ processes and $conn(G) \geq 2f+1$ are needed even for weak Byzantine agreement.*

We will just show that three processes cannot solve weak Byzantine agreement with one fault; the rest of the proof follows as before.

Suppose there exist three processes p, q, and r that can solve weak Byzantine agreement with one fault. Let P_0 be the execution in which all three processes start with 0, no failures occur, and therefore all three processes decide 0. Let P_1 be the same for 1 instead of 0. Let k be greater than or equal to the number of rounds in executions P_0 and P_1. We now create a new system S with at least $4k+2$ nodes by pasting enough copies of the sequence p-q-r into a ring. Let half the ring start with value 0 and the the other half start with 1 (see Figure 6.10).

By arguing as we did before, any two consecutive processes must agree; therefore, all processes in S must agree. Assume (without loss of generality) that they all decide 1.

Now consider a block of at least $2k+1$ consecutive processes that start with 0. For 1 round, the middle $2k-1$ processes in this block will all mimic their namesakes in P_0, because they do not receive any information from outside the block of 0's. Likewise, for 2 rounds the middle $2k-3$ processes will behave as in P_0, etc. Finally, the middle process of the group will behave as in P_0 for k rounds. Therefore, it will decide 0. This is a contradiction.

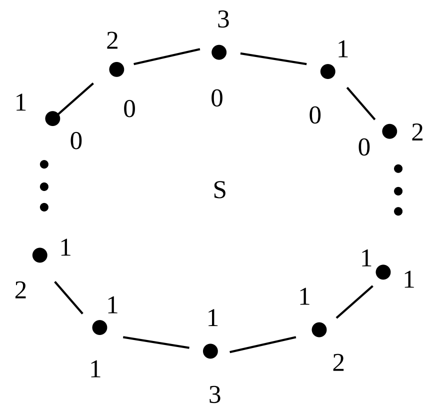


Figure 6.10: configuration in the proof of Theorem 4

6.4 Number of Rounds with Stopping Faults

We now turn to the question of whether Byzantine agreement can be solved in fewer than $f+1$ rounds. Once again the answer is no, even if we require only the weakest validity condition, namely, that processes must decide v only when all start with v, and no faults occur. In fact, at least $f+1$ rounds are required even to simply tolerate f stopping faults.

We assume the following model. The number of processes satisfies $n \geq f+2$. The algorithm terminates in a fixed number of rounds. We also assume, for simplicity, that every process sends to every other process at every round (until it fails). As usual, we assume that a process can fail in the middle of sending at any round, so that it can succeed in sending any subset of the messages.

As for two-generals problem, it is useful to carry out the proof using the notion of communication pattern.

Define a *communication pattern* to be an indication of which processes send to which other processes in each round. A communication pattern does not tell us the actual information sent but only "who sent to whom" in a round. A communication pattern can be depicted graphically as shown in Figure 6.11. As before, a communication pattern is a subset of set of (i, j, k) triples, where i and j are distinct processes and k is a round. However, in the stopping-failure case, if a message from some process i fails to reach its destination in round k, then no messages from i are delivered in any round after k.

In the figure, p_3 does not send to p_2 in round 2. Thus p_3 must have stopped and will send nothing further in round 2 and all future rounds. Essentially, a communication pattern depicts how processes fail in a run.

Define a *run* as an input assignment and a communication pattern. (This is identical to what we called *adversary* in the section on two-generals.) Given any run ρ, we can define

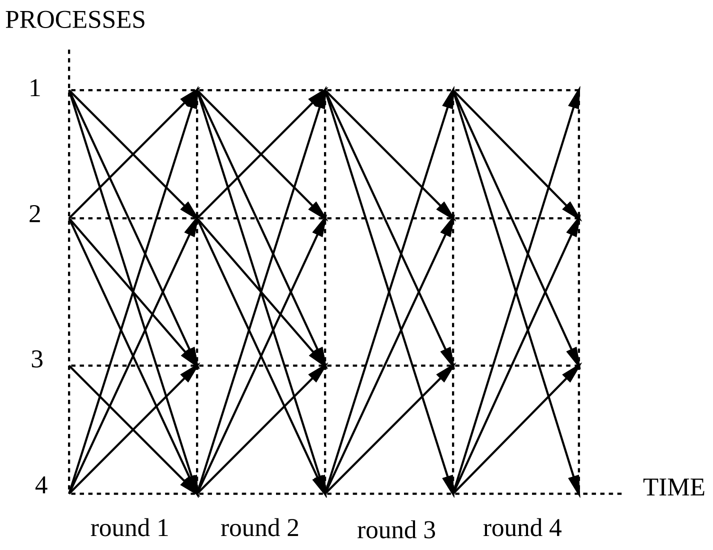


Figure 6.11: Example of a communication pattern. Process 3 stops before completing round 1, process 2 stops after completing round 2, processes 1 and 4 are up throughout the execution.

an execution exec(ρ) generated by ρ. This execution is an alternating sequence of vectors of states, one per process, and matrices of messages, one per pair of processes, starting with the initial states containing the given input values. These states and messages can be filled in using the deterministic message and transition functions. It is possible to infer when a process fails by the first time, if any, when it fails to send a message. After such a failure, it will not perform its state transition function. By convention, we say that a process that sends all its messages at round k but none at round $k+1$, performs the transition of round k in the execution exec(ρ).

A process is *faulty* in a run exactly if it stops sending somewhere in the communication pattern. We will only consider runs with at most f faulty processes.

The Case of One Failure. First, for intuition, we will consider the special case of $f = 1$. We show that two rounds are required to handle a single stopping fault. We do this by contradiction, so assume a 1-round algorithm exists.

We construct a chain of executions, each with at most one fault, such that (i) weak validity requires that the first execution must lead to a 0 decision and the last execution must lead to a 1 decision, and (ii) any two consecutive executions in the chain look the same to some process that is nonfaulty in both. This implies a contradiction, since if two executions look the same to a nonfaulty process, that process must make the same decision in both executions, and therefore, by the agreement requirement, both executions must have the same decision value. Thus every execution in the chain must have the same decision value, which is impossible since the first execution must decide 0, and the last must decide 1.

We start the chain with the execution exec(ρ_0) determined from the run ρ_0 having all 0 input values and the complete communication pattern, as in Figure 6.12. This execution must decide 0.

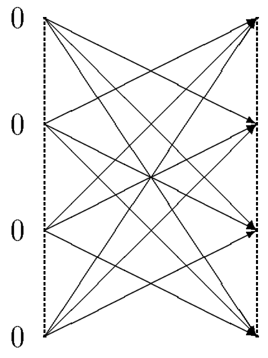


Figure 6.12: A run, ρ_0, which must decide 0.

Starting from execution exec(ρ_0), form the next execution by removing the message from p_1 to p_2. These two executions look the same to all processes except p_1 and p_2. Since $n \geq f+2 = 3$, there is at least one such other process, and it is nonfaulty in both, and hence must decide the same in both.

Next we remove the message from p_1 to p_3; these two look the same to all processes except p_1 and p_3. We continue in this manner removing one message at a time in such a way that every consecutive pair of executions looks the same to some nonfaulty process.

Once we have removed all the messages from p_1, we continue by changing p_1's input value from 0 to 1. Of course these two executions will look the same to every process except p_1 since p_1 sends no messages. Now we can add the messages back in one by one, and again every consecutive pair of executions will look the same to some nonfaulty process. This all yields exec(ρ_1), where ρ_1 has 1 as input value for p_1, 0 for all the others, and complete communication.

Next, we repeat this construction for p_2, removing p_2's messages one-by-one, changing p_2's input value from 0 to 1, and then adding p_2's messages back in. This yields ρ_2. Repeating this construction for p_3,...,p_n, we end the chain with the execution exec(ρ_n), where all start with 1 and there is complete communication. This execution must decide 1. Thus we have produced a chain as claimed, and this is a contradiction.

The Case of Two Failures. Before moving to the general case, we will do one more preliminary case. We will now show that two rounds are not sufficient to handle two stopping failures. Again, we proceed by contradiction. We form a chain as we did for the case of one fault and one round. We start with the execution determined by all 0's as input and the complete (2-round) communication pattern; this execution must decide 0.

To form the chain we want again to work toward killing p_1 at the beginning. When we were only dealing with one round we could kill messages from p_1 one-by-one. Now, if we delete a first-round message from p_1 to q in one step of the chain, then it is no longer the case that the two executions must look the same to some nonfaulty process. This is because in the second round q could inform all other processes as to whether or not it received a message from p_1 in the first round, so that at the end of the second round the two executions can differ for all non-faulty processes.

We solve this problem by using several steps to delete the first-round message from p_1 to q, and by letting q be faulty too (we are allowed two faults). We start with an execution in which p_1 sends to q in the first round and q sends to every process in the second round. Now we let q be faulty and remove second-round messages from q, one-by-one, until we have an execution in which p_1 sends to q in the first round and q sends no messages in the second round. Now we can remove the first-round message from p_1 to q, and these two executions will only look different to p_1 and q. Now we replace second-round messages from q one-by-one until we have an execution in which p_1 does not send to q in the first round and q sends to all in the second round. This achieves our goal of removing a first-round message from p_1 to q while still letting each consecutive pair of executions look the same to some nonfaulty process.

In this way, we can remove first-round messages from p_1 one-by-one until p_1 sends no messages. Then we can change p_1's input from 0 to 1 as before, and replace p_1's messages one-by-one. Repeating this for $p_2,\ldots,p_n$ gives the desired chain.

The General Case. The preceding two examples contain the main ideas for the general case, in which there are up to f faulty processes in a run. Suppose we have an f round protocol for f faults.

A little notation may help. If ρ and ρ' are runs with p nonfaulty in both then we write $\rho \overset{p}{\sim} \rho'$ to mean that exec(ρ) and exec(ρ') look the same to p through time f (same state sequence and same messages received).

We write $\rho \sim \rho'$ if there exists a process p that is nonfaulty in both ρ and ρ' such that $\rho \overset{p}{\sim} \rho'$. We write $\rho \approx \rho'$ for the transitive closure of the $\sim$ relation.

Every run ρ has a decision value denoted by dec(ρ). If $\rho \sim \rho'$ then dec(ρ) = dec(ρ') since ρ and ρ' look the same to some nonfaulty process. Thus if $\rho \approx \rho'$ then dec(ρ) = dec(ρ').

Now let F_ρ be the set of failures in run ρ.

Lemma 5 *Let ρ and ρ' be runs. Let p be a process. Let k be such that $0 \leq k \leq f-1$. If $|F_\rho \cup F_{\rho'}| \leq k+1$ and ρ and ρ' only differ in p's failure behavior after time k then $\rho \approx \rho'$.*

(This means that the patterns are the same, except for some differences in the existence of some outgoing messages from p at rounds $k+1$ or greater.)

To get our result we need to use the case $k = 0$ in Lemma 5. This says that if runs ρ and ρ' have only one failure between them and differ only in the failure behavior of one process then $\rho \approx \rho'$. We use this as follows.

Let ρ_0 and ρ_0' be two runs both starting with all 0's as input; in ρ_0 no process fails, and in ρ_0', process p_1 fails at the beginning and sends no messages. There is only one failure between them and they only differ in p_1's failure behavior only, so by the lemma, $\rho_0 \approx \rho_0'$.

Now let ρ_0'' be the same as ρ_0' except p_1 has a 1 as input instead of 0. Since p_1 sends no messages in ρ_0' this change cannot be seen by any process except p_1, so $\rho_0' \sim \rho_0''$. Now let ρ_1 have the same input as ρ_0'' except there are no failures in ρ_1. Again the lemma says $\rho_0'' \approx \rho_1$. Therefore $\rho_0 \approx \rho_1$.

We continue this, letting ρ_i be the execution with no failures and input defined by having the first i processes get 1 as input and the others get 0. We have $\rho_{i-1} \approx \rho_i$, so $\rho_0 \approx \rho_n$, and therefore $\text{dec}(\rho_0) = \text{dec}(\rho_n)$. But this is a contradiction since ρ_0 has all zeros as input with no failures and hence must decide 0 whereas ρ_n has all ones as input with no failures and hence must decide 1.

It remains to prove the lemma. This will be done in the next lecture.

Lecturer: Nancy Lynch

Lecture 7

7.1 Number of Rounds With Stopping Failures (cont.)

Last time, we were showing the lower bound of $f+1$ rounds on consensus with stopping. We had gotten it down to the following lemma. Today we will finish by proving the lemma.

Lemma 1 *Let ρ and ρ' be runs. Let p be a process. Let k be such that $0 \le k \le f-1$. If $|F_\rho \cup F_{\rho'}| \le k+1$ and ρ and ρ' only differ in p's failure behavior after time k then $\rho \approx \rho'$.*

Intuitively, the lemma means that the patterns are the same, except for some differences in the outgoing messages from p at round $k+1$ or later.

Proof: We prove the lemma by reverse induction on k, i.e., k goes from $f-1$ down to 0.

Base case: Suppose $k = f-1$. In this case ρ and ρ' agree up to the end of round $f-1$. Consider round f. If ρ and ρ' do not differ at all then we are done, so suppose they differ. Then p is faulty in at least one of ρ or ρ', and the total number of other processes that fail in either ρ or ρ' is no more than $f-1$. Consider two processes, q and r, that are nonfaulty in both ρ and ρ'. Note that such two processes exist since $n \ge f+2$.

Let ρ_1 be the same as ρ, except that p sends to q in round f of ρ_1 exactly if it does in ρ' (see Figure 7.1 for example).

We have $\rho \overset{r}{\sim} \rho_1$ and $\rho_1 \overset{q}{\sim} \rho'$, so $\rho \approx \rho'$.

Inductive step: We want to show that the lemma is true for k, where $0 \le k < f-1$, assuming that it holds for $k+1$. Runs ρ and ρ' agree up to the end of round k. If they also agree up to end of round $k+1$ then we can apply the inductive hypothesis and we are done, so assume process p fails in round $k+1$, and assume (without loss of generality) that p fails in ρ.

Let ρ_i, for $1 \le i \le n$, be the same as ρ except that p sends to $p_1, \ldots, p_i$ at round $k+1$ of ρ_i exactly if it does in ρ'.

Claim 2 *For $1 \le i \le n$, $\rho_{i-1} \approx \rho_i$ (where $\rho_0 = \rho$).*

Proof: Executions ρ_{i-1} and ρ_i differ at most in what p sends to p_i at round $k+1$. Let ρ'_i be the same as ρ_{i-1} except that p_i sends no messages after round $k+1$ in ρ'_i. Similarly let ρ''_i

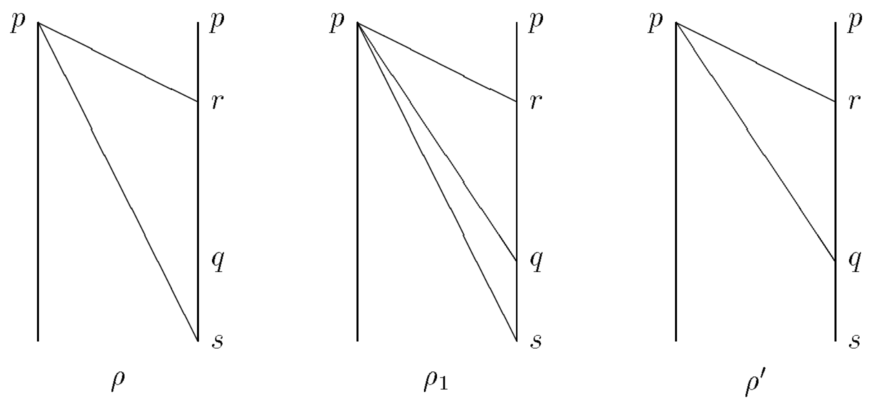


Figure 7.1: Example of construction used to prove base case of Lemma 1.

be the same as ρ_i except that p_i sends no messages after round $k+1$ in ρ_i''. This situation is illustrated in Figure 7.2.

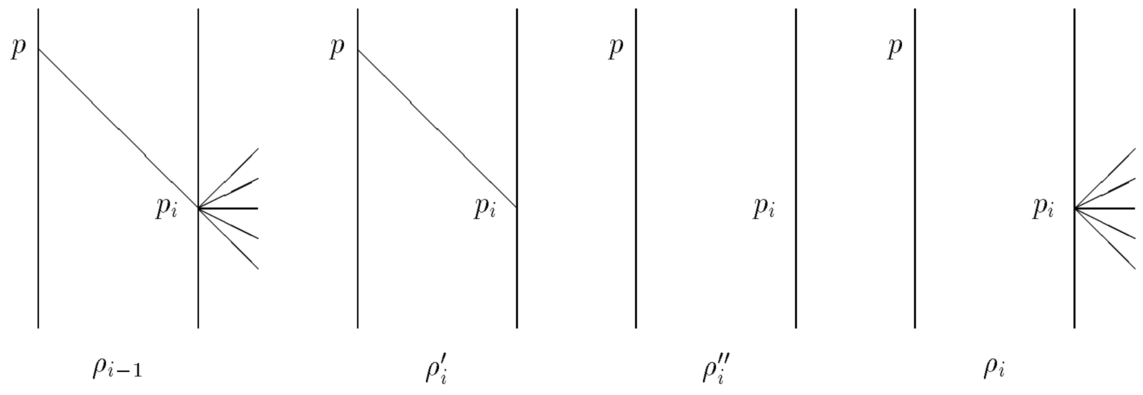


Figure 7.2: Example of construction used to prove inductive step of Lemma 1.

Notice that each of ρ_i, ρ_i', ρ_i' has no more than $k+2$ failures (since ρ has no more than $k+1$ failures). Therefore, by the inductive hypothesis, we have $\rho_{i-1} \approx \rho_i'$ and $\rho_i \approx \rho_i''$. Also $\rho_i' \sim \rho_i''$ since both look the same to any nonfaulty process other than p_i. Thus $\rho_{i-1} \approx \rho_i$. ▮

From the claim, and by the transitivity of $\approx$, we have $\rho = \rho_0 \approx \rho_n$, and since ρ_n and ρ' only differ in p's failure behavior after time $k+1$, we can apply the inductive hypothesis once again to get $\rho_n \approx \rho'$. Thus we have $\rho \approx \rho'$, as needed. ▮

Applying Lemma 1 with $k = 0$, we can summarize with the following theorem.

Theorem 3 *Any agreement protocol that tolerates f stopping failures requires $f+1$ rounds.*

7.2 The Commit Problem

The *commit problem* [Dwork, Skeen] is a variant of the consensus problem, and may be described as follows. We assume that the communication is realized by a complete graph. We assume reliable message delivery, but there may be any number of process stopping faults.

Agreement: There is at most one decision value.

Validity:

1. If any process starts with 0, then 0 is the only possible decision value.
2. If all start with 1 and there are no failures, then 1 is the only possible decision value.

Termination: This comes in two flavors. The *weak termination* requirement is that termination is required only in the case where there are no failures; the *strong termination* requirement specifies that all nonfaulty processes terminate.

The main differences between the between the commit problem and the consensus problem for stopping faults are, first, in the particular choice of validity condition, and second, in the consideration of a weaker notion of termination.

7.2.1 Two Phase Commit (2PC)

For weak termination, the standard algorithm in practice is *two-phase commit.* The protocol consists of everyone sending their values to a distinguished process, say p_1, p_1 deciding based on whether or not anyone has an initial 0 value, and broadcasting the decision (see Figure 7.3 for illustration).

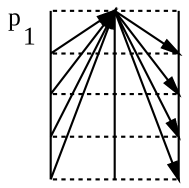


Figure 7.3: message pattern in two-phase-commit protocol

It is easy to see that 2PC satisfies agreement, validity, and weak termination.

However, 2PC does not satisfy strong termination, because if p_1 fails, the other processes never finish. In practice, if p_1 fails, then the others can carry out some kind of communication among themselves, and may sometimes manage to complete the protocol.

But this does not always lead to termination, since the other processes might not be able to figure out what a failed p_1 has already decided, yet they must not disagree with it. For example, suppose that all processes except for p_1 start with 1, but p_1 fails before sending any messages to anyone else. Then no one else ever learns p_1's initial value, so they cannot decide 1. But neither can they decide 0, since as far as they can tell, p_1 might have already decided 1.

This protocol takes only 2 rounds. The weaker termination condition, however, implies that this does not violate the lower bound just proved on the number of rounds. (Note that the earlier result could be modified to use the commit validity condition.)

7.2.2 Three-phase Commit (3PC)

2PC can be augmented to ensure *strong termination*. The key is that p_1 does not decide to commit unless everyone else knows that it intends to. This is implemented using an extra "phase". The algorithm proceeds in four rounds as follows (see also Figure 7.4).

Round 1: All processes send their values to p_1. If p_1 receives any 0, or doesn't hear from some process, then p_1 decides 0. Otherwise, it doesn't yet decide.

Round 2: If p_1 decided 0, it broadcasts this decision. If not (that is, it has received 1's from everyone), then p_1 sends *tentative 1* to all other processes. If a process p_i, $i \neq 1$, receives 0, it decides 0. If it receives *tentative 1*, it records this fact, thus removing its uncertainty about the existence of any initial 0's, and about p_1's intended decision.

Round 3: If a process p_i, $i \neq 1$ recorded a tentative 1, it sends *ack* to p_1. At the end of this round, process p_1 decides 1 (whether or not it receives *ack*s from all the others). Note that p_1 knows that each process has either recorded the tentative 1, or else has failed without deciding and can therefore be ignored.

Round 4: If p_1 decided 1, it broadcasts 1. Anyone who receives 1 decides 1.

Rounds 2 and 3 serve as an extra phase of informing everyone of the "intention to commit", and making sure they know this intention, before p_1 actually commits.[4]

[4]Note that there is an apparent redundancy in this presentation. It appears that the *ack*s, which arise in the practical versions of the protocol, are ignored in this abstract version. So this version can apparently be shortened to three rounds, eliminating the *ack*s entirely. This remains to be worked out.

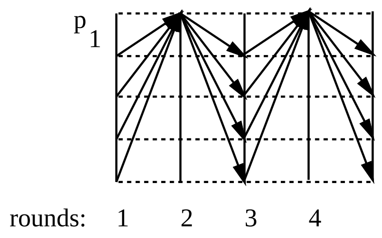


Figure 7.4: message pattern in three-phase-commit protocol

This protocol does not guarantee strong termination yet. If p_1 doesn't fail, every nonfaulty process will eventually decide, since p_1 never blocks waiting for the others. But if p_1 fails, the others can be left waiting, either to receive the tentative 1 or the final 1. They must do something at the end of the four rounds if they haven't decided yet.

At this point, the literature is a little messy — elaborate protocols have been described, with many alternative versions. We give below the key ideas for the case where p_1 has failed. After 4 rounds, every process $p \neq p_1$, even it has failed, is in one of the following states:

- decided 0,
- uncertain (hasn't decided, and hasn't received a round 2 "tentative 1" message),
- ready (hasn't decided, and *has* received a round 2 "tentative 1" message), or
- decided 1.

The important property of 3PC is stated in the following observation.

Observation 1 *Let p be any process (failed or not, including p_1). If p has decided 0, then all nonfaulty processes (in fact, all processes) are either uncertain or have decided 0, and if p has decided 1, then all nonfaulty processes are either ready or have decided 1.*

Using this observation, the non-failed processes try to complete the protocol. For instance, this could involve electing a new leader. One solution is to use p_2 (and if this fails, to use p_3, etc.). (This assumes that the processes know their indices.) The new leader collects the status info from the non-failed processes, and uses the following termination rule.

1. If some process reports that it decided, the leader decides in the same way, and broadcasts the decision.
2. If all processes report that they are uncertain, then the leader decides 0 and broadcasts 0.

3. Otherwise (i.e., all are uncertain or ready, and at least one is ready), the leader continues the work of the original protocol from the "middle" of round 2. That is, it sends *tentative 1* messages to all that reported uncertain, waits, and decides 1. (Notice that this cannot cause a violation of the validity requirement, since the fact that someone was ready implies that all started with initial value 1.)

Any process that receives a 0 or 1 message decides on that value.

7.2.3 Lower Bound on the Number of Messages

It is interesting to consider the number of messages that are sent in the protocol. In this section we prove the following theorem [Dwork-Skeen].

Theorem 4 *Any protocol that solves the commit problem, even with weak termination, sends at least $2n-2$ messages in the failure-free run starting with all 1's.*

Note that 2PC uses $2n-2$ messages in the failure-free case, and therefore the result is tight.

The key idea in the lower bound is stated in the following lemma.

Lemma 5 *Let ρ be the run where all processes start with 1, and there are no failures. Then there must be a path of messages from every node to every other node.*

Before we prove Lemma 5, let us consider a special case. Consider the failure-free execution ρ whose graph is depicted in Figure 7.5. Assume all processes start with 1 in ρ.

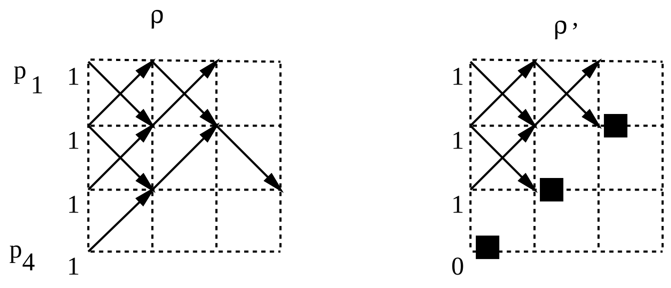


Figure 7.5: execution graphs for ρ and ρ' in the special case. Black square represents process failure.

In ρ, there is no path from p_4 to p_1. Consider now an alternative execution ρ', in which the initial value of p_4 is 0, and all processes stop when they first "hear" from p_4. (In other words, in ρ', all the edges reachable from the node of p_4 at time 0 are removed.) It is straightforward to show that $\rho \overset{p_1}{\sim} \rho'$ *regardless of the initial value of* p_4. However, validity requires that p_1 decides 1 in ρ, and 0 in ρ', a contradiction.

The general case proof uses the same argument.

Proof: By example. In ρ, the decision of all processes must be 1, by termination and validity. If the lemma is false, then there is some pair of processes p, q such that there is no path of messages from p to q in ρ. Thus in ρ, q never learns about p. Now construct ρ' by failing every process exactly when it first learns about p, and changing p's input to 0. The resulting run looks the same to q, i.e., $\rho \approx \rho'$, and therefore q decides 1 in ρ', which violates validity. ∎

The proof of Theorem 4 is completed by the following graph-theoretic lemma.

Lemma 6 *Let G be a communication graph for n processes. If each of i processes is connected to each of all the n processes (by a chain of messages), then there are at least $n+i-2$ messages in the graph.*

Proof: By induction on i.

Base case: $i = 1$. The $n-1$ follows from the fact that there must be some message to each of the $n-1$ processes other than the given process.

Inductive step: assume the lemma holds for i. Let S be the set of $i+1$ processes that are joined to all n. Without loss of generality, we can assume that in round 1, at least one of the processes in S sends a message. (If not, we can remove all the initial rounds in which no process in S sends a message — what is left still joins S to all n.) Call some such process p. Now consider the reduced graph, obtained by removing a single first-round message from p to anyone. The remaining graph connects everyone in $S - \{p\}$ to all n. By induction, it must have at least $n+i-2$ edges. So the whole graph has at least $n+i-2+1$ edges, which is $n+(i+1)-2$, as needed. ∎

Lecturer: Nancy Lynch

Lecture 8

8.1 Asynchronous Shared Memory

In this lecture we begin a new topic, which features a major switch in the flavor of the algorithms we study. Namely, we add the very important uncertainty of asynchrony — for processes, variable process speeds, and for messages, variable message delay. We shall start this part with shared memory rather than networks. These systems can be thought of as the asynchronous analog of PRAMs. At first, as a simplifying assumption, we will not consider failures: asynchrony is complicated enough to deal with.

Before we can express any algorithm, or make any meaningful statement, we need to define a new computation model. We first describe the model informally and consider an example to gain some intuition; only then will we come back to a more formal model.

8.1.1 Informal Model

The system is modeled as a collection of processes and shared memory cells, with external interface (see Figure 8.1).

Each process is a kind of state machine, with a set *states* of states and a subset *start* of *states* indicating the initial states, as before. But now it also has labeled *actions* describing activities it participates in. These are classified as either *input*, *output*, or *internal* actions. We further distinguish between two different kinds of internal actions: those that involve the shared memory, and those which involve strictly local computation.

There is no message generation function, since there are no messages in this model (all communication is via the shared memory).

There is a transition relation *trans*, which is a set of (s, π, s') triples, where π is the label of an input, output, or local computation action. The triple $(s, \pi, s') \in \mathit{trans}$ says that from a state s, it is possible to go to state s' while performing action π. (Note that *trans* is a *relation* rather than a function — this generality includes the convenience of nondeterminism.)

We assume that input actions can always happen (technically, this is an *input-enabling* model). Formally, for every s and input action π, there exists s' such that $(s, \pi, s') \in \mathit{trans}$.

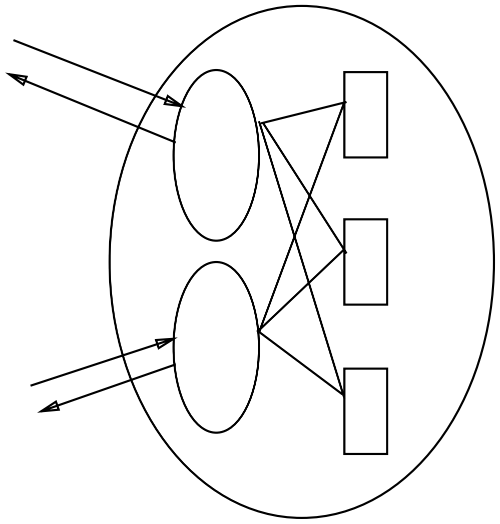


Figure 8.1: shared memory system. Circles represent processes, squares represent shared memory cells, and arrows represent the external interface (i.e., the input and output actions).

In contrast, output and local computation steps might be enabled only in a subset of the states. The intuition behind the input-enabling property is that we think of the input actions as being controlled by an arbitrary external user, while the internal and output actions are under the control of the system.

Shared memory access transitions are formalized differently. Specifically, the transitions have the form $((s, m), \pi, (s', m'))$, where s, s' are process states, and m, m' are shared memory states. These describe changes to the state and the shared memory as a result of accessing the shared memory from some state. A special condition that we impose here is that the *enabling* of a shared memory access action should depend on the local process state only, and not on the contents of shared memory; however, the particular *response*, i.e., the change to the states of the process and of the shared memory, can depend on the shared memory state. Formally, this property is captured by saying that if π is enabled in (s, m), then for all memory states m', π is enabled in (s, m'). In particular cases, the accesses are further constrained.

The most common restriction is to allow access only one location to be involved in a single step. Another popular restriction is that all accesses should be of type *read* or *write*. A read just involves copying the value of a shared memory location to a local variable, while a write just involves writing a designated value to a shared memory location, overwriting whatever was there before.

An execution is modeled very differently from before. Now processes are allowed to take steps one at a time, in *arbitrary* order (rather than taking steps in synchronized rounds).

This arbitrariness is the essence of asynchrony. More formally, an execution is an alternating sequence $s_0, \pi, s_1, \ldots$, consisting of system states (i.e., the states of all the processes and of the shared memory), alternated with actions (each identified with a particular process), where successive (state, action, state) triples satisfy the transition relation. An execution may be a finite or an infinite sequence.

There is one important exception to the arbitrariness: we don't want to allow a process to stop when it is supposed to be taking steps. This situation arises when the process is in a state in which any *locally controlled* action (i.e., non-input action) is enabled. (Recall that input actions are always enabled; however, we will not require that they actually occur.) A possible solution is to require that if a process takes only finitely many steps, its last step takes it to a state in which no more locally controlled actions are enabled. But that is not quite sufficient: we would like also to rule out another case, where a process does infinitely many steps but after some point all these steps are input steps. We want to make sure that the process itself also gets turns. But we have to be careful in saying this: consider the scenario in which infinitely many inputs occur, and no locally-controlled actions, but in fact no locally controlled actions are enabled. That seems OK, since vacuously, the process had "chances" to take steps but simply had none it wanted to take. We account for all these possibilities in the following definition. For each process i, either

1. the entire execution is finite, and in the final state no locally controlled action of process i is enabled, or

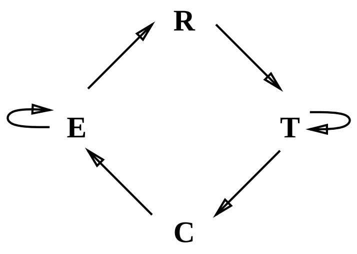
2. the execution is infinite, and there are either infinitely many places in the sequence where i does a locally controlled action, or else infinitely many places where no such action is enabled.

We call this condition the *fairness* condition for this system.

We will normally consider the case where the execution is fair. However, at a later point in the course (when we consider "wait-free objects" and other fault-tolerance issues), we will want to prove things about executions that are not necessarily fair.

8.2 Mutual Exclusion Problem

The problem of Mutual Exclusion involves allocating a single indivisible resource among n processes $p_1, \ldots, p_n$. Having the access to the resource is modeled by the process reaching a *critical region*, which is simply a designated subset of its states. Normally, that is, when it doesn't want access to the resource, the process is said to be in the *remainder region*. In order to gain access to the critical region, a process executes a *trying protocol*, while for symmetry, after the process is done with the resource, it executes an (often trivial) *exit protocol.* This procedure can be repeated, and the behavior of each process is thus cyclic, moving from the *remainder region* (R) to the *trying region* (T), then to the *critical region* (C), and finally to the *exit region* (E). The cycle of regions that a process visits is shown in Figure 8.2. In the diagram, self-loops indicate that a process can remain in these regions after it performs a step. In contrast, we will abstract away steps done within the critical and remainder regions, and not consider them here.



$$\begin{tikzcd} & \mathbf{R} \arrow[dr] & \\ \mathbf{E} \arrow[ur] \arrow[loop left] & & \mathbf{T} \arrow[dl] \arrow[loop right] \\ & \mathbf{C} \arrow[ul] & \end{tikzcd}$$

Figure 8.2: the cycle of regions of a single process

We will consider here only algorithms in which communication among processes is done through shared memory. For starters, we will suppose that the memory is read-write only, that is, in one step, a process can either read or write a single word of shared memory. To match this up with the shared memory model discussed above, we have n process automata accessing read-write shared memory. The two basic actions involving process p_i and a shared variable x are:

- p_i reads x and uses the value read to modify the state of p_i, and
- p_i writes a value determined from p_i's state to x.

The inputs to processes are try_i actions, modeling the rest of the program requesting access to the resource, and $exit_i$, modeling the rest of the program announcing it is done with the resource. The output actions are $crit_i$, which is interpreted as granting the resource, and rem_i, which tells the rest of the program that it can continue. The processes inside the diagram, then, are responsible for performing the trying and exit protocols.

It is sometimes useful to view the outside world as consisting of n *users*, which we model as other state machines that communicate with their respective processes via the designated actions. These users are assumed to execute some application programs. In between the $crit_i$ and $exit_i$ action, $user_i$ is free to use the resource (see Figure 8.3).

We will assume that each user obeys the cyclic behavior, that is, that each user is not the first to violate the cyclic order of actions between itself and its process.

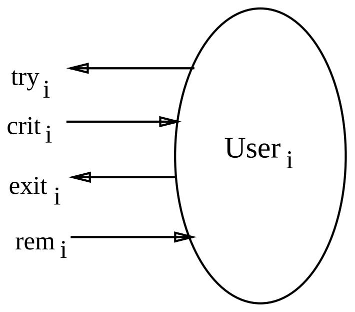


Figure 8.3: interface specification for the user

A global picture of the system architecture appears in Figure 8.4.

Within this setting, the protocol is itself supposed to preserve the cyclic behavior, and moreover, to satisfy the following basic properties.

Mutual exclusion: There is no reachable state in which more than one process is in region C.

Deadlock-Freedom: If at any point in a *fair execution*, at least one process is in the trying region and no process is in the critical region, then at some later point some process enters the critical region. Similarly, if at any point in a fair execution, at least one process is in the exit region (with no other condition here), then at some later point some process enters the critical region.

Note that deadlock-freedom presupposes fairness to all the processes, whereas we don't need to assume this for the other conditions. In the ensuing discussions of mutual exclusion, we might sometimes neglect to specify that an execution is fair, but it should be clear that we assume it throughout — until otherwise specified (namely, when we consider faults).

Note that responsibility for the entire system continuing to make progress depends not only on the protocol, but on the users as well. If a user gets the resource but never returns it (via an $exit_i$), then the entire system will grind to a halt. But if the user returns the resource every time, then the deadlock-freedom condition implies that the system continues to make progress (unless everyone remains in its remainder region).

Note that the deadlock-freedom property does not imply that any particular process ever succeeds in reaching its critical region. Rather, it is a global notion of progress, saying that *some* process reaches the critical region. For instance, the following scenario does not violate the deadlock-freedom condition: p_1 enters T, while p_2 cycles repeatedly through the regions (the rest of the processes are in R). The deadlock-freedom condition does not guarantee that p_1 will ever enter C.

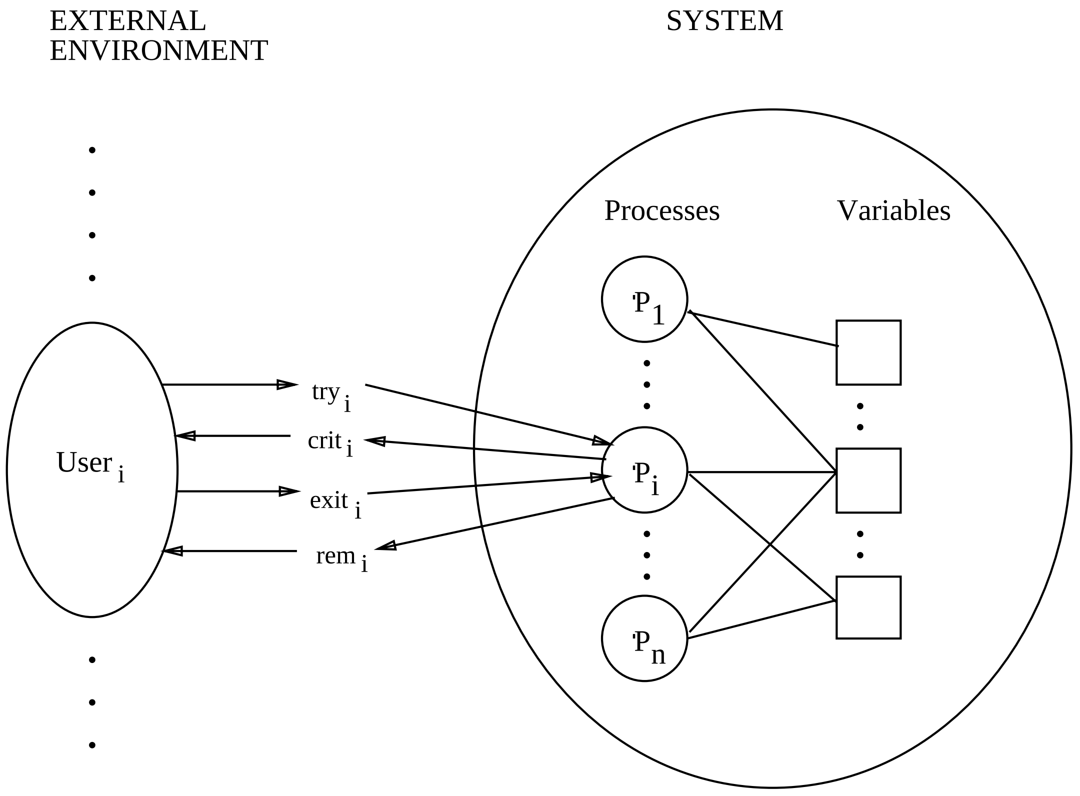


Figure 8.4: Architecture for the mutual exclusion problem

There is one other constraint: a process can have a locally controlled action enabled only when it is in $T \cup E$. This implies that the processes can actively execute the protocol only when there are active requests. The motivation for this constraint is as follows. In the original setting where this problem was studied, the processes did not have dedicated processors: they were logically independent processes executed on a single time-shared processor. In this setting, having special processes managing the mutual exclusion algorithm would involve extra context-switching, i.e., between the manager process and the active processes. In a true multiprocessor environment, it is possible to avoid the context-switching by using a dedicated processor to manage each resource; however, such processors would be constantly monitoring the shared memory, causing memory contention. Moreover, a processor dedicated to managing a particular resource is unavailable for participation in other computational tasks.

8.3 Dijkstra's Mutual Exclusion Algorithm

The first mutual exclusion algorithm for this setting was developed in 1965 by Edsger Dijkstra. It was based on a prior two-process solution by Dekker. Until then, it was not even clear if the problem is solvable.

We begin by looking at the code, though it will not be completely clear at first how this code maps to the state-machine model. The code is given in Figure 8.3.

The shared variables are $\mathit{flag}[i], 1 \leq i \leq n$, one per process, each taking on values from $\{0, 1, 2\}$, initially 0, and *turn*, an integer in $\{1, \ldots n\}$. Each $\mathit{flag}[i]$ is a *single-writer multi-reader* variable, writable only by process i but readable by everyone. The *turn* variable is *multi-writer multi-reader*, both writable and readable by everyone.

We translate the code to our model as follows. The state of each process consists of the values of its local variables, as usual, plus some other information that is not represented explicitly in the code, including:

- Temporary variables needed to remember values of variables read.
- A program counter, to say where in the code the process is up to.
- Temporary variables introduced by the flow of control of the program (e.g., the **for** loop introduces a set to keep track of the indices already processed).
- A region designation, T, C, E or R.

The state of the entire system consists of process states plus values for all the shared variables.

The initial state of the system consists of initial values for all local and shared variables, and program counters in the remainder region. (Temporary variables can be undefined.) The actions are try_i, crit_i, exit_i, rem_i, local computation steps, and the read-write accesses to the shared variables. The steps just follow the code in the natural way. The code describes the changes to the local and shared variables explicitly, but the implicit variables also need to be changed by the steps. For example, when a try_i action occurs, the region designation for i becomes T. The program counter changes as it should, e.g., when try_i occurs, it becomes a pointer to the beginning of the trying region code.

This code is adapted from Dijkstra's paper. Note that the code does not specify explicitly exactly which portions of the code comprise indivisible steps. However, since the processes are asynchronous, it is important to do this. For this algorithm, atomic actions are the try_i, etc., plus the single reads from and writes to the shared variables, plus some local computation steps. The code is rewritten in Figure 8.6 to make the indivisible steps more

Shared variables:

- *flag* : an array indexed by $[1..n]$ of integers from {0,1,2}, initially all 0, written by p_i and read by all processes.
- *turn* : integer from {1,...,n}, initially arbitrary, written and read by all processes.

Code for p_i:

```
  ** Remainder Region **
$try_i$

  ** begin first stage, trying to obtain $turn$ **
L: $flag[i] \leftarrow 1$
while $turn \neq i$ do
    if $flag[turn] = 0$ then $turn \leftarrow i$
    end if
end while

  ** begin second stage, checking that no other processor has reached this stage **
$flag[i] \leftarrow 2$
for $j \neq i$ do ** Note: order of checks unimportant **
    if $flag[j] = 2$ then goto L
    end if
end for
$crit_i$

  ** Critical Region **
$exit_i$
$flag[i] \leftarrow 0$
$rem_i$
```

Figure 8.5: Dijkstra's mutual exclusion algorithm

explicit. The atomic actions are enclosed in pointy brackets. Note that the read of *flag*[*turn*] does not take two atomic actions because *turn* was just read in the line above, and so a local copy of *turn* can be used. The atomicity of local computation steps is not specified — in fact, it is unimportant, so any reasonable interpretation can be used.

8.3.1 A Correctness Argument

In this section we sketch a correctness argument for Dijkstra's algorithm. Recall that for the synchronous model, the nicest proofs resulted from invariants describing the state of the system after some number of rounds. In the asynchronous setting, there is no notion of round, so it may not be readily clear how to use assertional methods. It turns out that they can still be used, and indeed are extremely useful, but typically it is a little harder to obtain the invariants. The arguments must be applied at a finer level of granularity, to make claims about the system state after any number of *individual process steps* (rather than rounds).

Before presenting an assertional proof, however, we will sketch a "traditional" *operational* argument, i.e., an argument based directly on the executions of the algorithm.

In the following series of lemmas we show that the algorithm satisfies the requirements. The first lemma is very each to verify.

Lemma 1 *Dijkstra's algorithm preserves cyclic region behavior for each i.*

The second lemma is more interesting.

Lemma 2 *Dijkstra's algorithm satisfies mutual exclusion.*

Proof: By contradiction. Assume p_i and p_j are both simultaneously in region C, in some reachable state, where $i \neq j$. Consider an execution that leads to this state. By the code, both p_i and p_j set their *flag* variables to 2 before entering their critical regions. Consider the last such step in which each sets its *flag* variable. Assume, without loss of generality, that *flag*[i] is set to 2 first. Then, *flag*[i] remains 2 from that point until p_i leaves C, which must be after p_j enters C, by the assumption that they both end up in C simultaneously. So, *flag*[i] has the value 2 throughout the time from when p_j sets *flag*[j] to 2 until p_j enters C (see Figure 8.7).

But during this time, p_j does a final test of *flag*[i] and finds it unequal to 2, a contradiction. ▮

We will redo this argument using an invariant assertion proof later. But first, we proceed with an argument for the deadlock-freedom property.

Lemma 3 *Dijkstra's algorithm is deadlock-free.*

Proof: We again argue by contradiction. Suppose there is some fair execution α that reaches a point where there is at least one process in T, and no process in C, and suppose

Shared variables:

- *flag* : an array indexed by $[1..n]$ of integers from {0,1,2} initially all 0, written by p_i and read by all
- *turn* : integer from {1,...,n}, initially arbitrary, written and read by all

Code for p_i

```
  **Remainder region**
$try_i$
L: $\langle flag[i] \leftarrow 1 \rangle$
while $\langle turn \neq i \rangle$ do
    if $\langle flag[turn] = 0 \rangle$ then $\langle turn \leftarrow i \rangle$
    end if
end while
$\langle flag[i] \leftarrow 2 \rangle$
for $j \neq i$ do
    if $\langle flag[j] = 2 \rangle$ then goto L
    end if
end for
$crit_i$
  **Critical region**
$exit_i$
$\langle flag[i] \leftarrow 0 \rangle$
$rem_i$
```

Figure 8.6: Dijkstra's algorithm showing atomic actions

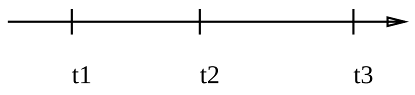


Figure 8.7: At time $t1$, p_i sets *flag*$[i]$ to 2; at time $t2$ p_j finds that *flag*$[i] \neq 2$; at time $t3$ p_i exits C.

that after this point, no process ever enters C. We begin by removing some complications. Note that all processes in $T \cup E$ continue taking steps in α, and hence, if some of the processes are in E, then after one step they will be in R. So, after some point in α, all processes are in $T \cup R$. Moreover, since there are only finitely many processes, after some point no new processes enter T, so after some point, no new processes enter T. That is, after some point in α, all the processes are in $T \cup R$, and no process every changes region. Write $\alpha = \alpha_1 \alpha_2$, where in α_2, there is a nonempty fixed group of processes continuing to take steps forever in T, and no region changes occur. Call these processes *contenders*.

After at most a single step in α_2, each contender i ensures that *flag*$[i] \geq 1$, and it remains ≥ 1 for the rest of α_2. So we can assume (without loss of generality) that this is the case for all contenders, throughout α_2.

Clearly, if *turn* changes during α_2, it is changed to a contender's index. Moreover, we have the following claim.

Claim 4 *In α_2, eventually turn acquires a contender's index.*

Proof: Suppose not, i.e., the value of *turn* remains equal to the index of a non-contender throughout α_2. Then when any contender p_i checks, it finds that $turn \neq i$ and *flag*$[turn] = 0$ and hence would set *turn* to i. There must exist an i for which this will happen, because all contenders are either in the while loop or in the second stage, destined to fail and return to the while loop (because, by assumption, we know they don't reach C in α_2). Thus, eventually in α_2, *turn* gets set to a contender's index. ∎

Once *turn* is set to a contender's index, it remains equal to a contender's index, although the value may change to different contenders. Then, any later reads of *turn* and *flag*$[turn]$ will yield *flag*$[turn] \geq 1$ (since for all contenders i, *flag*$[i] \geq 1$), and *turn* will not be changed. Therefore, eventually *turn* stabilizes to a final (contender's) index. Let α_3 be a suffix of α_2 in which the value of *turn* is stabilized at some contender's index, say i.

Now we claim that in α_3, any contender $j \neq i$ eventually ends up in the first while loop, looping forever. This is because if it is ever in the second stage, then since it doesn't succeed in entering C, it must eventually return to L. But then it is stuck in stage 1, because $turn = i \neq j$ and *flag*$[i] \neq 0$, throughout α_3. So let α_4 be a suffix of α_3 in which all contenders other than i loop forever in the first while loop. Note that this means that all contenders other than i have their *flag* variables equal to 1, throughout α_4.

We conclude the argument by claiming that in α_4, process i has nothing to stand in its way. This is true because when the while loop is completed, $\mathit{flag}[i]$ is set to 2, and no other process has $\mathit{flag} = 2$, so p_i succeeds in its second stage tests and enters C.

To complete the proof we remark that the argument for the exit region is straightforward, since no one ever gets blocked there. ∎

Lecturer: Nancy Lynch

Lecture 9

9.1 Dijkstra's Mutual Exclusion Algorithm (cont.)

In this section we finish our discussion of Dijkstra's mutual exclusion algorithm with an alternative proof of the mutual exclusion property of the algorithm. Last time we saw an operational argument. Today we'll sketch an assertional proof, using invariants, just as in the synchronous case (See Goldman-Lynch [LICS-90] for a more detailed proof).

9.1.1 An Assertional Proof

To prove mutual exclusion, we must show that

$$\neg[\exists(p_i, p_j) \quad | \quad (i \neq j) \wedge (p_i \text{ in C}) \wedge (p_j \text{ in C})]$$

We would like to prove this proposition by induction on the number of steps in an execution. But, as usual, the given statement is not strong enough to prove directly by induction. We need to augment it with other conditions, which may involve all the state components, namely shared variables, local variables, program counters, region designations, and temporary variables.

We have to be a little more precise about the use of temporary variables. We define, for each process p_i, a new local variable S_i; this variable is used by p_i during execution of the **for** loop, to keep track of the set of processes that it has already observed to have *flag* $\neq 2$. Initially $S_i = \emptyset$. When the process finds *flag*$[j] \neq 2$ during an iteration of the for loop, it performs the assignment $S_i \leftarrow S_i \cup \{j\}$. When $S_i = \{1, \ldots, n\} - \{i\}$, the program counter of p_i moves to after the **for** loop. When *flag*$[i]$ is set to 0 or 1, p_i assigns $S_i \leftarrow \emptyset$. Figure 9.1 shows where these uses of S would appear in the code for p_i. (Note that it becomes more convenient to rewrite the for loop as an explicit while loop.)

For a given system state, we define the following sets of processes.

- *before-C:* processes whose program counter is right after the final loop.
- *in-C:* processes whose program counter is in region C.

Shared variables:

- *flag* : an array indexed by $[1..n]$ of integers from {0,1,2}, initially all 0, written by p_i and read by all processes
- *turn* : integer from {1,...,n}, initially arbitrary, written and read by all processes

Code for p_i

```
  ** Remainder Region **
$try_i$
L: $S_i \leftarrow \emptyset$
$\langle flag[i] \leftarrow 1 \rangle$
while $\langle turn \neq i \rangle$ do
    if $\langle flag[turn] = 0 \rangle$ then $\langle turn \leftarrow i \rangle$
    end if
end while
$\langle flag[i] \leftarrow 2 \rangle$
while $S_i \neq \{1,\ldots,n\} - \{i\}$ do
    choose $j \in \{1,\ldots,n\} - \{i\} - S_i$
    if $\langle flag[j] = 2 \rangle$ then goto L
    end if
    $S_i \leftarrow S_i \cup \{j\}$
end while
$crit_i$
  ** Critical Region **
$exit_i$
$S_i \leftarrow \emptyset$
$\langle flag[i] \leftarrow 0 \rangle$
$rem_i$
```

Figure 9.1: Dijkstra's algorithm showing atomic actions and S_i.

(The actual entrance to C is a separate step from the last test in the loop.)

Put in the above terms, we need to prove that $|\mathit{in}\text{-}C| \leq 1$. We actually prove a stronger statement, namely that $|\mathit{in}\text{-}C| + |\mathit{before}\text{-}C| \leq 1$. This is a consequence of the following two claims:

1. $\neg[\exists(p_i, p_j) \quad | \quad (i \neq j) \wedge (i \in S_j) \wedge (j \in S_i)]$

2. $p_i \in (\mathit{before}\text{-}C \cup \mathit{in}\text{-}C) \Longrightarrow S_i = \{1, \ldots, n\} - \{i\}$

First a small lemma.

Lemma 1 *If $S_i \neq \emptyset$ then $\mathit{flag}[i] = 2$.*

Proof: From the definition of S_i and the code for the algorithm. ∎

Now the main nonexistence claim.

Lemma 2 $\neg[\exists(p_i, p_j) \quad | \quad (i \neq j) \wedge (i \in S_j) \wedge (j \in S_i)]$.

Proof: By induction on the length of executions. The basis case is easy since, in the initial state, all sets S_i are empty. Now consider the case where j gets added to S_i (the reader can convince him/herself that this is actually the only case of interest). This must occur when i is in its final loop, testing flag_j. Since j gets added to S_i, it must be the case that $\mathit{flag}[j] \neq 2$ because otherwise, i would exit the loop. By the contrapositive of Lemma 1 then, $S_j = \emptyset$, and so $i \notin S_j$.

∎

Now the second claim.

Lemma 3 $p_i \in (\mathit{before}\text{-}C \ \cup \mathit{in}\text{-}C) \Longrightarrow S_i = \{1, \ldots, n\} - \{i\}$.

Proof: By induction on the length of the execution. In the basis case, all processes are in region R, so the claim holds vacuously. For the inductive step, assume the claim holds in any reachable state and consider the next step. The steps of interest are those where p_i exits the second loop normally (i.e., doesn't **goto** L), or enters C. Clearly, if p_i exits the loop normally, S_i contains all indices except i, since this is the termination condition for the loop (when rewritten explicitly in terms of the S sets). Upon entry to the critical region, S_i does not change. Thus the claim holds in the induction step. ∎

We can now prove the main result by contradiction.

Lemma 4 $|\mathit{in}\text{-}C| + |\mathit{before}\text{-}C| \leq 1$.

Proof: Assume, for contradiction, that in some state reachable in an execution $|\mathit{in}\text{-}C| + |\mathit{before}\text{-}C| > 1$. Then there exist two processes, p_i and p_j, $i \neq j$, such that $p_i \in (\mathit{before}\text{-}C \cup \mathit{in}\text{-}C)$ and $p_j \in (\mathit{before}\text{-}C \cup \mathit{in}\text{-}C)$. By Lemma 3, $S_i = \{1, \ldots, n\} - \{i\}$ and $S_j = \{1, \ldots, n\} - \{j\}$. But by Lemma 2, either $i \notin S_j$ or $j \notin S_i$, a contradiction. ∎

9.1.2 Running Time

We would like to bound the time from any state in which there is some process in T and no one in C, until someone enters C. In the asynchronous setting, however, it is not clear what "time" should mean. For instance, unlike the synchronous case, there is no notion of rounds to count in this model. The way to measure time in this model is as follows. We assume that each step occurs at some real time point, and that the algorithm begins at time 0. We impose upper bounds on step time for individual processes, denoted by s, and on maximum time anyone uses the critical region, denoted by c. Using these bounds, we can deduce time upper bounds for the time required for interesting activity to occur.

Theorem 5 *In Dijkstra's algorithm, suppose that at a particular time there is some process in T and no process in C. Then within time $O(sn)$, some process enters C.*

We remark that the constant involved in the big-O is independent of s, c and n.

Proof: We analyze the time along the same lines as we proved deadlock-freedom. Suppose the contrary, and consider an execution in which, at some point, process p_i is in T and no process is in C, and in which no process enters C for time ksn, for some particular large k.

First we claim that the time elapsed from the starting point of the analysis until p_i tests *turn* is at most $O(sn)$. (We will assume that k has been chosen to be considerably larger than the constant hidden by this big-O.) This is because p_i can at worst spend this much time before failing in the second stage and returning to L. We know that it must fail because otherwise it would go to C, which we have assumed doesn't happen this quickly.

Second, we claim that the additional time elapsed until *turn* equals a contender index is at most $O(s)$. To see this, we need a small case analysis. If when p_i tests *turn*, *turn* holds a contender index, we're done, so suppose that this is not the case; specifically, suppose that *turn* $= j$, where j is not a contender. Then within time $O(s)$ after this test, p_i will test *flag*(j). If p_i finds *flag*$(j) = 0$, then p_i sets *turn*, to i, which is the index of a contender, and we are again done. On the other hand, if it finds *flag*$(j) \neq 0$, then it must be that in between the test of *turn* by p_i and the test of *flag*(j), process j entered the trying region and became a contender. If *turn* has not changed in the interim, then *turn* is equal to the index of a contender (j) and we are done. On the other hand, if *turn* has changed in the interim, then it must have been set to the index of a contender. So again, we are done.

Then after an additional time $O(s)$, no process can ever reset *turn* (or enter the second stage). Then time $O(sn)$ later, all others in the second stage will have left, since they don't succeed because it's too soon. Then within an additional time $O(sn)$, i must succeed in entering C. ∎

9.2 Improved Mutual Exclusion Algorithms

While Dijkstra's algorithm guarantees mutual exclusion and deadlock-freedom, there are other desirable properties that it does not have. Most importantly, it does not guarantee any sort of *fairness* to individual processes; that is, it is possible that one process will continuously be granted access to its critical region, while other processes trying to gain access are prevented from doing so. This situation is sometimes called *process starvation*. Note that this is a different kind of fairness from that discussed earlier: this is fairness to the invoked *operations*, rather than fairness of process steps.

Another unattractive property of Dijkstra's algorithm is that it uses a *multi-reader/multi-writer* variable (for *turn*). This may be difficult or expensive to implement in certain kinds of systems. Several algorithms that improve upon Dijkstra's have been designed. We shall look at some algorithm developed by Peterson, and by Peterson and Fischer.

9.2.1 No-Starvation Requirements

Before we look at alternative mutual exclusion algorithms, we consider what it means for an algorithm to guarantee fairness. Depending upon the context in which the algorithm is used, different notions of fairness may be desirable. Specifically, we shall consider the following three ideas that have been used to refine the requirements for mutual exclusion algorithms.

Lockout-freedom: In a fair execution (with respect to processes' steps) of the algorithm where the users always return the resource, any process in T eventually enters C. (Also, in any fair execution, any process in E eventually enters R.)

Time bound b: If the users always return the resource within time c of when it is granted, and processes always take steps in $T \cup E$ within time s, then any process in T enters C within time b. (Note: b will typically be some function of s and c.) (Also, if processes always take steps in $T \cup E$ within time s, then any process in E enters R within time b.)

Number of bypasses b: Consider an interval of an execution throughout which some process p_i is in T (more specifically, has performed the first non-input step in T). During this interval, any other process p_j, $j \neq i$, can only enter C at most b times. (Also, the same for bypass in E.)

In each case above, we have stated fairness conditions for the exit region that are similar to those for the trying region. However, all the exit regions we will consider are actually trivial, and satisfy stronger properties (e.g., are "wait-free").

Some implications. There are some simple relations among the different requirements from mutual exclusion protocols.

Theorem 6 *If an algorithm is lockout-free then it is deadlock-free.*

Proof: Consider a point in a fair execution such that i is in T, no one is in C, and suppose that no one ever enters C. Then this is a fair execution in which the user always returns the resource. So the lockout-freedom condition implies that i eventually enters C, as needed for deadlock-freedom.

Likewise, consider a point in a fair execution such that i is in E. By lockout-freedom, i eventually enters R, as needed for deadlock-freedom. ∎

Theorem 7 *If an algorithm is deadlock-free and has a bypass bound, then the algorithm is lockout-free.*

Proof: Consider a point in a fair execution. Suppose that the users always return the resource, and i is in T. Deadlock-freedom and the user returning all resources imply that the system keeps making progress as long as anyone is in T. Hence, the only way to avoid breaking the bypass bound, is that eventually i must reach C.

The argument for the exit region is similar. ∎

Theorem 8 *If an algorithm has any time bound then it is lockout-free.*

Proof: Consider a point in a fair execution. Suppose the users always return the resource, and i is in T. Associate times with the events in the execution in a monotone nondecreasing, unbounded way, so that the times for steps of each process are at most s and the times for all the critical regions are all at most c. By the assumption, i enters C in at most the time bound, so in particular, i eventually enters C, as needed for lockout-freedom.

The argument for the exit region is similar. ∎

In the following, we shall see some protocols that satisfy some of these more sophisticated requirements.

9.2.2 Peterson's Two-Process Algorithm

We begin with an algorithm that gives *lockout-freedom*, and a good *time bound* (with an interesting analysis). We start with a 2-process solution, for processes p_0 and p_1. We write $\bar{\imath}$ for $1 - i$, i.e., the index of the other process. The code is given in Figure 9.2.

We now argue that the algorithm is correct.

Theorem 9 *Peterson's two-process algorithm satisfies mutual exclusion.*

Proof Sketch: It is easy to show by induction that

$$level(i) = 0 \Longrightarrow i \notin (\textit{at-wait} \cup \textit{before-}C \cup \textit{in-}C)\ . \tag{9.1}$$

Shared variables:

- *level* : a Boolean array, indexed by $[0,1]$, initially 0
- *turn* : a Boolean variable, initial state arbitrary

Code for p_i:

```
  ** Remainder Region **
$try_i$
$level(i) := 1$
$turn := i$
wait for $level(\bar{\imath}) = 0$ or $turn \neq i$
$crit_i$
  ** Critical Region **
$exit_i$
$level(i) := 0$
$rem_i$
```

Figure 9.2: Peterson's algorithm for two processes

Using (9.1), we can show by induction that

$$i \in (\textit{before-C} \cup \textit{in-C}) \Longrightarrow (\bar{\imath} \notin (\textit{at-wait} \cup \textit{before-C} \cup \textit{in-C})) \vee (turn \neq i) \ . \qquad (9.2)$$

There are three key steps to check: First, when i passes the wait test, then we have one of the following. Either $turn \neq i$ and we are done, or else $level(\bar{\imath}) = 0$, in which case we are done by (9.1).

Second, when $\bar{\imath}$ reaches *at-wait*, then it explicitly sets $turn \neq i$. Third, when *turn* gets set to i, then this must be a step of process p_i, performed when it is not in the indicated region. ∎

Theorem 10 *Peterson's two-process algorithm satisfies deadlock-freedom.*

Proof Sketch: We prove deadlock-freedom by contradiction. That is, assume that at some point, $i \in T$, there is no process in C, and no one ever enters C later. We consider two cases. If $\bar{\imath}$ eventually enters T, then both processes must get stuck at the wait statement since they don't enter C. But this cannot happen, since the value of *turn* must be favorable to one of them.

On the other hand, suppose that $\bar{\imath}$ never reaches T. In this case, we can show by induction that $level(\bar{\imath})$ eventually becomes and stays 0, contradicting the assumption that i is stuck in its wait statement. ∎

Theorem 11 *Peterson's two-process algorithm satisfies lockout-freedom.*

Proof Sketch: We show the stronger property of 2-bounded bypass. Suppose the contrary, i.e., that i is in T after setting $level(i) = 1$, and $\bar{\imath}$ enters C three times. By the code, in the second and third time, $\bar{\imath}$ sets $turn := \bar{\imath}$ and then must afterwards see $turn = i$. This means that there are two occasions upon which i must set $turn := i$ (since only i can set $turn := i$). But by our assumption, $turn := i$ is only performed once during one pass by process i through the trying region, a contradiction. ∎

9.2.3 Tournament Algorithm

We now extend the basic two-process algorithm to a more general number of processes. Here, for simplicity, we assume that n is a power of 2. The basic idea is to run a *tournament*, using the 2-process strategy.

In the following presentation, we take the Peterson-Fischer tournament algorithm, and rephrase it using Peterson's simpler 2-process algorithm. We remark that the original Peterson-Fischer algorithm is quite messy (at least too messy to prove formally here). The code given here seems simpler, but it still needs careful checking and proof. A disadvantage of this code as compared to the Peterson-Fischer code is that this one involves multi-writer variables, while the original protocol works with single-writer registers.

Before we give the algorithm, we need some notation. For $1 \le i \le n$ and $1 \le k \le \log n$ we define the following notions.

- The (i,k)-*competition* is the number given by the high-order $(\log n - k)$ bits of i, i.e.,
$$comp(i,k) = \left\lfloor \frac{i}{2^k} \right\rfloor \ .$$
- The (i,k)-*role* is the $(\log n - k + 1)$st bit of the binary representation of i, i.e.,
$$role(i,k) = \left\lfloor \frac{i}{2^k} \right\rfloor \bmod 2 \ .$$
- The (i,k)-*opponents* is the set of numbers with the same high-order $(\log n - k)$ bits as i, and opposite $(\log n - k + 1)$st bit, i.e.,
$$opponents(i,k) = \{ j \ : \ comp(j,k) = comp(i,k) \ \textbf{and} \ role(j,k) \neq role(i,k) \} \ .$$

Shared variables:

- *level*: an array indexed by $[1..n]$ of $\{0,\ldots,\log n\}$, initially all 0
- *turn*: a Boolean array indexed by all binary strings of length at most $\log n - 1$, initial state arbitrary

Code for p_i:

```
  ** Remainder Region **
$try_i$
for $k := 1$ to $\log n$ do
    $level(i) := k$
    $turn(comp(i,k)) := role(i,k)$
    wait for $[\forall j \in opponents(i,k) : level(j) < k]$ or $[turn(comp(i,k)) \neq role(i,k)]$
end for
$crit_i$
  ** Critical Region **
$exit_i$
$level(i) := 0$
$rem_i$
```

Figure 9.3: Peterson's tournament algorithm

The code of the algorithm is given in Figure 9.3. We only sketch the correctness arguments for the algorithm.

Theorem 12 *The tournament algorithm satisfies mutual exclusion.*

Proof Sketch: The proof follows the same ideas as for two processes. This time, we must show that at most one process from each subtree of a level k node reaches level $k+1$ (or finishes, if $k = \log n$). This is proven using an invariant.

We think of the main loop of the algorithm unfolded. For $1 \leq k \leq \log n$, define $done_k$ to be the set of points in the code from right after the level k loop, until the end of C. Consider the following invariant.

$$\begin{aligned}\forall k, 1 \leq k \leq \log n\, [i \in done_k] \implies \; & opponents(i,k) \cap (wait_k \cup done_k) = \emptyset \\ \textbf{or} \; & turn(comp(i,k)) \neq role(i,k)\ . \end{aligned} \tag{9.3}$$

We first show that (9.3) implies the result: if two processes i, j reach the critical section together, consider their lowest common competition (that is, $\min_l \{comp(i,l) = comp(j,l)\}$). Suppose this competition is at level k. Then both i and j are in $done_k$, and so they must have conflicting settings of the *role* variable.

The inductive proof of (9.3) is roughly as follows. Fix a process i and level number k. We need to verify the following key steps: (a) when i passes the level k test (this case is easy), (b) an opponent setting *turn*, and (c) a relative setting *turn* (i can't, from where it is, and others can't reach there by inductive hypothesis, level $k-1$). ∎

Theorem 13 *The tournament algorithm satisfies deadlock freedom.*

Proof Sketch: Deadlock freedom follows from lockout freedom, which in turn follows from a time bound. We show a time bound. Specifically, we claim that in $O(s \cdot n^2 + cn)$ time any process in T enters C. We prove this bound using a recurrence as follows.

Define $T(0)$ to be the maximum time from when a process enters T until it enters C. For $1 \leq k \leq \log n$, let $T(k)$ be the maximum time it takes a process to enter C after winning (completing the body of the loop) at level k. We wish to bound $T(0)$.

By the code, we have $T(\log n) \leq s$, since only one step is needed to enter C after winning at the top level. Now, in order to find $T(0)$, we set up a recurrence for $T(k)$ in terms of $T(k+1)$.

Suppose that p_i has just won at level k, and advances to level $k+1$. First, p_i does a couple of assignments (taking $O(s)$ time), and then it reaches the wait loop. Now we consider two cases:

1. If p_i finds the wait condition to be true the first time it tests it, then p_i immediately wins at level $k+1$. Since the number of opponents at level k is 2^k, and since p_i must test the *level* of all its opponents, this fragment takes $O(s \cdot 2^k)$ time. The remaining time till it reaches the critical region is at most $T(k+1)$, by definition. The total time is, in this case, at most $O(s \cdot 2^k) + T(k+1)$.

2. If p_i finds the wait condition to be false the first time it tests it, then there is a competitor (say p_j) at a level $\geq k+1$. In this case, we claim that either p_i or p_j must win at level $k+1$ within time $O(s \cdot 2^k)$ of when p_i reaches the wait condition. (In fact, p_j might have already won before p_i reached the wait condition.) This is true because within time $O(s)$, p_j must reach its wait condition, and then one of the wait conditions must be true (because of the value of $turn(comp(i, k+1))$). Within an additional $O(s \cdot 2^k)$, someone will discover the wait condition is true, and win. Now, if p_i is the winner, then it takes an additional $T(k+1)$ until it reaches the critical region; if p_j is the winner, then within additional time $T(k+1)$, p_j reaches C, and within additional

time c, it leaves C. After another $O(s)$ time, it resets $level(j) := 0$. Thereafter, we claim that p_i will soon win at level $k+1$: right after p_j resets its *level*, all the *level* values of $opponents(i,k)$ are less than $k+1$. If they remain this way for time $O(s \cdot 2^k)$, p_i will find its condition true and win. If not, then within this time, some opponent of i reaches level $k+1$. Then within additional time s, it sets *turn* to be favorable to i, and within an additional $O(s \cdot 2^k)$, p_i will discover this fact and win.

The total time is thus at most the maximum of the two cases above, or

$$\max\left\{(s \cdot 2^k) + T(k+1)\ ,\ O(s \cdot 2^k) + 2T(k+1) + c\right\} = O(s \cdot 2^k) + 2T(k+1) + c\ .$$

Thus we need to solve the following recurrence.

$$\begin{aligned} T(k) &\le 2T(k+1) + O(s \cdot 2^k) + c \\ T(\log n) &\le s \end{aligned}$$

Let a be the constant hidden by the big-O notation. We have

$$\begin{aligned} T(0) &\le 2T(1) + as2^0 + c \\ &\le 2^2T(2) + as(2^0 + 2^2) + c(2^0 + 2^1) \\ &\vdots \\ &\le 2^kT(k) + as(2^0 + 2^2 + \ldots + 2^{2k-2}) + c(2^0 + 2^1 + \ldots + 2^{k-1}) \\ &\vdots \\ &\le 2^{\log n}T(\log n) + as(2^0 + 2^2 + \ldots 2^{2(\log n - 1)}) + c(2^0 + 2^1 + \ldots + 2^{\log n - 1}) \\ &= ns + O(s \cdot n^2) + O(cn) \\ &= O(s \cdot n^2 + cn) \end{aligned}$$

∎

Bounded Bypass. As a last remark, we point out the fact that Peterson's algorithm does *not* have any bound on the number of bypasses. To see this, consider an execution in which one process i arrives at its leaf, and takes its steps with intervening times equal to the upper bound s, and meanwhile, another process j arrives in the other half of the tree, going much faster. Process j can reach all the way to the top and win, and in fact it can repeat this arbitrarily many times, before i even wins at level 1. This is due to the fact that is no lower bound was assumed on process step times. Note that there is no contradiction between unbounded bypass and a time upper bound, because the unbounded bypass can only occur when some processes take steps very fast.

Shared variables:

- *level*: an array indexed by $[1..n]$ of $\{0, \ldots, n-1\}$, initially all 0
- *turn*: an array indexed by $[1..n-1]$ of $\{1, \ldots, n\}$, initial state arbitrary

Code for p_i:

```
  ** Remainder Region **
$try_i$
for $k := 1$ to $n-1$ do
    $level(i) := k$
    $turn(k) := i$
    wait for $[\forall j \neq i : level(j) < k]$ or $[turn(k) \neq i]$
end for
$crit_i$

  ** Critical Region **
$exit_i$
$level(i) := 0$
$rem_i$
```

Figure 9.4: Peterson's iterative algorithm

Remark. In the original Peterson-Fischer tournament algorithm (with single-writer variables), the *turn* information is not kept in a centralized variables, but is rather distributed around variables belonging to the separate processes. This leads to a complex system of repeated reads of the variables, in order to obtain consistent readings.

9.2.4 Iterative Algorithm

In Peterson's newer paper, there is another variant of the algorithm presented above. Instead of a tournament, this one conducts only one competition for each level, and rather than allowing only one winner for each competition, it makes sure there is at least one *loser*. In this algorithm, all n processes can compete at the beginning, but at most $n-1$ can have won at level 1 at any particular time, and in general, at most $n-k$ can have won at level k. Thus, after $n-1$ levels, get only one winner. The code is given in Figure 9.4.

6.852J/18.437J Distributed Algorithms October 15, 1992

Lecturer: Boaz Patt-Shamir

Lecture 10

10.1 Atomic Objects

In the past couple of lectures, we have been considering the mutual exclusion problem in the asynchronous shared-memory model. Today we shall consider a new problem in the same model. The new problem is the implementation of an interesting programming language primitive, a kind of data object called an *atomic object*. Atomic objects have recently been proposed as the basis for a general approach to solving problems in the asynchronous shared memory model. This approach can be thought of as "object-oriented" style of constructing asynchronous concurrent systems.

The system architecture is the same as before (see Figure 10.1).

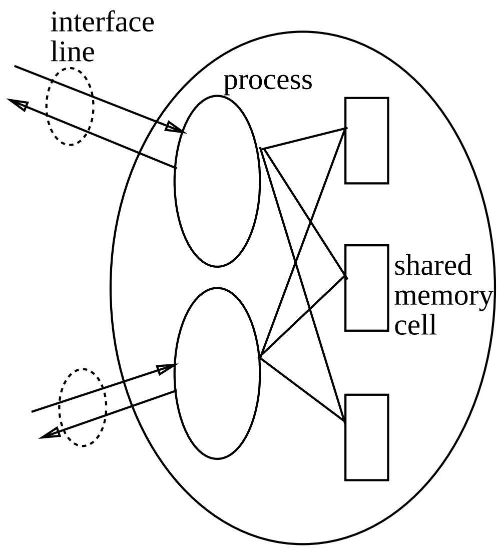


Figure 10.1: shared memory system

Before giving a formal definition of atomic objects, we give a simple example.

10.1.1 Example: Read-Write Object

The read-write object is somewhat similar to a shared read-write variable, but with separate invocation and response actions. More specifically, it has two types of input actions: $read_i$, where i is a process, and $write_i(v)$, where i is a process and v a value to be written. A $read_i$ is supposed to return a value. The "return" event is represented by a separate output action which we call $read\text{-}respond_i(v)$. For uniformity, also give the $write_i$ a corresponding response action $write\text{-}respond_i$. The $write\text{-}respond_i$ is merely an ack, whose meaning is that the $write_i$ action is done.

The separation of actions (which can be viewed as giving a finer atomicity) makes this kind of object a little different from a shared read-write register. In particular, it permits concurrent access to the object from different users.

The read-write object is one of the simplest useful objects. We can easily define other, fancier objects; for example

- queues (with *insert* and *delete* actions),
- read-modify-write objects (to be discussed later),
- snapshot objects (to be considered next),

and more. Each object has its own characteristic interface, with inputs being invocations of operations and outputs being responses.

10.1.2 Definition

In this section we give an elaborate definition of the requirements from an atomic object. First, recall the cyclic discipline for invocations and responses for the "mutual exclusion object". Something similar is required from any atomic object: we define a generalized notion of *well-formedness* at the interface. Here, this says that for each process i (whose connection with the external environment is called *line* i), the invocations and responses alternate, starting with an invocation. We stress that not everything needs to alternate in this way: only the invocations and responses on any particular line. There can be concurrency among the invocations on different lines. We think of the object when it is used in combination with user programs that preserve (i.e., are not the first to violate) this well-formedness condition, and the first requirement on an object implementation is that it also preserve well-formedness.

Property 1: Preserves well-formedness

Also, as before, we make a restriction on when our processes can take steps. Namely, in well-formed executions, process i is only permitted to take steps in between an invocation and a response at i (that is, while an invocation is *active* at i).

The next correctness condition involves the correctness of the responses. This correctness is defined in terms of a related *serial specification.* A serial specification describes the correct responses to a sequence of operation invocations, when they are executed sequentially (i.e, without concurrent invocations). For instance, for a read-write object having initial value 0, the correct sequences include the following (subscripts denote processes/line numbers).

$$read_1, read\text{-}respond_1(0), write_2(8), write\text{-}respond_2, read_1, read\text{-}respond_1(8)$$

Usually, these sequences are specified by a simple state machine in which each operation causes a change of state and a possible return value. We remark that this serial specification is now standard in theory of data types.

So let S be a serial specification for a particular set of operation types (e.g., read, write, etc.). An atomic object corresponding to S has the same interface as S. In addition to the well-formedness condition above, we require something about the contents of the sequences.

Property 2: Atomicity

We want to say that any well-formed execution must "look like" a sequence in the serial specification S. The way of saying this is a little complicated, since we want a condition that also makes sense for executions in which some of the operations don't return. That is, even if for some of the lines i, the final operation gets invoked and does not return, we would like that the execution obey some rules. Specifically, we require, for a (finite or infinite) well-formed execution of the object, that it be possible to select the following.

1. For each completed operation, a *serialization point* within the active interval.

2. An arbitrary subset T of the incomplete operations (i.e., those having an invocation but no response), such that for each operation in T we can select a serialization point sometime after the invocation, and a response action.

The points should be selected in a way such that if we move the invocation and response actions for all the completed operations, and all the operations in T, so that they flank their respective serialization points (i.e., if we "shrink" the completed operations and all the operations in T to contain only their respective serialization points), then the resulting sequence of invocations and responses is in S (i.e., is correct according to the serial specification).

So, the atomicity condition essentially stipulates that an execution looks as if the operations that were completed (and some of the incomplete ones) were performed instantaneously at some time in their intervals. Figure 10.2 shows a few scenarios involving a two-process read-write object.

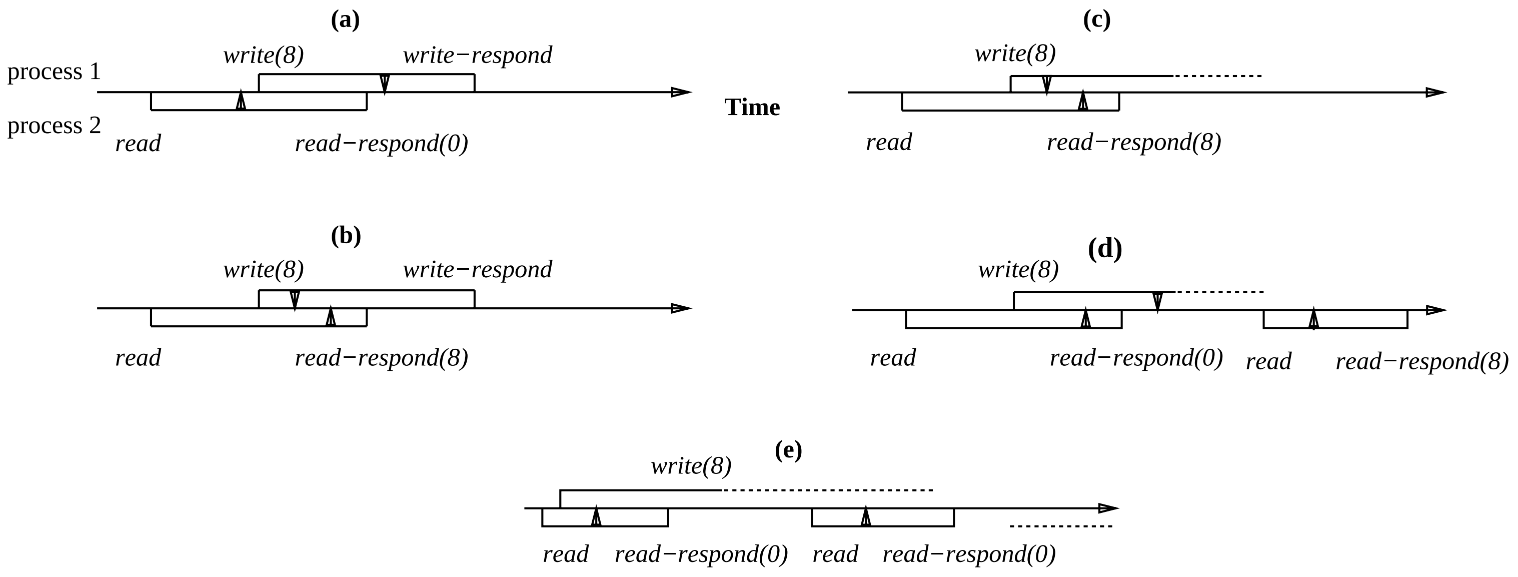


Figure 10.2: possible executions of a read-write object with two processes. The arrows represent serialization points. In the scenario (e), there are infinitely many *read*s that return 0, and consequently the incomplete *write* does not get a serialization point.

Property 3: Liveness

This property is simple: we require that in every well-formed *fair execution*, every invocation receives a response. This rules out, for example, the "dead object" that never returns responses.

Notice that here, we are using the underlying fairness notion for the model — i.e., that processes continue to take steps when those are enabled. In this case, the statement of Property 2 could be simplified. The reason we have given the more complicated statement of Property 2 is that in the sequel, we shall consider non-fair executions of atomic object implementations. This will be when we are discussing resiliency, and wait-freedom.[5] Note that Property 3 (so far) only makes sense in the particular model we are using, with processes to which it makes sense to be "fair".

Discussion. Intuitively, we can say that an atomic object is a "concurrent version" of a corresponding serial specification. It is appropriate to ask what good is this concept. The

[5]In the literature, wait-freedom is sometimes referred to as "wait-freeness".

important feature we get is that we can use an atomic object in modular construction of systems: suppose that we design a system to use instantaneous objects (e.g., read-write, queues, or something more interesting), but we don't actually have such instantaneous objects. If we have atomic versions of them, then we can "plug" those implementations in, and, as we will later see, under certain circumstances, the resulting system "behaves the same", as far as a user can tell. We shall return to this discussion later.

10.2 Atomic Snapshots

We shall now see how can simple atomic objects be used to implement (seemingly) much more powerful objects. Specifically, we'll see how to implement *atomic snapshot object*, using *atomic registers*. We follow the general ideas of [Afek Attiya et al]. We start with a description of the the problem, and then give a simple solution that uses registers of unbounded size. Finally, we refine the construction to work with bounded registers.

10.2.1 Problem Statement

Atomic snapshot object models an entire memory, divided into n *words*. It has n *update* and n *snap* lines (see Figure 10.3). The $update_i(v)$ writes v into $word_i$, and the *snap* returns the vector of all the latest values.

As usual , the atomic version of this object has the same responses as if shrunk to a point in the interval.

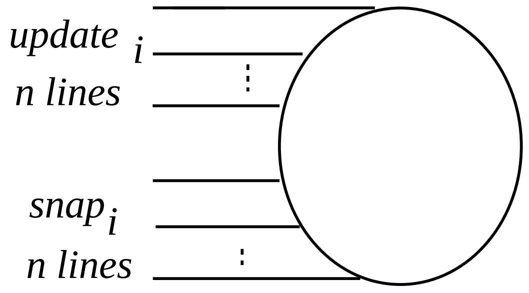


Figure 10.3: schematic interface of an atomic snapshot object

The model of computation we assume is that of 1-writer n-reader atomic registers.

10.2.2 Unbounded Algorithm

Suppose that each *update* writes into its register in a way such that each value can be verified to be new. A simple way to do it is to attach a "sequence number" to each value; whenever a newer value is written, it is written with an incremented sequence number. For this rule, we have the following simple property.

Suppose every update leaves a unique mark in its register. If two consecutive global reads return identical values, then the values returned are consistent, i.e., they constitute a true snapshot.

The observation above immediately gives rise to the following simplistic algorithm: each *update* increments the local sequence number; each *snap* collects repeatedly all the values, until two identical consecutive sets of collected values are seen (a *successful double collect*). This common set is returned by the *snap-respond.*

This algorithm always returns correct answers, but it suffers from the problem that a *snap* may never return, if there are *update*s invoked concurrently.

For this problem, however, we have the following observation (see Figure 10.4).

If a snap sees a value being updated 3 times, then the updater executes a complete update operation within the interval of the snap.

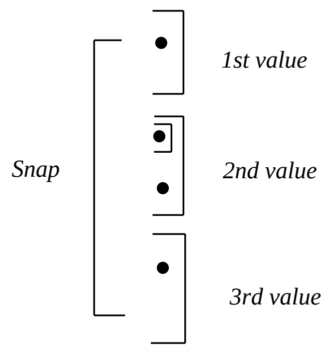


Figure 10.4: a complete *update* must be contained in the interval containing three changes of a single value. Notice the embedded *snap* in the middle *update.*

The observation above leads us to the main idea of the algorithm, which can be informally stated as *If you want to write, then read first!* Specifically, we extend the algorithm such that before an *update* process writes, it must take a *snap* action, and save the result of this internal *snap* somewhere accessible to all others. We shall use the term *embedded snap* for the *snap*s activated by *update*s.

We can now describe the unbounded algorithm. The system structure is depicted in Figure 10.5. First, we describe the state variables.

For each $update_i$ we have a register which is a record with the following fields.

- value v_i

- sequence number s_i
- view (i.e., a vector of values) G_i

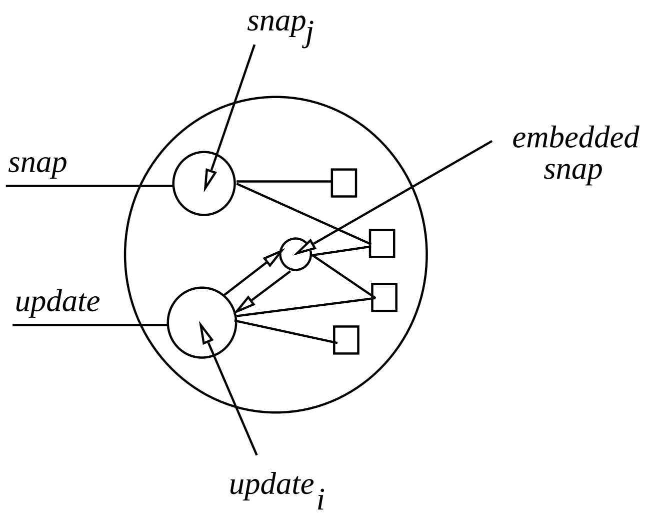


Figure 10.5: system structure for the algorithm

The code for $snap_i$ process is as follows.

```
Read all registers until either
    see 2 equal $s_j$ for all $j$ (“double collect”), or
    see 3 distinct values of $s_k$ for some $k$.
In the first case, return the repeated vector of $v_j$s.
In the second case, return the second $G_k$ for that $k$ (“borrowed view”).
```

The code for $update_i(v)$ process is as follows.

```
Do $snap_i$ (embedded snap procedure).
Write (atomically):
    incremented sequence number in $s_i$,
    $v$ in $v_i$, and
    the value returned by the snap in $G_i$
```

Correctness. The well-formedness condition is clearly maintained. The more interesting parts are serializability and termination.

Theorem 1 *The algorithm satisfies the atomicity condition.*

Proof: We construct explicit serialization points in the operations interval at which the read or write can said to have occurred. Clearly, *update* must be serialized at the point of write.

For the *snap* operation, the situation is a bit more complicated. Consider the sequence of writes that occurs in any execution. At any point in time, there is a unique vector of v_is. Call these "acceptable vectors". We show that only acceptable vectors are obtained by *snap*s. We will construct serialization point for all *snap*s ("real" and embedded) as follows.

First, consider *snap*s that terminate with successful double collects. For these, we can pick any point between the end of first collect and the beginning of second as a serialization point. It is clear that the vector returned by *snap-respond* is exactly the acceptable vector at this point, because there is no change in that interval, by the assumption that any change effects the sequence numbers.

Second, consider *snap*s that terminate with borrowed view. We pick serialization points for these by induction on their order of the completion. Base case is trivial (no such *snap*s). For the inductive step, note that the borrowed view of the kth such *snap* is the result of some other *snap* which is completely contained within the *snap* at hand. By induction, the former was already assigned a point. We choose that same point for the *snap* at hand. This point is in the bigger snap interval since it is in the smaller (see Figure 10.5).

By the induction, we also have that the value returned is exactly the acceptable vector at the serialization point. ∎

Theorem 2 *In a fair execution, each snap and update terminates in $O(n^2)$ memory access steps.*

Proof: For *snap*, it follows from the pigeon-hole principle, that after at most $2n+1$ "collects" of all the values, there must be a process with 3 changes. Since in each collect the *snap* process reads n values, the $O(n^2)$ bound follows. For *update*, we have $O(n^2)$ too, because of embedded *snap* (which is followed only by a single *write*). ∎

10.2.3 Bounded Register Algorithm

The requirement of the previous algorithm to have unbounded registers is problematic. Theoretically, this means that the algorithm needs unbounded space. The point is that the correctness only required that a change is detectable; having a sequence number gives us a total order, which is far more than we need.

If we think of the algorithm a little, we can see that the purpose of the sequence numbers was that *snap* processes could tell when $update_i$ produced a new value. This, however, can be communicated explicitly using *handshake bits*. More specifically, the main idea is as follows. Instead of keeping sequence numbers for $update_i$, keep n pairs of handshake bits for communication between $update_i$ and $snap_j$ for all j (actually, $2n$ — for real and embedded *snaps*). The handshake protocol is similar the Peterson-Fisher 2-process mutual exclusion algorithm: $update_i$ sets the bits to be unequal, and $snap_i$ sets them equal.

We use the following notation for the handshake bits. For each $(update_i, snap_j)$ pair, we have bits p_{ij} at i, and q_{ji} at j. with this notation, the handshake protocol is simply the following.

1. When $update_i$ writes: $p_{ij} \leftarrow \neg q_{ji}$.

2. Before $snap_j$ reads: $q_{ji} \leftarrow p_{ij}$

The handshake protocol is "nearly" sufficient. As we shall see, it can be the case that two consecutive values can be confused. To overcome this problem, each $update_i$ has an additional toggle bit, $toggle_i$, that it flips during each write. This ensures that each write changes the register value.

We can now give a sketch of the code of the bounded algorithm. (The complete algorithm can be found in page 10 of Handout 15.)

```
$update_i$:
 Read the handshake bits $q_{ji}$ for all $j$
 Snap
 Write the value, borrowed view, the negated handshake bits, and negated toggle bit


$snap_j$:
 Repeat
    Read handshake bits $p_{ij}$ and set $q_{ji} \leftarrow p_{ij}$ for all $i$
    Do two collect
    If both collects are equal in all $p_{ij}$ and $toggle_i$, then return common vector
    else record who has moved
 Until someone moved 3 times
 Return borrowed view
```

Correctness. The general idea is the same as for unbounded case. We serialize *update*s by the writes. For *snap*: we can serialize *snap* that return borrowed view using the same induction. Not surprisingly, the problem now is correctness of *snap*s that return by double-collect. The following argument shows that the same serialization rule (i.e., any point between the two collects).

Claim 3 *If a snap does a successful double collect, then no write occurs between the end of the first and the beginning of the second.*

Proof: By contradiction. Suppose two reads by $snap_j$ of r_i produce p_{ij} equal to q_{ji}'s most recent values and toggle bit, and assume a write by i occurs in between the 2 reads. Consider the last such $write_i$ — it must write the same p_{ij} and $toggle_i$ read by $snap_j$. Since during $update_i$, p_{ij} gets $\neg q_{ji}$, then the read must precede $snap_j$'s most recent write of q_{ji}.

Thus we must have the situation depicted in Figure 10.6.

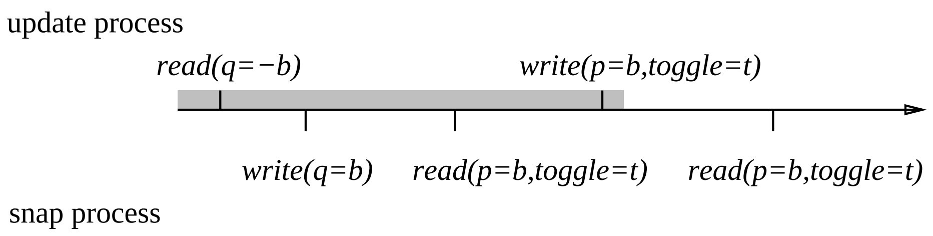


Figure 10.6: scenario described in the proof of Claim 3. The shaded area represent the active interval of the last $update_i$ before the second read of $snap_j$.

The $read_i$ and $write_i$ are part of the same $update_i$, so the two reads by $snap_j$ return values written by the 2 *successive* $update_i$ writes, with identical toggle bits — contradicting the code. ∎

Before we conclude the discussion of the atomic snapshot algorithm, we remark that it has the additional nice property of *wait-freedom*. Namely, in any execution, including those that are not necessarily fair to all processes, but just to some subset subset P, any operation of any $i \in P$ is guaranteed to terminate. Notice that processes not in P may even stop, without affecting the progress of the processes in P. We will discuss this concept extensively in future lectures.

Lecturer: Nancy Lynch

Lecture 11

So far, we have been working within a model based on read-write shared memory, while studying the mutual exclusion and atomic snapshot problems. We have digressed slightly by jumping from the mutual exclusion problem to the atomic snapshot problem. Today, we will return to the mutual exclusion problem once again. In the following lectures, we will consider the *consensus* problem in read-write shared memory, and then consider atomic (and other) objects once again.

11.1 Burns' Mutual Exclusion Algorithm

Both of the algorithms we have studied so far (Dijkstra's and Peterson's) use multi-writer variables (*turn*) along with a collection of single-writer variables (*flag*). Due to the fact that it might be difficult and inefficient to implement (i.e., physically build) multi-writer shared variables in certain systems (in particular, in distributed systems), algorithms that use only single-writer variables are worth investigating. We shall see two single-writer algorithms, by Burns and by Lamport.

The first algorithm, developed by Jim Burns, appears in Figure 11.1. It does not guarantee fairness, but only mutual exclusion and deadlock-freedom. Lamport's algorithm is fair also, but has the disadvantage of using unbounded variables.

Mutual exclusion. The proof that Burns' algorithm guarantees mutual exclusion is similar to the proof for Dijkstra's algorithm, except that the *flag* variable is set to 1 where in Dijkstra's it is set to 2. For example, consider an operational argument. If i and j both reach C, then assume that i set *flag* to 1 first. By the code, it keeps it 1 until it leaves C. Also by the code, after j sets its *flag* to 1, j must check that $flag(i) = 0$ before it can enter C. The argument is completed by showing that at least one of the processes must notice that the other has set it *flag*. Specifically, if $i < j$, then j must return to L, and if $i > j$, then i cannot proceed to the critical region.

Shared variables:

- *flag* : an array indexed by $[1..n]$ of $\{0, 1\}$, initially all 0, where $flag[i]$ is written by p_i and read by all

Code for p_i

```
L:
$flag[i] \leftarrow 0$
for $j \in \{1, \ldots, i-1\}$ do
    if $flag[j] = 1$ then goto L
    end if
end for
$flag[i] \leftarrow 1$
for $j \in \{1, \ldots, i-1\}$ do
    if $flag[j] = 1$ then goto L
    end if
end for
M:
for $j \in \{i+1, \ldots, n\}$ do
    if $flag[j] = 1$ then goto M
    end if
end for
  **Critical region**
$flag[i] \leftarrow 0$
  **Remainder region**
```

Figure 11.1: Burns' Mutual Exclusion Algorithm

Deadlock-freedom. This property can be argued by contradiction as follows. As for Dijkstra's algorithm, assume that there exists some execution in which all processes are in either R or T, and their are no further region changes. Partition the processes into those that ever reach label M and those that do not; call the first set P and the second set Q. Eventually, in this execution, there must exist a point where all processes in P have already reached M. Note that they never thereafter drop back to any point prior to label M. Now we claim that there is at least one process in P — specifically, the one with the lowest index among all the contenders — that will reach P. Let i be the largest index of a process in P. We claim that eventually after this point, any process $j \in Q$ such that $j > i$ has $flag(j)$ set permanently to 0. This is because if its *flag* is 1, it eventually detects the presence of a smaller index active process and returns to L, where it sets its *flag* to 0, and from that point it can never progress far enough to set it back to 1. Finally, we claim that p_i will eventually reach the critical region, a contradiction.

Remark. Burns' algorithm uses no multi-writer variables, but does use n shared (multi-reader) variables to guarantee mutual exclusion and deadlock-freedom for n processes. Later in this lecture we will see that this is optimal in terms of the number of registers, even if unbounded multi-writer registers are available.

11.2 Lamport's Bakery Algorithm

In this section we discuss Lamport's bakery algorithm. It is a very basic and interesting mutual exclusion algorithm; it is practical, and its ideas reappear in many other places. In the presentation here we assume for simplicity that the shared memory consists of (single-writer) read-write shared variables.[6] The algorithm guarantees nice fairness behavior: it features lockout-freedom, a good time bound, and bounded bypass. In fact, it has a stronger property: FIFO after a wait-free "doorway" (to be defined below). An unattractive property it has, however, is that it uses unbounded size registers. The code is given in Figure 11.2. We remark that the code given here can be simplified in the case of indivisible read-write registers. The given code works also for the safe registers model.

In the algorithm, the trying region is broken up into two subregions, which we call the *doorway*, and the rest of T. The doorway is the part from when the process enters until it sets $choosing[i]$ to 0. In the doorway, the process chooses a *number* that is greater than the numbers that it sees that other processes have already chosen. While it does this, it sets $choosing[i]$ to 1, to let the other processes know that it is currently choosing a number. Note

[6]The algorithm works also under a weaker model, called *safe registers*, in which the registers have much less predictable behavior in the presence of concurrent accesses. We discuss this model later in the course.

Shared variables:

- *choosing* : an array indexed by $[1..n]$ of integers from $\{0,1\}$, initially all 0, where $choosing[i]$ is written by p_i and read by all
- *number* : an array indexed by $[1..n]$ of integers from $\{0,1,\ldots\}$, initially all 0, where $number[i]$ is written by p_i and read by all

Code for p_i

```
  ** beginning of doorway **
$try_i$
L1:
$choosing[i] \leftarrow 1$
$number[i] \leftarrow 1 + \max\{number[1], \ldots, number[n]\}$
$choosing[i] \leftarrow 0$
  ** end of doorway **
  ** beginning of bakery**
for $j \in \{1, \ldots, n\}$ do
    L2:
    if $choosing[j] = 1$ then goto L2
    end if
    L3:
    if $number[j] \neq 0$ and $(number[j], j) < (number[i], i)$ then goto L3
    end if
end for
$crit_i$
  ** critical region**
  ** end of bakery **
$exit_i$
$number[i] \leftarrow 0$
$rem_i$
  **Remainder region**
```

Figure 11.2: Lamport's bakery mutual exclusion Algorithm

that it is possible for two processes to be in the doorway concurrently, and thus two processes may obtain the same number. To overcome this problem, the comparison is done between $(number, index)$ pairs lexicographically, and thus ties are broken in favor of the process with the smaller index (this is really an arbitrary rule, but it is commonly used). In the rest of the trying region, the process waits for its $(number, index)$ pair to be the lowest, and is also waiting for any processes that are choosing.[7]

The algorithm resembles the operation of a bakery: customers enter the doorway, where they choose a number, then exit the doorway and wait in the store until their number is the smallest.

11.2.1 Analysis

Basic Properties

Let D denote the doorway; $T - D$ is then the rest of the trying region. We first argue that the algorithm does not violate the mutual exclusion property.

Claim 1 *In any reachable state of the algorithm, if $p_i \in C$ and for some $j \neq i$ we have $p_j \in (T - D) \cup C$, then $(number[i], i) < (number[j], j)$.*

We give here an operational proof, since it can be extended more easily to the safe register case later.

Proof: Process i had to read $choosing[j] = 0$ in $L2$ before entering C. Thus, at the time of that read, j was not in the "choosing region" (i.e., in the doorway after setting $choosing[j]$ to 1). But since j is in $(T - D) \cup C$, j must have gone through the doorway at some point. There are two cases to consider.

Case 1: j entered the choosing region after process i read that $choosing[j] = 0$. Then i's number was chosen before j started choosing, ensuring that j saw $number[i]$ when it chose. Therefore, when i is in C, we have $number[j] > number[i]$, which suffices.

Case 2: j left the choosing region before i's read. Then when i reaches $L3$ and reads j's number, it gets the most recent value $number[j]$. Since i decided to enter C anyhow, it must be that $(number[i], i) < (number[j], j)$. ∎

Corollary 2 *The bakery algorithm satisfies mutual exclusion.*

Proof: Suppose that two processes, i and j, are both in C. Then by Claim 1, we must have both $(number[i], i) < (number[j], j)$ and $(number[j], j) < (number[i], i)$, a contradiction. ∎

[7] Here, we are using the processes indices in much the same way as we previously used the UID's. If the processes did not know their indices, but had UID's, we could use those here instead. This observation holds also for Dijkstra's algorithm, but not for Peterson's, since there, the indices distinguish the roles the processes play.

Theorem 3 *The algorithm satisfies deadlock-freedom.*

Proof: Again, we argue by contradiction. Suppose that deadlock occurs: then eventually a point is reached, after which a fixed set of processes are in T, and no new region changes occur. Also, by the code, eventually all of the processes in T get out of the doorway and into $T - D$. At this time point, the process with the lowest $(number, index)$ is not blocked. ▮

Theorem 4 *The algorithm satisfies lockout-freedom.*

Proof: Consider a particular process $i \in T$. It eventually exits the doorway. Thereafter, any new process that enters the doorway sees i's number, and hence chooses a higher number. Thus, if i doesn't reach C, none of these new processes can reach C either, since they all see that i has a smaller number, and by the code, they must wait "behind" i. But by Theorem 3, processes in T must continue to go to C, which means that i eventually goes. ▮

Time Analysis

In this section we show that any process enters C at most $O(n^2)$ time units after it starts executing T. We start by showing that the algorithm has bounded bypass after the doorway.

Lemma 5 *The total number of times a processor p_i enters the bakery while some processor p_j is continuously in the bakery is no more than 2.*

Proof: It suffices to show that any processor may exit the bakery at most once while p_j is in the bakery. While j is in the bakery, $number(j)$ is unchanged. Hence, if a process i exits the bakery, then when it enters the doorway again, $number(i)$ will be assigned a number strictly more than $number(i)$, and p_i will not be able to proceed from L2 from some state in the fragment until p_j exits C. ▮

We now analyze the the time for an individual process to enter C. Consider an execution fragment such that in the first state, some process p_t is in L1, and in the last state p_t is in C. Our goal is to bound the time of any such execution fragment. Denote by $T_d(i)$ the time a processor p_i spends in the doorway, and by $T_b(i)$ the time p_i spends in the bakery (we consider only the fragment in question). We shall bound $T_d(t) + T_b(t)$. We first bound the time spent in the doorway.

Lemma 6 *For any processor p_i, $T_d(i) = O(sn)$.*

Proof: The doorway consists of n reads and 3 writes. ▮

By the above lemma, after $O(sn)$ time, the process p_t reaches the bakery. We now consider only the suffix of the fragment, in which p_t is in the bakery. Consider the **if** statements. They can be described as "busy waiting" until some predicate holds. We call a test *successful* if the next step is *not* another execution of the test (specifically, if the predicate in L2 and L3 was false). The following lemma bounds the number of successful tests in the fragment we consider.

Lemma 7 *The total number of successful tests in the fragment is no more than $O(n^2)$.*

Proof: In all the Trying Region every processor makes $O(n)$ successful tests, and by Lemma 5, any processor passes the Trying Region no more than twice. ∎

We can now bound the total time of the fragment.

Theorem 8 *The total time of the execution fragment in question is no more than $T_d(t) + T_b(t) = O(sn^2) + O(cn)$.*

Proof: We know that in all states during the execution fragment, some processor can make progress (or otherwise the liveness property is contradicted). But this implies that either some process is making progress in the doorway, or that some process makes a successful comparison, or that some process makes progress in the C. The total amount of time in the fragment during which *some* processor in the C is $O(cn)$, since by Lemma 5, any processor enters the C at most once. Also, the total amount of time in the fragment during which *any* processor is in the doorway is $O(sn^2)$, by Lemmas 5 and 6. Taking into account the "successful test" steps from Lemma 7 we can conclude that

$$T_b(t) \leq O(sn^2) + O(sn^2) + O(cn) = O(sn^2) + O(cn) \ .$$

The result is obtained when combining the above bound with Lemma 6. ∎

Additional Properties

The algorithm has some other desirable properties. For example, we have *FIFO after a doorway.* This stronger fairness property says that if i finishes the doorway before j enters T, then i must precede j in entering C. This is an almost-FIFO property: it is not actually FIFO based on time of entry to T, or even time of first non-input step in T (i could set its *choosing* variable to 1, then delay while others choose, so all the others could beat it).

It isn't very useful to say (as a feature) that an algorithm is FIFO after a doorway, since so far there are no constraints on where the doorway begins. We might even place the doorway right at the entrance to C! But the doorway in this algorithm has a nice property: it's *wait-free*, which means that a process is guaranteed eventually to complete it, if that process continues to take steps, regardless of what the other processes do (continue taking steps or stop, in any combination, any speed).

The bakery algorithm has also some limited amount of fault-tolerance. Consider faults in which a user can "abort" an execution at any time when it is not in R, by sending an $abort_i$ input to its process. Process i handles this by setting its variables, local and shared, back to initial values. A lag is allowed in resetting the shared variables. We can see that mutual exclusion is still preserved. Moreover, we still have some kind of deadlock-freedom: suppose that at some point some process is in T and does not fail (abort), and that there are only finitely many total failures, and that C is empty. Then the algorithm guarantees that eventually some process enters C (even if all other processes fail).

11.3 The Number of Registers for Mutual Exclusion

We have seen several algorithms using read-write shared memory, for mutual exclusion, with deadlock-freedom, and also various fairness properties. The question arises as to whether the problem can be solved with fewer than n read-write variables.

If the variables are constrained to be single-writer, then the following lemma implies that at least n variables are needed.

Claim 9 *Suppose that an algorithm A solves mutual exclusion with deadlock-freedom for $n \geq 2$ processes, using only read-write shared variables. Suppose that s is a reachable state of A in which all processes are in the remainder region. Then any process i can go critical on its own starting from s, and along the way, i must write some shared variable.*

Proof: Deadlock-freedom implies that i can go critical on its own from s. Suppose that it does not write any shared variable. Then consider any other process i'; deadlock-freedom also implies that i' can go critical on its own, starting from s. Now consider a third execution, in which process i first behaves as it does in its solo execution, eventually entering C without writing any shared variable. Since process i does not write any shared variable, the resulting state s' "looks like" state s to process i'. (Here, we say that two states *look alike* to some process if the state of that process and the values of all shared variables are the same in the two states.) Therefore, i' is also able to go critical on its own starting from s'. This violates the mutual exclusion requirement. ∎

Claim 9 directly implies that if only single-writer registers are available, then the algorithm must use at least n of them. But note that we have not even beaten this bound when we used multi-writer variables. In this section we prove that this is not an accident. In fact, we have the following lower bound.

Theorem 10 *If algorithm A solves mutual exclusion with deadlock-freedom for $n \geq 2$ processes, using only read-write variables, then A must use at least n shared variables.*

Note that this result holds regardless of the size of the shared variables: they can be as small as a single bit, or even unbounded in size. Also note that no fairness assumption is needed; deadlock-freedom is sufficient to get the impossibility result.

To get some intuition, we start with two simple cases. We first consider the case of one variable and two processes; then we consider the case for two variables and three processes, and finally we shall extend the ideas to the general case.

We shall use the following definition extensively.

Definition 1 *Suppose that at some global state s, the action enabled in some process p is writing to a variable v (that is, the next time p takes a step, it writes to v). In this case we say that p* covers *v.*

11.3.1 Two Processes and One Variable

Suppose that the system consists of two processes p_1 and p_2, and only 1 variable. We construct a run that violates mutual exclusion for this system. Claim 9 implies that there is a solo run of p_1 that causes it to enter C, and to write the single shared variable v before doing so. So now run p_1 just until the first time it is about to write to v, i.e., until the first time it covers v. We then run p_2 until it goes to C (since p_1 hasn't written anything yet, for p_2 the situation is indistinguishable from the state in which p_1 is in R, and hence p_2 will behave as it does running solo). Then let p_1 continue running. Now, the first thing p_1 does is to write v, thereby overwriting anything that p_2 wrote and so eliminating all traces of p_2's execution. Thus, p_1 will run as if alone, and also go to C, contradicting the mutual exclusion requirement.

11.3.2 Three processes and Two Variables

Now suppose that we have three processes p_1, p_2, p_3 and two variables v_1, v_2. Again, we shall construct a run that violates mutual exclusion, using the following strategy. Starting from an initial state, we will maneuver p_1 and p_2 alone so that each is covering a different variable; moreover, the resulting state, s, is indistinguishable to p_3 from another reachable state, s', in which all three processes are in R. Then we run p_3 from s, and it must eventually go to C, since it can't tell that anyone else is there. Then we let p_1 and p_2 each take a step. Since they are covering the two variables, the first thing they do is to overwrite all traces of p_3's execution, and so they will run as if they are alone, and by deadlock-freedom, one will eventually go to C, yielding the desired violation of mutual exclusion.

It remains to show how we maneuver p_1 and p_2 to cover the two shared variables. We do this as follows (see Figure 11.3).

First, we run p_1 alone until it covers a shared variable for the first time. Call this point t_1. Then, we run p_1 alone until it enters C, then continues to E, R, back to T, and again covers some variable for the first time. Call this point t_2. We repeat this procedure to obtain a third point t_3. Notice that two of the three points t_1, t_2 an t_3 must involve covering the

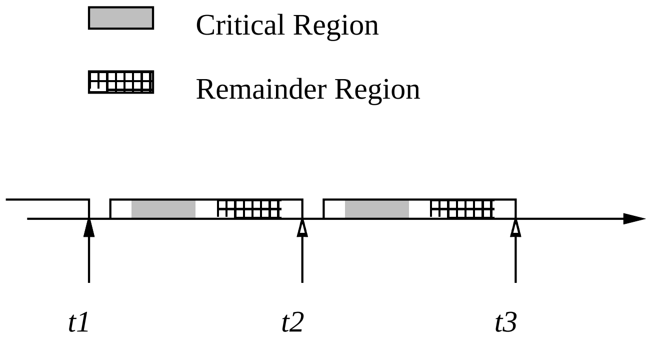


Figure 11.3: solo run of p_1. It covers a variable at each of t_1, t_2, and t_3, and hence it covers the same variable twice.

same variable (see Figure 11.3). Without loss of generality, suppose that in t_1 and t_3 p_1 covers v_1 (the same argument holds for all the other cases).

Now consider the following execution. Run p_1 until point t_1. Then let p_2 enter and run alone. We claim that eventually p_2 enters C, since the state looks to p_2 as if it is in the solo execution. Moreover, we claim that along the way p_2 must write the other variable, call it v_2. For otherwise, p_2 could go to C, then p_1 could immediately write v_1 and thus overwrite all traces of p_2, then go on and violate mutual exclusion.

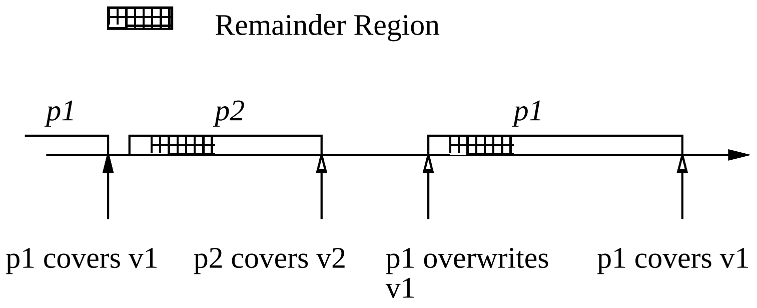


Figure 11.4: Execution for 2 variables. By the end of this fragment, p_1 and p_2 cover both shared variables, and the state is indistinguishable from the state in which p_1 and p_2 are in their last remainder state.

So now we construct the required execution as follows (see Figure 11.4.) Run p_1 until point t_1, when it covers v_1. Then run p_2 until it first covers v_2. Note that we are not done yet, because p_2 could have written v_1 since last leaving R. But now, we can resume p_1 until point t_3. The first thing it does is write v_1, thereby overwriting anything p_2 wrote. Then p_1 and p_2 cover variables v_1 and v_2, respectively. Moreover, by stopping p_1 and p_2 back in their most recent remainder regions, the shared memory and state of p_3 would still be the same. This completes the construction of the execution in which p_1 and p_2 cover the two shared variables, in a state that is indistinguishable for p_3 from the state in which both are in R. By the argument above, we conclude that in this case, mutual exclusion can be violated.

11.3.3 The General Case

The proof for the general case extends the examples above with an inductive argument. We call a state a *global remainder state* if all processes are in R in that state. We call a state *k-reachable* from another if it is reachable using steps of processes $p_1, \ldots, p_k$ only.

We start with two preliminary lemmas.

Lemma 11 *Suppose A solves mutual exclusion with deadlock-freedom for $n \geq 2$ processes, using only read-write variables. Let s and s' be reachable states of A, that are indistinguishable to p_i, and suppose that s' is a global remainder state. Then p_i can go, on its own, from s to its critical region.*

Proof: Process p_i can do so in s', by deadlock-freedom. Since s looks like s' to p_i, it can behave in the same way from s. ∎

Lemma 12 *Suppose A solves mutual exclusion with deadlock-freedom for $n \geq 2$ processes, using only read-write variables. Let s be a reachable state of A. Suppose that i goes from R to C on its own starting from s. Then along the way, i must write to some variable that is not covered in s.*

Proof: If not, then after p_i enters, can resume the others, who overwrite and hide it. ∎

Using the above easy properties, we shall prove the following main lemma.

Lemma 13 *Let A be an n process algorithm solving mutual exclusion with deadlock-freedom, using only read-write variables. Let s_0 be any reachable global remainder state. Suppose $1 \leq k \leq n$. Then there are two states s and s', each k-reachable from s_0, such that the following properties hold.*

1. *k distinct variables are covered by $p_1, \ldots, p_k$ in s.*
2. *s' is a global remainder state.*
3. *s looks like s' to $p_{k+1}, \ldots, p_n$.*

Notice that applying this lemma for $k = n$ yields the theorem.

Proof: By induction on k.

Base case: For $k = 1$, we have the first example above. To get s, just run p_1 till it covers a variable. In this case, we define $s' = s_0$.

Inductive step: suppose the lemma holds for k; we prove it holds for $k + 1 \leq n$. Starting from s_0, by induction, we run $p_1, \ldots, p_k$ until they reach a point where they cover k distinct variables, yet the state looks to $p_{k+1}, \ldots, p_n$ like some global remainder state that is k-reachable from s_0. Then, we let $p_1, \ldots, p_k$ write in turn, overwriting the k variables, and then let them progress to C, E, R, calling the resulting state s^1. We apply the inductive hypothesis again to reach another covering state s^2 that looks like a global remainder state k-reachable from s^1. We repeat this procedure $\binom{n}{k} + 1$ times. Now, by the Pigeonhole Principle, among the $\binom{n}{k} + 1$ covering states that have been obtained, there must be two that *cover the same set of variables.* Let t_1 be the first of these points in the execution, t_2 the second. Let s_1 and s_1' be the covering and global remainder states corresponding to t_1, and likewise s_2 and s_2' for t_2.

Consider running p_{k+1} alone from t_1. Since the state at t_1 looks to p_{k+1} like a reachable global remainder state, Lemma 11 implies that p_{k+1} will eventually enter C. Along the way, by Lemma 12, it must write to some variable not in V.

Now we construct the needed execution to prove the lemma as follows (see Figure 11.5). First, run $p_1, \ldots, p_k$ until t_1; then let p_{k+1} take steps until it first covers a variable not in V. Next, let $p_1, \ldots, p_k$ resume and go to t_2.

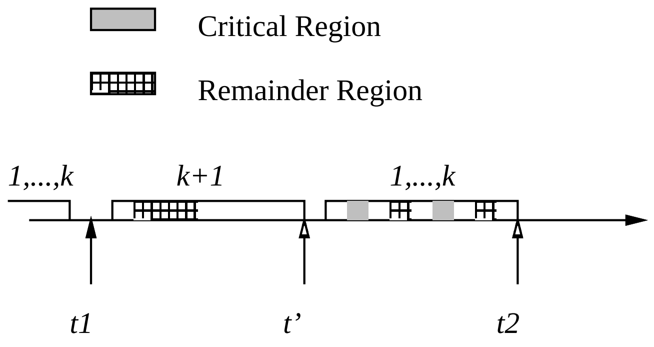


Figure 11.5: construction for the general case. In t_1, processes $p_1 \ldots p_k$ cover k variables of V. In t', p_{k+1} covers some variable not in V. In t_2, V and that variable are covered.

Call the resulting state s; this is the s that is required in the statement of the lemma. (This state s is the same as s_2, with the state of p_{k+1} changed to what it is when it stops.) Let $s' = s_2'$.

We now show that the required properties hold. First, note that the entire construction (including the uses of the inductive hypothesis), only involves running $p_1, \ldots, p_{k+1}$, so both s and s' are $k+1$-reachable from s_0. Also, directly from the construction, we have $k+1$ variables covered in s: the k variables in V, and a new one covered by p_{k+1}. By inductive hypothesis, $s_2' = s'$ is a global remainder state.

It remains only to show that s and s' look alike to $p_{k+2}, \ldots, p_n$. But this follows from the facts that s_2 and s_2' look alike to $p_{k+1}, \ldots, p_n$, and that s_2 and s look alike to all processes except p_{k+1}. ∎

Lecturer: Nancy Lynch

Lecture 12

12.1 Consensus Using Read-Write Shared Memory

In this section, we focus on the consensus problem in the shared memory setting. We shall work from papers by Herlihy; Loui, Abu-Amara; and Fischer, Lynch, Paterson. We first define the problem in this context. The architecture is the same architecture we considered in the previous lectures, namely, read-write shared variables (allowing multi-reader, multi-writer variables). Here, we assume that the external interface consists of input action $init_i(v)$, where v is the input value, and output action $decide_i(v)$, where v is the decision value. Just to keep consistent with the invocation-response style we are using for mutual exclusion and atomic objects, we assume that the initial values arrive from the outside in input actions.

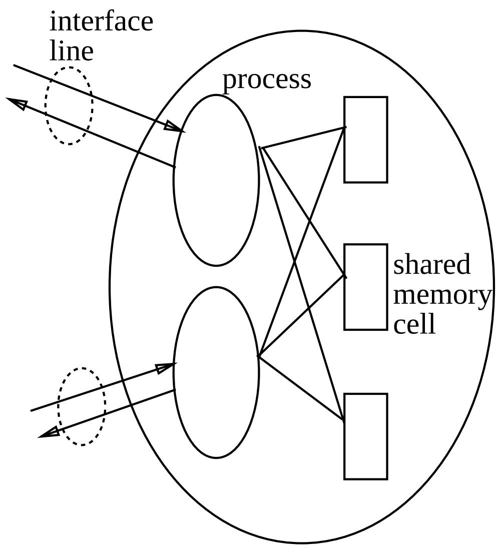


Figure 12.1: shared memory system

We require the following properties from any execution, fair or not:

Agreement: All decision values are identical.

Validity: If all processes start with the same value v, then v is the only possible decision.

In addition, we need some kind of termination condition. The simplest would be the following.

Termination: If *init* events occur at all nodes and the execution is fair, then all processes eventually decide.

This condition (which applies only when there are no faults), has simple solutions, even though there is asynchrony. So it is only interesting to consider the faulty case. In the following requirement, we give the formulation of the requirement for stopping faults.

Termination: Suppose *init* events occur at all processes, and let i be any process. If i does not fail (i.e., the execution is fair to i) then eventually a $decide_i$ event occurs.

We remark that this condition is similar to *wait-freedom*.

12.1.1 Impossibility for Arbitrary Stopping Faults

Our first result in this section is the following.

Theorem 1 *There is no algorithm for consensus that tolerates stopping faults.*

Proof: Assume that A is a solution. Suppose for simplicity that the values in A are chosen from $\{0, 1\}$. Without loss of generality, we can assume that the state-transition relation of A is "process-deterministic", i.e., from any global state, if a process i is enabled to take a non-input step, then there is a unique new global state that can result from its doing so. Also, for any global state and any input, there is a unique resulting global state. This does not restrict the generality since we could just prune out some of the transitions; the problem still has to be solved in the pruned algorithm.

We can further restrict attention to *input-first executions* of A, in which inputs arrive everywhere before anything else happens, in order of process index. That is, they have a prefix of the form $init_1(v_1), init_2(v_2), \ldots init_n(v_n)$. This is just a subset of the allowed executions. Now consider any finite input-first execution α of A. We say that α is 0-*valent* if the only value v that ever appears in a $decide(v)$ event in α or any continuation of α is 0; analogously, we say that α is 1-*valent* if the only such value v is 1. We say that α is *univalent* if it is either 0-valent or 1-valent, and *bivalent* if both values appear in some extensions.

Note that this classification is exhaustive, because any such α can be extended to a fair (i.e., failure-free) execution of A, in which everyone is required to eventually decide.

Define an *initial execution* of A to be an input-first execution of length exactly n. That is, it consists of exactly n $init_i$ actions, one for each i (in order of process index).

Lemma 2 *There exists an initial execution of A that is bivalent.*

Proof: If not, then all initial executions are univalent. Note that the one in which all start with 0 is 0-valent, and analogously for 1. Now consider changing one 0 at a time to a 1. This creates a chain of initial executions, where any two consecutive initial executions in this chain only differ in one process' input. Since every initial execution in the chain is univalent, by assumption, there must be two consecutive initial executions, α and α', such that α is 0-valent and α' is 1-valent. Without loss of generality, suppose that α and α' only differ in the initial value of process i. Now consider any extension of α in which i fails right at the beginning, and never takes a step. The rest of the processes must eventually decide, by the termination condition. Since α is 0-valent, this decision must be 0. Now run the same extension after α', still having i fail at the beginning. Since i fails at the beginning, and α and α' are the same except for process i, the other processes will behave in the same way, and decide 0 in this case as well. But that contradicts the assumption that α' is 1-valent. ▮

Remark. Note, for later reference, that this lemma still holds for a more constrained assumption about A: that it experiences at most a single fault. Technically, the lemma holds under the same conditions as above, with the additional constraint in the termination condition that there is at most one failure.

Next, we define a *decider* to be an input-first execution that is bivalent, but any one-step extension of it is univalent. We have the following lemma.

Lemma 3 *A has a decider.*

Proof: Suppose not. Then any bivalent input-first execution has a bivalent one-step extension. Then starting with a bivalent initial execution (as guaranteed by Lemma 2), we will produce an infinite (input-first) execution all of whose prefixes are bivalent. The construction is extremely simple — at each stage, we start with a bivalent initial execution, and we extend it by one step to another bivalent configuration. Since any bivalent input-first execution has a bivalent one-step extension, we know we can do this. But this is a contradiction because some process does not fail in this execution, so a decision is supposed to be reached by that process. ▮

Remark. Note that this lemma does require the full resiliency: assuming resiliency to a single fault is not sufficient to guarantee the existence of a decider.

Now we can complete the proof of the theorem as follows. Define $extension(\alpha, i)$ to be the execution obtained by extending execution α by a single step of i. By Lemma 3, we obtain a decider α.

Suppose that p_i's step leads to a 0-valent execution, and p_j's step leads to a 1-valent execution (see Figure 12.2). That is, $extension(\alpha, i)$ is 0-valent and $extension(\alpha, j)$ is 1-valent. Clearly, $i \neq j$. (Note that by "process determinism" assumption, each process can

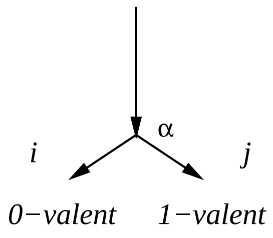


Figure 12.2: α is a decider. If i takes a step, then the resulting configuration is 0-valent, and if j takes a step, the resulting configuration is 1-valent.

only lead to a unique extension of α.)

We now proceed by case analysis, and get a contradiction for each possibility.

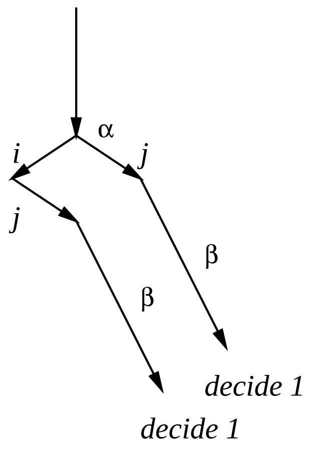


Figure 12.3: The extension β does not include steps of i, and therefore must result in decision value 1 in both cases.

Case 1: p_i's step is a read step. Consider extending $extension(\alpha, j)$ in such a way that all processes *except for* p_i continue to take steps. Eventually, they must decide 1. Now take the same extension, beginning with the step of p_j, and run it after $extension(\alpha, i)$ (see Figure 12.3). Since p_i's step is just a read, it does not leave any trace that would cause the other processes to behave any differently. So in this case also, they all decide 1 contradicting the assumption that $extension(\alpha, i)$ is 0-valent.

Case 2: p_j's step is a read step. This case is symmetric to case 1, and the same argument applies.

Case 3: p_i's and p_j's steps are both writes. We distinguish between the following sub-cases.

Case 3a: p_i and p_j write to different variables. In this case, the result of running either p_i and then p_j, or p_j and then p_i, after α, is exactly the same global state (see Figure 12.4).

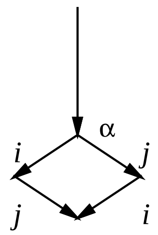


Figure 12.4: If i and j write to different variables, then applying j after i results in the same configuration as in applying i after j.

But then we have a common global state that can follow either a 0-valent or a 1-valent execution. If we run all the processes from this state, they must reach a decision, but either decision will yield a contradiction. For instance, if a decision of 0 is reached, we have a decision of 0 in an execution extending a 1-valent prefix.

Case 3b: They are writes to the same variable. As in Figure 12.3, we can run all but p_i until they decide 1. On the other hand, we can apply that extension also after $extension(\alpha, i)$ and obtain the same decision, because the very first step of the extension will overwrite the value written by p_i, and reach decision value 1 from a 0-valent state, a contradiction as in Case 1.

In summary, we have contradictions in all possible cases, and thus we conclude that no such algorithm A can exist. ∎

12.1.2 Impossibility for a Single Stopping Fault

We can strengthen Theorem 1 to show impossibility of even 1-resilient consensus algorithm. The stronger version is due to Loui, Abu-Amara, following the Fischer, Lynch, Paterson proof for message-passing systems.

Theorem 4 *There is no algorithm that solves consensus under the conditions above in the presence of a single stopping fault.*

Proof: As noted above, Lemma 2 holds also for the case of one stopping fault, and hence we are still guaranteed that there exists a bivalent initial execution. However, our proof will now proceed using a different argument. Specifically, we now show the following lemma.

Lemma 5 *For every bivalent input-first execution α, and every process i, there is an extension α' of α such that $extension(\alpha', i)$ is bivalent.*

Before we turn to prove the lemma, we show how it implies the theorem. Lemma 5 allows us to construct the following bad execution, in which no one fails, and yet no one ever decides: We start with the given initial bivalent execution, and repeatedly extend it, including at least one step of process 1 in the first extension, then at least one step of 2 in the second extension, etc., in round-robin order, while keeping all prefixes bivalent. The key step is the extension for any particular process, but that is done by applying the lemma. In the resulting execution, each process takes many steps, yet no process ever reaches a decision.

So it remains only to prove the lemma.

Proof: (of Lemma 5)
By contradiction: suppose the lemma is false. Let us say what this means: we assume that there exist some bivalent input-first execution α and some process i, such that for every extension α' of α, $extension(\alpha', i)$ is univalent. This, in particular, implies that $extension(\alpha, i)$ is univalent; suppose without loss of generality that it is 0-valent. Since α is bivalent, there is some extension α'' of α leading to a decision of 1. Now, we must have $extension(\alpha'', i)$ 1-valent. Consider applying i at all points along the path from α to α''. At the beginning, we get 0-valence, and at the end, 1-valence. At each intermediate step, we have univalence. Therefore, it must be that there are two consecutive steps at which i is applied, in the first of which it yields a 0-valent extension and in the second a 1-valent extension. See Figure 12.5.

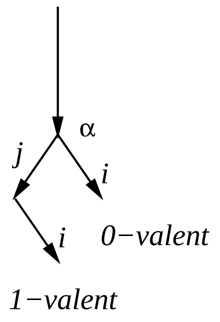


Figure 12.5: whenever we extend α with i, we get a 0-valent configuration.

Suppose that j is the process that takes the intervening step. We claim $j \neq i$. This is true because if $j = i$, we have one step of i leading to 0-valence while two steps lead to 1-valence, contradicting the process-determinism. So we must have somewhere the configuration depicted in Figure 12.6.

We again proceed by case analysis, and obtain a contradiction for each case.

Case 1: i's step is a read. In this case, after ji and ij, the values in all shared variables and the states of all processes other than i, are identical. So from either of these states, run all processes except i, and they will have to decide. But one of these positions is 0-valent and the other 1-valent, so we get a contradiction for any decision value.

Case 2: j's step is a read. This case is similar to Case 1. After i and ji, the values of all shared variables and the states of all processes other than j are identical, and therefore,

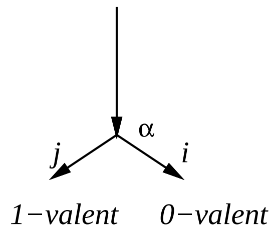


Figure 12.6: the point in the execution which we analyze by cases.

when we let all processes except j run from both states, we obtain the same contradiction as above.

Case 3: Both steps are writes.

Case 3a: The writes are to different variables. In this case we get a commutative scenario depicted in Figure 12.4, which immediately implies a contradiction.

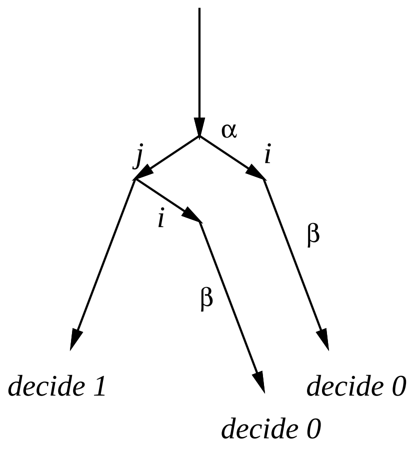


Figure 12.7: case *3b*. Process j does not take steps in β.

Case 3b: The writes are to the same variable. In this case, in a ji extension, the i step overwrites the j step (see Figure 12.7). Thus, after i and ji, the values of all shared variables and the states of all processes other than j are identical, and we obtain a contradiction as before. ∎

This completes the proof of Theorem 4. ∎

12.2 Modeling and Modularity for Shared Memory Systems

In this section we go back to considering the idea of atomic objects, and show how they might be used to describe modular system designs. An atomic object is a concurrent "version" of a corresponding serial specification. We want to prove a theorem that says, roughly speaking, that any algorithm written for the "instantaneous access" model we have already been using can also run correctly in a variant of the model where all the shared variables are replaced by corresponding atomic objects. This would allow us to decompose the task of solving a problem into the task of solving it, assuming some kind of instantaneous access memory, then separately implement atomic objects of the corresponding kind. This could be done in a series of stages, if the problem being solved is itself that of building a (more powerful) atomic object.

To make all of this precise, we should be a little more formal about the model we are using.

Recall the model we have been assuming — state machines with input, output, local computation and shared memory actions, where the latter have transitions of the form $((s, m), \pi, (s', m'))$. So far, all accesses to shared memory have been by instantaneous actions.

Now we would like to define a slightly different version of the model, where instead of sharing memory with instantaneous actions, the processes communicate with separate *objects*. The communication is no longer instantaneous, but involves separate invocations and responses. We must say what kind of fairness we want here too.

It is helpful at this point to introduce a very general, simple and precise formal model for asynchronous concurrent systems: *I/O Automata.* This model is useful as a general foundation for the various kinds of asynchronous systems we will be studying. By adding particular extra structure, it can be used to describe the "instantaneous access" shared memory systems we have been studying. It is also very natural for describing non-instantaneous access shared memory systems, as well as message-passing systems, dataflow systems, etc. Virtually, just about any asynchronous model.

12.2.1 The Basic Input/Output Automaton Model

In this section we give the basic definitions for the I/O Automaton model for asynchronous concurrent computation. For the moment, these definitions do not include any notion of fairness. In the subsequent sections, we define composition of I/O automata and show that it has the nice properties that it should. Then we introduce the notion of fairness and show

how it interacts with the composition notions. We then describe some useful conventions for stating problems to be solved by I/O automata, in terms of their external behavior, and describe what it means for an I/O automaton to solve a problem. Finally, we describe two important proof techniques for verifying systems described as I/O automata – based on modular decomposition and on hierarchical decomposition. These notes are adapted from the Lynch, Tuttle CWI paper.

Actions and Action Signatures

We assume a universal set of *actions*. Sequences of actions are used in this world, for describing the behavior of modules in concurrent systems. Since the same action may occur several times in a sequence, it is convenient to distinguish the different occurrences. We refer to a particular occurrence of an action in a sequence as an *event*.

The actions of each automaton are classified as either 'input', 'output', or 'internal'. The distinctions are that input actions are not under the automaton's control, output actions are under the automaton's control and externally observable, and internal actions are under the automaton's control but not externally observable. In order to describe this classification, each automaton comes equipped with an 'action signature'.

An *action signature* S is an ordered triple consisting of three disjoint sets of actions. We write $in(S)$, $out(S)$ and $int(S)$ for the three components of S, and refer to the actions in the three sets as the *input actions*, *output actions* and *internal actions* of S, respectively. We let $ext(S) = \in(S) \cup out(S)$ and refer to the actions in $ext(S)$ as the *external actions* of S. Also, we let $local(S) = out(S) \cup int(S)$, and refer to the actions in $local(S)$ as the *locally-controlled actions* of S. Finally, we let $acts(S) = \in S \cup out(S) \cup int(S)$, and refer to the actions in $acts(S)$ as the *actions* of S. An *external action signature* is an action signature consisting entirely of external actions, that is, having no internal actions. If S is an action signature, then the *external action signature* of S is the action signature $extsig(S) = (\in S, out(S), \phi)$, i.e., the action signature that is obtained from S by removing the internal actions.

Input/Output Automata

Now we are ready to define the basic component of our model. An *input/output automaton* A (also called an *I/O automaton* or simply an *automaton*) consists of five components:

- an action signature *sig(A)*,
- a set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of *start states*,

- a transition relation $trans(()A) \subseteq states(A) \times acts(sig(A)) \times states(A)$, with the property that for every state s and input action π there is a transition (s, π, s') in $trans(()A)$, and

- an equivalence relation $part(A)$ on $local(sig(A))$, having at most countably many equivalence classes.

We refer to an element (s, π, s') of $trans(A)$ as a *transition*, or *step* of A. The transition (s, π, s') is called an *input transition* of A if π is an input action. *Output transitions*, *internal transitions*, *external transitions* and *locally-controlled transitions* are defined analogously. If (s, π, s') is a transition of A, then π is said to be *enabled* in s. Since every input action is enabled in every state, automata are said to be *input-enabled.* The input-enabling property means that the automaton is not able to block input actions. The partition $part(A)$ is what was described in the introduction as an abstract description of the 'components' of the automaton. We shall use it to define fairness later.

An *execution fragment* of A is a finite sequence $s_0,\pi_1,s_1,\pi_2,...,\pi_n,s_n$ or an infinite sequence $s_0,\pi_1,s_1,\pi_2,...,\pi_n,s_n,...$ of alternating states and actions of A such that (s_i,π_{i+1},s_{i+1}) is a transition of A for every i. An execution fragment beginning with a start state is called an *execution.* We denote the set of executions of A by $execs(A)$, and the set of finite executions of A by it finexecs(A). A state is said to be *reachable* in A if it is the final state of a finite execution of A.

The *schedule* of an execution fragment α of A is the subsequence of α consisting of actions, and is denoted by *sched(α).* We say that β is a *schedule* of A if β is the schedule of an execution of A. We denote the set of schedules of A by $scheds(A)$ and the set of finite schedules of A by $finscheds(A)$. The *behavior* of an execution or schedule α of A is the subsequence of α consisting of external actions, and is denoted by $beh(\alpha)$. We say that β is a *behavior* of A if β is the behavior of an execution of A. We denote the set of behaviors of A by $behs(A)$ and the set of finite behaviors of A by $finbehs(A)$.

Composition

For motivation, consider the composition of user automata and a shared memory system automaton.

Generally speaking, we can construct an automaton modeling a complex system by composing automata modeling the simpler system components. The essence of this composition is quite simple: when we compose a collection of automata, we identify an output action π of one automaton with the input action π of each automaton having π as an input action. Consequently, when one automaton having π as an output action performs π , all automata having π as an input action perform π simultaneously (automata not having π as an action do nothing).

We impose certain restrictions on the composition of automata. Since internal actions of an automaton A are intended to be unobservable by any other automaton B, we cannot allow A to be composed with B unless the internal actions of A are disjoint from the actions of B, since otherwise one of A's internal actions could force B to take a step. Furthermore, in keeping with our philosophy that at most one system component controls the performance of any given action, we cannot allow A and B to be composed unless the output actions of A and B form disjoint sets. Finally, since we do not preclude the possibility of composing a countable collection of automata, each action of a composition must be an action of only finitely many of the composition's components. Note that with infinite products we can handle systems that can create processes dynamically.

Since the action signature of a composition (the composition's interface with its environment) is determined uniquely by the action signatures of its components, it is convenient to define a composition of action signatures before defining the composition of automata. The preceding discussion motivates the following definition. A countable collection $S_{i i \in I}$ of action signatures is said to be *strongly compatible* if for all $i, j \in I$ satisfying $i \neq j$ we have

1. $out(S_i) \cap out(S_j) = \emptyset$,
2. $int(S_i) \cap acts(S_j) = \emptyset$, and
3. no action is contained in infinitely many sets $acts(S_i)$.

We say that a collection of automata are *strongly compatible* if their action signatures are strongly compatible.

When we compose a collection of automata, internal actions of the components become internal actions of the composition, output actions become output actions, and all other actions (each of which can only an input action of a component) become input actions.

As motivation for this decision, consider one automaton A having π as an output action and two automata B_1 and B_2 having π as an input action. Notice that π is essentially a broadcast from A to B_1 and B_2 in the composition $A \cdot B_1 \cdot B_2$ of the three automata. Notice, however, that if we hide communication, then the composition $(A \cdot B_1) \cdot B_2$ would not be the same as the composition $A \cdot B_1 \cdot B_2$ since π would be made internal to $A \cdot B_1$ before composing with B_2, and hence π would no longer be a broadcast to both B_1 and B_2. This is problematic if we want to reason about the system $A \cdot B_1 \cdot B_2$ in a modular way by first reasoning about $A \cdot B_1$ and then reasoning about $A \cdot B_1 \cdot B_2$. We will define another operation to hide such communication actions explicitly.

The preceding discussion motivates the following definitions. The *composition* $S = \prod_{i \in I} S_i$ of a countable collection of strongly compatible action signatures $\{S_i\}_{i \in I}$ is defined to be the action signature with

- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$,
- $out(S) = \cup_{i \in I} out(S_i)$, and
- $int(S) = \cup_{i \in I} int(S_i)$.

Illustration: users and shared memory system automaton. List the various actions, show signatures are compatible.

The *composition* $A = \prod_{i \in I} A_i$ of a countable collection of strongly compatible automata $\{A_i\}_{i \in I}$ is the automaton defined as follows:[8]

- $sig(A) = \prod_{i \in I} sig(A_i)$,
- $states(A) = \prod_{i \in I} states(A_i)$,
- $start(A) = \prod_{i \in I} start(A_i)$,
- $trans(A)$ is the set of triples $(\vec{s_1}, \pi, \vec{s_2})$ such that, for all $i \in I$, if $\pi \in acts(A_i)$ then $(\vec{s_1}[i], \pi, \vec{s_2}[i]) \in trans(A_i)$, and if $\pi \notin acts(A_i)$ then $\vec{s_1}[i] = \vec{s_2}[i]$, and
- $part(A) = \cup_{i \in I} part(A_i)$.

When I is the finite set $1, ..., n$, we often denote $\prod_{i \in I} A_i$ by $A_1 \cdot \cdots \cdot A_n$.

Notice that since the automata A_i are input-enabled, so is their composition. The partition of the composition's locally-controlled actions is formed by taking the union of the components' partitions (that is, each equivalence class of each component becomes an equivalence class of the composition).

Three basic results relate the executions, schedules, and behaviors of a composition to those of the composition's components. The first says, for example, that an execution of a composition induces executions of the component automata. Given an execution $\alpha = \vec{s_0}\pi_1\vec{s_1}\ldots$ of A, let $\alpha|A_i$ be the sequence obtained by deleting $\pi_j\vec{s_j}$ when π_j is not an action of A_i and replacing the remaining $\vec{s_j}$ by $\vec{s_j}[i]$.

Proposition 6 *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \prod_{i \in I} A_i$. If $\alpha \in execs(A)$ then $\alpha|A_i \in execs(A_i)$ for every $i \in I$. Moreover, the same result holds if execs is replaced by it finexecs, scheds, finscheds, behs, or finbehs.*

[8]Here $start(A)$ and $states(A)$ are defined in terms of the ordinary Cartesian product, while $sig(A)$ is defined in terms of the composition of actions signatures just defined. Also, we use the notation $\vec{s}[i]$ to denote the ith component of the state vector $\vec{s}$.

Certain converses of the preceding proposition are also true. The following proposition says that executions of component automata can often be pasted together to form an execution of the composition.

Proposition 7 *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \prod_{i \in I} A_i$. Suppose α_i is an execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $acts(A)$ such that $\beta|A_i = sched(\alpha_i)$ for every $i \in I$. Then there is an execution α of A such that $\beta = sched(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$. Moreover, the same result holds when acts and sched are replaced by ext and beh, respectively.*

As a corollary, schedules and behaviors of component automata can also be pasted together to form schedules and behaviors of the composition.

Proposition 8 *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \prod_{i \in I} A_i$. Let β be a sequence of actions in $acts(A)$. If $\beta|A_i \in scheds(A_i)$ for every $i \in I$, then $\beta \in scheds(A)$. Moreover, the same result holds when acts and scheds are replaced by ext and behs, respectively.*

As promised, we now define an operation that 'hides' actions of an automaton by converting them to internal actions. This operation is useful for redefining what the external actions of a composition are. We begin with a hiding operation for action signatures: if S is an action signature and $\Sigma \subseteq acts(S)$, then $hide_\Sigma S = S'$ where $in(S') = in(S) - \Sigma$, $out(S') = out(S) - \Sigma$ and $int(S') = int(S) \cup \Sigma$. We now define a hiding operation for automata: if A is an automaton and $\Sigma \subseteq acts(A)$, then $hide_\Sigma A$ is the automaton A' obtained from A by replacing $sig(A)$ with $sig(A') = hide_\Sigma sig(A)$.

Fairness

We are in general only interested in the executions of a composition in which all components are treated fairly. While what it means for a component to be treated fairly may vary from context to context, it seems that any reasonable definition should have the property that infinitely often the component is given the opportunity to perform one of its locally-controlled actions. In this section we define such a notion of fairness.

As we have mentioned, the partition of an automaton's locally-controlled actions is intended to capture some of the structure of the system the automaton is modeling. Each class of actions is intended to represent the set of locally-controlled actions of some system component.

The definition of automaton composition guarantees that an equivalence class of a component automaton becomes an equivalence class of a composition, and hence that composition retains the *essential* structure of the system's primitive components.[9] In our model, therefore,

[9]It might be argued that retaining this partition is a bad thing to do since it destroys some aspects of

being fair to each component means being fair to each equivalence class of locally-controlled actions. This motivates the following definition.

A *fair execution* of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of *part*(A):

1. If α is finite, then no action of C is enabled in the final state of α.

2. If α is infinite, then either α contains infinitely many events from C, or α contains infinitely many occurrences of states in which no action of C is enabled.

This says that a fair execution gives fair turns to each class C of $part(A)$, and therefore to each component of the system being modeled. Infinitely often the automaton attempts to perform an action from the class C. On each attempt, either an action of C is performed, or no action from C *can* be performed since no action from C is enabled. For example, we may view a finite fair execution as an execution at the end of which the automaton repeatedly cycles through the classes in round-robin order attempting to perform an action from each class, but failing each time since no action is enabled from the final state.

We denote the set of fair executions of A by $fairexecs(A)$. We say that β is a *fair schedule* of A if β is the schedule of a fair execution of A, and we denote the set of fair schedules of A by $fairscheds(A)$. We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A, and we denote the set of fair behaviors of A by $fairbehs(A)$.

We can prove the following analogues to Propositions 1-3 in the preceding section:

Proposition 9 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. If $\alpha \in fairexecs(A)$ then $\alpha|A_i \in fairexecs(A_i)$ for every $i \in I$. Moreover, the same result holds if fairexecs is replaced by fairscheds or fairbehs.*

Proposition 10 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. Suppose α_i is a fair execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $acts(A)$ such that $\beta|A_i = sched(\alpha_i)$ for every $i \in I$. Then there is a fair execution α of A such that $\beta = sched(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$. Moreover, the same result holds when acts and sched are replaced by ext and beh, respectively.*

Proposition 11 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. Let β be a sequence of actions in $acts(A)$. If $\beta|A_i \in fairscheds(A_i)$ for every $i \in I$, then $\beta \in fairscheds(A)$. Moreover, the same result holds when acts() and fairscheds() are replaced by ext and fairbehs, respectively.*

abstraction. Notice, however, that any reasonable definition of fairness must lead to some breakdown of abstraction since being fair means being fair to the primitive components which must somehow be modeled.

We state these results because analogous results often do not hold in other models. As we will see in the following section, the fact that the fair behavior of a composition is uniquely determined by the fair behavior of the components makes it possible to reason about the fair behavior of a system in a modular way.

Lecturer: Nancy Lynch

Lecture 13

13.1 I/O Automata (cont.)

13.1.1 Problem Specification

We want to say that a problem specification is simply a set of allowable 'behaviors,' and that an automaton solves the specification if each of its 'behaviors' is contained in this set. The automaton solves the problem in the sense that every 'behavior' it exhibits is a 'behavior' allowed by the problem specification (but notice that there is no single 'behavior' the automaton is *required* to exhibit). The appropriate notion of 'behavior' (e.g., finite behavior, arbitrary behavior, fair behavior, etc.) used in such a definition depends to some extent on the nature of the problem specification. In addition to a set of allowable behaviors, however, a problem specification must specify the required interface between a solution and its environment. That is, we want a problem specification to be a set of behaviors together with an action signature.

We therefore define a *schedule module* H to consist of two components:

- an action signature $sig(H)$, and
- a set $scheds(H)$ of *schedules*.

Each schedule in $scheds(H)$ is a finite or infinite sequence of actions of H. We denote by $finscheds(H)$ the set of finite schedules of H. The *behavior* of a schedule β of H is the subsequence of β consisting of external actions, and is denoted by $beh(\beta)$. We say that β is a *behavior* of H if β is the behavior of a schedule of H. We denote the set of behaviors of H by $behs(H)$ and the set of finite behaviors of H by $finbehs(H)$. We extend the definitions of fair schedules and fair behaviors to schedule modules in a trivial way, letting $fairscheds(H) = scheds(H)$ and $fairbehs(H) = behs(H)$. We will use the term *module* to refer to either an automaton or a schedule module.

There are several natural schedule modules that we often wish to associate with an automaton. They correspond to the automaton's schedules, finite schedules, fair schedules, behaviors, finite behaviors and fair behaviors. For each automaton A, let $Scheds(A)$,

$Finscheds(A)$ and $Fairscheds(A)$ be the schedule modules having action signature $sig(A)$ and having schedules $scheds(A)$, $finscheds(A)$ and $fairscheds(A)$, respectively. Also, for each module M (either an automaton or schedule module), let $Behs(M)$, $Finbehs(M)$ and $Fairbehs(M)$ be the schedule modules having the external action signature $extsig(M)$ and having schedules $behs(M)$, $finbehs(M)$ and $fairbehs(M)$, respectively. (Here and elsewhere, we follow the convention of denoting sets of schedules with lower case names and corresponding schedule modules with corresponding upper case names.)

It is convenient to define two operations for schedule modules. Corresponding to our composition operation for automata, we define the composition of a countable collection of strongly compatible schedule modules $\{H_i\}_{i \in I}$ to be the schedule module $H = \prod_{i \in I} H_i$ where:

- $sig(H) = \prod_{i \in I} sig(H_i)$,
- $scheds(H)$ is the set of sequences β of actions of H such that $\beta | H_i$ is a schedule of H_i for every $i \in I$.

The following proposition shows how composition of schedule modules corresponds to composition of automata.

Proposition 1 *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \prod_{i \in I} A_i$. Then $Scheds(A) = \prod_{i \in I} Scheds(A_i)$, $Fairscheds(A) = \prod_{i \in I} Fairscheds(A_i)$, $Behs(A) = \prod_{i \in I} Behs(A_i)$ and $Fairbehs(A) = \prod_{i \in I} Fairbehs(A_i)$.*

Corresponding to our hiding operation for automata, we define hide $hide_{\Sigma} H$ to be the schedule module H' obtained from H by replacing $sig(H)$ with $sig(H') = hide_{\Sigma} sig(H)$.

Finally, we are ready to define a problem specification and what it means for an automaton to satisfy a specification. A *problem* is simply a schedule module P. An automaton A *solves*[10] a problem P if A and P have the same external action signature and $fairbehs(A) \subseteq fairbehs(P)$. An automaton A *implements* a problem P if A and P have the same external action signature (that is, the same external interface) and $finbehs(A) \subseteq finbehs(P)$. Notice that if A solves P, then A cannot be a trivial solution of P since the fact that A is input-enabled ensures that $fairbehs(A)$ contains a response by A to every possible sequence of input actions. For analogous reasons, the same is true if A implements P.

Since we may want to carry out correctness proofs hierarchically in several stages, it is convenient to state the definitions of 'solves' and 'implements' more generally. For example, we may want to prove that one automaton solves a problem by showing that the automaton 'solves' another automaton, which in turn 'solves' another automaton, and so on, until some final automaton solves the original problem. Therefore, let M and M' be modules

[10]This concept is sometimes called *satisfying*.

(either automata or schedule modules) with the same external action signature. We say that M *solves* M' if *fairbehs*$(M) \subseteq$ *fairbehs*(M') and that M *implements* M' if *finbehs*$(M) \subseteq$ *finbehs*(M').

As we have seen, there are many ways to argue that an automaton A solves a problem P. We now turn our attention to two more general techniques.

13.1.2 Proof Techniques

Modular Decomposition

One common technique for reasoning about the behavior of an automaton is *modular decomposition*, in which we reason about the behavior of a composition by reasoning about the behavior of the component automata individually.

It is often the case that an automaton behaves correctly only in the context of certain restrictions on its input. These restrictions may be guaranteed in the context of the composition with other automata comprising the remainder of the system, or may be restrictions defined by a problem statement describing conditions under which a solution is required to behave correctly. A useful notion for discussing such restrictions is that of a module 'preserving' a property of behaviors: as long as the environment does not violate this property, neither does the module.

In practice, this notion is of most interest when the property is prefix-closed, and when the property does not concern the module's internal actions. A set of sequences $\mathcal{P}$ is said to be *prefix-closed* if $\beta \in \mathcal{P}$ whenever both β is a prefix of α and $\alpha \in \mathcal{P}$. A module M (either an automaton or schedule module) is said to be *prefix-closed* provided that *finbehs*(M) is prefix-closed.

Let M be a prefix-closed module and let $\mathcal{P}$ be a nonempty, prefix-closed set of sequences of actions from a set Φ satisfying $\Phi \cap int(M) = \emptyset$. We say that M *preserves* $\mathcal{P}$ if $\beta\pi|\Phi \in \mathcal{P}$ whenever $\beta|\Phi \in \mathcal{P}$, $\pi \in out(M)$, and $\beta\pi|M \in$ *finbehs*(M).

In general, if a module preserves a property $\mathcal{P}$, then the module is not the first to violate $\mathcal{P}$: as long as the environment only provides inputs such that the cumulative behavior satisfies $\mathcal{P}$, the module will only perform outputs such that the cumulative behavior satisfies $\mathcal{P}$. This definition, however, deserves closer inspection. First, notice that we consider only sequences β with the property that $\beta\pi|M \in$ *finbehs*(M). This implies that we consider only sequences β that contain no internal actions of M. Second, notice that we require sequences β to satisfy only $\beta|\Phi \in \mathcal{P}$ rather than the stronger property $\beta \in \mathcal{P}$. Suppose, for example, that $\mathcal{P}$ is a property of the actions Φ at one of two interfaces to the module M. In this case, it may be that for no $\beta \in \mathcal{P}$ and $\pi \in out(M)$ is it the case that $\beta\pi|M \in$ *finbehs*(M), since all finite behaviors of M containing outputs include activity at both interfaces to M. By considering β satisfying only $\beta|\Phi \in \mathcal{P}$, we consider all sequences determining finite behaviors of M that, at the interface concerning $\mathcal{P}$, do not violate the property $\mathcal{P}$.

One can prove that a composition preserves a property by showing that each of the component automata preserves the property:

Proposition 2 *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata and let $A = \prod_{i \in I} A_i$. If A_i preserves $\mathcal{P}$ for every $i \in I$, then A preserves $\mathcal{P}$.*

In fact, we can prove a slightly stronger result. An automaton is said to be *closed* if it has no input actions. In other words, it models a closed system that does not interact with its environment.

Proposition 3 *Let A be a closed automaton. Let $\mathcal{P}$ be a set of sequences over Φ. If A preserves $\mathcal{P}$, then finbehs$(A)|\Phi \subseteq \mathcal{P}$.*

In the special case that Φ is the set of external actions of A, the conclusion of this proposition reduces to the fact that *finbehs*$(A) \subseteq \mathcal{P}$. The proof of the proposition depends on the fact that Φ does not contain any of A's input actions, and hence that if the property $\mathcal{P}$ is violated then it is not an input action of A committing the violation. In fact, this proposition follows as a corollary from the following slightly more general statement: If A preserves $\mathcal{P}$ and *in*$(A) \cap \Phi = \emptyset$, then *finbehs*$(A)|\Phi \subseteq \mathcal{P}$.

Combining Propositions 2 and 3, we have the following technique for proving that an automaton implements a problem:

Corollary 4 *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata with the property that $A = \prod_{i \in I} A_i$ is a closed automaton. Let P be a problem with the external action signature of A. If A_i preserves finbehs(P) for all $i \in I$, then A implements P.*

That is, if we can prove that each component A_i preserves the external behavior required by the problem P, then we will have shown that the composition A preserves the desired external behavior; and since A has no input actions that could be responsible for violating the behavior required by P, it follows that all finite behaviors of A are behaviors of P.

Hierarchical Decomposition

A second common technique for proving that an automaton solves a problem is *hierarchical decomposition* in which we prove that the given automaton solves a second, that the second solves a third, and so on until the final automaton solves the given problem. One way of proving that one automaton A solves another automaton B is to establish a relationship between the states of A and B and use this relationship to argue that the fair behaviors of A are fair behaviors of B. In order to establish such a relationship in between two automata we can use a "simulation" relation f. Below, for binary relation f, we use the notation $u \in f(s)$ as an alternative way of writing $(s, u) \in f$.

Definition 1 *Suppose A and B are input/output automata with the same external action signature, and suppose f is a binary relation from $states(A)$ to $states(B)$. Then f is a* forward simulation *from A to B provided that both of the following are true:*

1. *If $s \in start(A)$ then $f(s) \cap start(B) \neq \emptyset$.*
2. *If s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and (s, π, s') is a step of A, then there is an 'extended step' (u, γ, u') such that $u' \in f(s')$ and $\gamma|ext(B) = \pi|ext(A)$.*

An extended step of an automaton A is a triple of the form (s, β, s'), where s and s' are states of A, β is a finite sequence of actions of A, and there is an execution fragment of A having s as its first state, s' as its last state, and β as its schedule. The following theorem gives the key property of forward simulations:

Theorem 5 *If there is a forward simulation from A to B, then $behs(A) \subseteq behs(B)$.*

13.2 Shared Memory Systems as I/O Automata

13.2.1 Instantaneous Memory Access

As a case study, we consider a mutual exclusion system. In this section we model the *users* as IO automata. Specifically, each $user_i$ is an IOA, with inputs $crit_i$ and rem_i, and outputs try_i and $exit_i$. The *user* automata have arbitrary internal actions, states sets and start states. There is only one constraint — that any such automaton "preserves" the cyclic behavior, i.e., if the system is not the first to violate it, the user does not violate it. (The formal definition of "preserves" is given in the notes above.)

The following example shows a particular user IOA. Note the language that is used to describe the automaton. We first describe the IOA intuitively. The "example user" chooses an arbitrary number from 1 to 3, and then makes exactly that many requests for the resource in succession.

In the precondition-effects language, the user is described as follows.

states: As before, we can write it as consisting of components. Here we have the following.

- *region*, values in $\{R, T, C, E\}$, initially R
- *count*, a number from 1 to 3 or *nil*, initially arbitrary

- *chosen*, a Boolean, initially *false*

start: Given by the initializations.

acts: The external actions are described above. Also, there is an internal action *choose*.

part: we have only one class that contains all the non-input actions.

trans: We describe the allowed triples (s, π, s') by organizing them according to action.

```
choose_i
Precondition:
    $s.chosen = false$
Effect:
    $s'.chosen = true$
    $s'.count \in \{1, 2, 3\}$

try_i
Precondition:
    $s.chosen = true$
    $s.region = R$
    $s.count > 0$
Effect:
    $s'.region = T$
    $s'.count = s.count - 1$

crit_i
Effect:
    $s'.region = C$

exit_i
Precondition:
    $s.region = C$
Effect:
    $s'.region = E$

rem_i
Effect:
    $s'.region = R$
```

Note that the automaton is input-enabled, i.e., it can accommodate unexpected inputs (which do not necessarily induce interesting behavior in this case, though.)

Here, we are describing the transitions of the form (s, π, s') using explicit mentions of the pre- and post- states s and s'. The effect is described in terms of equations relating the two. We remark that this notation is very similar to that used by Lamport in his language TLA (Temporal Logic of Actions). An alternative notation that is also popular suppresses explicit mention of the states, and describes the effect using assignment statements. This is the syntax used in Ken Goldman's Spectrum system, for instance.

Fair Executions. Consider the executions in which the system does not violate the cyclic behavior. All these executions must involve first doing *choose*: this action is enabled, nothing else is enabled, and the fairness requires that the execution can't stop if something is enabled. By the code, the next step must be try_i. Then the execution can stop, if no $crit_i$ ever occurs. But if $crit_i$ occurs, a later $exit_i$ must occur. Again, at this point the execution can stop, but if a later rem_i occurs, then the automaton stops if it chose 1, else the execution continues.

That is, have fair executions (omitting states that are determined):

choose, 1, *try*

choose, 1, *try*, *crit*, *exit*

choose, 1, *try*, *crit*, *exit*, *rem*

choose, 2, *try*, *crit*, *exit*, *rem*, *try*,

etc.

There are also some fair executions in which the system violates the cyclic behavior, e.g.:

$$choose, 1, crit$$

But, for instance, the following is not fair:

$$choose, 2, try, crit, exit, rem$$

Note that for the automaton above, all the fair executions with correct cyclic behavior are finite.

To model the rest of the shared memory system, we use one big IOA to model all the processes and the memory together. The outputs are the *crit* and *rem* actions. The internal actions are any local computation actions and the shared memory actions. The *states* set consists of the states of all the processes plus the state of the shared memory. The partition is defined by having one class per process, which contains all the non-input actions belonging to that process. The IOA fairness condition captures exactly our requirement on fairness of process execution in the shared memory system. Of course, there are extra constraints that are not implied by the general model, e.g., types of memory (read-write, etc.), and locality of process action (i.e., that it can only affect its own state, only access one variable at a time, etc.).

13.2.2 Object Model

In this section we define formally an alternative way to model shared memory systems, the *object model*, which was discussed briefly in Lecture 10. We will use this model for the next few lectures. In the object model, each process is a separate IOA (usually with only one fairness class), and each shared memory object is also an IOA. Thus, for example, a read-write object is an IOA with inputs of the form $read_{i,x}$, where i is a process and x an object name, and $write_{i,x}(v)$, where i is a process, x is an object and v is a value to be written. Now, note that a *read* is supposed to return a value. Since this is an IOA, this will be a separate output action $read\text{-}respond_{i,x}(v)$. For uniformity, also give the *write* a $write\text{-}respond_{i,x}$. The $write\text{-}respond_{i,x}$ action serves only as an "ack", saying that the write activation has terminated.

This is essentially the same interface that we have already described for atomic read-write objects — the only difference is that now we are adding an identifier for the specific object into all the actions. (This is because IOA's have a "global naming scheme" for actions.)

Of course, we could also have other objects: queue, read-modify-write registers, and snapshot objects, as discussed earlier. Each has its own characteristic interface, with inputs being invocations of operations and outputs being responses.

The processes are automata with inputs as before, plus the responses to their data operations, e.g., $read\text{-}respond_{i,x}(v)$ is an input to process i. The outputs of the processes now also contain the invocations of their data operations, e.g., $read_{i,x}$. Usually, each process would have only one fairness class; that is, we think of it as a sequential process. (But we would like to be able to generalize this when convenient.)

For the objects, we usually don't want to consider just any automata with the given interface — we want some extra conditions on the correctness of the responses. For example, the atomic object conditions defined earlier are an important example. We recall these conditions below (see notes of Lecture 10 for more complete specification).

1. Preserving well-formedness. Recall that "preserves" has a formal definition for IOA's.

2. Giving "atomic" responses based on serialization points, in all executions. (Recall that it is required that all complete operations and an arbitrary subset of the incomplete operations have serialization points.)

3. In fair executions, every invocation has a response. Note that this now makes sense for automata that have some organization other than the process-variable architecture we have been studying, because IOA's in general have a notion of fairness.

After composing the process automata and the object automata, we hide the communication actions between them.

13.3 Modularity Results

Now that we have described the two kinds of models, instantaneous access and atomic objects, more-or-less precisely in terms of IOA's, we can state our modularity results.

13.3.1 Transforming from Instantaneous to Atomic Objects

In this section we discuss how to transform algorithms which are specified for instantaneous objects to work with atomic versions of the objects. Let A be an algorithm in the instantaneous shared memory model satisfying the following conditions.

- Each process can only access one shared variable at a time (this is the usual restriction).
- For each $user_i$ of A, suppose that there is a notion of *user well-formedness*, consisting of a subset of alternating sequences of input and output actions. We require that A preserve user well-formedness.
- Suppose that p_i's states are classified as "input states" and "output states", based on which is supposed to happen next in a well-formed execution (initially, only "input" actions are enabled). We require that non-input steps of p_i are only enabled in "output" states.

We remark that this is the case for all the examples we have considered. For mutual exclusion algorithms, the output states are the $T \cup E$ states, which occur between *try* and *crit* actions, and between *exit* and *rem* actions, respectively. For snapshot implementations, the output states are just those while an invocation of *update* or *snap* is active. For consensus algorithms, the output states are those between the *init* and *decide*.

Assuming A satisfies the above conditions, we can now transform A into a new algorithm $T(A)$, designed for the atomic object model. $T(A)$ includes an atomic object for each shared memory location. Each access to an object is expanded to an invocation and a response. After the invocation, the process waits until the response occurs, doing nothing else in the meantime. When the response comes in, the process resumes as in A.

Lemma 6

$T(A)$ preserves user well-formedness for each user i.

2. *If α is any (not necessarily fair) user well-formed execution of $T(A)$, then there is a user well-formed execution α' of A such that $beh(\alpha) = beh(\alpha')$.*

3. *If α is any fair user well-formed execution of $T(A)$ (i.e., both the processes and the atomic objects have fair executions), then there is a fair user well-formed execution α' of A such that $beh(\alpha) = beh(\alpha')$.*

Proof Sketch: Here is a sketch of the argument. We start with α and modify it to get α' as follows. First, by the definition, we have the serialization points within the operation intervals, for all the invocations of the atomic objects. Now consider the execution obtained by moving the invocation and response actions to the serialization points, and adjusting the states accordingly. In the case of incomplete operation intervals, there are two cases. If there is a serialization point for the interval, then put the invocation from α, together with a *newly manufactured* response event, at the serialization point (the response event should be the one whose existence is asserted by the atomic object definition). If there is no serialization point, then just remove the invocation.

We now claim that we can move all the events we have moved *without changing the order of any events of a particular process p_i*. This follows from two facts. First, by construction, p_i does no locally-controlled actions while waiting for a response from an operation. Second, when p_i invokes an operation, it is in an output state, and by assumption, no inputs will come in when p_i is in an output state. Similarly, we can remove the invocations we have removed and add the responses we have added without affecting the external behavior, since a process with an incomplete invocation does not do anything afterwards anyway (it doesn't matter if it stops just before issuing the invocation, while waiting for the response, or just after receiving the response).

In the resulting execution, all the invocations and responses occur in consecutive pairs. We can replace those pairs by instantaneous access steps, thus obtaining an execution α of the instantaneous access system A. Thus α and α' have the same behavior at the user interface, proving Part 2.

For Part 3, notice that if α is fair, then all embedded executions of the atomic objects always respond to all invocations. Coupling this with the fairness assumption for processes, yields fairness for the simulated algorithm A (i.e., that the simulated processes continue taking steps). ▮

So, Lemma 6 says that any algorithm for the instantaneous shared memory model can be used (in a transformed version) in the atomic object shared memory model, and no external user of the algorithm could ever tell the difference.

In the special case where A itself is an atomic object implementation (in the instantaneous shared memory model), this lemma implies that $T(A)$ is also an atomic object implementation. To see this, note that Property 1 (preserving well-formedness) follows immediately from Part 1 of Lemma 6; Property 2 follows from Part 2 of Lemma 6 and the fact that A is an atomic object (property 2 of the atomic object definition for A); and Property 3 follows from Part 3 of Lemma 6 and the fact that A is an atomic object (Property 3 of the atomic object definition for A). Therefore, we can build atomic objects *hierarchically*.

13.3.2 Composition in the Object Model

Now we want to consider building objects that are not necessarily atomic hierarchically. To do this, we have to generalize the notion of an object specification to include other kinds of objects. Later in the course we shall see some examples for such objects, including *safe* and *regular* objects, and the special concurrent timestamp object to be considered next time.

Object Specifications

We define the general notion of an *object specification*, generalizing the specification for atomic objects. Formally, an object specification consists of the following.

1. An external signature consisting of input actions, called *invocations*, and output actions, called *responses*. There is also a fixed binary relation that relates responses (syntactically) to corresponding invocations.

2. A set of sequences of external actions; all sequences in the set are "well-formed", i.e., they contain alternating invocations and (corresponding) responses on each of the lines (with invocation first). The set is prefix-closed.

This is a very general notion. Basically, this is an arbitrary behavior specification at the object boundary. The behavior specification for atomic objects is a special case.

An IOA with the appropriate interface is said to *satisfy*, or *implement*, this specification if the following conditions hold.

1. It preserves well-formedness.

2. All its well-formed behaviors are in the set.

3. All its fair behaviors include responses to all invocations.

Compositionality of Object Implementations

Suppose that we start with a system A in the object specification model, using a particular set of object specifications for its objects. We need to be a little more careful about what this means. The executions of this system are the executions of the processes that get responses allowed by the object specifications. The fair executions of the system are then defined to be those executions that are fair to all the processes and in which all invocations to the object specifications eventually return. We also assume that each process of A preserves well-formedness for each object specification.

Suppose that A itself satisfies another, "higher-level" object specification. Consider the result of replacing all the object specifications of A with object IOA's that satisfy the corresponding specifications (in the object specification model). We claim that the resulting system still implements the external object specification.

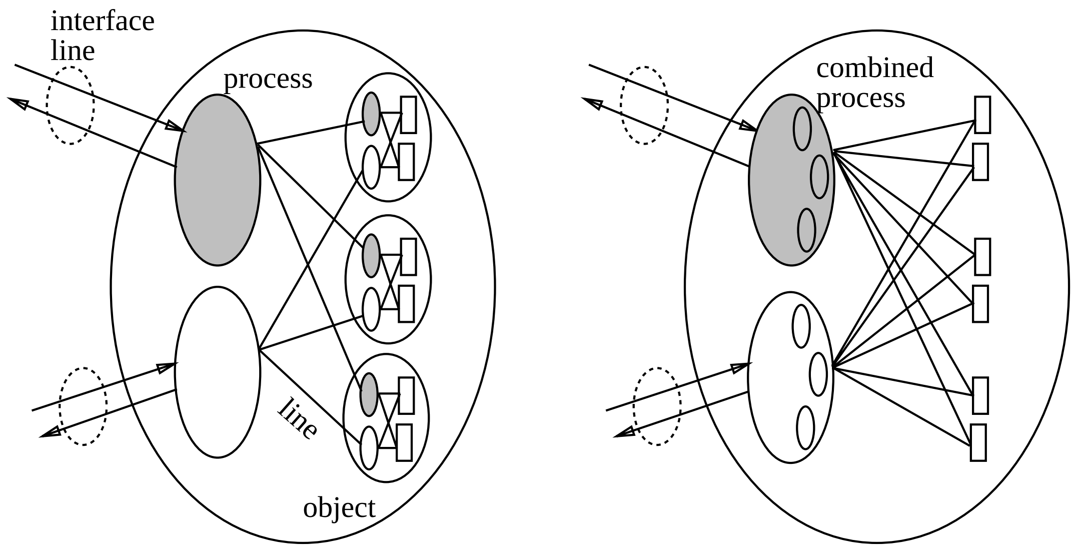


Figure 13.1: On the left we see an object composed of smaller objects. All the shaded processes are associated with the same external interface line. On the right we see the combined process for fairness purposes.

Note that fairness of the composed implementation refers to fair execution of all the processes, high-level and low-level, plus the object specification liveness condition for the low level objects.

Note that the resulting architecture is different from the model we have been using. It doesn't have one process per line, but rather a distributed collection of processes. We can convert this to a system in the usual architecture by composing some of these processes to give the processes we want, as follows (see figure 13.1). Compose the distributed processes that operate on behalf of any particular line i — the high-level process plus, for every line of the intermediate-level objects that it uses, the corresponding low-level process (or processes, if one high-level process invokes several different operation types on a single object). Call this composition q_i. Let the composed processes q_i be the processes of the composed implementation. (Notice that they have several fairness classes each: this is why we didn't want to rule this out earlier.)

Remark: We can get similar result composing a system that uses object specifications with object implementations that use instantaneous memory, getting a composed implementation that uses instantaneous memory.

Lecturer: Nancy Lynch

Lecture 14

14.1 Modularity (cont.)

14.1.1 Wait-freedom

In this section we consider the *wait-freedom* property. We define this property only for implementations in some special forms as follows.

1. In the instantaneous shared memory model, wait-freedom deals with an intermediate notion between ordinary IOA fairness (fairness to all processes) and no fairness: namely, fairness to particular processes. For this setting, the wait freedom condition is that if an execution is fair to any particular process i, then any invocation of i eventually gets a response.

2. In the object specification model, where in place of the objects we have object specifications, the wait freedom condition is that if an execution is fair to any particular process i, *and all invocations to memory objects by process i return*, then any invocation of i eventually gets a response.

Let us consider the compositionality of wait-free implementations. To be specific, suppose that we start with a system A in the object spec model, that by itself satisfies some object spec, using a particular set of object specs for its objects. Suppose that each process of A preserve well-formedness for each object. Assume also that A is *wait-free*. Suppose we then replace all the object specs used by A with *wait-free* objects that satisfy the corresponding specs; again, these are required to be expressed in the object spec model. We already know how to get the composition, and we also know that the result satisfies the external object spec. We now want to claim that the resulting system is also wait-free. We argue that as follows. Let α be an execution of the composed system, in which q_i does not fail, and in which all invocations by q_i on low-level objects return. In α, the high-level p_i does not fail, since by assumption, q_i doesn't fail. Also, any invocation by p_i on the intermediate-level object must return, by the wait-freedom of the implementation of the intermediate objects.

Therefore, by the wait-freedom of the high-level implementation, any invocation on line i eventually returns.

Remark. We can get a similar result about composing a wait-free implementation that uses object specs, with wait-free object implementations that use instantaneous memory, getting a wait-free composed implementation that uses instantaneous memory.

14.2 Concurrent Timestamp Systems

Today we shall go through some constructions to show how we could build some powerful objects out of others. We shall do this hierarchically. We have already seen (in Lecture 10) how we can get a wait-free implementation of an atomic snapshot object, in terms of instantaneous access single-writer multi-reader read-write objects. In this section we show how to get a wait-free implementation of another type of object, called *concurrent timestamp object* (CTS), in terms of instantaneous snapshot objects. By the compositionality results above, we can combine these and get a wait-free implementation of a concurrent timestamp object in terms of instantaneous access single-writer multi-reader read-write variables.

We shall then see how we can use a CTS object as a building block for other tasks. Specifically, we give an improvement to Lamport's bakery algorithm, and an implementation of a wait-free multi-writer multi-reader register. CTS objects can also be use for other tasks, e.g., resource allocation systems.

14.2.1 Definition of CTS Objects

A concurrent timestamp system object has two operations.

- $label_i(v)$: always gets the response "OK".
- $scan_i$: gets a response that describes a total ordering of the n processes, together with an associated value for each process.

The rough idea is that each newly arrived process gets an associated label of some kind, and we require that later labels can be detected as "larger" than earlier labels. The *scan* operation is supposed to return a reasonable ordering of the processes, based on the order of their most recent labels (i.e., the approximate order of their most recent arrivals). And, for good measure, it also returns the values associated with those labels.

Note that this is not an atomic object; it has a more general kind of object spec. In the literature (see Gawlick's thesis, and Dolev-Shavit), the precise requirements are given by a set of axioms about the well-formed behaviors. These look a little too complicated to present in class, so instead we define the correct behaviors as the well-formed behaviors of a fairly simple algorithm, given in Figures 14.2 and 14.1.

This "spec algorithm" uses a different programming notation from what we have previously used (e.g., for mutual exclusion). The notation presented here describes I/O automata more directly: each action is specified separately, with an explicit description of the states in which it can occur, and the changes it causes to the process state. Note also the explicit manipulation of the program counter (*pc*). This notation is very similar to that used in the previous lecture to describe a user automaton; the main differences are that here we use assignment statements rather than equations and leave the pre- and post-states implicit.

The "begin" and "end" actions just move the *pc* and transmit the results. The real work is done by embedded *snap* and *update* actions. In the *scan*, the process does a *snap* to get an instantaneous snapshot of everyone's latest label and value. Then it returns the order determined by the labels, with the associated values. In the *label*, the process first does a *snap* to collect everyone's labels. Then if the process does not already have the max, it chooses some larger value (this could be the integer one greater than the max, as in Lamport's bakery, or it could be some more general real value). Finally, it does an instantaneous *update* to write the new label and value in the shared memory.

A straightforward observation is that the algorithm is wait-free. The most interesting property of the algorithm, however, is that it can be implemented using a *bounded domain* for the labels. This property will allow us to obtain bounded versions for bakery-style mutual exclusion algorithm, and a wait-free implementation of multi-writer multi-reader atomic objects in terms of single-writer multi-reader (which uses three levels of implementation: snapshot in terms of single-writer objects, CTS object in terms of snapshot, and multi-writer registers in terms of CTS object).

14.2.2 Bounded Label Implementation

We start with a bounded label implementation. First, we recall what this means. We are using the program above (Figures 14.2,14.1) as an object specification. It describes the interface, and gives a well-defined set of well-formed behaviors (whatever they are). To get an implementation of this spec, we need to design another algorithm that also preserves well-formedness, that has the required liveness properties, and whose well-formed behaviors are a *subset* of those of this algorithm.

The algorithm we use is very close to the unbounded algorithm above given by the spec. The only difference is in the label domain — for the bounded case we shall use a finite domain of size 5^{n-1}. The new domain has a binary relation defined for the purpose of determining order of labels, maximum, etc., only now it can't be a total order because elements have

Shared variables A snapshot object, whose value is a vector of pairs, $\overline{(label, value)}$, with one component for each "labeler" process i. The *label*s are positive real numbers, and the domain of *value* is arbitrary.

$scan_i$ **process**

Local state:

- pc: program counter
- $\overline{(snaplabel, snapvalue)}$: to hold values returned by a *snap*
- $order$: to hold an ordering of processes to be returned

Actions:

```
$beginscan_i$
  Effect:
    $pc \leftarrow snap$

$snap_i$
  Precondition:
    $pc = snap$
  Effect:
    $\overline{(snaplabel, snapvalue)} \leftarrow (\overline{label, value})$
    $pc \leftarrow define\text{-}response$

$define\text{-}response_i$
  Precondition:
    $pc = define\text{-}response$
  Effect:
    $order \leftarrow$ the total order on indices where $j$ appears
        before $k$ iff $(snaplabel_j, j) < (snaplabel_k, k)$
  $pc \leftarrow endscan$

$endscan_i(o, \overline{v})$
  Precondition:
    $pc = endscan$
    $o = order$
    $\overline{v} = \overline{snapvalue}$
  Effect:
    $pc \leftarrow beginscan$
```

Figure 14.1: Specification algorithm for timestamp system, part I

label$_i$ **process**

Local state:

- *pc*
- $\overline{(snaplabel, snapvalue)}$, to hold values returned by a *snap*.
- *newlabel*, to hold the newly-chosen label

Actions:

```
$beginlabel_i$
  Effect:
    $pc \leftarrow snap$

$snap_i$
  Precondition:
    $pc = snap$
  Effect:
    $\overline{(snaplabel, snapvalue)} \leftarrow \overline{label, value}$
    if $\forall j$, $(snaplabel[j], j) \leq (snaplabel[i], i)$ then
        $newlabel \leftarrow snaplabel[i]$
    else $newlabel \leftarrow max(\overline{snaplabel}) + r$, where $r$
        is any positive real (nondeterministic)
    $pc \leftarrow update$

$update_i$
  Precondition:
    $pc = update$
  Effect:
    $label[i] \leftarrow newlabel$
    $pc \leftarrow endlabel$

$endlabel_i$
  Precondition:
    $pc = endlabel$
  Effect:
    $pc \leftarrow beginscan$
```

Figure 14.2: Specification algorithm for timestamp system, part II

to be reused. The algorithm will ensure, however, that the labels that are in use at any particular time are totally ordered by the given relation.

The only changes in the code are in the actions: *define-response* of *scan*, in particular, where the order of indices is determined from the scanned labels, and *snap* of *label*, in particular, in determining if the process executing the step has the maximum label, and in determining the next label to choose.

Atomic Labels and Scans

To motivate the value domain chosen, first suppose that each *label* and *scan* operation is done atomically. Consider two processes. We use the label domain consisting of $\{1,2,3\}$, with relation given by the arrows: see Figure 14.3.

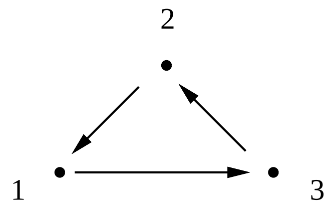


Figure 14.3: 3-element domain for two processes. $a \leftarrow b$ indicates that $a \prec b$.

That is, order relation "$\prec$" is $\{(1,2),(2,3),(3,1)\}$. Suppose that the two processes, p and q, initially both have $label = 1$ (ties are broken by process index). If p does a *label* operation, it sees that q has the maximum (assuming that $p < q$), so when p chooses a new label, it gets the "next label", which is 2. Now if q does a *label*, it sees that the maximum is 2, and therefore chooses the next label, which is now 3. The two, if they take turns, just chase each other around the circle in a leapfrog fashion. It is important to see that the effect is just as if they were continuing to choose from an unbounded space.

What about three processes? We can't simply extend this to a ring of a larger size: consider the following scenario. Suppose one of the processes, say p, retains label 1 and does no new *label* operations, while the other two processes, say q and r, "leapfrog" around the ring. This can yield quite different behavior from the unbounded algorithm, when eventually, q and r bump into p. In fact, we don't even have a definition of how to compare nonadjacent labels in this ring.

A valid solution for the 3 processes case is given in the recursive label structure depicted in Figure 14.4. In this domain, we have three "main" level 1 components, where each of them is constructed from three level 2 components. In each level, the ordering is given by the arrows. A label now consists of a string of two "sub-labels", one for each level. The ordering

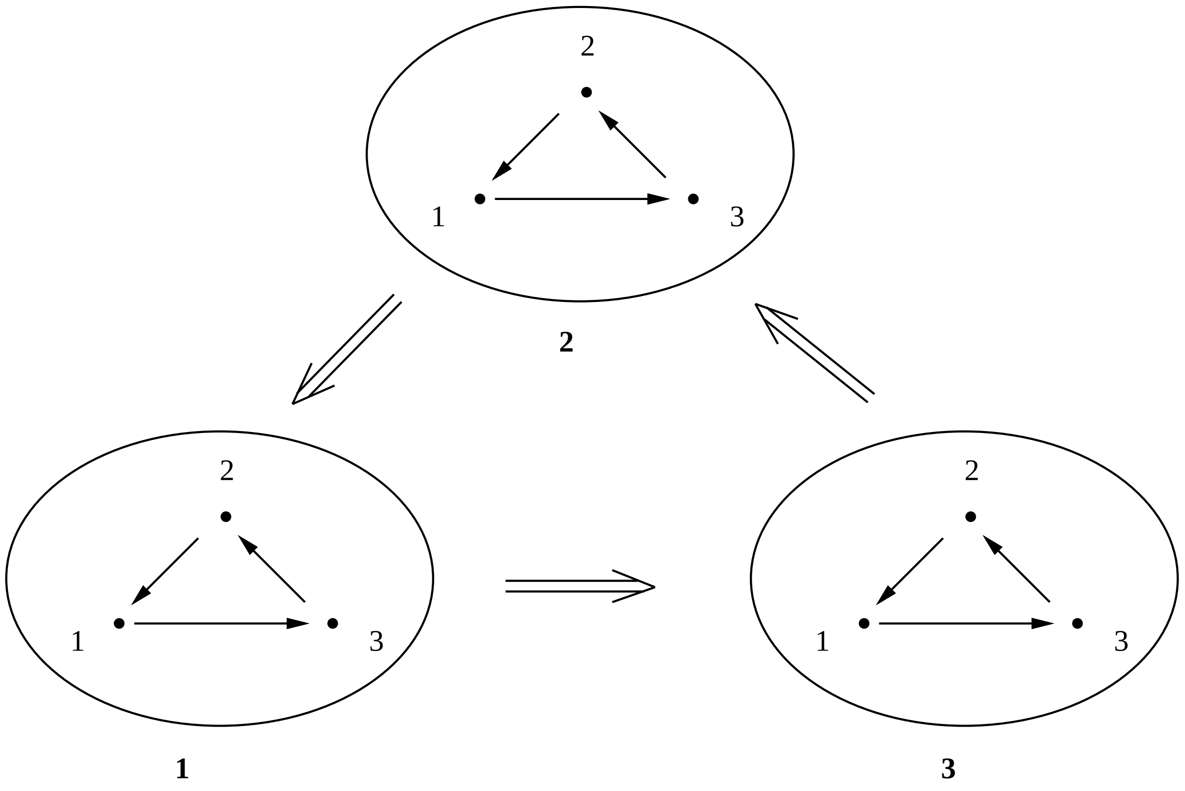


Figure 14.4: 9-element domain for three processes. Each element has a label that consists of two strings of $\{1, 2, 3\}$, and the relation is taken to be lexicographical.

is determined lexicographically. To get a feeling why this construction works, consider again the three-process scenario described above. If p keeps the initial label 1.1, q will choose 1.2 next. But then r will not wrap around and choose 1.3, because the "1-component" of level 1 is already "full" (i.e., it contains 2 processes). So instead, it goes to 2.1, starting in the next component. Thus, the total order of processes is p, q, r, since the given relation relates all three of the pairs of labels involved here (we have $(1.1, 1.2)$ $(1.2, 2.1)$, and $(1.1, 2.1)$ in the relation). In the next step, q would choose 2.2. Then r could choose $(2, 3)$ and still maintain the total order property (because the component is not considered full if the 2 processes it contains include the one now choosing). This can go on forever, since now q and r alone would just cycle within component 2 without violating the requirement.

It is now quite easy to extend this scheme for the general case of n processes. We will have a domain of size 3^{n-1}, where each label is a length $n-1$ string of $\{1, 2, 3\}$, that corresponds to a depth $n-1$ nesting of 3-cycles. We say that a level k component, $1 \leq k \leq n-1$, is "full" if it contains at least $n-k$ processes. An invariant can be proved saying that for any cycle, at any level, at most two of the components are "occupied", which suffices to yield a total order (cf. homework problem).

The following rule is used to pick a new value. If the choosing process is the maximum (i.e., it dominates all the others in the given relation), it keeps its old value. If it is not the maximum, then it looks at the current maximum value. It finds the first level k, $1 \leq k \leq n-1$, such that the level k component of the maximum is full (i.e., contains at least $n-k$ processes' values, excluding the process currently choosing). The chosen value is the *first* label (according to the relation) in the *next* component at level k.

Note that there must be some full level, since fullness at level $n-1$ just means that there is a process (i.e., the maximum itself) in the lowest-level component (viz., the single node containing the maximum value).

It is perhaps helpful to trace this procedure for $n = 4$ processes, with 3 levels of nesting (see Figure 14.5).

If the max's level 1 component is full, it means that it contains all 3 other processes, so we can choose the next level 1 component, since we are guaranteed that no one has a value outside the max's level 1 component. If the max's level 1 component is not full, then there are at most 2 processes in that component. We then look in the level 2 component of the max. If it is full, it contains 2 processes' values, so that means there are none left for the other two level 2 components, and hence it's safe to choose there. Else, there are at most 1 process in the level 2 component. We can repeat the argument for this case to see that is OK to choose the next element within the level 2 component, i.e., the next level 3 component.

Nonatomic Labels

The "recursive triangles" construction above doesn't quite give us what we want. The problems arise when we consider concurrent execution of the label operations. Consider the 3^2 structure with 3 processes. Suppose that we get to the point where p, q and r have labels 1.1, 1.2 and 2.1, respectively (see Figure 14.6).

Suppose that both p and q initiate *label* operations. First, both do their embedded *snap* steps, discovering that the max label in use is 2.1. They both choose label 2.2. Process p writes its new label 2.2, but process q delays writing. Now processes p and r can leapfrog within the 2 component, eventually reaching a point where the occupied labels are 2.3 and 2.1. So far so good, but now let q perform its delayed write. This will make all three labels in component 2 occupied, which will break the total ordering.

In order to handle such race conditions when processes first enter a component, the size 3 ring is modified by adding a "gateway" (see Figure 14.7). The labels for n processes are obtained by nesting recursively to depth $n-1$ as before (see Figure 14.8); again, we say that a level k component, $1 \leq k \leq n-1$, is "full" if it contains at least $n-k$ processes. We use the same choice rule as before, looking for the first level at which the max's component is

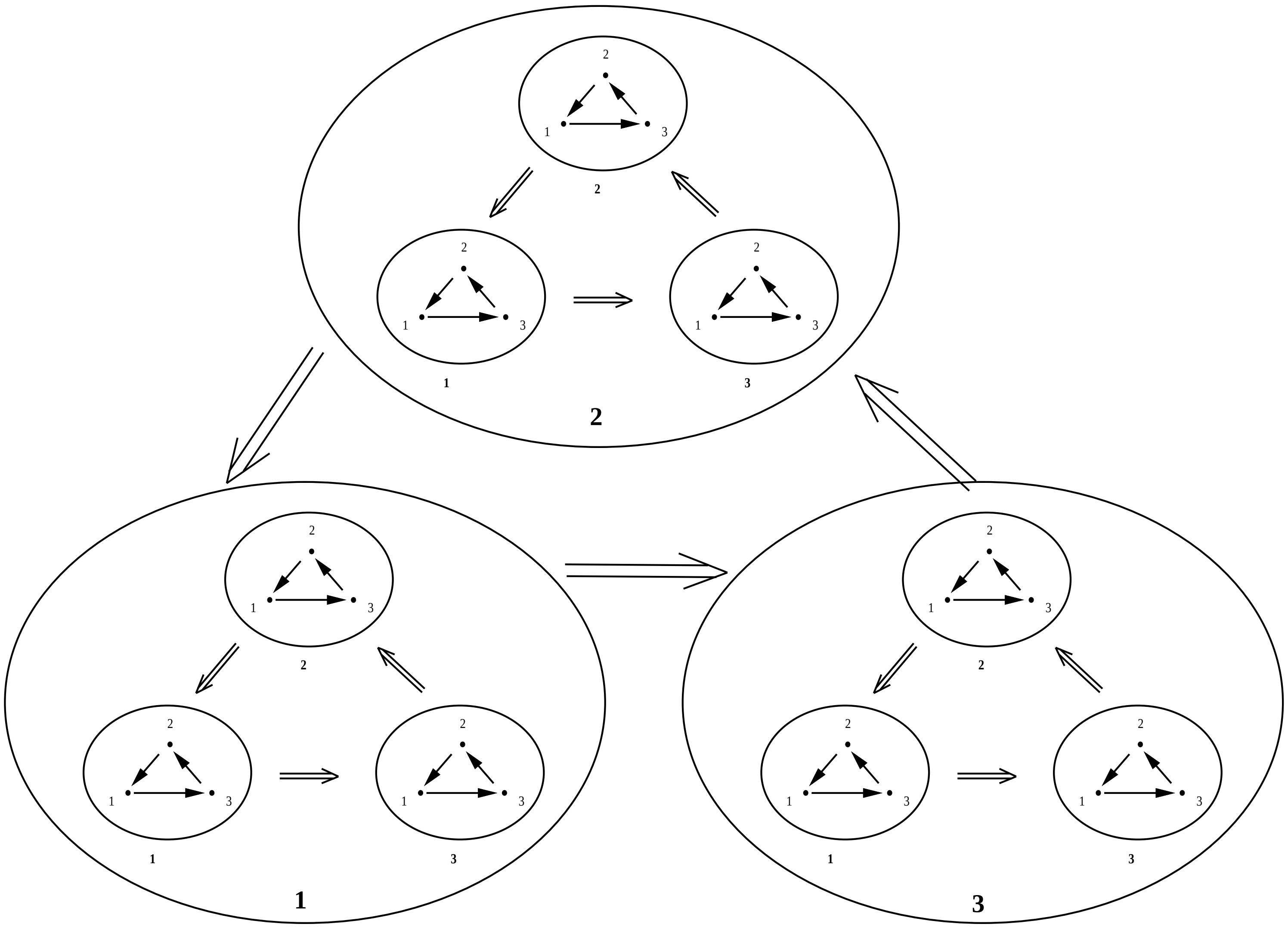


Figure 14.5: 27-elements domain for four processes.

full. Now we claim that this construction works in the concurrent case also.

For intuition, re-consider the previous example, now with the gateway. Suppose that we manage to get the processes to the point where p and q have 3.4 and 3.5, resp., and r has 4.1 (see Figure 14.8).

Processes p and q can choose 4.2 concurrently. Now let p write, and delay q as before. If p and r now leapfrog around the 4 component, they do so around the cycle. But this means that they never get back to the position 4.2, and thus, when q eventually writes there, it will be ordered earlier than the "leapfrogging" processes, and the total order will be preserved. Note that there are now 3 processes in the 4 component, i.e., it is "overfull". But that isn't too harmful, since there are only 2 within the cycle. If q now picks another label, it will

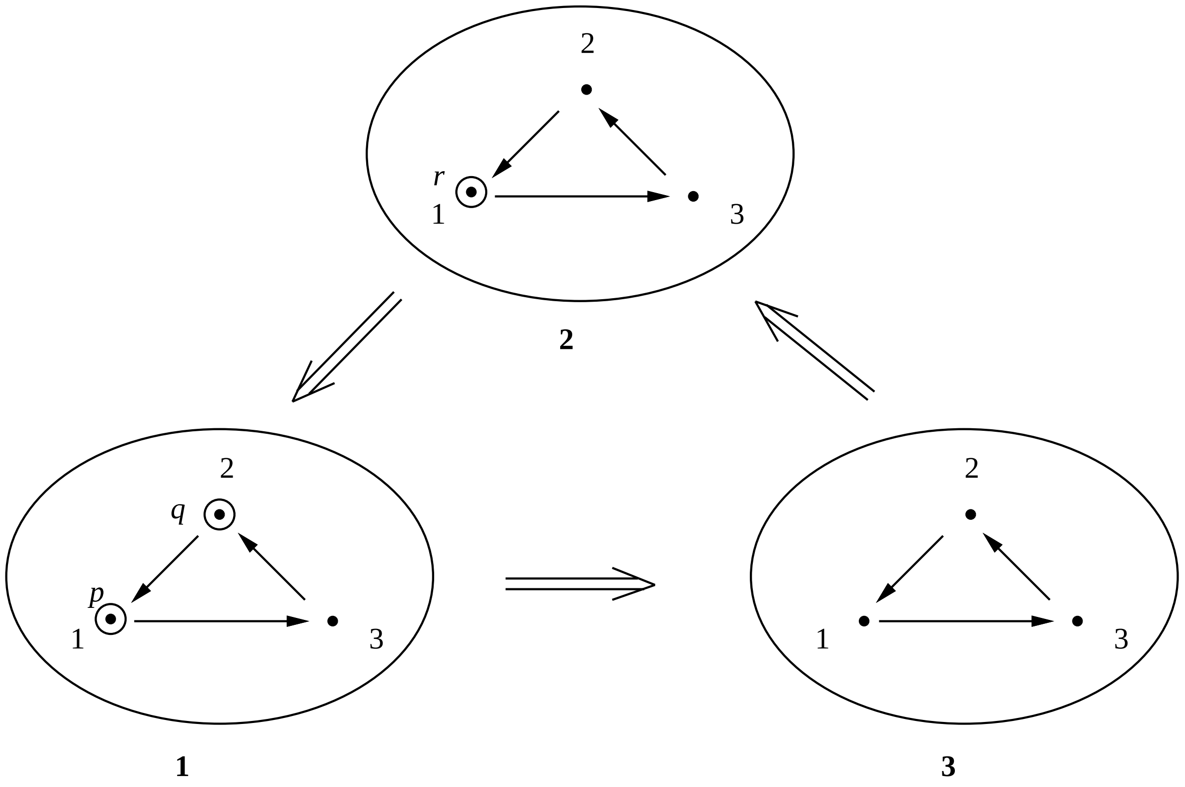


Figure 14.6: initial configuration for counter-example in the concurrent case.

observe that component 4 is already full, and will choose 5.1.

For general n, we claim that although components can become overfull, (i.e., for a level $1 \leq k \leq n-1$, a component could have more than $n-k$ processes), the *cycle* of level k never contains more than $n-k$ processes. For general n, we can prove the invariant that in every cycle at every level, at most two of the three subcomponents are occupied, which implies a total ordering of the labels. We remark that the precise statements of the invariants are a little trickier than this (see below).

14.2.3 Correctness Proof

Gawlick, Lynch and Shavit have carried out an invariant proof for the CTS algorithm. The following invariants are used.

Invariant 1 *Let L be any set of n labels obtained by choosing, for each i, exactly one of $label_i$ and $newlabel_i$. Then L is totally ordered by the label order relation.*

Note that Invariant 1 says not only that the written labels are totally ordered, but also that so are all the pending ones, and any combination of the two kinds. Given this, we can define *lmax* to be the max *label* value, and *imax* to be the largest index of a process having

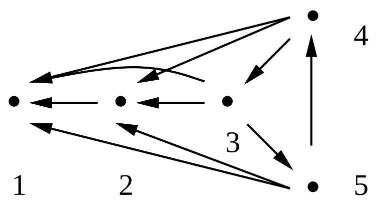


Figure 14.7: structure of one level of the concurrent-case domain.

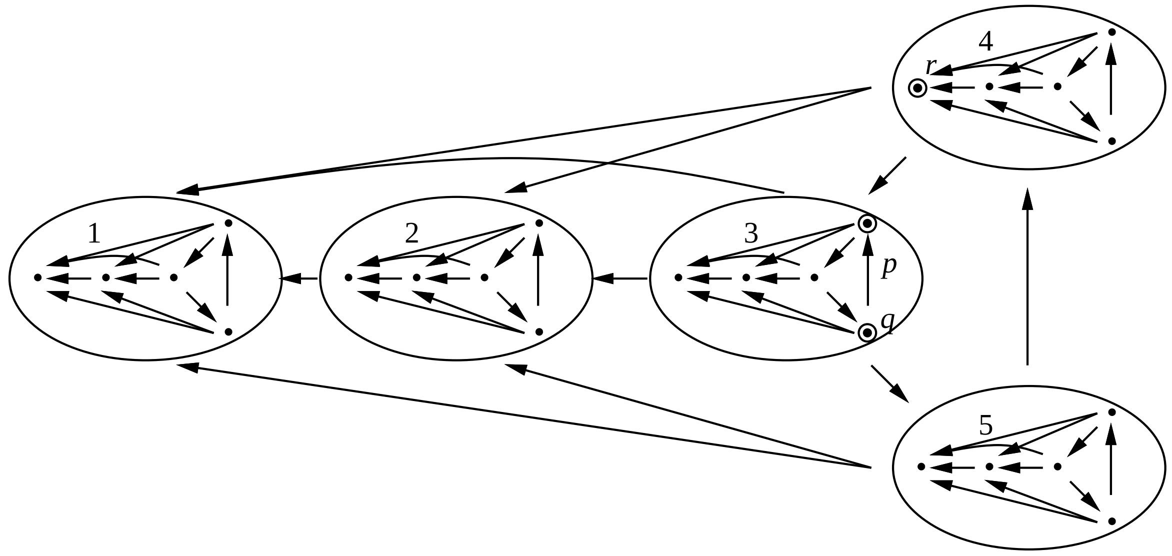


Figure 14.8: initial configuration for concurrent case.

label *lmax*.

Invariant 2 *If* $i = imax$ *then* $newlabel_i = label_i$.

Invariant 2 says that any pending label from the known max is the same as its old label (it would never have had any reason to increase its label if it's the max).

The following invariant describes a property that any pending label that's greater than the current known max must satisfy. Define the *nextlabel* function to yield the very next value at the indicated level.

Invariant 3 *If* $lmax < newlabel_i$ *then* $newlabel_i = nextlabel(lmax, k)$ *for some* k, $1 \leq k \leq n - 1$.

The previous three invariants, 1, 2, and 3, are sufficient to complete the proof. The others are just used to make the induction go through. However, in order to prove these three invariants by induction, we require three additional technical invariants. Thus, for all levels k, $1 \leq k \leq n - 1$, we have the following three technical claims. First, we have one about "old" pending labels; it constrains how different these pending labels can be from the corresponding actual labels.

Invariant 4 *If $newlabel_i \preceq lmax$ and $label_i$ is in the same level k component as $lmax$, then so is $newlabel_i$.*

The next invariant describes some more constraints on the *newlabel* values, based on the corresponding actual labels. This time, the constraint refers to *newlabel* values in the cycle of a component containing the max label.

Invariant 5 *If $newlabel_i$ is in the cycle of the level k component containing $lmax$, then $label_i$ is in that level k component.*

And finally, we have an invariant saying that it is sometimes possible to deduce that the component containing the max label contains lots of processes.

Invariant 6

1. *If $newlabel_i = nextlabel(lmax, k)$ then there are at least $n - k$ processes excluding i whose label values are in the same $k - 1$ component as $lmax$.*

2. *If $lmax$ is not in level k component numbered 1, then there are at least $n - k + 1$ processes whose label values are in the same $k - 1$ component as $lmax$.*

This invariants are proved by induction, as usual. We remark that the proofs are not short – they are fairly long, detailed case analyses.

The upshot of the invariants is that we have the total ordering property preserved, but this doesn't exactly show that we have behavior inclusion.

Recall that we need to show that all well-formed behaviors of the bounded object are also well-formed behaviors of the unbounded object. The right way to show this is to give a formal correspondence between the two objects. Recall the case of the stopping-failure synchronous consensus algorithm, where we showed a formal correspondence between an inefficient but understandable algorithm and an optimized version. We did this using invariants relating the states of the two algorithms when they run using the same inputs and failure pattern. We are going to do essentially the same thing here. Now, however, we don't have any convenient notion analogous to the "(inputs, failure-pattern)" pair to determine the course of the execution. We are in a setting in which there is much more nondeterminism, both in the order in which various actions occur, and in what state changes happen when a particular action happens (consider the choice of the new label in the unbounded object). But note that now, we don't have to show an *equivalence*, saying that everything that either algorithm does the other does also. Rather, we only need a *one-directional relationship*, showing that everything that the bounded algorithm does, the unbounded can also do (this is exactly inclusion of external behaviors).

We shall use the notion of *forward simulation relation* defined in a Lecture 13 applying it to get a simulation from B, the bounded object, to U, the unbounded object. Incidentally, in this case we don't need to use the well-formedness restriction — the correspondence between the objects also holds in the non-well-formed case (although it is not so interesting in that case). We remark that if we only wanted to consider inclusion in the restricted case, say for some other implementation that didn't work in the case of non-well-formed executions, we could just enlarge the system by explicit composition with a "most general" well-formed-preserving environment and prove inclusion for the resulting composed systems.

It turns out that the forward simulation mapping f is quite simple. (Note the similarity in style to the earlier proof for synchronous consensus, with invariants about the pair of states.) We define $u \in f(s)$ provided that the following hold for all i, j, where $i \neq j$:

1. $s.pc_i = u.pc_i$

2. $$\begin{aligned} s.label_i \prec s.label_j &\iff u.label_i < u.label_j \\ s.label_i \prec s.newlabel_j &\iff u.label_i < u.newlabel_j \\ s.newlabel_i \prec s.label_j &\iff u.newlabel_i < u.label_j \\ s.newlabel_i \prec s.newlabel_j &\iff u.newlabel_i < u.newlabel_j \end{aligned}$$

3. $s.order_i = u.order_i$

4. $s.value_i = u.value_i$

5. $s.snapvalue_i = u.snapvalue_i$

In other words, everything is exactly the same except for the two kinds of label values, and in these cases, the ordering relationship is the same. Now we need to show that this correspondence is preserved. Formally, we have to show the conditions of the definition of a simulation relation. To help us, we use Invariants 1, 2, and 3. The interesting step is the *snap* step embedded in a *label* operation, since it is the one that has the potential to violate the nice ordering relationships by choosing *newlabel*. So consider a $snap_i$ step within a *label* operation. If i detects itself to be the max (based on the *label* values that it reads in the snapshot), it doesn't make any changes in either algorithm, so the correspondence is preserved. So suppose that it doesn't detect itself to be the max (which is same in both). So a particular label is chosen in B. We must define some corresponding label in U, which should satisfy the same ordering conditions. We know that this new label will be strictly greater than all the *labels* in B, so we might be tempted to just choose one plus the max of those, for use in U. But we have to also preserve the relative order of this *newlabel* and all the other *newlabels*. There are scenarios in which this new one might not be greater than all the others, in fact, it possibly could be strictly less than some. But Invariant 2 implies that we don't have to worry about the *newlabel*s that are not greater than the maximum *label* in B, and Invariant 3 characterizes the form of the *newlabel*'s that are greater than the maximum *label* in B. Now it is easy to see how the new label now being chosen fits in relative to these, and we can choose a new label in U in the same way.

14.2.4 Application to Lamport's Bakery Algorithm

We can use a CTS algorithm to provide a bounded-register variant of the Lamport's bakery algorithm. We describe the algorithm in Figure 14.9 below in a sort of pseudocode, roughly similar to the original Lamport algorithm, rather than precondition-effects notation.

The invocations of *label* and *scan* are written as single statements, but they denote pairs of invocation and response actions to the CTS object, e.g., $beginlabel_i$ and $endlabel_i$.

Let us go over the differences from the original bakery algorithm. First, instead of explicitly reading all the variables and taking the maximum, we just do a label operation to establish the order (everything is hidden within the CTS object). Second, the structure of the checks is changed a little. Instead of looping through the processes one at a time, the modified algorithm first checks that no one is choosing, and then checks the priorities. Note that the algorithm is not sensitive to this difference. Third, in the loop where we check the priorities, instead of reading the numbers, the modified algorithm at process i does *scan* to obtain the order, and then looks for i to be ahead of everyone else. Note that in this variant, we only check for i to be ahead of those that are currently competing; in the prior algorithm, the number was explicitly set back to 0 when a process left, but now we no longer have sufficient explicit control to assign those numbers. So instead, we leave the order as it is, but explicitly discount those that aren't competing.

To understand the above algorithm, consider it first when the unbounded CTS algorithm is employed. We claim that in this case, similar arguments to those used in the correctness proof for the original bakery algorithm show the same properties, i.e., mutual exclusion, deadlock-freedom, lockout-freedom, and FIFO after a wait-free doorway (now the doorway is the portion of the code from entry to the trying region until the process sets its *flag* to 2). The unbounded variant is more complicated, has worse time complexity, and so far, does not seem to be better than the original algorithm. But now note that we can replace the unbounded CTS object within this algorithm by the bounded CTS object. Since every well-formed behavior of the bounded algorithm is also a well-formed behavior of the unbounded algorithm, we obtained an algorithm that uses bounded-size registers, and still works correctly! Recall that the bounded algorithm is based on snapshot, based on single-writer registers. And the only values that need to be written in the registers are the timestamps, chosen from the bounded domain. Note that this application did not use the value part of the CTS; only the ordering part is used.

Shared variables:

- *flag* : an array indexed by $[1..n]$ of integers from $\{0,1\}$, initially all 0, where $flag[i]$ is written by p_i and read by all
- *CTS* : a concurrent timestamp object

Code for p_i

```
$try_i$
L1:
$flag[i] \leftarrow 1$
$label_i$
$flag[i] \leftarrow 2$
L2:
for $j \in \{1, \ldots, n\}$ do
    if $flag[j] = 1$ then goto L2
    end if
end for
L3:
$scan_i(\prec)$
for $j \in \{1, \ldots, n\}$ do
    if $j \prec i$ and $flag[j] = 1$ then goto L3
    end if
end for
$crit_i$

  **Critical region**
$exit_i$
$flag[i] \leftarrow 0$
$rem_i$

  **Remainder region**
```

Figure 14.9: the bakery algorithm with concurrent timestamps

Lecturer: Nancy Lynch

Lecture 15

15.1 Multi-Writer Register Implementation Using CTS

In this section we show another simple and interesting application of CTS, namely implementing multi-writer multi-reader registers from single-writer multi-reader registers. In this application, we use the value part of the CTS as well as the ordering part.

15.1.1 The Algorithm

The code is very simple:

$read_i$ **process:** $scan_i(o, \bar{v})$, and return v_j, where j is the maximum index in o.

$write_i(v)$ **process:** $label_i(v)$.

The algorithm is straightforward: a *write* simply inserts the new value as part of a *label* action, while *read* returns the latest value, as determined by the order returned by an embedded *scan*. In fact, the algorithm is easier to understand by considering the high level code and the CTS level code together (but keeping the snapshot abstraction instantaneous). In this more elaborate description, the low-level memory consists of a vector of $(value, timestamp)$ pairs. The *write* action snaps all of memory instantaneously, chooses a timestamp bigger than the biggest timestamp already existing in the memory (with the exception that if it already is the biggest, it keeps its value), and (in a separate step) writes back the new value with that new timestamp. The *read* process snaps all of memory instantaneously, and returns the value that is associated with the biggest timestamp.

Note that if the algorithm works correctly using the unbounded CTS algorithm, then it will also work using the bounded algorithm, which can be implemented using bounded space. Thus, it is sufficient to prove that the algorithm works correctly using unbounded timestamps.

It is easy to see that the algorithm has the nice properties of wait-freeness, fairness, and well-formedness. The interesting thing here is to show the existence of the needed serialization points. We do it by first proving the following lemma.

15.1.2 A Lemma for Showing Atomicity

Lemma 1 *Let β be a well-formed sequence of invocations and responses for a read-write register. Let T be any subset of the incomplete operations in β. Let S be the union of the complete operations and T. Suppose that $\prec$ is a partial ordering of all the operations in S, satisfying the following properties.*

1. *Let i be a write operation in S, and let j be any operation in S. Then we have $i \prec j$ or $j \prec i$. In other words, $\prec$ totally orders all the write operations, and orders all the read operations with respect to the write operations.*

2. *The value returned by each complete read operation in S is the value written by the last preceding write operation according to $\prec$ (or the initial value, if none).*

3. *If i is a complete operation and end_i precedes $begin_j$ in β, then it cannot be the case that $j \prec i$.*

4. *For any operation i, there are only finitely many operations j such that $j \prec i$.*

Then β is an atomic read-write register behavior (i.e., satisfies the atomicity condition).

Proof: We construct serialization points for all the operations in S as follows. We insert each serialization point $*_i$ immediately after the later of $begin_i$ and all $begin_j$ such that $j \prec i$. (Note that Condition 4 ensures that the position is well defined.) For consecutive $*$'s, order them in any way that is consistent with $\prec$. Also, for each (incomplete) *read* in T, assign a return value equal to the value of the last *write* operation in S that is ordered before it by $\prec$ (or the initial value, if none).

To prove that this serialization satisfies the requirements, we have to show that

1. the serialization points are all within the respective operation intervals, and

2. each *read* i returns the value of the write whose serialization point is the last one before $*_i$ (or the initial value, if none).

For the first claim, consider the $*_i$ serialization point. By construction, it must be after $begin_i$. If i is a complete operation, it is also before end_i: if not, then end_i precedes $begin_j$ for some j having $j \prec i$, contradicting Condition 3.

For the second claim, consider *read* operation i. By Condition 2 for complete *read*s, and the definition for incomplete *read*s, the value returned by *read* i is the value written by the last *write* operation in S that is ordered before op i by $\prec$ (or the initial value, if none). Condition 1 says that $\prec$ orders all the *write*s with respect to all other *write*s and all *read*s. To show that *read* i returns the value of the *write* whose serialization point is the last one before $*_i$ (or the initial value, if none), it is sufficient to show that the (total) order of the serialization points is consistent with $\prec$, i.e., if $i \prec j$ then $*_i$ precedes $*_j$. This is true since by definition, $*_i$ occurs after the last of $begin_i$ and all $begin_k$, for $k \prec i$, while $*_j$ occurs after the last of $begin_j$ and all $begin_k$ where $k \prec j$. Thus, if $i \prec j$ then for all k such that $k \prec i$ we have $k \prec j$, and hence $*_j$ occurs after $*_i$. ▮

15.1.3 Proof of the Multi-writer Algorithm

We now continue verifying the implementation of multi-writer registers. We define an ordering on all the completed operations and T, where T is the set of all the incomplete *write* operations that get far enough to perform their embedded *update* steps. Namely, order all the *write* operations in order of the timestamps they choose, with ties broken by process index as usual. Also, if two operations have the same timestamp *and* the same process index, then we order them in order of their occurrence — this can be done since one must totally precede the other. Order each complete *read* operation after the *write* whose value it returns, (or before all the *write*s, if it returns the initial value). Note that this can be determined by examining the execution. To complete the proof, we need to check that this ordering satisfies the four conditions. It is immediate from the definitions that Conditions 1 and 2 are satisfied. Condition 4 is left to the reader — we remark only that it follows from the fact that any *write* in T eventually performs its update, and so after that time, other new *write* operations will see it and choose bigger timestamps.

The interesting one is Condition 3. We must show that if i is a complete operation, and end_i precedes $begin_j$, then it is not the case that $j \prec i$.

First, note that for any particular process, the timestamps written in the vector location for that process are monotone nondecreasing. This is because each process sees the timestamp written by the previous one, and hence chooses a larger timestamp (or the same, if it's the maximum). We proceed by case analysis, based on the types of the operations involved.

Case 1: Two writes. First suppose that i and j are performed by different processes. Then in the snap step, j sees the update performed by i (or something with a timestamp at least as big), and hence j chooses a larger timestamp than i's, and gets ordered after i by $\prec$.

Now suppose that i and j are performed by the same process. Then since j sees the update performed by i, j chooses either the same or a larger timestamp then i. If it chooses a larger timestamp, then as before, j gets ordered after i by $\prec$. On the other hand, if it chooses the same timestamp, then the explicit tie-breaking convention says that the two *write*s are ordered by $\prec$ in order of occurrence, i.e., with j after i.

Case 2: Two reads. If *read* i finishes before *read* j starts, then by the monotonicity above,

j sees at least as large timestamps for all the processes as i does. This means that *read* j returns a value associated with a $(\mathit{timestamp}, \mathit{index})$ pair at least as great as that of i, and so j does not get ordered before i.

Case 3: read i, write j. Since i completes before j begins, the maximum $(\mathit{timestamp}, \mathit{index})$ pair that j sees is at least as great as the maximum one that i sees.

First suppose that j does not see its own process with the maximum pair. In this case, j chooses a new timestamp that is strictly larger than the maximum timestamp it sees, so strictly larger than the maximum timestamp that i sees. Now, i gets ordered right after the *write* whose value it gets, which is by definition the maximum pair *write* that it sees. Since this pair has a smaller timestamp than that of j, it must be that i is ordered before j.

On the other hand, suppose that j sees its own process with the maximum pair. In this case, j chooses the same timestamp that it sees. The resulting pair is either greater than or equal to the maximum pair that i sees. If it is greater, then as above, i is ordered before j. If it is equal, then the explicit tie-breaking rule for the same pair implies that *write* j gets ordered by $\prec$ after the *write* whose value i gets, and hence after the *read* by i. This again implies that i gets ordered before j.

Case 4: write i, read j. Since i completes before j begins, j sees a timestamp for i's process that is at least as great as that of operation i, and hence the value that j returns comes from a *write* operation that has at least as great a $(\mathit{timestamp}, \mathit{index})$ pair as does i. So again i is ordered before j.

This sort of case analysis is useful for other read-write register algorithms. Before we conclude the discussion on implementing multi-writer registers from single-writer registers, we remark that this problem has a rich history of of complicated/incorrect algorithms, including papers by Peterson and Burns, by Schaffer, and by Li, Tromp and Vitanyi.

15.2 Algorithms in the Read-Modify-Write Model

In this section we consider shared memory with a different memory access primitive, called *read-modify-write*. Our model is the instantaneous shared memory model, only now the variables are accessible by a more powerful operation. Intuitively, the idea is that a process should be able, in one instantaneous step, to access a shared variable, to use the variable's value and the process' state to determine the new value for the variable, and the new process state. We can formulate this in the invocation-response style by using $\mathit{apply}(f)$ to describe the information needed to update the variable, and the old value v comes back as a response (cf. Homework 5). The response value can be used to an update the state, since input to the process can trigger an arbitrary state change.

In its full generality, this is a very powerful type of shared memory, since it allows arbitrary computation based on the state and shared memory values. It is not exactly clear how this would be implemented – remember that the basic model implies *fair* turns to all the processes, which means *fair* access to the shared memory. So it seems that in this model we are implicitly assuming some low-level arbitration mechanism to manage access to these variables. Still, we might be able to implement the fair access with high probability — if accesses are fast, for example, then interference is unlikely, and by repeated retry, maybe with some backoff mechanism, we would eventually gain access to the variable.

In what follows, we shall consider various problems in this model, including consensus, mutual exclusion, and other resource allocation problems.

15.2.1 Consensus

It is very easy to implement wait-free consensus using a single read-modify-write shared variable as follows. The variable starts out with the value "undefined". All processes access the variable; if a process sees "undefined", it changes the value to its own initial value, and decides on this value. If a process sees 0 or 1, it does not change the value written there, but instead accepts the written value as the decision value. It is easy to verify correctness and wait-freedom for this scheme.

Let us formulate the idea using the *apply*(f) style. Each process uses f_0 or f_1, depending on its initial value, where

$$f_b(x) = \begin{cases} b, & \text{if } x = \textit{undefined} \\ x, & \text{otherwise} \end{cases}$$

An input (i.e., return value) of "undefined" causes *decision* to be set to the initial value, and an input of b causes it to be set to b.

Another style for writing this algorithm is describing flow of control, with explicit brackets indicating the beginning and ending of steps (involving the shared memory and the local state). Since arbitrarily complicated computations can be done in one step, there can be many steps of code within one pair of brackets. There shouldn't be nonterminating loops, however, since a step must always end. The code is given in this style in Figure 15.1, and in the precondition-effect style in Figure 15.2.

In a result by Herlihy, it is proven that it is impossible to obtain a wait-free implementation of atomic read-modify-write objects in the instantaneous shared memory model, where the memory is read-write. This can now be seen easily as follows. If we could solve wait-free consensus in this model, then by using transitivity, applied to the simple solution above for the read-modify-write model and the claimed implementation of read-modify-write objects,

```
$init_i(b)$
$value \leftarrow b$

$lock$
if $x = undefined$ then
   $x \leftarrow b$
   $decision \leftarrow b$
else $decision \leftarrow x$
$unlock$

$decide_i(d)$
```

Figure 15.1: Consensus in the lock-unlock memory style

we could solve wait-free consensus in the instantaneous read-write model, which we have already shown to be impossible. Also, a similar argument implies that we can't get a wait-free implementation of atomic read-modify-write objects in the atomic read-write object model.

15.2.2 Mutual Exclusion

Consider the mutual exclusion problem with the more powerful read-modify-write memory. In some sense, this seems almost paradoxical: it sounds as if we're assuming a solution to a very similar problem – fair exclusive access to a shared variable – in order to get fair exclusive access to the critical region. This seems likely to make the problem trivial. And indeed, it does simplify things considerably, but not completely. In particular, we shall see some nontrivial lower bounds.

Deadlock-Freedom

Recall that for read-write registers, we needed n variables to guarantee only the weak properties of mutual exclusion and deadlock-freedom. To see how different read-modify-write model is from read-write, consider the trivial 1-variable algorithm in Figure 15.3. It is straightforward to see that the algorithm guarantees both mutual exclusion and deadlock-freedom.

State:

- pc, with values $\{init, access, decide, done\}$
- $value$
- $decision$

Actions:

```
$init_i(b)$
  Effect: $value \leftarrow b$
          $pc \leftarrow access$

$access_i$
  Precondition: $pc = access$
  Effect: if $x = undefined$ then
              $x \leftarrow value$
              $decision \leftarrow value$
          else $decision \leftarrow x$
          $pc \leftarrow decide$

$decide_i(d)$
  Precondition: $pc = decide$, and $decision = d$
  Effect: $pc \leftarrow done$
```

Figure 15.2: Consensus algorithm in the precondition-effect language

Shared variables: shared variable v, values $\{0,1\}$, initially 0

Local variables: (for process i) pc, values in $\{R, T1, T2, C, E1, E2\}$, initially R

Code for process i:

```
$try_i$
   Effect: $pc \leftarrow T1$

$test_i$
   Precondition: $pc = T1$
   Effect: if $v = 0$ then
               $v \leftarrow 1$
               $pc \leftarrow T2$

$crit_i$
   Precondition: $pc = T2$
   Effect: $pc \leftarrow C$

$exit_i$
   Effect: $pc \leftarrow E1$

$return_i$
   Precondition: $pc = E1$
   Effect: $v \leftarrow 0$
           $pc \leftarrow E2$

$rem_i$
   Precondition: $pc = E2$
   Effect: $pc \leftarrow R$
```

Figure 15.3: Mutual exclusion algorithm for the read-modify-write model

Bounded Bypass

The simple algorithm of Figure 15.3 does not guarantee any fairness. But we could even get FIFO order (based on the first locally controlled step of a process in the trying region), e.g., by maintaining a queue of process indices in the shared variable. An entering process adds itself to the end of the queue; when a process is at the beginning of the queue, it can go critical, and when it exits, it deletes itself from the queue. This is simple, fast, and only uses one variable, but it is space-consuming: there are more than $n!$ different queues of at most n indices, and so the variable would need to be able to assume more than $n!$ different values, i.e., $\Omega(n \log n)$ bits. One might try to cut down the size of the shared variable. The interesting question is how many states we need in order to guarantee fairness. Can we do it in constant number of values? Linear?

We can achieve fairness with n^2 values ($2 \log n$ bits) by keeping only two numbers in the (single) shared variable: the next "ticket" to be given out, and the number of the "ticket" that has permission to enter the critical region. (This can be viewed as a distributed implementation of the queue from the previous algorithm.) Initially, the value of the shared variable is $(1,1)$. When a process enters, it "takes a ticket" (i.e., copies and increments the first component). If a process' ticket is equal to the second component, it goes to critical region. When a process exits, it increments the second component. This algorithm is also FIFO (based on the first non-input step in trying region). We can allow the tickets to be incremented mod n, with a total of n^2 values.

The obvious question now is whether we can do better. The following theorem gives the answer for the case of bounded bypass (which is a stronger requirement than lockout-freedom).

Theorem 2 *Let A be an n-process mutual exclusion algorithm with deadlock-freedom and bounded bypass, using a single read-modify-write shared variable. Then the number of distinct values the variable can take on is at least n.*

Proof: By contradiction: we shall construct an execution in which some process will be bypassed arbitrarily (but finitely) many times. We start by defining a sequence of finite executions, where each execution is an extension of the previous one, as follows. The first execution α_1 is obtained by letting process p_1 run alone from the start state until it enters C. In the second execution α_2, after α_1 we let p_2 enter the trying region and take one non-input step. (Obviously, p_2 remains in the trying region.) And α_i, for $3 \leq i \leq n$, is defined by starting after the end of α_{i-1}, then letting p_i enter the trying region and take one non-input step.

Define v_i to be the value of the shared variable just after α_i, $1 \leq i \leq n$. We first claim that $v_i \neq v_j$ for $i \neq j$. For assume the contrary, i.e., that $v_i = v_j$, and without loss of generality, assume $i < j$. Then the state after α_i looks like the state after α_j to $p_1, p_2, \ldots, p_i$, since the value of the shared variable and all these processes' states are the same in both. Now, there is a fair continuation of α_i involving $p_1, \ldots, p_i$ only, that causes some process to enter C infinitely many times. This follows from the deadlock-freedom assumption (which only applies in fair executions).

The same continuation can be applied after α_j, with the same effect. But note that now the continuation is not fair: $p_{i+1}, \ldots, p_j$ are not taking any steps, although they are in T. However, running a sufficiently large (but finite) portion of this extension after α_j is enough to violate the bounded bypass assumption — p_j will get bypassed arbitrarily many times by some process. ∎

Is this lower bound tight, or can it be raised, say to $\Omega(n^2)$? Another result in [BFJLP] shows that it can't. This is demonstrated by a *counterexample algorithm* that needs only $n + c$ values for bounded-bypass (where the bound is around 2) mutual exclusion with deadlock-freedom.

The main idea of the algorithm is as follows. The processes in the trying region are divided into two sets, called *buffer* and *main*. When processes enter the trying region, they go into *buffer*, where they lose their relative order. At some time, when *main* is empty, all processes in *buffer* go to *main*, thereby emptying *buffer*. Then processes are chosen one at a time, in an arbitrary order, to go to the critical region.

The implementation of this idea needs some communication mechanisms, to tell processes when to change regions. We'll sketch the ideas here, and leave the details to the reader. Assume, for a start, that we have a *supervisor process* that is always enabled (regardless of user inputs), and that can tell processes when to change regions and go critical. Clearly, the supervisor does not fit within the rules of our models; we'll see how to get rid of it later.

With such a supervisor, we have the following strategy. First, let the variable have 2 components, one for a count $0, \ldots, n$ and one to hold any of a finite set of messages. This is cn values, but we can optimize to $n + c$ by employing a priority scheme to allow preemption of the variable for different purposes. The supervisor keeps counts of number of processes it knows about in the *buffer* and *main* regions. When a process enters, it increments the count in the shared variable to inform the supervisor that someone new has entered, and then waits in *buffer*. The supervisor, whenever it sees a count in the variable, absorbs it in its local *buffer-count* and resets the variable count to 0. Thus, the supervisor can figure out when to move processes to *main*. This is done (sequentially) by putting messages in the message component of the shared variable. We have the following messages:

- ENTER-MAIN: the first process in *buffer* that sees this message can "pick it up", and be thereby told to go to *main*.

- ACK: process response.
- ENTER-CRIT: can be picked up by anyone in *main*. The process that picks it up can go to the critical region.
- ACK: process response.
- BYE: the process in the critical region says it's done and leaves.

Let's see briefly how can we reuse the variable. We need the variable for two purposes: counting, and message delivery. Note that message communication proceeds more or less sequentially (see Figure 15.4 for example).

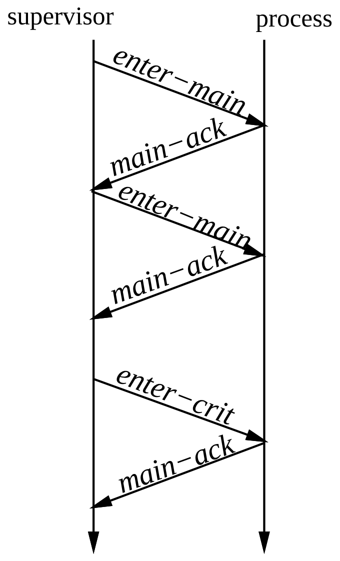


Figure 15.4: A typical communication fragment through the shared variable.

We have an implicit "control thread" here. If this thread is ever "broken" by overwriting a message with a count increase, the rest of the system will eventually quiesce: the supervisor will eventually absorb all counts, and count will become 0. At that point, the over-written message could be replaced (count = 0 will be default if message is there). More specifically, the following occurs. When a process that enters the system sees a message in the variable, it picks it up and puts down a count of 1 to announce its presence. This process holds the message until count is 0 again, and then replaces the message it is holding in the shared variable.

Now we turn back to our model, in which there is no supervisor. The idea is to have a "virtual supervisor" simulated by the processes. E.g., at any time, the process that controls C will be the "designated supervisor", and it must pass responsibility on when it leaves C. This involves passing on its state information, and we need to use the variable to communicate this too. This can be done by using the variable as a message channel, where we again optimize multiple uses of message channel as before. Note that if ever there's no other process to pass it to, it means that no one is waiting. But then there's no interesting information in the state anyway, so there is no problem in "abandoning responsibility", when the process leaves and puts an indicator in the variable.

Lockout-Freedom

The lower bound of Theorem 2 can be beefed up to get a similar lower bound for lockout-freedom [BFJLP]. This is harder, because lockout freedom, unlike bounded bypass, is a property of (possibly infinite) fair executions, and not only finite executions. The bad executions that are constructed must be fair.

We can get a lower bound of $n/2$, by a complicated construction with an extra assumption, that processes don't remember anything when they return to their remainder regions. Some tries were made to raise the bound to n, since it seems as if an algorithm has to have sufficiently many different values of the variable to record any number of entering processes. But the search for a lower bound was halted by another clever counterexample algorithm, giving lockout-freedom using only $n/2 + c$ values.

The idea that algorithm is as follows. Similarly to the $n+c$ value algorithm, the algorithm has incoming processes increment a count, which now only takes on values $0, \ldots, n/2$, and then wraps around back to 0. The count is absorbed by a (conceptual) supervisor, as before. If it wraps around to 0, it seems like a group of $n/2$ processes is hidden. But this is not quite the case: there's someone who knows about them — called the "executive" — the one that did the transition from $n/2$ to 0. The executive can send SLEEP messages to (an arbitrary set of) $n/2$ processes in *buffer*, to put them to sleep. It doesn't matter which processes in *buffer* go to sleep. Then, having removed $n/2$ from the fray, the executive reenters the system. Now the algorithm runs exactly as the bounded bypass algorithm, since it can't overflow (but it only gets up to count $n/2$). When the executive reaches C, it can take care of the sleepers by sending them messages to awaken, and telling the supervisor about them. Again, we must share the variable, now with a more complicated priority scheme.

Note: we can't have 2 executives overlapping and confusing their sleepers, because it's not possible for that many to enter while the others are asleep.

Lecturer: Nancy Lynch

Lecture 16

16.1 Read-Modify-Write Algorithms (cont.)

16.1.1 Mutual Exclusion (cont.)

Last time, we described read-modify-write algorithms for mutual exclusion with bounded bypass, and mutual exclusion with lockout-freedom. We showed that we can get algorithms that use a linear number of values for either. Today, we conclude this topic with one more lower bound result.

Though we will not give here the full $n/2$ lower bound for lockout-freedom, (too complicated!), we will show the kinds of ideas that are needed for such a proof. We shall only sketch a weaker lower bound for lockout-freedom, of approximately $\sqrt{n}$. Specifically, we prove the following theorem.

Theorem 1 *Any system of $n \geq \frac{k^2-k}{2} - 1$ or more processes satisfying mutual exclusion and lockout-freedom requires at least k values of the shared variable.*

Proof: By induction on k. The base case $k = 2$ is trivial. Assume now that theorem holds for k; we show that it holds for $k+1$. Suppose $n \geq \frac{(k+1)^2-(k+1)}{2} - 1$, and suppose for contradiction that number of shared values is no more than k. From the inductive hypothesis, it follows that the number of values is exactly k. We now construct a bad execution to derive a contradiction.

Define an execution α_1 by running p_1 alone until it enters C. Define further an execution α_2 by running p_2 after α_1, until p_2 reaches a state with shared variable value v_2, such that p_2 can run on its own, causing v_2 to recur infinitely many times. Such a state must exist since x can assume only finitely many values. Likewise, get α_i, for $3 \leq i \leq n$, by running p_i after α_{i-1} until it reaches a point where it could, on its own, cause the current value of the shared variable x to recur infinitely many times. Let v_i be the value corresponding to α_i. Since there are only k values for x, by the pigeonhole principle, there must exist i and j, where $n-k \leq i < j \leq n$, such that $v_i = v_j$.

Now, processes $p_1, \ldots, p_i$ comprise a system with at least $\frac{k^2-k}{2} - 1$ processes, solving mutual exclusion with lockout-freedom. Thus, by induction, they must use all k values of shared memory. More specifically, for every global state s that is i-reachable from the state q_i just after α_i, and every value v of x, there is a global state reachable from s in which the value of x is v. (If not, then we could run the system to global state s, then starting from there we have a reduced system with fewer than k values, contradicting the induction hypothesis.) This implies that there exists a fair of execution of processes $p_1, \ldots, p_i$ that starts from q_i, such that all k values of shared memory recur infinitely many times.

Now we can construct the bad execution as follows. First run $p_1, \ldots, p_j$ through α_j, to state q_j, which looks like q_i to $p_1, \ldots, p_i$. Then run $p_1, \ldots, p_i$ as above, now from q_j instead of q_i, in the execution in which each shared value recurs infinitely often. Now recall that in q_j, each process $p_{i+1}, \ldots, p_j$ is able to cause the value it sees to recur infinitely many times. So intersperse steps of the main run of $p_1, \ldots, p_i$ with steps of $p_{i+1}, \ldots, p_j$ as follows: each time the shared variable is set to some v_l, $i+1 \le l \le j$, run p_l for enough steps (at least one) to let it return the value to v_l. This yields an infinite execution that is fair to *all* processes and that locks out, for example, p_j. ▮

The key idea to remember in the above proof is the construction of bad fair executions by splicing together fair executions. Also, note the apparent paradox: there is a nontrivial inherent cost to lockout-free mutual exclusion, even though we are assuming fair exclusive access to a shared variable, which seems very close.

16.1.2 Other Resource Allocation Problems

Mutual exclusion is only one kind of resource allocation problem, modeling access to a single shared resource. In this section we generalize the problem. There are two ways to describe the generalizations. The first is exclusion problems, and the second is resource problems. We start with the new definitions.

Definitions

In the first variant, we describe *exclusion problems*. Formally, we describe these problems by a collection $\mathcal{E}$ of "conflicting sets of processes" that are not allowed to coexist in C. $\mathcal{E}$ could be any collection of sets of processes that is closed under containment, i.e., if $S \in \mathcal{E}$ and $S' \supseteq S$ then $S' \in \mathcal{E}$. For example, the mutual exclusion problem is characterized by

$$\mathcal{E} = \{S : |S| \ge 2\} .$$

A common generalization is the *k-exclusion problem*, modeling k shared resources, where at most k processes may be in the critical region at any given time. In our language, the problem is defined by

$$\mathcal{E} = \{S : |S| \ge k+1\} .$$

Clearly, we can be more arbitrary, e.g., say $\mathcal{E}$ is defined to contain $\{1,2\}, \{1,3\}, \{2,4\}, \{3,4\}$ and other containing subsets. But note that in this case, 1 doesn't exclude 4, and 2 doesn't exclude 3.

This leads us to the second variant of the definition, in terms of *resources*. In this case, we have a set of explicit resources in the problem statement, and for each p_i, we give a list of *requirements*, in the form of a monotone Boolean formula, e.g., $(R_1 \wedge R_2) \vee (R_3 \wedge R_4)$, where the R_i's are resources. Intuitively, the meaning of the example is that p_i "needs" either R_1 or R_2, and either R_3 or R_4 in order to go critical. More formally, a set of resources is *acceptable* for a process to go critical iff the setting all the corresponding variables to true yields a satisfying assignment for the formula.

Let's consider another simple example to illustrate this style of formulation. Suppose p_1 wants resources R_1 and R_4, p_2 wants R_2 and R_3, p_3 wants R_3 and R_4, and p_4 wants R_4 and R_1. In this example, the corresponding formulae have only a single clause (i.e, there are no options here).

Note that any resource problem gives rise to an exclusion problem, where the combinations excluded by the formulas cannot be simultaneously satisfied. E.g., for the example above, we get an equivalent formulation by

$$\mathcal{E} = \{S : S \supseteq \{1,2\}, \{2,1\}, \{2,3\}, \{3,4\}, \{4,1\}\}$$

It is also true that every exclusion condition can be expressed as a resource problem, and thus these formulations are equivalent.

For stating a general problem to be solved in concurrent systems, we again assume the cyclic behavior of the requests as before. The mutual exclusion condition is now replaced by a general exclusion requirement. The definitions of progress and lockout-freedom remain the same as before. We actually want to require something more, since there are some trivial solutions that satisfy the requirements listed so far: any solution to mutual exclusion is *a fortiori* a solution to more general exclusion. But there's something wrong with this — we want to insist that *more concurrency* is somehow possible. We don't know what is the best way to express this intuitive notion. We give here a slightly stronger condition in terms of the resource formulation as follows.

> Consider any reachable state in which process i is in T, and every process that has *any of the same-named resources* in its resource requirement formula is in R. Then if all the conflicting processes stay in R and i continues to take steps, eventually i proceeds to C. (Note: We allow for other processes to halt.)

Dijkstra had originally defined a special case, called the *Dining Philosophers* problem, described below as a resource problem.

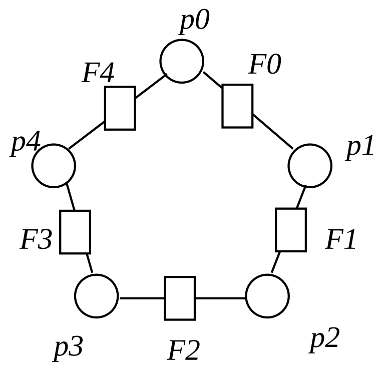


Figure 16.1: The dining philosophers problem for $n = 5$.

In the traditional scenario (see Figure 16.1), we have five (say) philosophers at a table, usually thinking (i.e., in R). From time to time, each philosopher wants to eat (i.e., to enter C), which requires two forks (representing the resources); we assume that each philosopher can only take the forks adjacent to him (or her). Thus, the formula for p_i is $F_{i-1} \wedge F_i$ (counting mod 5). We can represent this as an exclusion problem, namely:

$$\mathcal{E} = \{S \ : \ S \supseteq \{F_i, F_{i+1}\} \text{ for some } 0 \leq i < 5\} \ .$$

We assume that there is a shared variable associated with each fork, and the access model for the variables is read-modify-write. If the processes are all identical and they don't use their indices, then we have the same kind of symmetry problem we saw in electing a leader in a ring, which cannot be solved. To be concrete, suppose they all try to pick up their left fork first, and then their right. This satisfies the exclusion condition, but it can deadlock: if all wake up at the same time and grab their left forks, then no one can proceed (this is worse than a deadlocking execution — the system actually wedges). Thus, we must use some kind of symmetry breaking technique, and here we make the assumption that the process indices (i.e., locations) are known to the processes.

Dijkstra's original solution requires simultaneous locking of the two adjacent forks, a centralized shared read-modify-write variable, and has no fairness. We'll see a better solution below.

The Left-Right Algorithm

This algorithm appeared in Burns paper, but it is generally considered as "folklore". This algorithm ensures exclusion, deadlock-freedom (in the stronger version stated above), and lockout-freedom. Also, it has a good time bound, independent of the size of the ring.

This is a simple style of algorithm based on running around and collecting the forks, one at a time. We have to be careful about the order in which this is done. For instance, if the order is arbitrary, then we risk deadlock as above. Even if there is no danger of deadlock, we might be doing very poorly in terms of time performance. E.g., if everyone gets their forks in numerical order of forks, then although they don't deadlock, this might result in a long *waiting chain*. To see this, consider the following scenario. In Figure 16.1 above, suppose that p_4 gets both forks, then p_3 gets its right fork, then waits for its left, p_2 gets its right fork, then waits for its left, p_1 gets its right fork, then waits for its left, and p_0 waits for its right fork. (See Figure 16.2 for final configuration).

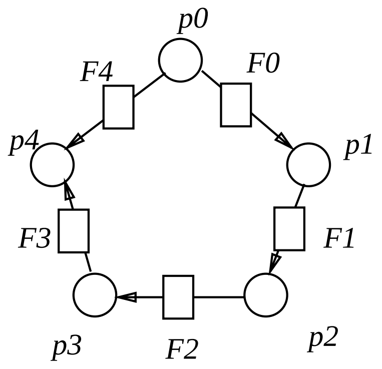


Figure 16.2: A configuration with long waiting chain. Arrows indicate possession of forks.

In the last state we have a chain where p_0 waits for p_1, which waits for p_2, which waits for p_3, which waits for p_4. The processes in the waiting chain can go into the critical region only sequentially. In general, this means that time can be at least linear in the size of the ring.

In contrast, the Burns algorithm has a constant bound, independent of the size of the ring. The algorithm accomplishes this by avoiding long waiting chains. We describe the algorithm here for n even. The variant for odd n is similar, and is left as an exercise. The algorithm is based on the following simple rule: even numbered processes pick up their left forks first, and odd numbered processes get their right forks first. Thus, the asymmetry here is the knowledge of the parity. Variables correspond to the forks, as before. Now, each variable contains a queue of processes waiting for the fork, of length at most 2. The first process on the queue is assumed to own the fork. The code is given in Figure 16.3.

Properties. Mutual exclusion is straightforward. Deadlock-freedom follows from lockout-freedom, which in turn follows from a time bound, which we prove now. Assume as usual that the step time at most s, and that the critical region time at most c. Define $T(n)$ to be the worst time from entry to trying region until entry to critical region (in an n-process ring). Our goal is to bound $T(n)$. As an auxiliary quantity, we define $S(n)$ to be the worst time from when a process has its first fork until entry to critical region.

```
$try_i$
  Effect: $pc \leftarrow$ $look\text{-}left$

$look\text{-}left$
  Precondition: $pc = look\text{-}left$
  Effect: if $i$ is not on $x_i.queue$ then add $i$ to end
          else if $i$ is first on $x_i.queue$ then $pc \leftarrow look\text{-}right$

$look\text{-}right$
  Precondition: $pc = look\text{-}right$
  Effect: if $i$ is not on $x_{i-1}.queue$ then add $i$ to end
          else if $i$ is first on $x_{i-1}.queue$ then $pc \leftarrow before\text{-}C$

$crit_i$
  Precondition: $pc = before\text{-}C$
  Effect: $pc \leftarrow C$

$exit_i$
  Effect: $return - set \leftarrow \{L, R\}$
          $pc \leftarrow return$

$return\text{-}left$
  Precondition: $pc = return$
          $L \in return\text{-}set$
  Effect: $return\text{-}set \leftarrow return\text{-}set - \{L\}$
          remove $i$ from $x_i.queue$

$return\text{-}right$
  Precondition: $pc = return$
          $R \in return\text{-}set$
  Effect: $return\text{-}set \leftarrow return\text{-}set - \{R\}$
          remove $i$ from $x_{i-1}.queue$

$rem_i$
  Precondition: $pc = return$
          $return\text{-}set = \emptyset$
  Effect: $pc \leftarrow rem$
```

Figure 16.3: Left-right dining philosophers algorithm: code for process p_i, i even.

Let us start by bounding $T(n)$ in terms of $S(n)$: p_i tests its first fork within time s of when it enters T. When p_i first tries, either it gets its first fork immediately or not. If so, then $S(n)$ time later, it goes critical. If not, then the other process has the first fork. At worst, that other process just got the fork, so it takes at most $S(n) + c + 2s$ until it gets to C, finishes C, and returns both forks. Then since p_i recorded its index in the queue for its first fork, within additional time s, p_i will re-test the first fork and succeed in getting it, and then in additional $S(n)$ time p_i reaches C. This analysis says that

$$T(n) \leq s + \max\{S(n), S(n) + c + 2s + s + S(n)\} = 4s + c + 2S(n) \ . \tag{16.4}$$

We now bound $S(n)$: p_i tests its second fork within time s of when it gets the first fork. Again we need to consider two cases. If it gets the second fork immediately, then within an additional time s, p_i goes critical. If not, then the other process has it, and because of the arrangement, since it's p_i's second fork, it must be its neighbor's second fork also. So within time at most $s + c + 2s$, the neighbor gets to C, leaves, and returns both forks. Then within additional time s, p_i will retest the second fork and succeed in getting it. Then additional time s will take p_i to C. This analysis says that

$$S(n) \leq s + \max\{s, s + c + 2s + s + s\} = 6s + c \ . \tag{16.5}$$

Putting Eq. (16.4) and (16.5) together, we conclude that

$$T(n) \leq 4s + c + 2(6s + c) = 16s + 3c \ .$$

Note that the bound is independent of n. This is a very nice property for a distributed algorithm to have. It expresses a strong kind of locality, or "network transparency".

Extension to More General Resource Problems

Due to the nice features of the Left-Right algorithm, it is natural to ask whether can we extend this alternating left-right idea to more general resource allocation problems. In this section we show an extension that is not completely general, but rather applies only to *conjunctive* problems, where the resource requirement of a process is exactly one particular set. (This case does not cover k-exclusion, for example, which is a disjunctive problem, with more than one alternative allowed).

Again, for each resource we have an associated variable, shared by all processes that use that resource. The variable contains a FIFO queue to record who's waiting (as before). As before, the processes will wait for the resources one at a time. To avoid deadlock, we arrange resources in some global total ordering, and let each process seek resources in order according to this ordering, smallest to largest. It is easy to see that this prevents deadlock: if process

i waits for process j, then j holds a resource which is strictly larger than the one for which i is waiting, and hence the process holding the largest-numbered resource can always advance. The FIFO nature of the queues also prevents lockout.

However, the time performance of this strategy is not very good, in general. There is no limit on the length of waiting chains (except for the number of processes), and so the time bounds can be linearly dependent on the number of nodes in the network. We saw such an example above, for rings. So we must refine the ordering scheme by giving a way to choose a good total ordering, i.e., one that doesn't permit long waiting chains. We use the following strategy for selecting a good total order. Define the *resource problem graph*, where the nodes represent the resources, and we have an edge from one node to another if there is a user that needs both corresponding resources. See Figure 16.4 for an example of Dining Philosophers with 6 nodes.

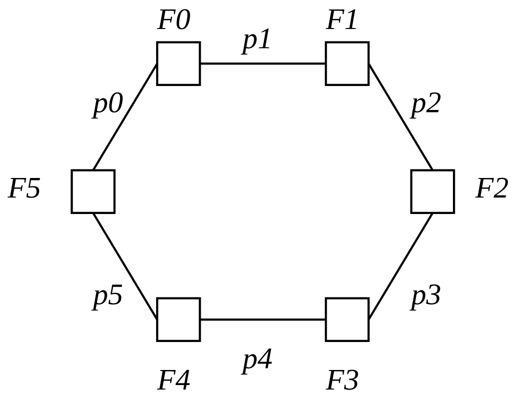


Figure 16.4: Resource graph for six dining philosophers.

Now *color* the nodes of the graph so that no two adjacent nodes have the same color. We can try to minimize the number of colors: finding minimal coloring is NP-hard, but a greedy algorithm is guranteed to color the graph in no more than $m+1$ colors.

Now totally order the colors. This only partially orders the resources, but it does totally order the resources needed by any single process. See Figure 16.5 for an example.

Now each process accesses the forks in increasing order according to the color ordering. (Equivalently, take any topological ordering of the given partial ordering, and let all processes follow that total ordering in the earlier strategy.) Carrying on with our example, this boils down to the alternating left-right strategy.

It is easy to see that the length of waiting chains is bounded by the number of colors. This is true since a process can be waiting for a resource of minimum color, which another process has. That process could only be waiting for a resource of "larger" color, etc.

The algorithm has the nice property that the worst-case time is not directly dependent

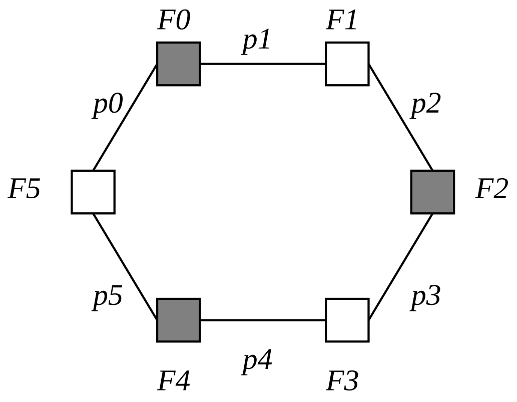


Figure 16.5: coloring.

on the total number of processes and resources. Rather, let s be an upper bound on step time, and c an upper bound on critical section time, as before. Let k be the number of colors, and let m be the maximal number of users for a single resource. We show that the worst case time bound is $O\left((m^k)c + (km^k)s\right)$. That is, time depends only on "local" parameters. (We can make a reasonable case that in a realistic distributed network, these parameters are actually local, not dependent on the size of the network.) Note that this a big bound — it is exponential in k, so complexity is not proportional to the length of the waiting chain, but is actually more complicated. There are some complex interleavings that get close to this bound.

To prove the upper bound, we use the same strategy is as before. Define $T_{i,j}$, for $1 \leq i \leq k$ indicating colors, and $1 \leq j \leq m$ indicating positions on a queue, to be the worst-case time from when a process reaches position j on any queue of a resource with color i, until it reaches C. Then set up inequalities as for the Left-Right algorithm. The base case is when the process already has the last resource:

$$T_{k,1} \leq s \tag{16.6}$$

Also, when a process is first on any queue, within time s it will be at worst at position m on the next higher queue:

$$T_{i,1} \leq s + T_{i+1,m} \quad \text{for all } i \leq k \tag{16.7}$$

And lastly, when a process in in position $j > 1$ on queue i, it only has to wait for its predecessor on the queue to get the resource and give it up, and then it gets the resource:

$$T_{i,j} \leq T_{i,j-1} + c + ks + T_{i,1} \quad \text{for all } i \text{ and for all } j > 1 \tag{16.8}$$

The claimed bound follows from solving for $T_{1,m}$ (after adding an extra s).

Cutting Down the Time Bound This result shows that a very general class of resource allocation problems can have time independent of global network parameters. But it would be nice to cut down the time bound from exponential to linear in the number of colors. There are some preliminary results by Awerbuch-Saks, and Choy-Singh. But there is probably still more work to be done.

16.2 Safe and Regular Registers

In this section we switch from the very powerful read-modify-write shared memory model to some much weaker variants. We start with some definitions.

16.2.1 Definitions

We now focus on read-write objects (registers) again, and consider generalizations of the notion of an atomic read-write register: *safe* and *regular* registers. These variants are weaker than atomic registers, but can still be used to solve interesting problems. They fit our general object definition, in particular, inputs are $read_{i,x}$ and $write_{i,x}(v)$, with outputs as before. The registers are modeled as I/O automata as in Figure 16.6.

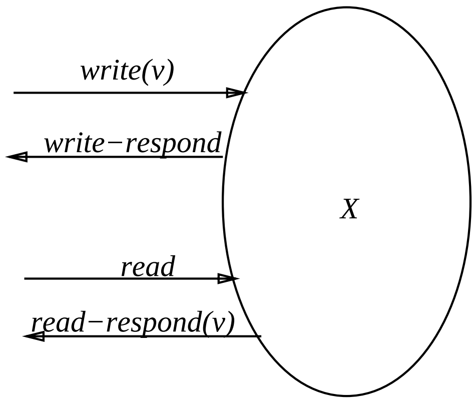


Figure 16.6: Concurrent Read/Write Register X. Subscripts mentioned in the text are omitted from the diagram.

As before, the index i on the read and write operations corresponds to a particular line on which a process can access the register. Also as before, we assume that operations on each line are invoked sequentially, i.e., no new operations are invoked on a line until all previous operations invoked on that line have returned. But otherwise, operations can overlap.

Recall the definition of atomic registers. An atomic register has three properties: it preserves well-formedness, it satisfies the atomicity condition, and fairness implies that operations complete. Also, depending on the architecture of the implementation, we can have a wait-free condition. Now we generalize the second (atomicity) condition, without changing the other conditions. This still fits the general object spec model, so the modularity results hold; this allows us to build objects hierarchically, even preserving the wait-free property.

For these weaker conditions, we only consider single-writer registers. This means that writes never overlap one another. We require in all cases that any read that doesn't overlap a write gets the value of the closest preceding write (or the initial value if none). This is what would happen in an atomic register. The definitions differ from atomic registers in allowing for more possibilities when a read does overlap a write.

The weakest possibility is a *safe register*, in which a read that overlaps a write can return an *arbitrary* value. The other possibility is a *regular register*, which falls somewhere in between safe and atomic registers. As above, a read operation on a regular register returns the correct value if it does not overlap any write operations. However, if a read overlaps one or more writes, it has to return the value of the register either before or after any of the writes it overlaps.

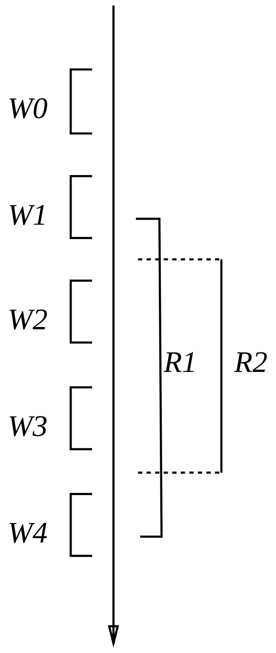


Figure 16.7: Read Overlapping Writes

For example, consider the scenario in Figure 16.7. The set of *feasible writes* for R_1 is $\{W_0, W_1, W_2, W_3, W_4\}$ because it is allowed to return a value written by any of these write operations. Similarly, the set of feasible writes for R_2 is $\{W_1, W_2, W_3\}$. The important difference between regular and atomic registers is that for regular registers, consecutive reads can return values in inverted order.

16.2.2 Implementation Relationships for Registers

Suppose we only have safe registers because that's all our hardware provides, and we would like to have atomic registers. We shall show that in fact, we can start with safe, single-reader single-writer, binary registers, and build all the way up to atomic, multi-reader multi-writer, k-ary registers for arbitrary k. Lamport carries out a series of wait-free constructions, starting with the simplest registers and ending up with single-reader single-writer k-ary registers. Others have completed the remaining constructions. We have already seen how to get multi-writer multi-reader registers from single-writer multi-reader registers (although there is room for improved efficiency). Also, Singh, Anderson, and Gouda have given an implementation of single-writer multi-reader registers from single-writer single-reader registers.

In this general development, some of the algorithms are logically complex and/or time-consuming, so there is some question about their practicality.

The general development we consider here can be formalized in the object spec model. So we get transitivity as discussed earlier. Recall the architecture.

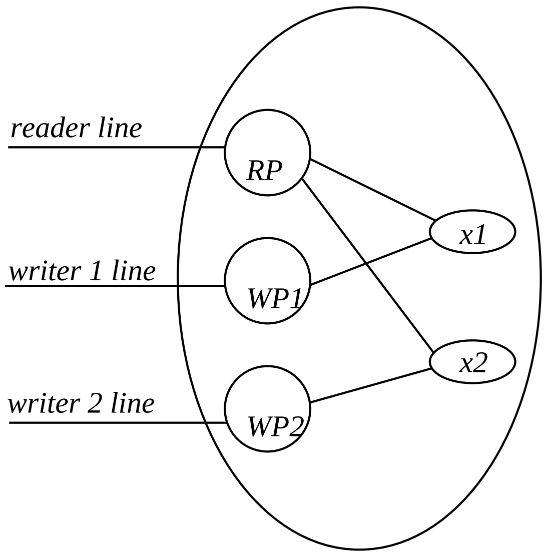


Figure 16.8: Example: Implementing a 2-writer 1-reader register with two 1-writer 2-reader registers

In Figure 16.8 we see a particular implementation of a 2-writer 1-reader register in terms of two 1-writer 2-reader registers. There is a process for each line of the register being implemented, i.e., for each of the high-level writers and the high-level reader. These processes are connected to the low-level registers in such a way that each line of a low-level register is connected to only one process.

In general, we call the high-level registers *logical* registers and the low-level registers *physical* registers. In all of the constructions we will consider, a logical register is constructed from one or more physical registers. Each input line to the logical register is connected to a process. These processes in turn are connected to one or more of the physical registers using internal lines. Exactly one process is connected to any internal line of a physical register. This permits the processes to guarantee sequentiality on each line. Processes connected to external write lines are called write processes, and processes connected to external read lines are called read processes. Note that nothing prevents a read process from being connected to an internal write line of a physical register, and vice versa.

We follow Lamport's development. We have safe vs. regular vs. atomic; single-reader vs. multi-reader; and binary vs. k-ary registers. We thus consider the twelve different kinds of registers this classification gives rise to, and see which register types can be used to implement which other types. The implementation relationships in Figure 16.9 should be obvious, because they indicate types of registers that are special cases of other types.

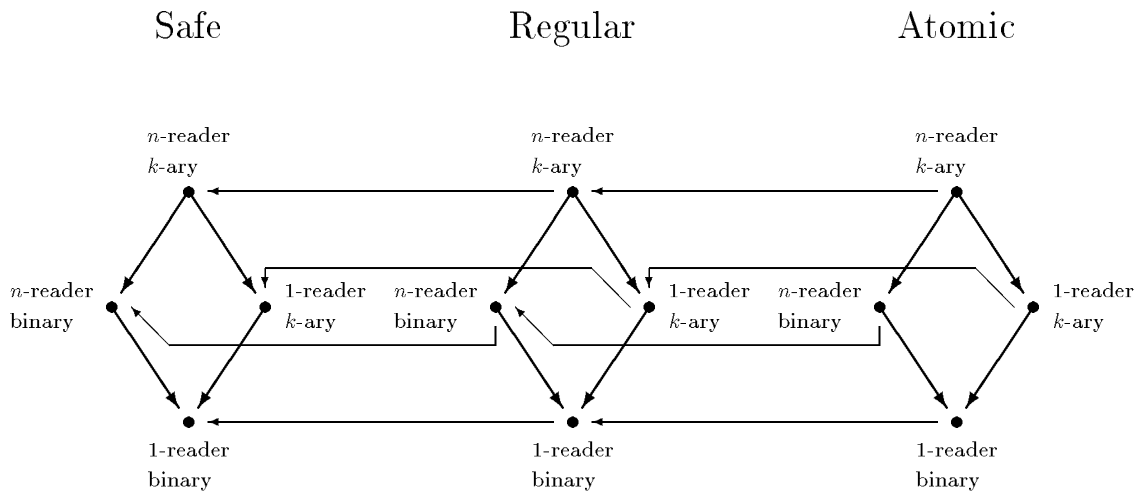


Figure 16.9: Obvious implementation relationships between register types. An arrow from type A to type B means that B can be implemented using A

.

16.2.3 Register Constructions

Lamport presents five constructions to show other implementation relationships. In the following constructions, actions on external lines are always specified in upper-case, whereas actions on internal lines are specified in lower-case. For example, *WRITE* and *READ* denote external operations whereas *write* and *read* denote internal operations.

N-Reader Registers from Single-Reader Registers

The following construction (Construction 1 in the paper) implements an n-reader safe register from n single-reader safe registers. The same algorithm also implements an n-reader regular register from n single-reader regular registers. The write process is connected to the write lines of all n internal registers as in Figure 16.10.

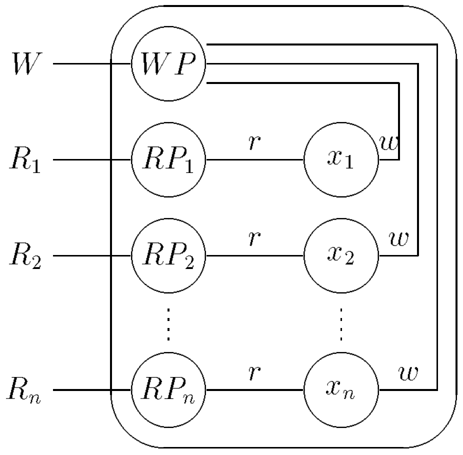


Figure 16.10: N-Reader Registers from n 1-Reader Registers

Read process i is connected to the read line of the ith physical register. Thus, the write process writes all the physical registers, while each read process only reads one physical register. The code is as follows.

WRITE(v): For all i in $\{1, \ldots, n\}$, invoke *write*(v) on x_i. Wait for *acks* from all x_i and then do *ACK*. (The *writes* can be done in any order.)

READ$_i$: Invoke *read* on x_i. Wait for *return*(v) and then do *RETURN*(v).

Claim 2 *If $x_1, \ldots, x_n$ are safe registers, then so is the logical register.*

Proof: Within each *WRITE*, exactly one *write* is performed on each particular register x_i. Therefore, since *WRITE* operations occur sequentially, *write* operations for each particular x_i are also sequential. Likewise, *read* operations for a particular x_i are also sequential.

Therefore, each physical register has the required sequentiality of accesses. This means that the physical registers return values according to their second conditions (behavior specs).

Now if a *READ*, say by RP_i, does not overlap any *WRITE*, then its contained *read* does not overlap any *write* to x_i. Therefore, the correctness of x_i assures that the *read* operation gets the value written by the last completed *write* to x_i. This is the same value as written by the last completed *WRITE*, and since RP_i returns this value, this *READ* returns the value of the last completed *WRITE*. ▮

Claim 3 *If $x_1, \ldots, x_n$ are regular registers, then so is the logical register.*

Proof: We can "reuse" the preceding proof to get the required sequentiality of accesses to each physical register, and to prove that any *READ* that does not overlap a *WRITE* returns the correct value. Therefore, we only need to show that if a *READ* process R overlaps some *WRITE* operations, then it returns the value written by a feasible *WRITE*. Since the *read* r for R falls somewhere inside the interval of R, the set of feasible *writes* for r corresponds to a subset of the feasible *WRITES* for R.

Therefore, regularity of the physical register x_i implies that r gets one of the values written by the set of feasible *writes*, and hence R gets one of the values written by the set of feasible *WRITES*. ▮

Consider two *READ*s done sequentially at different processes, bracketed within a single *WRITE*. They can obtain out-of-order results, if the first sees the result of the *WRITE* but the later one does not (since *WRITE* isn't done atomically). This implies that the construction does not make the logical register atomic even if the x_i are atomic.

With this construction, Figure 16.8 reduces to Figure 16.11.

Wait-freedom. The previous construction guarantees that all logical operations terminate in a bounded number of steps of the given process, regardless of what the other processes do. Therefore, the implementation is wait-free. In fact, the property satisfied is stronger, since it mentions a bounded number of steps. Sometimes in the literature, wait-freedom is formulated in terms of such a bound.

K-ary Safe Registers from Binary Safe Registers

If $k = \lfloor 2^l \rfloor$, then we can implement a k-ary safe register using l binary safe registers (Construction 2 in the paper). We do this by storing the ith bit of the value in binary register x_i. The logical register will allow the same number of readers as the physical registers do (see Figure 16.12).

The code for this construction is as follows.

WRITE(v): For i in $\{1, \ldots, l\}$ (in any order), write bit i of the value to register x_i.

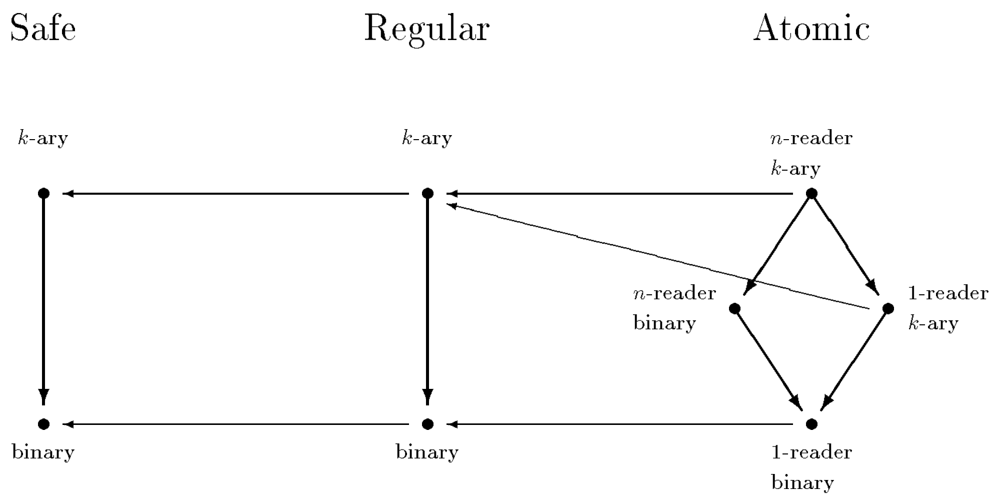


Figure 16.11: Collapsed Implementation Relationships

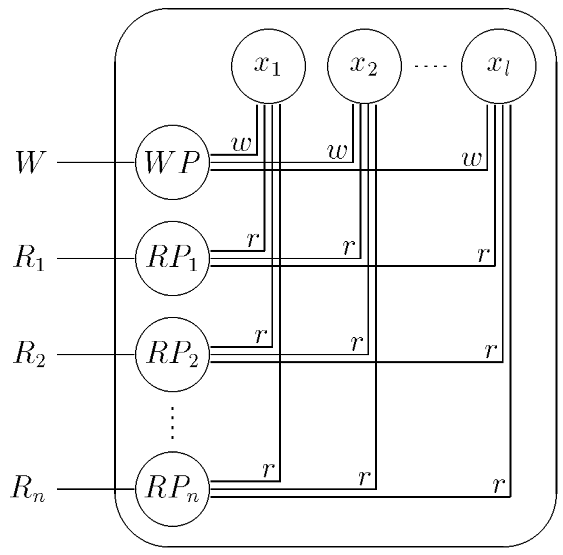


Figure 16.12: K-ary Safe Registers from Binary Safe Registers

$READ_i$: For i in $\{1, \ldots, l\}$ (in any order), read bit i of value v from register x_i. Then $RETURN(v)$.

Note that unlike the Construction 1, this construction works for safe registers only, i.e., a k-ary regular register cannot be constructed from a binary regular register using this method. With this construction, Figure 16.11 reduces to Figure 16.13.

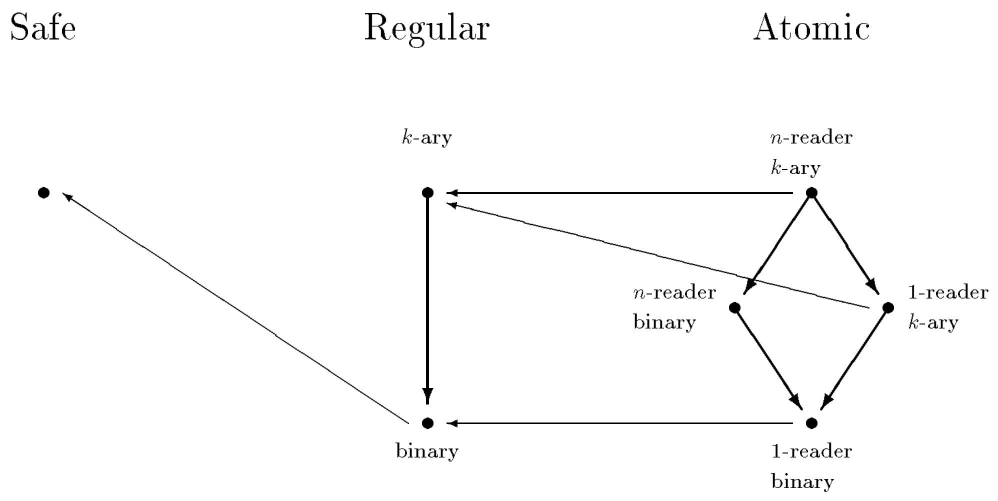


Figure 16.13: Collapsed Implementation Relationships

Lecture 17

17.1 Register Constructions (cont.)

We continue investigating the implementation relationships among the different register models. In the last lecture, we saw Constructions 1 and 2, and we have consequently reduced the relationship to the one described in Figure 17.1.

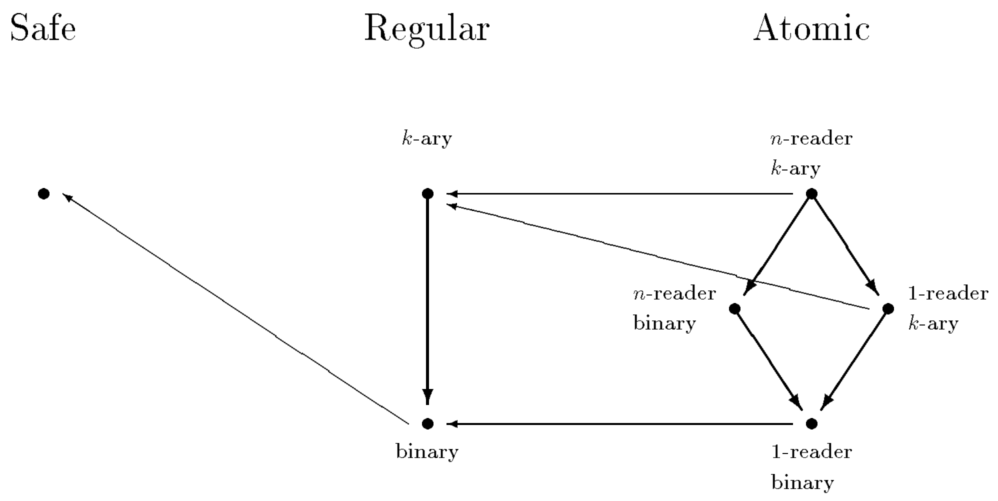


Figure 17.1: Collapsed Implementation Relationships

17.1.1 Construction 3: Binary Regular Register from Binary Safe Register

A binary regular register can easily be implemented using just one binary safe register (see Figure 17.2).

Recall that a read from a binary safe register may return any value from its domain the read overlaps a write. For binary registers, the domain consists of only 0 and 1. Notice that since a binary regular register is only required to return a feasible value, it suffices to ensure

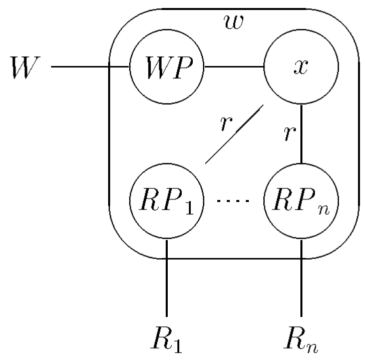


Figure 17.2: Binary Regular Register from Binary Safe Register

that all writes done on the low-level register actually change the value — because in this case either value is feasible.

The basic idea is that the write process, *WP*, keeps track of the contents of register x. (This is easy because *WP* is the only writer of x.) *WP* does a low-level *write* only when it is performing a *WRITE* that would actually change the value of the register. If the *WRITE* is just rewriting the old value, then *WP* finishes the operation right away without touching x. Therefore, all low level *write*s toggle the value of the register. Specifically, the "code" is as follows.

write(v): if v is the last recorded value, then do *WRITE-RESPOND*; else do *write*(v), and upon receiving *write-respond* do *WRITE-RESPOND*.

read: do *read*; upon receiving *read-respond*(x), do *READ-RESPOND*(x).

Now consider what happens when a *READ* is overlapped by a *WRITE*. If the corresponding *write* is not performed (i.e., the value is unchanged), then register x will just return the old value, and this *READ* will be correct. If the corresponding *write* is performed, x may return either 0 or 1. However, both 0 and 1 are in the feasible value set of this *READ* because the overlapping *WRITE* is toggling the value of the register. Therefore, the *READ* will be correct.

The implementation relations of Figure 17.1 now reduce to Figure 17.3.

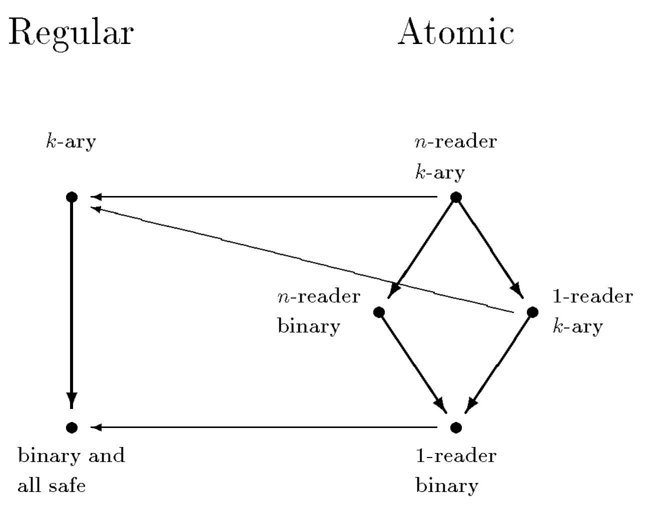


Figure 17.3: Collapsed Implementation Relationships

17.1.2 Construction 4: K-ary Regular Register from Binary Regular Register

We can implement a k-ary regular register using k binary regular registers arranged as in Figure 17.4. Here, we assume that the k values of the register are encoded as $0, 1, \ldots, k-1$. We will use a unary encoding of these values, and the idea (which has appeared in other papers, by Lamport and by Peterson) of writing a sequence of bits in one order, and reading them in the opposite order. We remark that Lamport also gives an improved implementation for binary representation, thus reducing the space requirement logarithmically.

If the initial value of the logical register is v_0, then initially x_{v_0} is 1, and the other physical registers are all 0 (using unary representation). The code is as follows.

WRITE(v): First, *write* 1 in x_v. Then, in order, *write* 0 in $x_{v-1}, \ldots, x_0$.

READ: Do *read* $x_0, x_1, \ldots, x_{k-1}$ in order, until $x_v = 1$ is found for some v. Then *RETURN* v.

Note that RP_i is guaranteed to find a non-zero x_v because whenever a physical register is zeroed out, there is already a 1 written in a higher index register. (Alternatively, we could initialize the system so that $x_{k-1} = 1$, and then we would have x_{k-1} always equal to 1.) We now prove the correctness of Construction 4 formally.

Claim 1 *If a READ R sees 1 in x_v, then v must have been written by a WRITE that is feasible for R.*

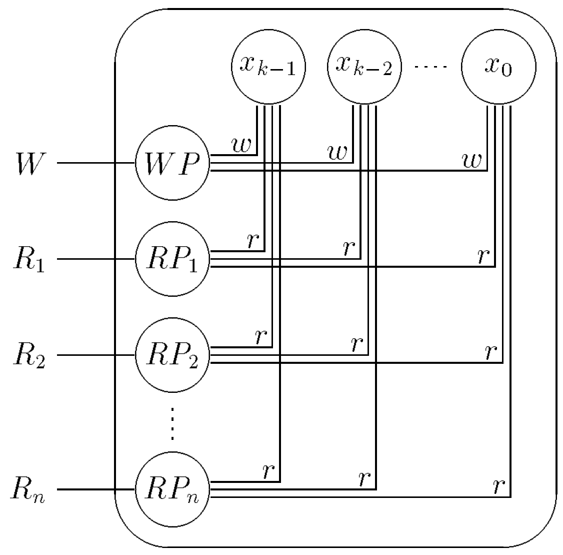


Figure 17.4: K-ary Regular Registers from Binary Regular Registers

Proof: By contradiction. Suppose R sees $x_v = 1$, and neither an overlapping nor an immediately preceding *WRITE* wrote v to the logical register. Then v was written either sometime in the past, say by *WRITE* W_1, or v is the initial value. We analyze below the former case; the initial value case can be treated similarly.

So let W_1 be the last *WRITE* completely preceding R that wrote v. Since v is not a feasible value for R, there must be another *WRITE* W_2 after W_1 which completely precedes R. Any such W_2 (i.e., a *WRITE* that follows W_1 and completely precedes R) can write only values strictly smaller than v, because if it wrote a value more than v, it would set $x_v = 0$ before R could see $x_v = 1$, and it cannot write v by the choice of W_1. Note that if such a W_2 writes value v', then the contained *write*(1) to $x_{v'}$ completely precedes the *read* of $x_{v'}$ in R.

So, consider the *latest* *write*(1) to a register $x_{v'}$ such that $v' < v$, and such that the *write*(1) follows W_1 and completely precedes the *read* of $x_{v'}$ in R. By the above arguments, there is at least one such *write*. We now proceed with case analysis.

Since R reads the registers in order $x_0, \ldots, x_{k-1}$, and it returns value $v > v'$, R must see $x_{v'} = 0$. But we have just identified a place where $x_{v'}$ is set to 1. Therefore, there exists some *write*(0) to $x_{v'}$ which follows the *write*(1) to $x_{v'}$ and either precedes or overlaps the *read* of v' in R. This can only happen if there is an intervening *write*(1) to some $x_{v''}$ such that $v'' > v'$. As above, in this case we must have $v'' < v$. (If not, then before doing the *write*(0) to $x_{v'}$, the enclosing *WRITE* would overwrite the 1 in x_v, and so R could not get it.) But then this is a contradiction to the choice of the *write* of v' as the latest. (Note the *write*(1) to $x_{v''}$ completely precedes the *read* of $x_{v''}$ in R, because the *write*(0) to $x_{v'}$ either precedes or overlaps the *read* of $x_{v'}$ in R, and $v'' > v'$.)

∎

The implementation relationships now collapse as shown in Figure 17.5.

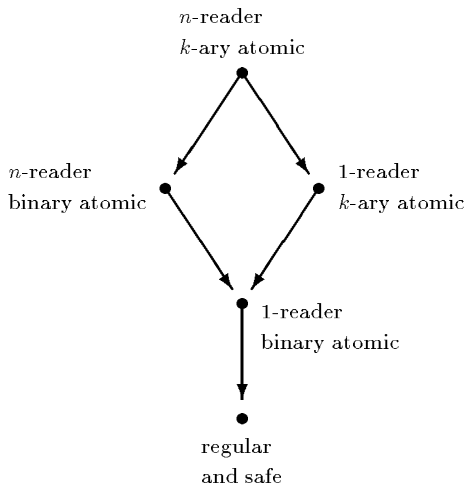


Figure 17.5: Collapsed Implementation Relationships

17.1.3 Construction 5: 1-Reader K-ary Atomic Register from Regular Register

We now show how to construct a 1-writer, 1-reader k-ary atomic register from two 1-writer, 1-reader k-ary regular registers arranged as in Figure 17.6. The code is given in Figure 17.7.

The cases in the code are designed to decide whether to return the new or the old value read. Intuitively, if $x'.num = 3$, it means that there has been much progress in the write (or it may be finished), so it is reasonable to return the new value. At the same time, it is useful to remember that the new value has already been returned, in the *new-returned* variable. This is because a later read of x that overlaps a write could get an earlier version of the variable, with $num = 2$ (because x is a regular register, and not an atomic register). Notice that it couldn't get $num = 1$. The second case covers the situation in which the previous read could have already returned the value of a write that overlaps both. The conditions look a little *ad hoc*. Unfortunately, we can't give a correctness proof in class. For the correctness proof, we refer the reader to [Lamport86]. This is not the sort of proof that can be done in any obvious way using invariants, because the underlying registers are not atomic. Instead,

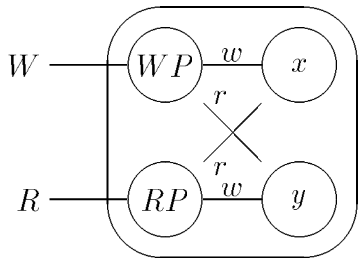


Figure 17.6: 1-Reader Atomic Register from Regular Registers

Lamport developed an alternative theory of concurrency, based on actions with duration, overlapping or totally preceding each other. Another approach to prove the correctness is to use the Lemma presented in Lecture 15.

The implementation relations now collapse as in Figure 17.8. This concludes Lamport's constructions.

17.1.4 Multi-Reader Atomic Registers

Notice that the constructions thus far still leave a gap between the n-reader, 1-writer atomic registers and the 1-reader, 1-writer atomic registers. This gap has been closed in a couple of other papers: [SinghAG87], [NewmanW87,]. We remark that these algorithms are somewhat complicated. It is also not clear if they are practical, since their time complexity is high. Singh-Anderson-Gouda have a reasonable construction, related to Lamport's Construction 5.

17.2 Lamport's Bakery Revisited

We now show that the original Lamport bakery algorithm presented in class still works if the underlying registers are only *safe*. The explanation is as follows.

- A read of the *choosing* variable while it is being written will be guaranteed to get either 0 or 1, since these are the only possible values. But those correspond to either the point before or after the write step.

- If a number is read by a chooser while it is being written, the chooser can get an arbitrary value. When the chooser takes the maximum of this arbitrary value with

Shared Variables: (regular registers)

- x: stores tuples of $(old, new, num, color)$, where old, new are from the value domain of the atomic register, and $num \in \{1,2,3\}$. Initially $x = (v_0, v_0, 3, red)$.
- y: stores values from $\{red, blue\}$. Initially $y = blue$.

Code for $write(v)$:

```
    $newcolor \leftarrow \neg y$
    $oldvalue \leftarrow x.new$                (record value of $x$ locally: no need to read it)
    for $i \in \{1,2,3\}$ do
        $x \leftarrow (oldvalue, v, i, newcolor)$
```

Code for $read$:

```
    (remember last two reads in $x'$ and $x''$)
    $x'' \leftarrow x'$
    $x' \leftarrow x$
    $y \leftarrow x'.color$.
    case
        $x'.num = 3$:
            $new\text{-}returned \leftarrow true$
            return $x'.new$
        $x'.num < 3$ and $x'.num \geq (x''.num - 1)$ and $new\text{-}returned$ and $x'.color = x''.color$:
            return $x'.new$
        otherwise:
            $new\text{-}returned \leftarrow false$
            $return\ x'.old$
    end case
```

Figure 17.7: Code for constructing single-read k-ary atomic register from regular registers

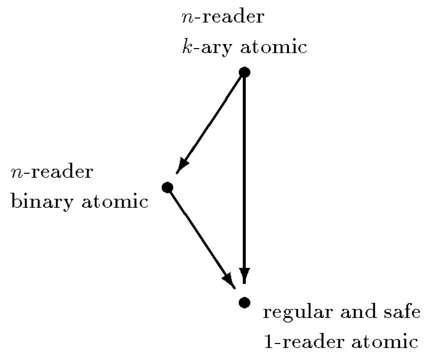


Figure 17.8: Collapsed Implementation Relationships

the others, the result is that the chooser is guaranteed to choose something bigger than all the values it sees without overlap, but there is no guarantee it will choose something bigger than a concurrent writer. However, the concurrent writer is choosing concurrently, and the algorithm didn't need any particular relation between values chosen concurrently.

- In L3, suppose someone reads a number while it's being written. If the test succeeds, the writer is back at the point of choosing its value, and everything is fine. (That is, it must be that the reader at $L2$ saw the writer before it begin choosing its number, so the writer is guaranteed to choose a larger number.) The problem is if the writer's value comes back as very small, which can delay the reader. However, it cannot delay the reader forever.

17.3 Asynchronous Network Algorithms

Now we make another major switch in the model we consider. Namely, we switch from asynchronous shared memory to asynchronous networks. As in synchronous networks, we assume a graph (or a digraph) G, with processes at the nodes, and communication over (directed) edges. But now, instead of synchronous rounds of communication, we have asynchrony in the communication as well as the process steps.

We model each process as an IOA, generally with some inputs and outputs connecting it with the outside world. (This is the boundary where the problems will usually be stated.) In addition, process i has outputs of the form $send_{i,j}(m)$, where j is an outgoing neighbor, and m is a message (element of some message alphabet M), and inputs of the form $receive_{j,i}(m)$, for j an incoming neighbor. Except for these interface restrictions, the process can be an arbitrary IOA. (We sometimes restrict the number of classes, or the number of states, etc., to obtain complexity results).

To model communication, we give a behavior specification for the allowable sequences of $send(i,j)$ and $receive(i,j)$ actions, for each edge (i,j). This can be done in two ways: by a list of *properties* or *axioms*, or by giving an IOA whose fair behaviors are exactly the desired sequences. The advantage of giving an explicit IOA is that in this case, the entire system is described simply as a composition, and we have information about the state of the entire system to use in invariants and simulation proofs. But sometimes it's necessary to do some unnatural programming to specify the desired behavior as an IOA , especially when we want to describe *liveness* constraints. Let us now make a short digression to define the important notions of safety and liveness.

Let S and L be properties (i.e., sets) of sequences over a given action alphabet. S is a *safety property* if the following conditions hold.

1. S is nonempty (i.e., there is some sequence that satisfies S),
2. S is prefix-closed (i.e., if a sequence satisfies S, then all its prefixes satisfy S), and
3. S is limit-closed (i.e., if every finite prefix of an infinite sequence satisfies S, then the infinite sequence satisfies S).

L is a *liveness property* if for every finite sequence, there is some extension that satisfies L. Informally, safety properties are appropriate for expressing the idea that "nothing bad ever happens" — if nothing bad has happened in a finite sequence, then nothing bad has happened in any prefix either; moreover, if something bad happens, it happens at a finite point in time. Liveness, on the other hand, is most often used to express the idea that "something good eventually happens".

We now return to the asynchronous message passing model. The most common communication channel studied in the research literature is a *reliable FIFO channel*. Formally, this is an IOA with the appropriate interface, whose state is a queue of messages. The $send(i,j)$ action puts its argument at the end of the queue. The $receive_{i,j}(m)$ action is enabled if m is at the head of the queue, and its effect is to remove the message from the queue. Of course, it also delivers the message to the recipient process (which should handle it somehow). But that is a part of the process' job, not of the channel's. The right division of responsibilities is obtained by the IOA composition definition. The fairness partition here can put all the locally controlled actions in a single class.

To gain some experience with asynchronous network algorithms (within this model), we go back to the beginning of the course and reconsider some of the examples we previously considered in synchronous networks.

17.3.1 Leader Election

Recall the algorithms we considered earlier. First, the graph is a ring, with UIDs built into the initial states of the processors as before. In the action signature there are no inputs from the outside world, and the only output actions are $leader_i$, one for each process i.

LeLann-Chang Algorithm

In this algorithm, each process sends its ID around the (unidirectional) ring. When a node receives an incoming identifier, it compares that identifier to its own; if the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, then the identifier is discarded; and if it is equal, it outputs the leader action. This idea still works in an asynchronous system: Figure 17.9 gives the code for the algorithm.

Essentially, the behavior of the algorithm is the same as in the synchronous, but "skewed" in time. We might prove it correct by relating it to the synchronous algorithm, but the relationship would not be a forward simulation, since things happen here in different orders. We need a more complex correspondence. Instead of pursuing that idea, we'll discuss a more common style of verifying such algorithms, namely, direct use of invariants and induction. Invariant proofs for asynchronous systems work fine. The induction is now on individual processes steps, whereas the induction in the synchronous model was on global system steps. Here, as in the synchronous case, we must show two things:

1. that no one except the maximum ever outputs *leader*, and

2. that a *leader* output eventually occurs.

Note that (1) is a safety property (asserting the fact that no violation ever occurs), and (2) is a liveness property.

Safety properties can be proved using invariants. Here, we use a similar invariant to the one we stated for the synchronous case:

Invariant 7 *If* $i \neq i_{max}$ *and* $j \in [i_{max}, i)$ *then* $own(i) \notin send(j)$.

We want to prove this invariant by induction again, similar to the synchronous case. (Recall a homework problem about this.) But now, due to the introduction of the message channels with their own states, we need to add another property to make the induction go through:

Invariant 8 *If* $i \neq i_{max}$ *and* $j \in [i_{max}, i)$ *then* $own(i) \notin queue(j, j+1)$.

With these two statements together, the induction works much as before. More specifically, we proceed by case analysis based on one process at a time performing an action.

The key step, as before, is where i gets pruned out by process i_{max}. This proves the safety property.

We now turn to the liveness property, namely that a $leader_i$ action eventually occurs. For this property, things are quite different from the synchronous case. Recall that in the synchronous case, we used a very strong invariant, which characterized exactly where the maximum value had reached after k rounds. Now there is no notion of round, so the proof must be different. We can't give such a precise characterization of what happens in the computation, since there is now so much more uncertainty. So we need to use a different method, based on making progress toward a goal. Here, we show inductively on k, for $0 \leq k \leq n-1$, that *eventually*, i_{max} winds up in $send_{i_{max}+k}$. We then apply this to $n-1$, and show that i_{max} eventually gets put into the $(i_{max}-1, i_{max})$ channel, and then that i_{max} message eventually is received at i_{max}, and therefore eventually $leader_{i_{max}}$ action gets done. All these "eventually" arguments depend on using the IOA fairness definition.

For example, suppose that there is a state s appearing in some fair execution α, where a message m appears at the head of a *send* queue in s. We want to show that eventually $send(m)$ must occur. Suppose not. By examination of the allowable transitions, we conclude that m remains at the head of the *send* queue forever. This means that the *send* class stays enabled forever. By the IOA fairness, some *send* must eventually occur. But m is the only one at the head of the queue, so the only one for which an action can be enabled, so $send(m)$ must eventually occur.

Likewise, if m appears in the kth position in the *send* queue, we can prove that eventually $send(m)$ occurs. This is proven by induction on k, with the base case above. The inductive step says that something in position k eventually gets to position $k-1$, and this is based on the head eventually getting removed.

We remark that this type of argument can be formalized within temporal logic, using statements of the form $P \Rightarrow \Diamond Q$ (see Manna and Pnueli).

State of process i:

- *own*: type UID, initially i's UID
- *send*: type queue of UID's, initially containing only i's UID
- *status*: takes values from $\{unknown, chosen, reported\}$, initially *unknown*.

Actions of process i:

```
$send_{i,i+1}(m)$
  Precondition: $m$ is first on $send$
  Effect: Remove first element of $send$

$receive_{i-1,i}(m)$
  Effect: case
            $m > own$: add $m$ to end of $send$
            $m = own$: $status \leftarrow chosen$
            otherwise: do nothing
          endcase

$leader_i$
  Precondition: $status = chosen$
  Effect: $status \leftarrow reported$
```

Fairness partition classes: For any process, all *send* actions are in one class, and the *leader* action in another (singleton) class.

Figure 17.9: LeLann-Chang algorithm for leader election in asynchronous rings

Lecturer: Nancy Lynch

Lecture 18

18.1 Leader Election in Asynchronous Rings (cont.)

18.1.1 The LeLann-Chang Algorithm (cont.)

We conclude our discussion of this algorithm with time analysis. In the synchronous case, we had an easy way to measure time — by counting the number of synchronous rounds; for the synchronous version of this algorithm, we got about n. In the asynchronous case, we have to use a *real time* measure similar to what we used for asynchronous shared memory algorithms.

At this point, we pause and generalize this time measure to arbitrary IOA's. This turns out to be straightforward — thinking of the fairness classes as representing separate tasks, we can just associate a positive real upper bound with each class. This generalizes what we did for shared memory in a natural way: there, we had upper bounds for each process, and for each user. (Recall that we could model the user as a separate I/O automaton.) In specializing version to the reliable network model, we put an upper bound of l on each class of each process, and an upper bound of d on time for the message queue to do a step (that is, the time to deliver the *first* message in the queue).

Using these definitions, we proceed with the analysis of the LeLann-Chang algorithm. A naive approach (as in the CWI paper) gives an $O(n^2)$ bound. This is obtained by using the progress argument from the last lecture, adding the time bounds in along the way: in the worst case, it takes$nl + nd$ time for the maximum UID to get from one node to the next (nl time to send it, and then nd to receive it). Note that in this analysis, we are allowing for the maximum number of messages (n) to delay the message of interest in all the queues. The overall time complexity is thus $O(n^2(l + d))$.

But it is possible to carry out a more refined analysis, thus obtaining a better upper bound. The key idea is that although the queues can grow to size at worst n and incur nd time to deliver everything, this cannot happen everywhere. More specifically, in order for a long queue to form, some messages must have been traveling faster than the worst-case upper bound. Using this intuition, we can carry out an analysis that yields an $O(n(l + d))$

upper bound on time. Informally, the idea is to associate a progress measure with each state, giving an upper bound on the time from any point when the algorithm is in that state until the leader is reported. Formally, we prove the following claim.

Claim 1 *Given a state in an execution of the algorithm, consider the distance around the ring of a token (i.e., message) i from its current position to its home node i. Let k be the maximum of these distances. Then within $k(l+d)+l$ time, a leader event occurs.*

Applying this claim to the initial state yields an upper bound for the algorithm of $n(l+d)+l$, as desired.

Proof: By induction on k.

Base case: $k=0$, i.e., the i_{max} token has already arrived at node i_{max}; in this case, within l time units the *leader* will occur.

Inductive step: Suppose the claim holds true for $k-1$, and prove it holds for k. Suppose that k is the maximum of the indicated distances. Consider any token that is at distance greater than $k-1$. By definition of k, no token is in distance strictly greater than k. This means that the token is either in the *send* queue at distance k, or in the channel's *queue* from distance k to $k-1$. We now argue that this token is not "blocked" by any other token, i.e., that it is the only token in the combination of this *send* buffer or this *queue*. To see this, notice that if another token were there also, then that other token would be at distance *strictly greater than* k from its goal, a contradiction to the assumption that k is the maximum such value. So within time at most $l+d$, the token on which we focus gets to the next node, i.e., to within distance $k-1$ of its goal, and then, by the inductive hypothesis, in additional time $(k-1)(l+d)+l$, a *leader* event occurs. This proves the inductive step. ∎

18.1.2 The Hirshberg-Sinclair Algorithm

Recall the Hirshberg-Sinclair algorithm, which used the idea of two-directional exploration as successively doubled distances. It is straightforward to see that this algorithm still works fine in the asynchronous setting, with the same time and communication upper bounds: $O(n \log n)$.

18.1.3 The Peterson Algorithm

Hirshberg and Sinclair, in their original paper, conjectured that in order to get $O(n \log n)$ worst case message performance, a leader election algorithm would have to allow bidirectional message passing. Peterson, however, constructed an algorithm disproving this conjecture. More specifically, the assumptions used by the Peterson algorithm are the following.

- The number of nodes in the ring is unknown.

- Unidirectional channels.
- Asynchronous model.
- Unbounded identifier set.
- Any node may be elected as the leader (not necessarily the maximum).

The Peterson algorithm not only has $O(n \log n)$ worst case performance, but in fact the constant factor is low; it is easy to show an upper bound of $2 \cdot n \log n$, and Peterson used a trickier, optimized construction to get a worst case performance of $1.44 \cdot n \log n$. (The constant has been brought even further down by other researchers.)

The reason we didn't introduce this algorithm earlier in the course is that synchrony doesn't help in this case: it seems like a "naturally asynchronous" algorithm.

In Peterson's algorithm, processes are designated as being either in an *active* mode or *relay* mode; all processes are initially active. We can consider the active processes as the ones "doing the real work" of the algorithm — the processes still participating in the leader-election process. The relay processes, in contrast, just pass messages along.

The Peterson algorithm is divided into (asynchronously determined) *phases*. In each phase, the number of active processes is divided at least in half, so there are at most $\log n$ phases.

In the first phase of the algorithm, each process sends its UID two steps clockwise. Thus, everyone can compare its own UID to those of its two counterclockwise neighbors. When it receives the UID's of its two counterclockwise neighbors, each process checks to see whether it is in a configuration such that the immediate counterclockwise neighbor has the highest UID of the three, i.e., process $i+1$ checks if $id_i > id_{i-1}$ and $id_i > id_{i+1}$. If process $i+1$ finds that this condition is satisfied, it remains "active," adopting as a "temporary UID" the UID of its immediate counterclockwise neighbor, i.e., process i. Any process for which the above condition does not hold becomes a "relay" for the remainder of the execution. The job of a "relay" is only to forward messages to active processes.

Subsequent phases proceed in much the same way: among active processors, only those whose immediate (active) counterclockwise neighbor has the highest (temporary) UID of the three will remain active for the next phase. A process that remains active after a given phase will adopt a new temporary UID for the subsequent phase; this new UID will be that of its immediate active counterclockwise neighbor from the just-completed phase. The formal code for Peterson's algorithm is given in Figures 18.1 and 18.2.

It is clear that in any given phase, there will be at least one process that finds itself in a configuration allowing it to remain active (unless only one process participates in the phase,

State of process i:

- *mode*: taking values from $\{active, relay\}$, initially *active*
- *status*: taking values from $\{unknown, elected, reported\}$, initially *unknown*
- $id(j)$, where $j \in \{1,2,3\}$: type UID or *nil*, initially $id(1) = i$'s UID, and $id(2) = id(3) = nil$.
- *send*: queue of UID's, initially containing i's UID
- *receive*: queue of UID's

Actions of process i:

```
get-second-uid_i
   Precondition: $mode = active$
                 $receive \neq \emptyset$
                 $id(2) = nil$
   Effect: $id(2) \leftarrow head(receive)$
           remove head of $receive$
           add $id(2)$ to end of $send$
           if $id(2) = id(1)$ then $status \leftarrow elected$

get-third-uid_i
   Precondition: $mode = active$
                 $receive \neq \emptyset$
                 $id(2) \neq nil$
                 $id(3) = nil$
   Effect: $id(3) \leftarrow head(receive)$
           remove head of $receive$
```

Figure 18.1: Peterson's algorithm for leader election: part 1

Code for process i (cont.):

```
$advance\text{-}phase_i$
  Precondition: $mode = active$
                $id(3) \neq nil$
                $id(2) > max\{id(1), id(3)\}$
  Effect: $id(1) \leftarrow id(2)$
          $id(2) \leftarrow nil$
          $id(3) \leftarrow nil$
          add $id(1)$ to $send$

$become\text{-}relay_i$
  Precondition: $mode = active$
                $id(3) \neq nil$
                $id(2) \leq max\{id(1), id(3)\}$
  Effect: $mode \leftarrow relay$

$relay_i$
  Precondition: $mode = relay$
                $receive \neq \emptyset$
  Effect: move head of $receive$ to tail of $send$

$send\_message_i(m)$
  Precondition: $head(send) = m$
  Effect: delete head of $send$

$receive\_message_i(m)$
  Effect: add $m$ to the tail of $receive$
```

Figure 18.2: Peterson's algorithm for leader election: part 2

in which case the lone remaining process is declared the winner). Moreover, at most half the previously active processes can survive a given phase, since for every process that remains active, there is an immediate counterclockwise active neighbor that must go into its relay state. Thus, as stated above, the number of active processes is at least halved in each phase, until only one active process remains.

Communication and Time Analysis. The total number of phases in Peterson's algorithm is at most $\lfloor \log n \rfloor$, and during each phase each process in the ring sends and receives exactly two messages. (This applies to both active and relay processes.) Thus, there are at most $2n \lfloor \log n \rfloor$ messages sent in the entire algorithm. (Note that this a much better constant factor than in the Hirshberg-Sinclair algorithm.)

As for time performance, one might first estimate that the algorithm should take $O(n \log n)$ time, since there are $\log n$ phases, and each phase could involve a chain of message deliveries (passing through relays) of total length $O(n)$. As it turns out, however, the algorithm terminates in $O(n)$ time. We give a brief sketch of the time analysis. Again, we may neglect "pileups", which seem not to matter for the same reason as in the analysis of LeLann. This allows us to simplify the analysis by assuming that every node gets a chance to send a message in between every two receives. For simplicity, we also assume that local processing time is negligible compared to the message-transmission time d, so we just consider the message-transmission time. Now, our plan is to trace backwards the longest sequential chain of message-sends that had to be sent in order to produce a leader.

Let us denote the eventual winner by $P0$. In the final phase of the algorithm, $P0$ had to hear from two active counterclockwise neighbors, $P1$ and $P2$. In the worst case, the chain of messages sent from $P2$ to $P0$ is actually n in length, and $P2 = P0$, as depicted in Figure 18.3.

Now, consider the previous phase. We wish to continue pursuing the dependency chain backwards from $P2$ (which might be the same node as $P0$). The key point is that, for any two consecutive phases, it must be the case that between any two active processes in the later phase, there is at least one active process in the previous phase. Thus, at the next-to-last phase, there must have been an additional process in the interval starting from $P1$ counterclockwise to $P2$, and another additional process in the interval starting from $P0$ counterclockwise to $P1$.

Thus, the chain of messages pursued backward from $P2$ in the next-to-last phase, from $P3$ and $P4$, can at worst only extend as far as $P1$ (i.e., in the worst case, $P1 = P4$), as depicted in Figure 18.4. And there is an additional process $Q1$ between $P1$ and $P0$.

At the phase *preceding* the next-to-last phase, $P4$ waits to hear from $P5$ and $P6$, where $P6$ is at worst equal to $Q1$; also, there is an additional process $Q2$ between $Q1$ and $P0$ (see

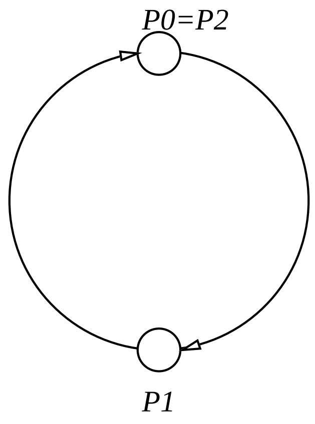


Figure 18.3: The last phase of the Peterson algorithm. $P0$ is the winner.

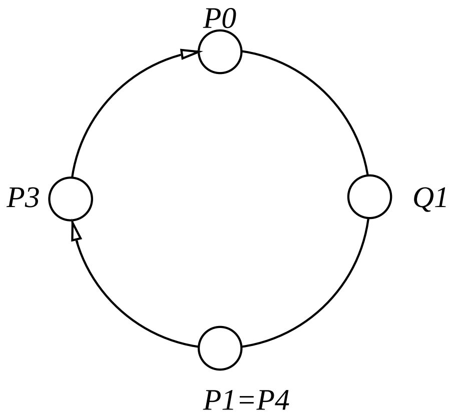


Figure 18.4: The next-to-last phase of the Peterson algorithm.

Figure 18.4). At the phase before this, $P6$ waits to hear from $P7$ and $P8$, etc. This analysis can go on, but we never get back around to $P0$, and so the total length of the dependency chain is at worst $2n$. We conclude therefore that the time is at most around $2 \cdot nd$.

18.1.4 Burns' Lower Bound Result

All the algorithms in the previous section are designed to minimize the number of messages, and the best achieve $O(n \log n)$ communication. Recall that in the synchronous case, we had a matching $\Omega(n \log n)$ lower bound under the assumption that the algorithms are *comparison based*. That carries over to the asynchronous model, since the synchronous model can be formulated as a special case of the asynchronous model. Note that the algorithms to which this lower bound applies are not allowed to use the UIDs in arbitrary ways, e.g., for counting. As we saw, if we lift this restriction, then we can get $O(n)$ message algorithms in

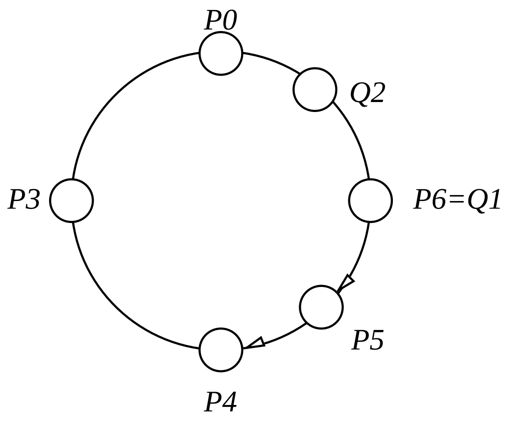


Figure 18.5: The next preceding phase of the Peterson algorithm.

the synchronous model.

It turns out, however, that the asynchronous network model has enough extra uncertainty so that the $\Omega(n \log n)$ lower bound applies regardless of how the identifiers are used. The proof of this fact is completely different proof from the synchronous case: now the asynchrony is used heavily. We consider leader election algorithms with the following properties.

- The number of nodes in the ring is unknown.
- Bidirectional channels.
- Asynchronous model.
- Unbounded identifier set.
- Any node may be elected as leader.

For this setting, Burns proved the following result.

Theorem 2 *Any leader election algorithm with the properties listed above sends at least* $\frac{1}{4}n \log n$ *messages in the worst case, where* n *is the number of processes in the ring.*

For simplicity, we assume that n is a power of 2. (The proof can be extended to arbitrary n: cf. homework.) We model each process as an I/O automaton, and stipulate that each automaton is distinct (in essence, that each process has a unique identifier). The automaton can be represented as in Figure 18.6. Each process has two output actions, **send-right** and **send-left**, and two input actions, **receive-right** and **receive-left**.

Our job will ultimately be to see how a collection of automata of this type behave when arranged into a ring; however, in the course of this exploration we would also like to see how the automata behave when arranged *not* in a ring, but simply in a straight line, as in

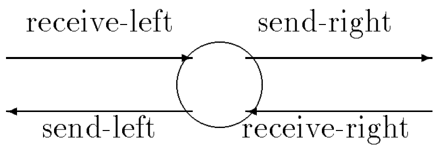


Figure 18.6: A process participating in a leader election algorithm.

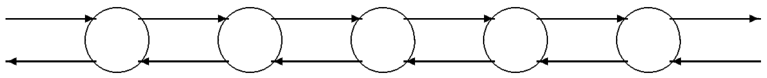


Figure 18.7: A line of leader-electing automata.

Figure 18.7. Formally, we can say that a line is a linear composition (using I/O automaton composition) of distinct automata, chosen from the universal set of automata.

The executions of such a line of automata can be examined "in isolation", where the two terminal automata receive no input messages; in this case the line simply operates on its own. Alternatively, we might choose to examine the executions of the line when certain input messages are provided to the two terminal automata.

As an added bit of notation, we will say that two lines of automata are *compatible* when they contain no common automaton between them. We will also define a *join* operation on two compatible lines which simply concatenates the lines; this operation interposes a new message queue to connect the rightmost **receive-right** message of the first line with the leftmost **send-left** message of the second, and likewise to connect the leftmost **receive-left** message of the second line with the rightmost **send-right** message of the first, and then uses ordinary IOA composition. Finally, the *ring* operation on a single line interposes new queues to connect the rightmost **send-right** and leftmost **receive-left** actions of the line, and the rightmost **receive-right** and leftmost **send-left** actions. The *ring* and *join* operations are depicted graphically in Figure 18.8.

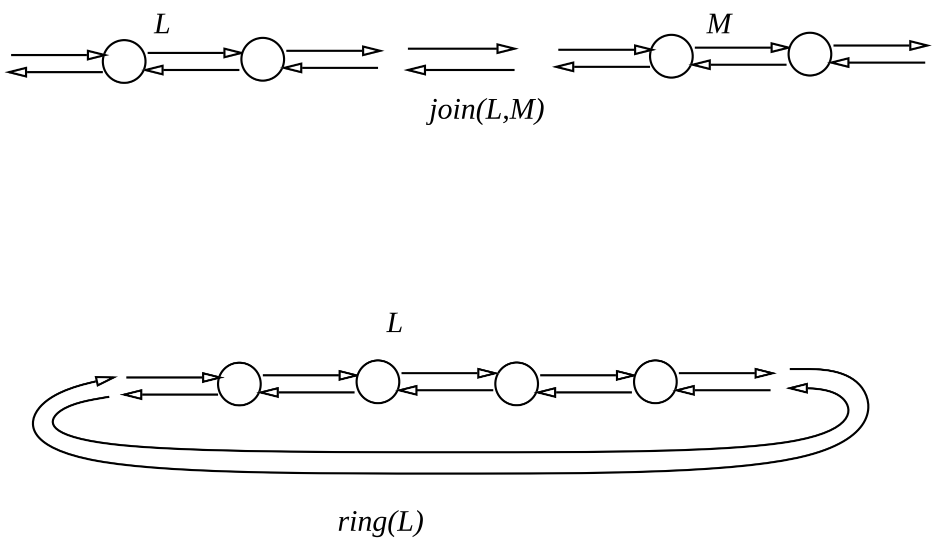


Figure 18.8: The *join* and *ring* operations.

We proceed with a proof that $\frac{1}{4}n \log n$ messages are required to elect a leader in a bidirectional asynchronous ring, where n is unknown to the processes and process identifiers are unbounded. For a system S (line or ring), and an execution α of S, we define the following notions.

- $COMM(S, \alpha)$ is the number of messages sent in execution α of system S.
- $COMM(S) = \sup_\alpha COMM(S, \alpha)$. Here we consider the number of messages sent during any execution of S. For lines, we only consider executions in which no messages come in from the ends.
- A state q of a ring is *quiescent* if there is no execution fragment starting from q in which any new message is sent.
- A state q of a line is *quiescent* if there is no execution fragment starting from q in which no messages arrive on the incoming links at the ends, and in which any new message is sent.

We now state and prove our key lemma.

Lemma 3 *For every $i \geq 0$, there is an infinite set of disjoint lines, $\mathcal{L}_i$, such that for all L $\in \mathcal{L}_i$, $| L |= 2^i$ and $COMM(L) \geq 1 + \frac{1}{4}i2^i$.*

Proof: By induction on i.

Base case: For $i = 0$, we need an infinite set of different processes such that each can send at least 1 message without first receiving one. Suppose, for contradiction, that there are 2 processes, p and q, such that neither can send a message without first receiving one. Consider rings $R1$, $R2$, and $R3$ as shown in Figure 18.9.

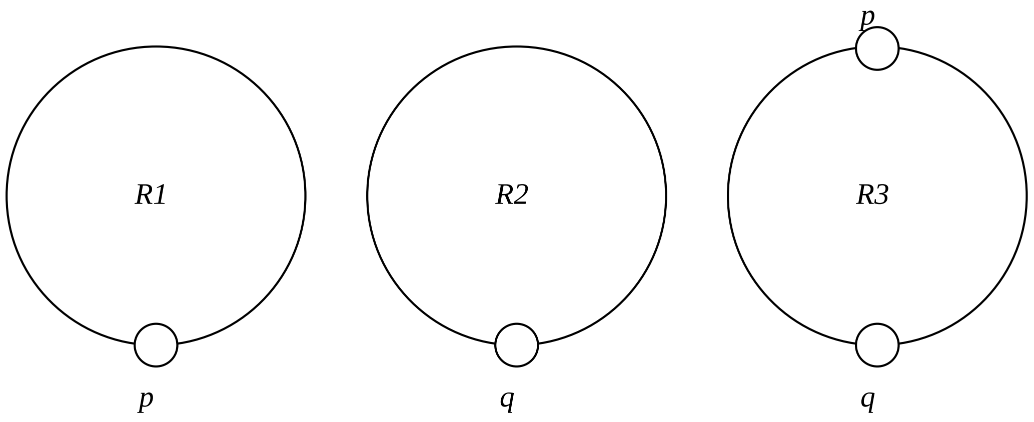


Figure 18.9: Basis for proof of Lemma 15.1

In all three rings, no messages are ever sent, so each process proceeds independently. Since $R1$ solves election, p must elect itself, and similarly for $R2$ and q. Then $R3$ elects two leaders, a contradiction. It follows that there is at most one process that can't send a message before receiving one. If there is an infinite number of processes, removing one leaves an infinite set of processes that will send a message without first receiving one. Let $\mathcal{L}_0$ be this set, which proves the basis.

Inductive step: Assume the lemma is true for $i - 1$. Let $n = 2^i$. Let L, M, N be any 3 lines from $\mathcal{L}_{i-1}$. Consider all possible combinations of two of these lines into a line of double size: LM, LN, ML, NL, MN, and NM. Since infinitely many disjoint sets of three lines can be chosen from $\mathcal{L}_{i-1}$, the following claim implies the lemma.

Claim 4 *At least one of the 6 lines can be made to send at least* $1 + \frac{n}{4}\log n$ *messages.*

Proof: Assume that the claim is false. By the inductive hypothesis, there exists a finite execution α_L of L for which $COMM(L, \alpha_L) \geq 1 + \frac{n}{8}\log\frac{n}{2}$, and in which no messages arrive from the ends.

We can assume without loss of generality that the final configuration of α_L is quiescent, since otherwise α_L can be extended to generate more messages, until the number $1+\frac{n}{4}\log n$ of messages is reached. We can assume the same condition for α_M and α_N by similar reasoning.

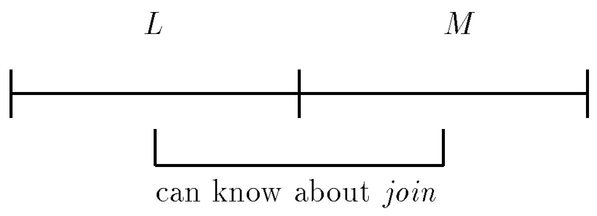


Figure 18.10: $join(L, M)$

Now consider any two of the lines, say L and M. Consider $join(L, M)$. Consider an execution that starts by running α_L on L and α_M on M, but delays messages over the boundary.

In this execution there are at least $2(1 + \frac{n}{8} \log \frac{n}{2})$ messages. Now deliver the delayed messages over the boundary. By the assumption that the claim is false, the entire line must quiesce without sending more than $\frac{n}{4}$ additional messages, messages, for otherwise the total exceeds $2(1 + \frac{n}{8} \log \frac{n}{2}) + \frac{n}{4} = 2 + \frac{n}{4} \log n$. This means that at most $\frac{n}{4}$ processes in $join(L, M)$ "know about" the *join*, and these are contiguous and adjacent to the boundary as shown in Figure 18.10. These processes extend at most halfway into either segment. Let us call this execution α_{LM}. Similarly for α_{LN}, etc.

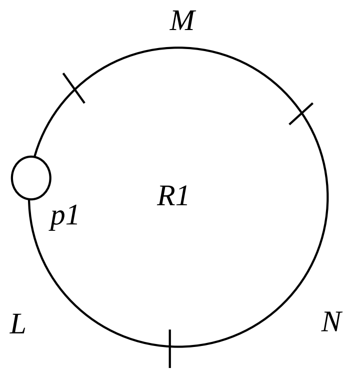


Figure 18.11: $ring(join(L, M, N))$: case 1

In ring $R1$ of Figure 18.11, consider an execution in which α_L, α_M, and α_N occur first, quiescing in three pieces separately. Then quiesce around boundaries as in α_{LM}, α_{LN}, and α_{NL}. *Since the processes that know about each join extend at most half way into either segment, these messages will be non-interfering.* Similarly for $R2$.

Each of $R1$ and $R2$ elects a leader, say p_1 and p_2. We can assume without loss of generality that p_1 is between the midpoint of L and the midpoint of M as in Figure 18.11. We consider cases based on the position of p_2 in $R2$ (Figure 18.12) to get a contradiction.

If p_2 is between the midpoint of L and the midpoint of N as in Figure 18.12, then let $R3$ be assembled by joining just M and N. Consider an execution of $R3$ in which the two

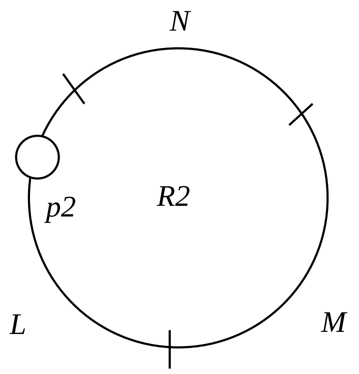


Figure 18.12: $ring(join(L, N, M))$

segments are first run to quiescence using α_M and α_N, and then quiescence occurs around the boundaries, as in α_{MN} and α_{NM}.

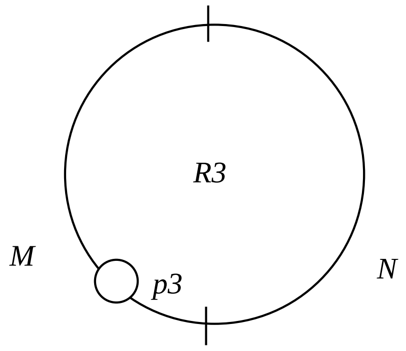


Figure 18.13: $ring(join(M, N))$: leader elected in the lower half

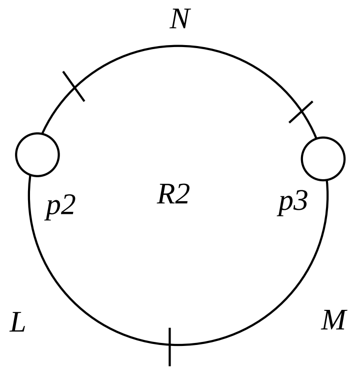


Figure 18.14: $ring(join(L, N, M))$

By the problem definition, a leader must be elected in $R3$, say p_3. First suppose it is in lower half as in Figure 18.13. Then it also occurs in $R2$ and gets elected there too as in

Figure 18.14. There are two leaders in this case which is a contradiction. If it is in the upper half of $R3$, then we arrive at a similar contradiction in $R1$.

If p_2 is between the midpoint of L and the midpoint of M, then we arrive at a similar contradiction based on $R3$, again.

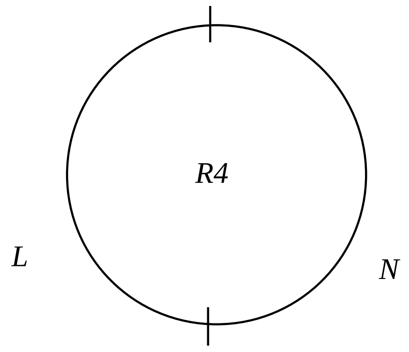


Figure 18.15: $ring(join(L, N))$

If p_2 is between the midpoint of M and the midpoint of N, then we arrive at a similar contradiction based on $R4$, a ring containing LN as in Figure 18.15. ▮

The lemma follows from the claim. ▮

Lemma 3 essentially proves Theorem 2: let n be a power of 2. Pick a line L of length n with $COMM(L) \geq 1 + \frac{n}{4} \log n$, and paste it into a circle using the *ring* operator. Let the processes in L behave exactly as they would if they were not connected, in the execution that sends the large number of messages, and delay all messages across the pasted boundary until all the large number of messages have been sent. This gives us the desired bound.

Note the crucial part played by the asynchrony in this argument.

18.2 Problems in General Asynchronous Networks

So far, we have revisited several synchronous ring leader election algorithms from the beginning of the course, and have seen that they extend directly to the asynchronous setting. Now we will reexamine the algorithms from more general synchronous networks. In the sequel we shall assume that the underlying communication network is modeled by a general undirected graph.

18.2.1 Network Leader Election

The synchronous algorithm we saw earlier for leader election in a general graph does not extend directly to the asynchronous model. In the synchronous algorithm, every node maintains a record of the maximum UID it has seen so far (initially its own). At each round, the node propagates this maximum on all its incident edges. The algorithm terminates after a number of rounds equal to the diameter; if the process has its own ID as its known maximum, then it announces itself as the leader.

In the asynchronous setting we don't have rounds, and we don't have any information about time that would allow us to wait for enough time to hear from everyone, and so the above idea doesn't extend. We could still follow the basic strategy of propagating maximum UID's asynchronously: whenever a process gets a new maximum, it propagates it sometime later. The problem is now that we don't know when to stop. To overcome this problem, we shall use other techniques.

18.2.2 Breadth-first Search and Shortest Paths

Reconsider the shortest paths algorithms. (Recall that breadth-first search is a special case of shortest paths, when all edges have weights 1.) In these algorithms, we assume that we are given an initiator node i_0, and we want that each node (eventually) outputs a pointer to a parent in a shortest-paths (or breadth-first) tree. We can also output the distance on the shortest path.

The algorithms we have presented earlier used the synchrony very heavily. E.g., for breadth-first search, we marked nodes in layers, one layer for each synchronous round. This was very efficient: $O(E)$ total messages, and time proportional to the diameter.

We cannot run that algorithm directly in an asynchronous network, since different branches of the tree can grow more quickly than others. More specifically, this means that a node can get some information about one path first, then later find out about a shorter path. Thus, we need to be able to adjust estimates downward when new information arrives. Recall that the synchronous shortest-paths algorithm we saw earlier already did this kind of adjustment. (We called it a "relaxation step".)

This adjustment idea is used in a famous shortest-paths (and BFS) algorithm that imitates the Bellman-Ford algorithm from sequential complexity theory. In fact, this algorithm was the routing algorithm used in the ARPANET between 1969 and 1980. Note that this will not be a terminating algorithm — the nodes will not know when they are done. We give the specification of this algorithm in precondition/effects style in Figure 18.16.

State variables:

best-dist, for i_0 this is initially 0, for others ∞
parent, initially undefined
new-send, for i_0 initially $neighbors(i_0)$, for others initially $\emptyset$
(says whom they should send a newly-discovered distance to).

Signature: We assume that the weights are known a priori, so no inputs are needed from the outside world. We need send and receive actions, as usual; the only message is a distance value (nonnegative real). There are no outputs (since not known when this terminates).

Actions:

```
$send_{i,j}(m)$
  Precondition: $j \in new\text{-}send$
         $m = best\text{-}dist$
  Effect: $new\text{-}send \leftarrow new\text{-}send - \{j\}$

$receive_{j,i}(m)$
  Effect: if $m + weight(j,i) < best\text{-}dist$ then
            $best\text{-}dist \leftarrow m + weight(j,i)$
            $new\text{-}send \leftarrow neighbors(i) - \{j\}$
            $parent \leftarrow j$
```

Figure 18.16: Bellman-Ford algorithm for shortest paths

Analysis. If the algorithm were executed in synchronous rounds, then the time would be $O(n)$, where n is the total number of nodes, and the number of messages would be $O(n|E|)$. This does not seem too bad. But in asynchronous execution the time bound can be much worse: in fact, we show below that the time and message complexity are *exponential* in n.

Claim 5 *The algorithm sends $\Omega(2^n)$ messages in the worst case.*

Proof: Consider the example illustrated in Figure 18.17.

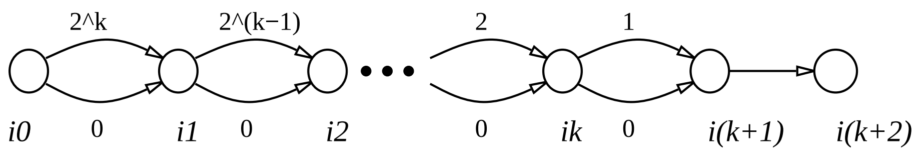


Figure 18.17: Bad scenario for the Bellman-Ford Algorithm

The possible estimates that i_{k+1} can have for its *best-dist* are $0, 1, 2, 3, \ldots 2^{k+1} - 1$. Moreover, we claim that it is possible for i_{k+1} to acquire all of these estimates, in order from the largest to the smallest. To see how, consider the following scenario. Suppose that the messages on the upper paths all propagate first, giving i_{k+1} the estimate $2^{k+1} - 1$. Next, the "alternative" message from i_k arrives, giving i_{k+1} the adjusted estimate of $2^{k+1} - 2$. Next, the "alternative" message from i_{k-1} to i_k arrives at i_k, cause i_k to reduce its estimate by 2. It then propagates this revision on both paths. Again, suppose the message on the upper path arrives first, etc. ∎

Since i_{k+1} gets 2^{k+1} distinct estimates, it is possible, *if i_{k+1} takes steps fast enough*, for i_{k+1} to actually put all these values in the outgoing channel to i_{k+2}. This yields exponentially many messages (actually, $\Omega(2^{n/2})$). Moreover, the time complexity is similarly bad, because these messages all get delivered sequentially to i_{k+2}.

We now consider upper bounds on the Bellman-Ford algorithm. First, we claim that the number of messages is $O((n+1)^n \cdot e)$ at most. This is because each node only sends new messages out when it gets a new estimate. Now, each estimate is the total weight of a simple path to that node from i_0, and the bound follows from the fact that there are at most $(n+1)^n$ such paths. (This may be a loose estimate.) Now consider any particular edge. It only gets messages when one of its endpoints gets a new estimate, leading to a total of $2(n+1)^n$ messages on that edge.

Lecturer: Nancy Lynch

Lecture 19

19.1 Asynchronous Broadcast-Convergecast

In an earlier lecture, we described a synchronous discipline for broadcasting and convergecasting (sometimes called broadcast-with-echo), using a breadth-first spanning tree. In the asynchronous setting, the breadth-first spanning tree is not as usable, since we don't know when the construction is complete. We can still do broadcast-convergecast reasonably efficiently, on an arbitrary (not necessarily breadth-first) spanning tree.

First we consider how to construct an arbitrary *spanning tree*. The distinguished source node i_0 begins by sending out *report* messages to all its neighbors. Any other node, upon receipt of its first *report* message, marks itself as *done*, identifies the sender j as its parent, sends a $parent_i(j)$ announcement to the outside world, and sends out *report* messages on its outgoing links. The node ignores additional *report* messages. This idea works fine asynchronously: nodes know when they are done, $O(E)$ messages are used, and it takes $O(diam)$ time until termination. (There are no pileups on links since only two messages ever get sent on each edge, one in each direction.)

Now we proceed to the task of *broadcasting* a message to all nodes. We can construct an arbitrary spanning tree as above, and then i_0 uses it to send messages over the tree links. To do this, nodes need to be informed as to which other nodes are their children, and this requires local communication from children to parents upon discovery. Note the interesting *timing anomaly* here: tree paths could take $\Omega(n)$ time to traverse the second time. To avoid that, we can send the message in the course of building the tree, by "piggybacking" the message on all *report* messages.

We can also use an asynchronously-constructed spanning tree to "fan in" results of a computation back to i_0: as in the synchronous case, each node waits to hear from all its children in the spanning tree, and then forwards a message to its parent.

This brings us to the task of *broadcast and convergecast*. Now suppose we want to broadcast a message and collect acknowledgments (back at i_0) that everyone has received it. To solve this problem we do a combination of the asynchronous spanning tree construction above, followed (asynchronously) by fanning in results from the leaves.

In fact, if the tree is being constructed solely for the purpose of sending and getting "acks" for a particular message, it is possible to delete the tree information after fanning in

State:

- *msg*, for the value being broadcast, at i_0 initially the message it is sending, elsewhere *nil*
- *send*, buffer for outgoing messages, at i_0 initially contains $(bcast, m)$ to all neighbors of i_0, elsewhere $\emptyset$
- *parent*, pointer for the parent edge, initially *nil*
- *acked*, a set to keep track of children that have acked, everywhere initially $\emptyset$
- *done*, Boolean flag, initially *false*

Actions:

```
send action: as usual, just sends anything in send buffer.

$receive_{j,i}(bcast, m)$
  Effect: if $msg = nil$ then
              $msg \leftarrow m$
              $parent \leftarrow j$
              for all $k \in neighbors - \{j\}$, put message $(bcast, m)$ in $send$
              $acked \leftarrow \emptyset$
          else put message $(ack)$ to $j$ in $send$

$receive_{j,i}(ack)$
  Effect: $acked \leftarrow acked \cup \{j\}$

$finish_i$ (for $i \neq i_0$)
  Precondition: $acked = neighbors - \{parent\}$
          $done = false$
  Effect: put $ack$ message to $parent$ in $send$
          $done \leftarrow true$

$finish_i$ (for $i = i_0$)
  Precondition: $acked = neighbors$
          $done = false$
  Effect: output $done$ message to outside world
          $done \leftarrow true$
```

Figure 19.1: Code for the broadcast-convergecast task, without deleting the tree

the results back to the parent. However, we have to be careful that the node doesn't forget *everything* — since if it later gets a *report* message from another part of the tree whose processing is delayed, it should "know enough" to ignore it. In other words, we need to keep some kind of flag at each node saying that it's done. The code is given in Figure 19.1.

Analysis. Communication is simple: it takes $O(E)$ messages to set up the tree, and after the tree is set up, it's only $O(n)$ messages to do the broadcast-convergecast work. The time analysis is trickier. One might think that in $O(diam)$ time the algorithm terminates, since that's how long it takes to do the broadcast. The convergecast, however, can take longer. Specifically, we have the following *timing anomaly:* a message can travel fast along a long network path, causing a node to become attached to the spanning tree on a long branch, even though there is a shorter path to it. When the response travels back along the long path, it may now take time proportional to the length of the path, which is $\Omega(n)$ for general graphs. To fix this problem, we shall need new ideas, such as a *synchronizer*, which we shall discuss next lecture.

Example: application to leader election. The broadcast-convergecast algorithm provides an easy solution to the leader election problem. Recall that the simple leader election algorithm we mentioned above didn't terminate. Now, we could do an alternative leader election algorithm as follows. Starting from every node, run broadcast-convergecast in parallel, while collecting the maximum ID in the convergecast on each tree. The node that finds that the maximum is equal to its own UID gets elected.

19.2 Minimum Spanning Tree

19.2.1 Problem Statement

Now we return to the problem of constructing a minimum-weight spanning tree (MST), this time in an asynchronous network. Let us recall the problem definition. We are given an undirected graph $G = (V, E)$ with weighted edges, such that each vertex is associated with its own process, and processes are able to communicate with each other via messages sent over edges. We wish to have the processes (vertices) cooperate to construct a minimum-weight spanning tree for the graph G. That is, we want to construct a tree covering the vertices in G, whose total edge weight is less than or equal to that of every other spanning tree for G.

We assume that processes have unique identifiers, and that each edge of the graph is associated with a unique weight known to the vertices on each side of that edge. (The assumption of unique weights on edges is not a strong one, given that processes have unique identifiers: if edges had non-distinct weights, we could derive "virtual weights" for all edges by appending the identifier numbers of the end points onto the edge weights, and use them as tie-breakers between the original weights.) We will also assume that a process does not know the overall topology of the graph — only the weights of its incident edges — and that it can learn non-local information only by sending messages to other processes over those edges. The output of the algorithm will be a "marked" set of tree edges; every process will mark those edges adjacent to it that are in the final MST.

There is one significant piece of input to this algorithm: namely, that any number of nodes will be "awakened" from the outside to begin computing the spanning tree. A process can be awakened either by the "outside world" (asking that the process begin the spanning tree computation), or by another (already-awakened) process during the course of the algorithm.

The motivation for the MST problem comes mainly from the area of communications. The weights of edges might be regarded as "message-sending costs" over the links between processess. In this case, if we want to broadcast a message to every process, we would use the MST to get the message to every process in the graph at minimum cost.

19.2.2 Connections Between MST and Other Problems

The MST problem has strong connections to two other problems: that of finding *any* (undirected, unrooted) spanning tree at all for a graph, and that of electing a leader in a graph.

If we are given an (undirected, unrooted) spanning tree, it is pretty easy to elect a leader; this proceeds via a "convergecast" of messages from the leaves of the tree until the incoming messages converge at some particular node, which can then be designated as the leader. (It is possible that the messages converge at some edge rather than some node, in which case one of the endpoints of this edge, say the one with the larger ID, can be chosen.)

Conversely, if we are given a leader, it is easy to find an arbitrary spanning tree, as discussed above (under broadcast-convergecast): the leader just broadcasts messages along each of its neighboring edges, and nodes designate as their parent in the tree that node from which they first receive an incoming message (after which the nodes then broadcast their own messages along their remaining neighboring edges). So, modulo the costs of these basic algorithms, the problems of leader election and finding an arbitrary spanning tree are equivalent.

A minimum spanning tree is of course a spanning tree. The converse problem, going from an arbitrary spanning tree (or a leader) to a minimum spanning tree, is much harder.

One possible idea would be to have every node send information regarding its surrounding edges to the leader, which then computes the MST centrally and distributes the information back to every other node in the graph. This centralized strategy requires a considerable amount of local computation, and a fairly large amount of communication.

19.2.3 The Synchronous Algorithm: Review

Let us recall how the synchronous algorithm worked. It was based on two basic properties of MST's:

Property 1 *Let G be an undirected graph with vertices V and weighted edges E. Let $(V_i, E_i) : 1 \leq i \leq k$ be any spanning forest for G, with $k > 1$. Fix any i, $1 \leq i \leq k$. Let e be an edge of lowest cost in in the set $\{e' : e' \in E - \bigcup_j E_j$ and exactly one endpoint of e' is in $V_i\}$. Then there is a spanning tree for G that includes $\{e\} \cup (\bigcup_j E_j)$ and this tree is of as low a cost as any spanning tree for G that includes $\bigcup_j E_j$.*

Property 2 *If all edges of a connected graph have distinct weights, then the MST is unique.*

These properties justify a basic strategy in which the MST is built up in *fragments* as follows. At any point in the algorithm, each of a collection of fragments may independently and concurrently find its own minimum-weight outgoing edge (MWOE), knowing that all such edges found must be included in the unique MST. (Note that if the edge weights were not distinct, the fragments couldn't carry out this choice independently, since it would be possible for them to form a cycle.)

This was the basis of the synchronous algorithm. It worked in synchronous levels, where in each level, *all* fragments found their MWOE's and combined using these, to form at most half as many fragments.

If we try to run the given synchronous algorithm in an asynchronous network, we see some problems.

Difficulty 1: In the synchronous case, when a node queries a neighbor to see if it is in the same fragment, it knows that the neighbor node is up-to-date, at the same level. Therefore, if the neighbor has a distinct fragment id, then this fact implies that the neighbor is not in the same fragment. But in the asynchronous setting, it could be the case that the neighbor has not yet heard that it is in the same fragment.

Difficulty 2: The synchronous algorithm achieves a message cost of $O(n \log n + E)$, based on a balanced construction of the fragments. Asynchronously, there is danger of constructing the fragments in an unbalanced way, leading to many more messages. The number of messages sent by a fragment to find its MWOE will be proportional to the number of nodes in the fragment. Under certain circumstances, one might imagine the algorithm proceeding by having one large fragment that picks up a single node at a time, each time requiring $\Omega(f)$ messages, where f is the number of nodes in the fragment (see Figure 19.2) . In such a situation, the algorithm would require $\Omega(n^2)$ messages to be sent overall.

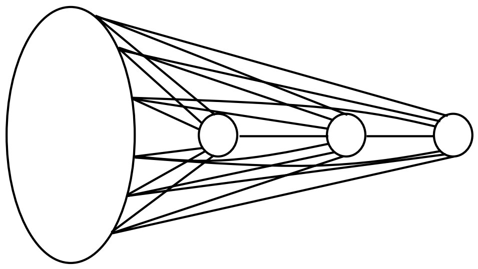


Figure 19.2: How do we avoid a big fragment growing by one node at a time?

19.2.4 The Gallager-Humblet-Spira Algorithm: Outline

These difficulties lead us to re-think the algorithm to modify it for use in an asynchronous network. Luckily, the basic ideas are the same. The algorithm we shall see below is by Gallager-Humblet-Spira. They focus on keeping the number of messages as small as possible, and manage to achieve the same $O((n \log n) + E)$ message bound as in the synchronous algorithm.

Before we continue, let us make a few preliminary remarks. First, consider the question whether this is the minimum bound possible. Note that, at least for some graphs (e.g., rings), the $n \log n$ term is necessary — it comes from the lower bound on the number of messages for leader election that we saw in Burns' theorem. What about the E term? This seems essential. In a work by Awerbuch-Goldreich-Peleg-Vainish, they show that the number of *bits* of communication must be at least $\Omega(E \log n)$. If we assume that each message contains only a constant number of id's, then this means that the number of messages is $\Omega(E)$. If the messages are allowed to be of arbitrary size, however, then it can be shown that $O(n)$ messages suffice.

Another thing we would like to say about the Gallager-Humblet-Spira algorithm is that not only is it interesting, but it is also extremely clever: as presented in their paper, it is about two pages of tight, modular code, and there is a good reason for just about *every* line in the algorithm. In fact, only one or two tiny optimizations have been advanced over the original algorithm. The algorithm has been proven correct via some rather difficult formal proofs (see [WelchLL88]); and it has been referenced and elaborated upon quite often in subsequent research. It has become a sort of test case for algorithm proof techniques.

As in the synchronous algorithm we saw earlier, the central idea of the Gallager-Humblet-Spira algorithm is that nodes form themselves into collections — i.e., fragments — of increasing size. (Initially, all nodes are considered to be in singleton fragments.) Each fragment is itself connected by edges that form a MST for the nodes in the fragment. Within any fragment, nodes cooperate in a distributed algorithm to find the MWOE for the entire fragment (that is, the minimum weight edge that leads to a node outside the fragment). The strategy for accomplishing this involves broadcasting over the edges of the fragment, asking each node separately for its own MWOE leading outside the fragment. Once all these edges have been found, the minimal edge among them will be selected as an edge to include in the (eventual) MST.

Once an MWOE for a fragment is found, a message may be sent out over that edge to the fragment on the other side. The two fragments may then combine into a new, larger fragment. The new fragment then finds its own MWOE, and the entire process is repeated until all the nodes in the graph have combined themselves into one giant fragment (whose edges are the final MST).

This is not the whole story, of course; there are still some problems to overcome. First, how does a node know which of its edges lead outside its current fragment? A node in fragment F can communicate over an outgoing edge, but the node at the other end needs some way of telling whether it too is in F. We will therefore need some way of naming fragments so that two nodes can determine whether they are in the same fragment. But the issue is still more complicated: it may be, for example, that the other node (at the end of the apparently outgoing edge) *is* in F but hasn't learned this fact yet, because of communication delays. Thus, some sort of overall synchronization process is needed—some sort of strategy that ensures that nodes won't search for outgoing edges until all nodes in the fragment have been informed of their current fragment.

And there is also the second problem, discussed above, of the unbalanced merging behavior causing excessive message cost. This second problem should suggest a "balanced-tree algorithm" solution: that is, the difficulty derives from the merging of data structures that are very unequal in size. The strategy that we will use, therefore, is to merge fragments of roughly equal size. Intuitively, if we can keep merging fragments of nearly equal size, we can keep the number of total messages to $O(n \log n)$.

The trick we will use to keep the fragments at similar sizes is to associate a *level number* with each fragment. We will ensure that if $level(F) = l$ for a given fragment F, then the number of nodes in F is greater than or equal to 2^l. Initially, all fragments are just singleton nodes at level 0. When two fragments at level l are merged together, the result is a new fragment at level $l + 1$. (This preserves the condition for level numbers: if two fragments of size at least 2^l are merged, the result is a new fragment of size at least 2^{l+1}.)

So far, this may look similar to the way the level numbers were used in the synchronous algorithm, but it will actually be somewhat different. E.g., in the synchronous case, we could merge some arbitrary number of level l fragments to get a new level $l + 1$ fragment.

The level numbers, as it turns out, will not only be useful in keeping things balanced, but they will also provide some identifier-like information helping to tell nodes whether they are in the same fragment.

19.2.5 In More Detail

There are two ways of combining fragments.

1. *Merging.* This combining rule is applied when we have two fragments F and F' with the same level and the same minimum-weight outgoing edge:

$$\begin{aligned} level(F) &= level(F') = l \\ \mathrm{MWOE}(F) &= \mathrm{MWOE}(F') \end{aligned}$$

   The result of a merge is a new fragment at a level of $l + 1$.

2. *Absorbing.* There is another case to consider. It might be that some nodes are forming into huge fragments via merging, but isolated nodes (or small fragments) are lagging behind at a low level. In this case, the small fragments may be absorbed into the larger ones without determining the MWOE of the large fragment. Specifically, the rule for absorbing is that if two fragments F and F' satisfy $level(F) < level(F')$, and the MWOE(F) leads to F', then F can be absorbed into F' by combining them along MWOE(F). The larger fragment formed is still at the level of F'. In a sense, we don't want to think of this as a "new" fragment, but rather as an augmented version of F'.

These two combining strategies are illustrated (in a rough way) by Figure 19.3. It is worth stressing the fact that $level(F) < level(F')$ does not imply that fragment F is smaller than F'; in fact, it could be larger. (Thus, the illustration of F as a "small" fragment in Figure 19.3 is meant only to suggest the typical case.)

Level numbers also serve, as mentioned above, as identifying information for fragments. For fragments of level 1 or greater, however, the specific fragment identifier is the *core edge* of the fragment. The core edge is the edge along which the merge operation resulting in the current fragment level took place. (Since level numbers for fragments are only incremented by merge operations, any fragment of level 1 or greater must have had its level number incremented by some previous merge along an edge.) The core edge also serves as the site where the processing for the fragment originates and where information from the nodes of the fragment is collected.

To summarize the way in which core edges are identified for fragments:

- For a *merge* operation, *core* is the common MWOE of the two combining fragments.
- For an *absorb* operation, *core* is the core edge of the fragment with the larger level number.

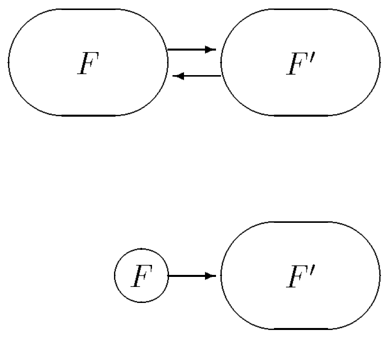


Figure 19.3: Two fragments combine by merging; a fragment absorbs itself into another

Note that identifying a fragment by its core edge depends on the assumption that all edges have unique identifiers. Since we are assuming that the edges have unique weights, the weights could be the identifiers.

We now want to show that this strategy, of having fragments merge together and absorb themselves into larger fragments, will in fact suffice to combine all fragments into a MST for the entire graph.

Claim 1 *If we start from an initial situation in which each fragment consists of a single node, and we apply any possible sequence of merge and absorb steps, then there is always some applicable step to take until the result is a single fragment containing all the nodes.*

Proof: We want to show that no matter what configuration we arrive at in the course of the algorithm, there is always some merge or absorb step that can be taken.

One way to see that this is true is to look at all the current fragments at some stage in the running algorithm. Each of these fragments will identify its MWOE leading to some other fragment. If we view the fragments as vertices in a "fragment-graph," and draw the MWOE for each fragment, we get a directed graph with an equal number of vertices and edges (see Figure 19.4). Since we have k nodes and k edges (for some k), such a directed graph *must* have a cycle; and because the edges have distinct weights, only cycles of size 2 (i.e., cycles involving two fragments) may exist. Such a 2-cycle represents two fragments that share a single MWOE.

Now, it must be the case that the two fragments in any 2-cycle can be combined. If the two fragments in the cycle have the same level number, a merge operation can take place; otherwise, the fragment with the smaller level number can absorb itself into the fragment

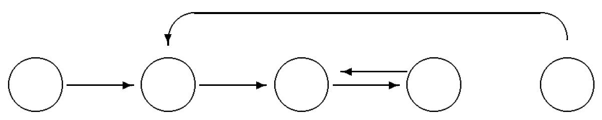


Figure 19.4: A collection of fragments and their minimum-weight outgoing edges.

with the larger one. ∎

Let's return to the question of how the MWOE is found for a given fragment. The basic strategy is as follows. Each node in the fragment is finds its own MWOE leading outside the fragment; then the information from each node is collected at a selected process, which takes the minimum of all the edges suggested by the individual nodes.

This seems straightforward, but it re-opens the question of how a node knows that a given edge is outgoing—that is, that the node at the other end of the edge lies outside the current fragment. Suppose we have a node p that "looks across" an edge e to a node q at the other end. How can p know if q is in a different fragment or not?

A fragment name (or identifier) may be thought of as a pair $(core, level)$. If q's fragment name is the same as p's, then p certainly knows that q is in the same fragment as itself. However, if q's fragment name is different from that of p, then it is still possible that q and p are indeed in the same fragment, but that q has not yet been informed of that fact. That is to say, q's information regarding its own current fragment may be out of date.

However, there is an important fact to note: if q's fragment name has a core unequal to that of p, and it has a level value at least as high as p, then q can't be in the fragment that p is in currently, and never will be. This is so because in the course of the algorithm, a node will only be in one fragment at any particular level. Thus, we have a general rule that q can use in telling p whether both are in the same fragment: if the value of $(core, level)$ for q is the same as that of p then they are in the same fragment, and if the value for *core* is different for q and the value of *level* is at least as large as that of p then they are in different fragments.

The upshot of this is that MWOE(p) can be determined only if $level(q) \geq level(p)$. If q has a lower level than p, it simply delays answering p until its own level is at least as great as p's.

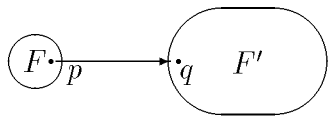


Figure 19.5: Fragment F absorbs itself into F' while F' is still searching for its own MWOE.

However, notice that we now have to reconsider the progress argument, since this extra precondition may cause progress to be blocked. Since a fragment can be delayed in finding its MWOE (since some individual nodes within the fragment are being delayed), we might ask whether it is possible for the algorithm to reach a state in which neither a merge or absorb operation is possible. To see that this is not the case, we use essentially the same argument as before, but this time we only consider those MWOE's found by fragments with the lowest level in the graph (call this level l). These succeed in all their individual MWOE(p) calculations, so succeed in determining their MWOE for the entire fragment. Then the argument is as before: If any fragment at level l finds a MWOE to a higher-level fragment, then an absorb operation is possible; and if every fragment at the level l has a MWOE to some other fragment at level l, then we must have a 2-cycle between two fragments at level l, and a merge operation is possible. So again, even with the new preconditions, we conclude that the algorithm must make progress until the complete MST is found.

Getting back to the algorithm, each fragment F will find its overall MWOE by taking a minimum of the MWOE for each node in the fragment. This will be done by a "broadcast-convergecast" algorithm starting from the core, emanating outward, and then collecting all information back at the core.

This leads to yet another question: what happens if a "small" fragment F gets absorbed into a larger one F' while F' is still in the course of looking for its own MWOE?

There are two cases to consider (consult Figure 19.5 for the labeling of nodes). Suppose first that the minimum edge leading outside the fragment F', has not yet been determined. In this case, we search for a MWOE for the newly-augmented fragment F' in F as well (there is no reason it cannot be there).

On the other hand, suppose MWOE(F') has already been found at the time that F absorbs itself into F'. In that event, the MWOE for q cannot possibly be e, since the only way that the MWOE for q could be known is for that edge to lead to a fragment with a level at least as great as F', and we know that the level of F is smaller than that of F'. Moreover, the fact that the MWOE for q is not e implies that the MWOE for the entire fragment F' cannot possibly be in F. This is true because e is the MWOE for fragment F, and thus there can be no edges leading out of F with a smaller cost than the already-discovered MWOE for node q. Thus, we conclude that if MWOE(F') is already known at the time the absorb operation takes place, then fragment F' needn't look for its overall MWOE among the newly-absorbed nodes. (This is fortunate, since if F' did in fact have to look for its MWOE among the new nodes, it could be too late: by the time the absorb operation takes place, q might have already reported its own MWOE, and fragment F' might already be deciding on an overall MWOE without knowing about the newly-absorbed nodes.)

19.2.6 A Summary of the Code in the GHS Algorithm

We have described intuitively the major ideas of the Gallager-Humblet-Spira algorithm; this explanation above should be sufficient to guide the reader through the code presented in their original paper.

Although the actual code in the paper is dense and complicated, the possibility of an understandable high-level description turns out to be fairly typical for communications algorithms. In fact, the high-level description that we have seen can serve as a basis for a correctness proof for the algorithm, using simulations.

The following message types are employed in the actual code:

- INITIATE messages are broadcast outward on the edges of a fragment to tell nodes to start finding their MWOE.

- REPORT messages are the messages that carry the MWOE information back in. (These are the convergecast response to the INITIATE broadcast messages.)

- TEST messages are sent out by nodes when they search for their own MWOE.

- ACCEPT and REJECT messages are sent in response to TEST messages from nodes; they inform the testing node whether the responding node is in a different fragment (ACCEPT) or is in the same fragment (REJECT).

- CHANGE-ROOT is a message sent toward a fragment's MWOE once that edge is found. The purpose of this message is to change the root of the (merging or currently-being-absorbed) fragment to the appropriate new root.

- CONNECT messages are sent across an edge when a fragment combines with another. In the case of a merge operation, CONNECT messages are sent both ways along the edge between the merging fragments; in the case of an absorb operation, a CONNECT message is sent by the "smaller" fragment along its MWOE toward the "larger" fragment.

In a bit more detail, INITIATE messages emanate outward from the designated "core edge" to all the nodes of the fragment; these INITIATE messages not only signal the nodes to look for their own MWOE (if that edge has not yet been found), but they also carry information about the fragment identity (the core edge and level number of the fragment). As for the TEST-ACCEPT-REJECT protocol: there's a little bookkeeping that nodes have to do. Every node, in order to avoid sending out redundant messages testing and re-testing edges, keeps a list of its incident edges in the order of weights. The nodes classify these incident edges in one of three categories:

- *Branch* edges are those edges designated as part of the building spanning tree.
- *Basic* edges are those edges that the node doesn't know anything about yet — they may yet end up in the spanning tree. (Initially, of course, all the node's edges are classified as basic.)
- *Rejected* edges are edges that cannot be in the spanning tree (i.e., they lead to another node within the same fragment).

A fragment node searching for its MWOE need only send messages along basic edges. The node tries each basic edge in order, lowest weight to highest. The protocol that the node follows is to send a TEST message with the fragment level-number and core-edge (represented by the unique weight of the core edge). The recipient of the TEST message then checks if its own identity is the same as the TESTer; if so, it sends back a REJECT message. If the recipient's identity (core edge) is different and its level is greater than or equal to that of the TESTer, it sends back an ACCEPT message. Finally, if the recipient has a different identity from the TESTer but has a lower level number, it delays responding until such time as it can send back a definite REJECT or ACCEPT.

So each node finds the MWOE if it exists. All of this information is sent back to the nodes incident on the core edge via REPORT messages, who determine the MWOE for the entire fragment. A CHANGEROOT message is then sent back towards the MWOE, and the endpoint node sends a CONNECT message out over the MWOE.

When two CONNECT messages cross, this is the signal that a merge operation is taking place. In this event, a new INITIATE broadcast emanates from the new core edge and the newly-formed fragment begins once more to look for its overall MWOE. If an absorbing CONNECT occurs, from a lower-level to a higher-level fragment, then the node in the high-level fragment knows whether it has found its own MWOE and thus whether to send back an INITIATE message to be broadcast in the lower-level fragment.

19.2.7 Complexity Analysis

Message complexity. The analysis is similar to the synchronous case, giving the same bound of $O(n \log n + E)$. More specifically, in order to analyze the message complexity of the GHS algorithm, we apportion the messages into two different sets, resulting separately (as we will see) in the $O(n \log n)$ term and the $O(E)$ term.

The $O(E)$ term arises from the fact that each edge in the graph must be tested at least once: in particular, we know that TEST messages and associated REJECT messages can occur at most once for each edge: this follows from the observation that once a REJECT message has been sent over an edge, that edge will never be tested again.

All other messages sent in the course of the algorithm—the TEST-ACCEPT pairs that go with the acceptances of edges, the INITIATE-REPORT broadcast-convergecast messages, and the CHANGEROOT-CONNECT messages that occur when fragments combine—can be considered as part of the overall process of finding the MWOE for a given fragment. In performing this task for a fragment, there will be at most one of these messages associated with each node (each node receives at most one INITIATE and one ACCEPT; each sends at most one successful TEST, one REPORT, and one of either CHANGEROOT or CONNECT). Thus, the number of messages sent within a fragment in finding the MWOE is $O(m)$ where m is the number of nodes in the fragment.

The total number of messages sent in the MWOE-finding process, therefore, is proportional to:

$$\sum_{all\ fragments\ F} number\ of\ nodes\ in\ F$$

which is

$$\sum_{all\ level\ numbers\ L} \left(\sum_{all\ fragments\ F\ of\ level\ L} number\ of\ nodes\ in\ F \right)$$

Now, the total number of nodes in the inner sum at each level is at most n, since each node appears in at most one fragment at a given level-number L. And since the biggest possible value of L is $\log n$, the sum above is bounded by:

$$\sum_{1}^{\log n} n = O(n \log n)$$

Thus, the overall message complexity of the algorithm is $O(E + (n \log n))$.

Time Complexity: It can be shown by induction on l that the time for all the nodes to reach level at least l is at most $O(ln)$. Hence, the time complexity of the GHS algorithm is $O(n \log n)$.

19.2.8 Proving Correctness for the GHS Algorithm

A good deal of interesting work remains to be done in the field of proving correctness for communication algorithms like the Gallager-Humblet-Spira algorithm. The level of complexity of this code makes direct invariant assertion proofs quite difficult to do, at least at the detailed level of the code. Note that most of the discussion in this lecture has been at the higher level of graphs, fragments, levels, MWOE's, etc. So it seems that a proof ought to take advantage of this high-level structure.

One promising approach is to apply invariant-assertion and other techniques to prove correctness for a high-level description of the algorithm and then prove independently that the code in fact correctly simulates the high-level description (see [WelchLL88]).

A proof can be formalized by describing the high-level algorithm as an I/O automaton (in which the state consists of fragments, and actions include merge and absorb operations), describing the low-level code as another I/O automaton, and then producing that there is a simulation mapping between the two automata.

Lecturer: Nancy Lynch

Lecture 20

20.1 Minimum Spanning Tree (cont.)

20.1.1 Simpler "Synchronous" Strategy

Note that in the asynchronous algorithm we have complications we did not encounter in the synchronous algorithm (e.g., absorbing vs. merging, waiting for fragments to catch up). Another idea to cope with these difficulties is to try to simulate the synchronous algorithm more closely, somehow synchronizing the levels.

More specifically, we can design an algorithm based on combining fragments, where each fragment has an associated level number, as follows. Each individual node starts as a single-node fragment at level 0. This time, fragments of level l can only be combined into fragments of level $l+1$, as in the synchronous algorithm. Each node keeps a *local-level* variable, which is the latest level it knows of the fragment it belongs to; so initially every node's *local-level* is 0, and when it finds out about its membership in a new fragment of level l, it raises its local level to l. The basic idea is that a node at level l tries not to participate in the algorithm for finding its level l fragment's MWOE until it knows that all the nodes in the system have reached level at least l. Implementing this test requires global synchronization. (This is *barrier synchronization.*) But in fact, some kind of a weaker local synchronization suffices. Namely, each node only tests that all of its *neighbors* have *local-level* at least l. This requires each node to send a message on all of its incident edges every time its level is incremented.

This algorithm has the same time performance as before, i.e., $O(n \log n)$. The message complexity gets worse, however: it is now $O(E \log n)$.

20.2 Synchronizers

The idea discussed above suggests a general strategy for running a synchronous algorithm in an asynchronous network. This will only apply to a fault-free asynchronous algorithm (i.e., no failed processes or messages being lost, duplicated, etc.), since, as we shall soon see, the fault-tolerance capability is very different in synchronous and asynchronous systems. We are working from Awerbuch, with some formalization and proofs done recently by Devarajan (a student).

The strategy works for general synchronous algorithms (without faults); the idea is to synchronize at *each round* rather than *every so often* as in the MST algorithm sketched above. Doing this every so often depends on special properties of the algorithm, e.g., that it works correctly if it is allowed to exhibit arbitrary interleavings of node behavior between synchronization points. We shall only describe how to synchronize at every round.

20.2.1 Synchronous Model

Recall that in the synchronous model, each node had message generation and transition functions. It is convenient to reformulate each node as an IOA with output actions *synch-send* and input actions *synch-receive*. The node must alternate these actions, starting with *synch-send*. At rounds at which the synchronous algorithm doesn't send anything, we assume that a *synch-send*$(\emptyset)$ action occurs. We call such a node automaton a *client* automaton. The client automaton may also have other external actions, interacting with the outside world.

The rest of the system can be described as a *synchronous message system* (called "synch-sys" for short in the sequel), which at each round, collects all the messages that are sent at that round in *synch-send* actions, and then delivers them to all the client automata in *synch-receive* actions. In particular, this system synchronizes globally, after all the *synch-send* events and before all the *synch-receive* events of each round.

Note that this composition of IOA's is equivalent to the more tightly coupled synchronous model described earlier, where the message system was not modeled explicitly.

We will describe how to "simulate" the synch-sys using an asynchronous network, so that the client automata running at the network nodes cannot tell the difference between running in the simulation system and running in a real synchronous system.

20.2.2 High-Level Implementation Using a Synchronizer

The basic idea discussed by Awerbuch is to separate out the processing of the actual messages from the synchronization. He splits up the implementation into several pieces: a *front-end* for each node, able to communicate with the front-ends of neighbors over special channels, and a *pure synchronizer* S. The job of each front end is to process the messages received from the local client in *synch-send* events. At a particular round, the front end collects all the messages that are to be sent to neighbors (note that a node might send to some neighbors and not to some others). It actually sends those messages to the neighbors, along the special channels. For each such message it sends, it waits to obtain an acknowledgment along the special channel. When it has received acks for all its messages, it is *safe* (using Awerbuch's terminology – p. 807 of Awerbuch paper). For a node to be safe means that it knows that all its neighbors have received its messages. Meanwhile, while waiting for its own messages to be acknowledged, it is collecting and acknowledging the messages of its neighbors.

When is it OK for the node front-end to deliver all the messages it has collected from its neighbors to the client? Only when it knows that it won't receive any others. It is therefore sufficient to determine that *all the node's neighbors are safe* for that round (i.e., that those neighbors know that *their* messages for that round have all been delivered).

Thus, the job of the synchronizer is to tell front-ends when all their neighbors are safe. To do this, the synchronizer has OK input actions, outputs from the front-ends, by which the front ends tell the synchronizer that they are safe. The synchronizer S sends a GO message to a front end when it has received an OK message from each of its neighbors.

For now, we are just writing an abstract spec for the S component; of course, this will have to be implemented in the network. We claim that this combination "simulates" the synchronous system (combination of clients and synch-sys). Notice we did not say that it "implements" the system, in the formal sense of behavior inclusion. In fact, it doesn't; this simulation is local, in the sense that the rounds can be skewed at distances in the network. Rounds are only kept close for neighbors. But in this architecture, where the client programs do not have any means of communicating directly with each other (outside of the synch-sys) this is enough to preserve the view of each client.

Theorem 1 *If α is any execution of the implementation (clients, front ends, channels and synchronizer) then there is an execution α' of the specification (clients and synch-sys) such that for all clients i, $\alpha|client_i = \alpha'|client_i$.*

That is, as far as each client can tell, it's running in a synchronous system. So the behavior is the same — the same correctness conditions hold at the outside boundary of the clients, subject to reordering at different nodes.

Proof Sketch: We cannot do an implementation proof (as we did, for example, for the CTSS algorithm), since we are reordering the actions at different nodes. Note that certain events "depend on" other events to enable them, e.g., an OK event depends on prior ack-input events at the same front-end, and all events at one client automaton may depend on each other. (This is a conservative assumption.) We define this *depends on* relation D formally as follows. For every schedule β of the implementation system, there is a partial-order relation D_β describing events occurring in β that depend on each other. The key property of this relation D_β is that for any schedule β of the implementation system, and any permutation β' of β that preserves the partial order of events given by D_β, we have that β' is also a schedule of the implementation system. This says that the D relations are capturing enough about the dependencies in the schedule to ensure that any reordering that preserves these dependencies is still a valid schedule of the implementation system.

Now, we start with any schedule β of the implementation, and reorder the events to make the successive rounds "line up" globally. The goal is to make the reordered schedule look as much as possible like a schedule of the synchronous system. We do this by explicitly putting all the *synch-receive*(i) events after all the *synch-send*(i) events for the same i. We now claim this new requirement is consistent with the dependency requirements in D_β — since those never require the reverse order, even when applied transitively. We get β_1 in this way. By the key claim above, β_1 is also a schedule of the *implementation system*.

We're not done yet — although β_1 is fairly "synchronous", it is still a schedule of the implementation system; we want a schedule of the specification system. In the next step, we suppress the internal actions of the synch-sys, getting a "reduced" sequence β'. This still looks the same to the clients as β, and now we claim that it is a schedule of the specification system. This claim can be proved by induction.

Note that the theorem started with an execution, not a schedule. To complete the proof, we extract the schedule β, get β' as above, and then fill in the states to get an execution of the specification system (filling in the client states as in α). ▮

Note that if we care about order of events at different clients, then this simulation strategy does not work.

20.2.3 Synchronizer Implementations

Now we are left with the job of implementing the *synchronizer* part of this implementation. Its job is quite simple: it gets OK inputs from each node at each round, and for each node, it can output GO when it knows that all its neighbors in the graph (strictly speaking, including the node itself) have done input OK for that round. We want to implement this by a distributed algorithm, one piece per node, using a message system as usual. There are several ways to do this.

Synchronizer α. The α synchronizer works as follows. When a node gets an OK, it sends this information to all its neighbors. When a node hears that all its neighbors are OK (and it is OK), it outputs GO. The correctness of this algorithm is fairly obvious. The complexity is as follows. For every round, we generate $O(E)$ messages (since all nodes send messages on all edges at every round). Also, each round takes constant time (measuring time from when all the OK's at a round have happened until all the GO's for that round have happened).

We conclude that this algorithm is fast, but may be too communication-inefficient for some purposes. This brings us to the other extreme below.

Synchronizer β. For this synchronizer, we assume that there is a rooted spanning tree in G. The rule is now as follows. Convergecast all the OK info to the root, and then broadcast permission to do the GO outputs. The cost of this algorithm is $O(n)$ messages for every round, but the time to complete a round is proportional to the height of the spanning tree (which is $\Omega(n)$ in the worst case).

20.2.4 Hybrid Implementation

By combining synchronizer β and α, it is possible to get a hybrid algorithm that (depending on the graph structure) might give (simultaneously) better time efficiency than β and better message efficiency than α. The basic idea is to divide the graph up into *clusters* of nodes, each with its own spanning tree, i.e., a spanning forest. (We assume that this spanning forest is constructed using some preprocessing.) The rule now will be to use synchronizer β within each cluster-tree, and use α to synchronize between clusters. To see how this works, we found it useful to describe a high-level decomposition (see Figure 20.1).

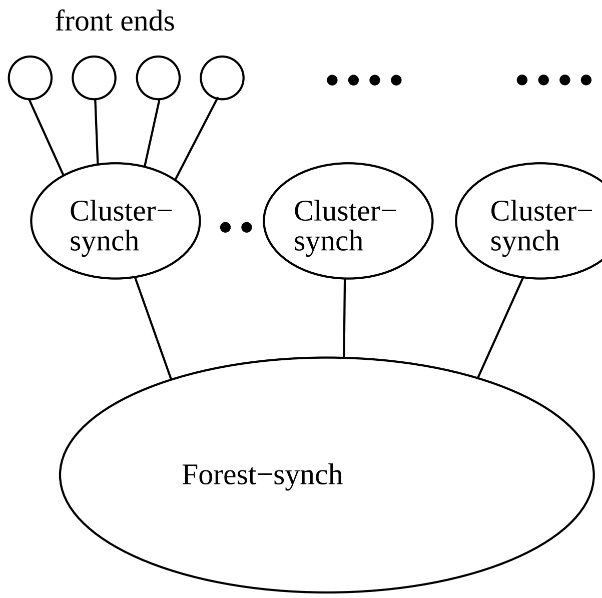


Figure 20.1: Decomposition of the synchronization task to *cluster-synch* for intra-cluster synchronization and *forest-synch* for inter-cluster synchronization.

The behavior of *cluster-synch* is to take OK's from all nodes in one particular cluster (which is a connected subset of nodes in the graph), and output a single "cluster-OK" to *forest-synch*. Then, when a single cluster-GO is input from *forest-synch*, it produces a GO for each node in the cluster. A possible way to implement this on the nodes of the given cluster, of course, is just to use the idea of synchronizer β, doing convergecast to handle the

OKs and broadcast to handle the GOs, both on a spanning tree of the cluster.

The other component is essentially (up to renaming) a synchronizer for the *cluster graph* G' of G, where the nodes of G' correspond to the clusters of G, and there is an edge from C to D in G' if in G there is an edge from some node in C to some node in D. Ignoring for the moment the problem of how to implement this forest-synchronizer in a distributed system based on G, let us first argue why this decomposition is correct. We need to show that any GO that is output at any node i implies that OKs for i and all its neighbors in G have occurred. First consider node i. The GO(i) action means that cluster-GO for i's cluster C has occurred, which means that cluster-OK for C has occurred, which means that OK(i) has occurred. Now consider $j \in \mathit{neighbors}_G(i) \cap C$. Same argument holds: cluster-OK for C means that OK(j) has occurred. Finally, let $j \in \mathit{neighbors}_G(i) - C$. The GO($i$) action means, as before, that cluster-GO for C has occurred, which implies that cluster-OK for D has occurred, where D is the cluster containing j. (The clusters are neighbors in G' because the nodes i and j are neighbors in G.) This implies as before that OK(j) has occurred.

Implementation of *forest-synch.* We can use any synchronizer to synchronize between the clusters. Suppose we want to use synchronizer α. Note that we can't run α directly, because it is supposed to run on nodes that correspond to the clusters, with edges directly connecting these nodes (clusters); we aren't provided with such nodes and channels in a distributed system. However, it is not hard to simulate them. We do this as follows. Assume we have a leader node in each cluster, and let it run the node protocol of α for the entire cluster. (This can—but doesn't have to—be the same node as is used as the root in the implementation of β for the cluster, if the cluster is synchronized using β.) The next problem to solve is the way two leaders in different clusters can communicate. We simulate direct communication using a path between them. Note that there must exist such a path, because the leaders that need to communicate are in adjacent clusters (and each cluster is connected). We need some preprocessing to determine these paths; we ignore this issue for now.

We need also to specify, for the final implementation, where the cluster-OK action occurs. This is done as an output from some node in the *cluster-synch* protocol; if synchronizer β is used, it is an output of the root of the spanning tree of the cluster. It is also an input to the leader node in the *forest-synch* for that same cluster. If these two are the same node, then this action just becomes an internal action of the actual node in the distributed system. (If these node are different, then we need to have them communicate along some path, also determined by some preprocessing; this requires an extra piece of the implementation.)

The formal structure of this hybrid algorithm is quite nice: each node in the distributed network is formally an IOA which is the composition of two other IOA's, one for each of the two protocols (intra-cluster and inter-cluster synchronizers). We can consider two orthogonal decompositions of the entire implementation: vertical (i.e., to nodes and channels), or horizontal (i.e., by the two algorithms, each distributed one piece per node).

Analysis. We shall consider the specific implementation where the cluster-synch is done using β, and the forest-synch is done using α, and we assume that the leader for α within a cluster is same node that serves as the root for β.

Consider one round. The number of messages is $O(n)$ for the work within the clusters, plus $O(E')$, where E' is the number of edges on all the paths needed for communication among the leaders. We remark that depending on the decomposition, E' could be significantly smaller than E. The time complexity is proportional to the maximum height of any cluster spanning tree.

Thus, the optimal complexity of the hybrid algorithm boils down to the combinatorial problem of finding a graph decomposition such that both the sum of the path lengths and the maximum height are small. Also, we need to establish this decomposition with a distributed algorithm. We will not address these problems here.

20.2.5 Applications

We can use the synchronizers presented above to simulate any synchronous algorithm (in the original model for synchronous systems presented earlier in the course) on an asynchronous system. Note that the synchronizer doesn't work for fault-tolerant algorithms such as Byzantine agreement, because it is not possible in this case to wait for all processes.

Ring leader election algorithms such as LeLann-Chang, Hirshberg-Sinclair, and Peterson can thus be run in an asynchronous system. But note that the message complexity goes way up if we do this. Since they already worked in an asynchronous system without any such extra synchronization, this isn't interesting. The following applications are more interesting.

Network leader election. Using the synchronizer, the algorithm that propagates the maximal ID seen so far can work as in the synchronous setting. This means that it can *count rounds* as before, waiting only *diam* rounds before terminating. It also means that the number of messages doesn't have to be excessive: it's not necessary for nodes to send constantly, since each node now knows exactly when it needs to send (once to each neighbor at each asynchronous round). Using synchronizer α, the complexity of the resulting algorithm is $O(diam)$ time, and $O(E \cdot diam)$ messages, which is better than the aforementioned naive strategy of multiple spanning trees.

Breadth-first search. For this problem, the synchronizer proves to be a big win. Recall the horrendous performance of the Bellman-Ford algorithm in the asynchronous model, and how simple the algorithm was in the synchronous setting. Now we can run the synchronous algorithm using, say synchronizer α, which leads to an $O(E \cdot diam)$ message, $O(diam)$ time algorithm. (Compare with the synchronous algorithm, which needed only $O(E)$ messages.) Note: $O(E \cdot diam)$ might be considered excessive communication. We can give a direct asynchronous algorithm based on building the BF tree in levels, doing broadcast-convergecast at each level, adding in the new leaves for that level, and terminating when no new nodes are discovered. For this algorithm, the time is then $O(diam^2)$, which is worse than the algorithm based on the synchronizer. But the message complexity is only $O(E + n \cdot diam)$, since each edge is explored once, and tree edges are traversed at most a constant number of times for each level. It is also possible to obtain a time-communication tradeoff by combining the two strategies. More specifically, we can explore k levels in each phase using α, obtaining time complexity of $O(\frac{diam^2}{k})$ and communication complexity of $O(E \cdot k + \frac{n \cdot diam}{k})$, for $1 \leq k \leq diam$.

Weighted Single-source Shortest Paths. Using the synchronous algorithm with synchronizer α yields an algorithm with $O(n)$ time complexity, and $O(En)$ messages. We remark that there is an algorithm with fewer messages and more time (cf. Gabow, as developed by Awerbuch in his notes).

Maximal Independent Set. We can apply the synchronizer to a randomized synchronous algorithm like MIS too. We omit details.

20.3 Lower Bound for Synchronizers

Awerbuch's synchronizer result suggests that a synchronous algorithm can be converted into a corresponding asynchronous algorithm without too great an increase in costs. In particular, by using synchronizer α, it is possible not to increase the time cost at all. The following result by Arjomandi-Fischer-Lynch shows that this approach cannot be adopted universally. In particular, it establishes a lower bound on time for an asynchronous algorithm to solve a particular problem. Since there is a very fast synchronous algorithm for the problem, this means that not every fast synchronous algorithm can be converted to an asynchronous algorithm with the same time complexity. Note that the difference is *not* caused by requiring any fault-tolerance, so the result may seem almost contradictory to the synchronizer results.

We start by defining the problem. Let $G = (V, E)$ be a graph. Recall that $diam(G)$ denotes the maximum distance between two nodes in G. The system's external interface consists of $flash_i$ output actions, for all $i \in V$, where $flash_i$ is an output of the I/O automaton at node i. As an illustration, imagine that the $flash_i$ is a signal that node i has completed some task. Define a *session* as a sequence of flashes in which at least one $flash_i$ occurs for every i. We can now define the *k-session problem:* we require simply that the algorithm should perform at least k disjoint sessions.

The original motivation for this setting was that the nodes were performing some kind of coordinated calculation on external data, e.g., a Boolean matrix transitive closure, in the PRAM style. Each node (i, j, k) was responsible for writing a 1 in location (i, j), in case it ever saw 1's in both locations (i, k) and (k, j). Notice that for this problem, it doesn't matter whether the nodes do excessive checking and writing. Correctness is guaranteed if we have at least $\log n$ sessions (i.e., if there is "enough" interleaving).

The reason the problem is stated this way is that it is a more general problem than the simpler problem in which all processes do exactly one step in each session (so it strengthens the impossibility result). Note also that this is a weak problem statement: it doesn't even require the nodes to know when k sessions have completed.

Before we turn to prove the lower bound result, let us make the problem statement more precise. As usual, we model the processes as IOAs, connected by FIFO channels. For simplicity, we let the node automata partition consist of a single-class (intuitively, the nodes are executing sequential programs).

Our goal is to prove an inherent time complexity result. (Note that this is the first lower bound on time that we are proving in this course.) We need to augment the model by associating times, as usual, with the events, in a monotone nondecreasing way (approaching infinity in an infinite fair execution). Let l be a bound for local step time, and d be the bound for delivery of first message in each channel. (This is a special case of the general notion of time bounds for IOA's.) We assume that $d \gg l$. An execution with times associated with events satisfying the given requirements is called a *timed execution.*

Next, we define the *time measure* $T(A)$ for algorithm A as follows. For each execution α of the algorithm, define $T(\alpha)$ to be the supremum of the times at which a *flash* event occurs in α. (There could be infinitely many such events, hence we use a supremum here.) Finally, we define

$$T(A) = \sup_{\alpha}(T(\alpha)) \ .$$

We can now state and prove our main result for this section

Theorem 2 *Suppose A is an algorithm that solves the k-session problem on graph G. Then $T(A) \geq (k-1) \cdot diam(G) \cdot d$.*

Before we turn to prove the theorem, consider the k-session problem in the synchronous case. We can get k sessions without the nodes ever communicating — each node just does a *flash* at every round, for k rounds. The time would be only kd (for a time measure normalized so that each round counts for d time). This discrepancy proves that the inherent multiplicative overhead due to asynchrony for some problems is proportional to $diam(G)$.

Proof Sketch: *(of Theorem 2)*
By contradiction. Suppose that there exists an algorithm A with $T(A) < (k-1) \cdot diam(G) \cdot d$.

Call a timed execution *slow* if all the actions take their maximum times, i.e., if the deliveries of the first messages in the channels always takes time d and the step times always take l. Let α be any slow (fair) timed execution of the system. By the assumption that A is correct, α contains k sessions. By the contradiction assumption, the time of the last *flash* in α is strictly less than $(k-1) \cdot diam(G) \cdot d$. So we can write $\alpha = \alpha'\alpha''$, where the time of the last event in α' is less than $(k-1) \cdot diam(G) \cdot d$, and where there are no *flash* events in α''. Furthermore, because of the time bound, we can decompose $\alpha' = \alpha_1\alpha_2 \ldots \alpha_{k-1}$ where each of the α_r has the difference between the times of its first and last events strictly less than $diam(G) \cdot d$.

We now construct another fair execution of the algorithm, $\beta = \beta_1\beta_2 \ldots \beta_{k-1}\beta''$. This will be an ordinary *untimed* fair execution; that is, we do not assign times to this one. The execution β is constructed by reordering the actions in each α_r (and removing the times) to obtain β_r, and by removing the times from α'' to get β''. (Of course, when we reorder events, we will have to make some adjustments to the intervening states.) We will show that the modified execution contains fewer than k sessions, which contradicts the correctness of A. The reordering must preserve certain dependencies in order to produce a valid execution of A. Formally, we shall prove the following claim.

Claim 3 *Let $i_0 = i_2 = i_4 = \cdots = i$, and $i_1 = i_3 = i_5 = \cdots = j$, where $dist(i,j) = diam(G)$. For all $r = 1, \ldots, k-1$ there exists a "reordering" $\beta_r = \gamma_r\delta_r$ of α_r (i.e., the actions are reordered, with the initial and final states unchanged), such that the following properties hold.*

1. *The reordering preserves the following* dependency *partial order.*
   (a) *The order of all actions of the same node.*
   (b) *The order of each $send_{i,j}(m)$ event and its corresponding $receive_{i,j}(m)$ event.*
2. *γ_r contains no event of i_{r-1}.*
3. *δ_r contains no event of i_r.*

Let us first show how to complete the proof of Theorem 2 using Claim 3. Since the reordering preserves the dependencies, this means that $\beta_1\beta_2 \ldots \beta_{k-1}\beta''$ is a fair execution of A. But we can show that this execution contains at most $k-1$ sessions as follows. No session can be entirely contained within γ_1, since γ_1 contains no event of i_0. Likewise, no session can be entirely contained within $\delta_{r-1}\gamma_r$, since this sequence contains no event of process i_{r-1}. This implies that each session must contain events on both sides of some $\gamma_r\delta_r$ boundary. Since there are only $k-1$ such boundaries, the theorem follows.

It remains to prove Claim 3. To do this, we produce the required reorderings. The construction is done independently for each r. So fix some r, $1 \leq r \leq k-1$. We consider two cases.

Case 1: α_r contains no event of i_{r-1}. In this case we define $\gamma_r = \beta_r$ and δ_r is empty.

Case 2: α_r contains an event of i_{r-1}. Let π be the first event of i_{r-1} in α_r. Now we claim that there is no step of i_r in α_r that depends on π (according to the dependency relation defined in Claim 3). This is true since the time for a message to propagate from i_{r-1} to i_r in a slow execution is at least $diam \cdot d$, whereas we have constructed α_r so that the time between its first and last events is strictly less than $diam \cdot d$. Thus we can reorder α_r so that all the steps of i_r *precede* π: we keep the initial and final states the same, and mimic the local state changes as in α_r. This still yields allowable transitions of the algorithm, by a dependency theorem similar to the one we used for the synchronizer construction (but simpler). Let γ_r be the part of the reordered sequence before π, and δ_r the part of the sequence starting with π. It is straightforward to verify that $\beta_r = \gamma_r \delta_r$ has the properties required to prove Claim 3. ∎

Note that the reordered sequence is *not* a timed execution — if we were trying to preserve the times somehow during the reordering, we would be stretching and shrinking some time intervals. We would have to be careful to observe the upper bound requirements. We could avoid these problems by making all the time intervals very small, but we don't need to worry about this at all, since all we need to get the contradiction is an untimed fair execution.

Theorem 2 almost looks like a contradiction to the synchronizer result — recall that the synchronizer gives a simulation with *constant* time overhead. This is not a contradiction, however, because the synchronizer simulation doesn't preserve the total external behavior: it only preserves the behavior at each node, while it may reorder the events at different nodes. We remark that for most purposes, we might not care about global reordering, but sometimes, when there is some "out-of-band" communication, the order of events at different nodes might be important.

Lecturer: Nancy Lynch

Lecture 21

21.1 Time, Clocks, etc.

The idea that it is OK to reorder events at different nodes motivates Lamport's important notion of "logical time", defined in his famous paper *Time, Clocks, and the Ordering of Events in a Distributed System.*

21.1.1 Logical Time

The model assumed is the one we have been using. In this model, there is no built-in notion of real-time (in particular, there are no clocks), but it is possible to impose a logical notion of time, namely Lamport's *logical time.* Specifically, every event of the system (send or receive of messages, external interface event, internal event of node) gets assigned a *logical time*, an element of some fixed totally ordered set. Typically, this set is either the nonnegative integers or the nonnegative reals (perhaps with tie-breakers). These logical times don't have to have any particular relationship to any notion of real time. However, they do need to satisfy the following properties.

1. No two events get assigned the same logical time.
2. Events at each node obtain logical times that are strictly increasing, in the same order as the events occur.
3. For any message, its send event gets a strictly smaller logical time than its receive event.
4. For any event, there are only finitely many other events that get assigned logical times that are smaller.

The important result about such an assignment is as follows. Suppose that we are given an execution α of a network system of IO automata and a logical time assignment *ltime* that satisfies the conditions above. Then there is another execution α' of the same system, which looks the same to each node (i.e., $\alpha|i = \alpha'|i$ for all i), and in which the times from the assignment *ltime* occur in increasing order *in real time*. In other words, we can have a reordered execution, such that the logical time order of the original execution is the same as the real time order in the reordered execution. So we can say that programming in terms of logical time looks to all the users as if they are programming in terms of real time.

We can view this as another way to reduce the uncertainty of an asynchronous network system. Roughly speaking, we can write asynchronous programs where we assume that each node has "access to" real time. (Real time modeling doesn't fit the IOA model, which is why this is rough. It will be done more carefully later in the course when we do timing-based models.) In the basic model, the nodes don't have this access. Instead, we can have them implement logical time somehow, and let them use the logical time in place of the real time. The reordering result says that the resulting executions will look the same to all nodes (separately).

One generalization is useful: suppose we want to augment the model to allow *broadcast* actions, which serve to send the same message to all the other nodes. We can implement this, of course, with a sequence of individual *send* actions. There is nothing too interesting here thus far. But the important thing here is that we can allow a broadcast to have a single logical time, and require that all the associated receives have larger logical times. The reordering result still holds with this new action.

21.1.2 Algorithms

Lamport's implementation. Lamport presents a simple algorithm for producing such logical times. It involves each node process maintaining a local variable *clock* that it uses as a local clock. The local clock gets incremented at each event (input, output or internal) that occurs at that node. The logical time of the event is defined to be the value of the variable *clock*, *immediately after* the event, paired with the process index as a tie-breaker.

This algorithm is easily seen to ensure Properties 1 and 2 above. In order to ensure Property 3, the following rule is observed. Whenever a node does a *send* (or *broadcast*) event, it first increments its *clock* variable to get the logical time for the *send* event, and it attaches that clock value to the message when it sends it out. When the receiver of the message performs a *receive* event, it makes sure that it increments its *clock* variable to be not only larger than its previous value, but also strictly larger than the clock value in the message. As before, it is this new clock value that gets assigned to the *receive* event. To ensure Property 4, it suffices to allow each increase of a *clock* variable to increase the value by some minimum amount (e.g., 1).

Welch's Implementation. In Welch's algorithm, as in Lamport's, the logical time at each node is maintained by a state variable named *clock*, which can take on nonnegative integer or real values. But this time, we assume that there is an input action $tick(t)$ that is responsible for setting this variable. The node sets its *clock* variable to t when $tick(t)$ is received. We can imagine this input as arriving from a separate "clock" IOA. (The node implements both this clock and the process IOA.)

Now each non-*tick* action gets its logical time equal to the value of *clock* when it is performed, with tie breakers: first, by the process index, and second, by execution order at the same process. Note that the *clock* value does not change during the execution of this action. Some additional careful handling is needed to ensure Property 3. This time, we have an extra FIFO *receive-buffer* at each node, which holds messages whose clock values are at least the clock value at the local node. Assuming that the local *clock* value keeps increasing without bound, eventually it will be big enough so that it dominates the incoming message's clock value. The rule is to wait for the time where the first message in the *receive-buffer* has an associated clock less than the local *clock*. Then this message can be processed as usual: we define the logical time of the receive event in terms of the local *clock* variable when this simulated receive occurs.

The correctness of this algorithm is guaranteed by the following facts. Property 1 is ensured by the the tie-breakers; Property 2 relies on the monotonicity of the local clocks plus the tie-breakers; Property 3 is guaranteed by the buffer discipline; and Property 4 requires a special property of the clock automaton, namely that it must grow unboundedly.[11] This could be achieved, e.g., by having a positive lower bound on the tick increment, and having the clock automaton execute fairly.

Note that this algorithm leads to bad performance when the local clocks get out of synch. For this to work in practice, we need some synchronization among the clocks, which is nor really possible in a purely asynchronous system. We could use a synchronizer for the local clocks, but it is too expensive to do this at every step. Instead, we could use a synchronizer every so often, say every 1000 steps. Still, in a pure asynchronous system, the worst-case behavior of this strategy will be poor. This algorithm makes more sense in a setting with some notion of real-time.

21.1.3 Applications

It is not clear that the above algorithms can always be used to implement some *ltime* abstraction, which then can be used as a "black box". The problem is that the logical time

[11]In other words, we don't allow *Zeno* behaviors, in which infinite executions take a finite amount of time.

abstraction does not have a clear interface description. Rather, we'll consider uses of the general idea of logical time. Before we go into more sophisticated applications, we remark that both algorithms can be used to support logical time for a system containing broadcast actions, as described above.

Banking system

Suppose we want to count the total amount of money in a banking system in which no new money is added to or removed from the system, but it may be sent around and received (i.e., the system satisfies *conservation of money*). The system is modeled as some collection of IOAs with no external actions, having the local amount of money encoded in the local state, with send actions and receive actions that have money parameters. The architecture is fixed in our model; the only variation is in the decision of when to send money, and how much.

Suppose this system somehow manages to associate a logical time with each event as above. Imagine that each node now consists of an augmented automaton that "contains" the original automaton. This augmented automaton is supposed to "know" the logical time associated with each event of the basic automaton. In this case, in order to count the total amount of money in the bank, it suffices to do the following.

1. Fix some particular time t, known to all the nodes.

2. For each node, determine the amount of money in the state of the basic automaton after processing all events with logical time less than or equal to t (and no later events).

3. For each channel, determine the amount of money in all the messages sent at a logical time at most t but received at a logical time strictly greater than t.

Adding up all these amounts gives a correct total. This can be argued as follows. Consider any execution α of the basic system, together with its logical time assignment. By the reordering result, there's another execution α' of the same basic system as above and same logical time assignment, that looks the same to all nodes, and all events occur in logical time order. What this strategy does is "cut" execution α' immediately after time t, and record the money in all the nodes, and the money in all the channels. This simple strategy is thus giving an *instantaneous snapshot* of the system state, which gives the correct total amount of money in the banking system.

Let us consider how to do this with a distributed algorithm. Each augmented automaton i is responsible for overseeing the work of basic automaton i. Augmented automaton i is assumed able to look inside basic automaton i to see the amount of money. (Note that this is not ordinary IOA composition: the interface abstractions are violated by the coupling.) It is also assumed to know the logical time associated with each event of the basic automaton. The augmented automaton i is responsible for determining the state of the basic automaton at that node, plus the states of all the incoming channels.

The augmented automaton i attaches the logical time of each send event to the message being sent. It records the basic automaton state as follows. Automaton i records the state after each step, until it sees some logical time greater than t. Then it "backs up" one step and returns the previous state. For recording the channel state, we need to know the messages that are sent at a logical time (at the sender) at most t and received at a logical time (at the receiver) strictly greater than t. So as soon as an event occurs with logical time exceeding t (that is, when it records the node state), it starts recording messages coming in on the channel. It continues recording them as long as the attached timestamp is no more than t. Note: to ensure termination, we need to assume that each basic node continues sending messages every so often, so that its neighbors eventually get something with timestamp strictly larger than t. Alternatively, the counting protocol itself could add some extra messages to determine when all the sender's messages with attached time at most t have arrived.

We remark that the augmented algorithm has the nice property that it doesn't interfere with the underlying operation of the basic system.

General Snapshot

This strategy above can, of course, be generalized beyond banks, to arbitrary asynchronous systems. Suppose we want to take any such system that is running, and determine a global state at some point in time. We can't actually do this without stopping everything in the system at one time (or making duplicate copies of everything). It is not practical in a real distributed system (e.g., one with thousands of nodes) to really stop everything: it could even be impossible. But for some applications, it may be sufficient to get a state that just "looks like" a correct global state, as far as the nodes can tell. (We shall see some such applications later.) In this case, we can use the strategy described above.

Simulating a Single State Machine

Another use, mentioned but not emphasized in Lamport's paper (but very much emphasized by him in the ensuing years) is the use of logical time to allow a distributed system to simulate a centralized state machine, i.e., a single object. The notion of an object that works here is a very simple one – essentially, it is an IO automaton with input actions only; we require that there be only one initial value and that the new state be determined by the old state and the input action. (This may not seem very useful at first glance, but it can be used to solve some interesting synchronization problems, e.g., we will see how to use it to solve mutual exclusion. It's also a model for a database with update operations.)

Suppose now that there are n users supplying inputs (updates) at the n node processes of the network. We would like them all to apply their updates to the same centralized object. We can maintain one copy of this object in one centralized location, and send all the updates there. But suppose that we would like all the nodes to have access to (i.e., to be able to read) the latest available state of the object. (E.g., suppose that reads are more frequent than writes, and we want to minimize the amount of communication.) This suggests a replicated implementation of the object, where each node keeps a private copy; we can broadcast all the updates to all the nodes, and let them all apply the updates to their copies. But they need to update their copies in the same way, at least eventually. If we just broadcast the updates, nothing can stop different nodes from applying them in different orders. We require a total order of all the updates produced anywhere in the system, known to all the nodes, so that they can all apply them in the same order. Moreover, there should only be finitely many updates ordered before any particular one, so that all can eventually be performed. Also, a node needs to know when it's safe to apply an update — this means that no further updates that should have been applied before (i.e., before the one about to be applied) will ever arrive. We would like the property of monotonicity of successive updates submitted by a particular node, in order to facilitate this property.

The answer to our problems is logical time. When a user submits an update, the associated process broadcasts it, and assigns it a *timestamp*, which is the logical time assigned to the broadcast event for the update. (If a node stops getting local updates as inputs, it should still continue to send some *dummy* messages out, just to keep propagating information about its logical time.) Each node puts all the updates it receives, together with their timestamps, in a "request queue", not yet applying them to the object. (Note: this queue must include all the updates the node receives in broadcasts by other nodes, plus those the node sends in its own broadcasts.) The node applies update u if the following conditions are satisfied.

1. Update u has the smallest timestamp of any request on the request queue.

2. The node has received a message (request or dummy) from every node with send time at least equal to the timestamp of u.

Condition 2 and the FIFOness of the channels guarantee that the node will never receive anything with smaller timestamp than this first update. Furthermore, any node eventually succeeds in applying all updates, because logical time keeps increasing at all nodes. Note that we didn't strictly need Property 3 of logical time (relating the times of message send and receive) here for correctness. But something like that is important for performance.

Example: Banking distributed database. Suppose each node is a bank branch, and updates are things like deposit, withdraw, add-interest, withdraw-from-checking-if-sufficient-funds-else-from-savings, etc. Note that in some of these cases, the order of the updates matters. Also suppose that all the nodes want to keep a recent picture of all the bank information (for reading, deciding on which later updates should be triggered at that branch, etc.). We can use logical time as above to simulate a centralized state machine representing the bank information.

Note that this algorithm is not very fault-tolerant. (Also, in general, it does not seem very efficient. But it can sometimes be reasonable — see the mutual exclusion algorithm below.)

21.2 Simulating Shared Memory

In this section we consider another simplifying strategy for asynchronous networks. We use this strategy to simulate (instantaneous) shared memory algorithms.

The basic idea is simple. Locate each shared variable at some node. Each operation gets sent to the appropriate node, and the sending process waits for a response. When the operation arrives at the simulated location, it gets performed (possibly being queued up before being performed), and a response is sent back. When the response is received by the sending process, it completes its step. If new inputs arrive from the outside world at a process during a "waiting period", the process only queues them. These pending inputs are processed only when the anticipated response returns.[12] We now claim that this simulation is correct, i.e., that *every* external behavior is also an external behavior of the instantaneously shared memory system being simulated; also, every fair behavior (of nodes and channels) is also a fair behavior of the instantaneous shared memory system; also, some kind of wait-freedom carries over. To prove this claim, we use the corresponding result for atomic shared memory simulating instantaneous shared memory, and the observation that we are actually simulating atomic objects using a centralized object and the delays in the message system. We omit details.

The problem of where to put the shared variables depends on the characteristics of the application at hand. E.g., for a system based on single-writer multi-reader shared variables, where writes are fairly frequent, we would expect to locate the variables at the writer's node.

[12]Earlier in the course we had a restriction for the atomic shared memory simulation of instantaneous shared memory, that no new inputs arrive at any node while the node process is waiting for an invocation to return. The reason for this was to avoid introducing any new interleavings. It seems, however, that we could have handled this by just forcing the process to queue up the inputs as above, and so remove the need for this restriction.

Note that the objects have to be simulated even while the nodes don't have active requests.

There are other ways of simulating shared memory algorithms.

Caching. For single-writer multi-reader registers, note that someone who wants to repeatedly test the value of a register has, in the implementation described above, to send repeated messages even though the variable is not changing. It is possible to develop an alternative strategy by which writers notify readers before they change the value of the variable, so if the readers haven't heard about a modification, they can just use their previous value. This is the basic idea of *caching*.

Replicated copies. Caching is one example of a general style of implementation based on *data replication*. Data can be replicated for many reasons, e.g., fault-tolerance (which we will discuss later). But often it is done only for *availability*. Suppose we have a multi-writer multi-reader shared register, in which writes are very infrequent. Then we can locate copies of the register *only* at all the readers' nodes (compare with the caching example). A reader, to read the register, can always look at its local copy. A writer, to write the register, needs to write all the copies. Note that it needs to do this atomically (since one at a time may cause out-of-order reads). Therefore we need some extra protocol here to ensure that the copies are written as if atomically. This requires techniques from the area of *database concurrency control*.

21.3 Mutual Exclusion and Resource Allocation

In this section we consider the problem of resource allocation in a distributed message-passing network.

21.3.1 Problem Definition

We have the same interface as before, but now the inputs and outputs for user i occur at a corresponding node i (see Figure 21.1). The processes, one for each i, communicate via messages over FIFO channels.

Consider the resource requirement formulation of the problem (in terms of a monotone Boolean formula involving the needed resources). Each node i has a static (fixed) resource requirement. Assume that any two nodes that have any common resource in their two resource requirements are neighbors in the network graph, so they can communicate to negotiate priority for the resources. We do not model the resources separately.

In this setting, we typically drop the restriction that nodes can perform locally-controlled steps only when they are between requests and responses. This is because the algorithms

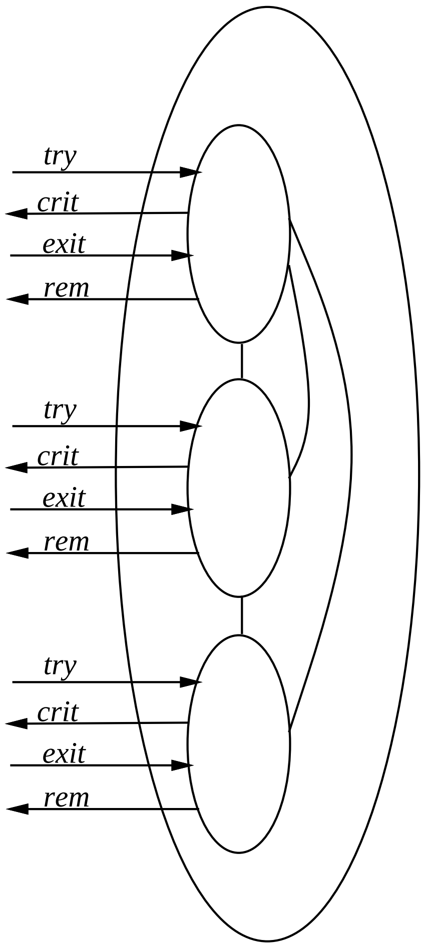


Figure 21.1: Interface specification for resource allocation on a message passing system

will typically need to respond to requests by their neighbors for the resources.

Th correctness conditions are now as follows. As before, we require a solution to preserve *well-formedness* (cyclic behavior), and also *mutual exclusion* or a more general exclusion condition.

We also require *deadlock-freedom*, i.e., if there is any active request and no one in C, then some request eventually gets granted, and if there is any active exit region then some exit eventually occurs. This time the hypothesis is that all the node and channel automata exhibit fair behavior (in the IOA sense, i.e., all the nodes keep taking steps and all the channels continue to deliver all messages).

The *lockout-freedom* condition is defined as before, under the hypothesis that all the node and channel automata exhibit fair behavior.

For the *concurrency* property, suppose that a request is invoked at a node i, and all neighbors are in R and remain in R. Suppose that execution proceeds so that i and all its neighbors continue to take steps fairly, and the connecting links continue to operate fairly. (Note that other processes can remain in C forever.) Then the concurrency condition requires that eventually i reach C.

We remark that this is analogous to the weak condition we had for shared memory systems, only here we add fairness requirements on the neighbors and the links in between.

21.3.2 Mutual Exclusion Algorithms

Raynal's book is a good reference (also for the shared memory mutual exclusive algorithms). We have many possible approaches now.

Simulating Shared Memory

We have learned about several shared memory algorithms for mutual exclusion. We can simply simulate one of these in a distributed system. E.g., use the Peterson multi-writer multi-reader shared register tournament algorithm (or the variant of this that we didn't have time to cover), or some version of the bakery algorithm (maybe with unbounded tickets).

LeLann

LeLann proposed the following simple solution. The processes are arranged in a logical ring $p_1, p_2, \ldots, p_n, p_1$. A *token* representing control of the resource is passed around the ring in order. When process p_i receives the token, it checks for an outstanding request for the resource from $user_i$. If there is no such request, the token is passed to the next process in the ring. If there is an outstanding request, the resource is granted and the token is held until the resource is returned and then passed to the next process.

The code for process p_i is given in Figure 21.2.

Let us go over the properties of LeLann's algorithm.

Mutual Exclusion is guaranteed in normal operation because there is only one token, and only its holder can have any resource.

Deadlock-Freedom: in fair executions, when no one is in C, the process that holds the token is either:

- in T, and then it can go to C, or
- in $E \cup R$, and then it has to pass the token to the next process.

Similarly, the deadlock-freedom for the exit region is satisfied.

Lockout-Freedom is straightforward.

Local variables:

- $token \in \{none, available, in\_use, used\}$, initially $available$ at p_1, and $none$ elsewhere
- $region \in \{R, T, C, E\}$, initially R

Actions:

```
$try_i$
  Effect: $region \leftarrow T$

$crit_i$
  Precondition: $region = T$
                $token = avail$
  Effect: $region \leftarrow C$
          $token \leftarrow in\_use$

$exit_i$
  Effect: $region \leftarrow E$

$rem_i$
  Precondition: $region = E$
  Effect: $region \leftarrow R$
          $token \leftarrow used$

$receive_{i-1,i}(token)$
  Effect: $token \leftarrow available$

$send_{i,i+1}(token)$
  Precondition: $token = used \bigvee (token = available \bigwedge region \neq T)$
  Effect: $token \leftarrow none$
```

Figure 21.2: LeLann mutual exclusion algorithm for message passing systems

Performance: First, let's consider the number of messages. It is not clear what to measure, since the messages aren't naturally apportioned to particular requests. E.g., we can measure the worst-case number of messages sent in between a try_i and corresponding $crit_i$, but it seems more reasonable to try to do some kind of *amortized analysis*. Note that each virtual link in the logical ring is really of unit length, since the graph is complete. In the worst case, n messages are sent between try_i and $crit_i$. For the amortized cost, we consider the case of "heavy load", where there is always an active request at each node; in this case there are only a constant number of messages per request. If a request comes in alone, however, we still get n messages in the worst case. Also, it is hard to get a good bound statement here, since messages are sent even in the absence of any requests at all.

For the time complexity, we assume worst-case bounds. Let c be the time spent in C, d be the message delay for the first message in any channel, and s be the process step time. The worst-case time is approximately $(c + d + O(s)) \cdot n$. Note that, unfortunately, this time bound has a $d \cdot n$ term, regardless of the load, and this might be big.

Resiliency: LeLann discusses various types of resiliency in his paper.

- *Process failure:* If a process stops and announces its failure, the rest of the processes can reconfigure the ring to bypass the failed process. This requires a distributed protocol. Note that a process that doesn't announce its failure can't be detected as failed, since in an asynchronous system, there is no way for a process to distinguish a failed process from one that is just going very slowly.

- *Loss of token:* When a token loss is detected (e.g., by timeout), a new one can be generated by using a *leader-election* protocol, as studied earlier.

Lamport's algorithm

Another solution uses Lamport logical time. In particular, the state machine simulation approach is applied. The state machine here has as its state a *queue* of process indices, which is initially empty. The operations are try_i and $exit_i$, where try_i has the effect of adding i to the end of *queue* and $exit_i$ has the effect of removing i from *queue*, provided it is at the front. When user i initiates a try_i or $exit_i$ event, it is regarded as submitting a corresponding "update" request. Then, as described above, the state machine approach broadcasts the updates, waits to be able to perform them, i.e., a node can perform an update when that update has the smallest timestamp of any update in the request buffer, and the node knows it has all the smaller timestamp updates. When the node is able to perform the update, it does so. The replicated object state is used as follows. When the *queue* at node i gets i at the front, the process at node i is allowed to do a $crit_i$. When the *queue* at node

i gets i removed, the process at node i is allowed to do a rem_i.

Let us now argue that this algorithm guarantees mutual exclusion. Suppose not; say p_i and p_j are simultaneously in C, and suppose, without loss of generality, that p_i has the smaller timestamp. Then when j was added to p_j's queue, the state machine implementation ensures that i must have already been there, ahead of j. But i has not been removed (since it is still in C). So j could not have done a $crit_i$ action, a contradiction.

It is a little hard to analyze the algorithm in this form, since it combines three protocols: the logical time generation protocol, the replicated state machine protocol, and the extra rules for when to do *crit* and *rem* actions. We can merge and "optimize" these protocols somewhat as follows. Assume that the logical time generation is done according to Lamport's algorithm, based on the given messages. We combine the request-buffer and the simulated state machine's state. Also, the *ack* messages below are used as the *dummy* messages discussed in the general approach.

In this algorithm, every process p_i maintains a local variable *region* as before, and for each other process p_j a local queue $queue_j$. This $queue_j$ contains all the messages ever received from p_j (we remark that this can be optimized). There are three types of messages:

- *try-msg*(i): Broadcast by p_i to announce that it is trying.
- *exit-msg*(i): Broadcast by p_i to announce that it is exiting.
- *ack*(i): Sent by p_i to p_j, acknowledging the receipt of a *try-msg*(j) message.

The code is written so as to send all these messages at the indicated times. We assume that the sending or broadcast logical times are piggybacked on the messages, and the queues contain these logical times as well as the messages themselves. Also, the logical time of each of the broadcast events (for try or exit requests) is used as the timestamp of the corresponding update.

This already tells what messages are sent and when. We won't have a separate application step to apply the update to the queue, since we do not have an explicit representation of the queue. All we need now is rules for p_i telling when to perform $crit_i$ and rem_i actions. Below we give these rules.

- $\underline{p_i \rightarrow R:}$ Can be done anytime after $exit_i$ occurs.
- $\underline{p_i \rightarrow C:}$ Must ensure that $region = T$ and that the following conditions hold.

  1. Mutual exclusion is preserved.
  2. There is no other request pending with a smaller timestamp.

Process p_i can ensure that the above conditions are met by checking for each $j \neq i$:

1. Any *try-msg* in $queue_j$ with timestamp less than the timestamp of the current $try - msg$ of p_i has an *exit-msg* in $queue_j$ with a larger timestamp than that of j's *try-msg*.
2. $queue_j$ contains some message (possibly an *ack*) with timestamp larger than the timestamp of the current *try-msg* of p_i

Lamport's algorithm has the following properties.

Mutual Exclusion. This can be proven by contradiction as follows. Assume that two processes, p_i and p_j, are in C at the same time, and (without loss of generality) that the timestamp of p_i's request is smaller than the timestamp of p_j's request. Then p_j had to check its $queue_i$ in order to enter C. The second test and FIFO behavior of channels imply that p_j had to see p_i's *try-msg*, so by the first test it had also to see an *exit-msg* from p_i, so p_i must have already left C.

Lockout-freedom. This property results from servicing requests in *timestamp* order. Since each event (in particular, request event) has a finite number of predecessors, all requests will eventually get serviced.

Complexity. Let us first deal with the number of messages. We shall use amortized analysis here as follows. Every request involves sending *try-msg*, *ack* and *exit-msg* messages between some process and all the others, thus $3(n-1)$ messages are sent per request. For the time complexity, note that when there is a single request in the system, the time is $2d+O(s)$, where d is the communication delay and s is the local processing time. We assume that the broadcast is done as one atomic step; if $n-1$ messages are treated separately, the processing costs are linear in n, but these costs are still presumed to be small compared to the communication delay d. (Recall that, in contrast, the time complexity of Lelann's algorithm had a dn term.)

Lecturer: Nancy Lynch

Lecture 22

22.1 Mutual Exclusion (cont.)

Last time, we gave Lamport's mutual exclusion algorithm based on his state-machine approach. Recall that we have *try* and *exit* messages as input to the state machine, and *ack* messages to respond to *try* messages. The following two variants provide interesting optimizations of this algorithm.

22.1.1 Ricart & Agrawala (1981)

Ricart and Agrawala's algorithm uses only $2(n-1)$ messages per request. It improves Lamport's original algorithm by acknowledging requests in a careful manner that eliminates the need for *exit* messages. This algorithm uses only two types of messages: *try* and *OK*. Process p_i sends $try(i)$ as in Lamport's algorithm, and can go critical after *OK* messages have been received from all the others. So the interesting part is a rule for when to send an *OK* message. The idea is to use a priority scheme. Specifically, in response to a *try*, a process does the following.

- Replies with *OK* if it is not critical or trying.
- If it is critical, it defers the reply until it exits, and then immediately sends all the deferred *OK*s.
- If it is trying, it compares the *timestamp* of its own request to the timestamp of the incoming *try*. If its own is bigger, its own is interpreted as lower priority, and the process sends *OK* immediately; else (having higher priority) the process defers acknowledging the request until it finishes with its own critical region.

In other words, when there is some conflict, this strategy is to resolve it in favor of the "earlier" request, as determined by the timestamps.

Properties of the Ricart-Agrawala Algorithm.

Mutual Exclusion: Proved by contradiction. Assume that processes p_i and p_j are both in C and (without loss of generality) that the timestamp of p_i's request is smaller than the timestamp of p_j's request. Then there must have been *try* and *OK* messages sent from each to the other, prior to their entry to C; at each node, the receipt of the *try* precedes the sending of the matching *OK*. But there are several possible orderings of the various events (see Figure 22.1).

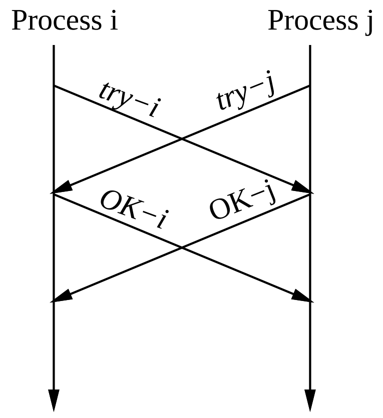


Figure 22.1: A possible scenario for the Ricart-Agrawala algorithm.

Now, the timestamp of i is less than that of j, and the logical time of the receipt of j's *try* by i is greater than the timestamp of j's request (by the logical time property). So it must be that the receipt of p_j's *try* message at i occurs after p_i has broadcast its *try*. Then at the time p_i receives p_j's *try*, it is either trying or critical. In either case, p_i's rules say it has to defer the *OK* message, thus p_j could not be in C, a contradiction.

Deadlock-freedom is also proved by contradiction. Assume some execution that reaches a point after which no progress is achieved. That is, at that point all the processes are either in R or T, none are in C, and from that point on, no process changes regions. By going a little further out in the execution, we can also reach a point after which no messages are ever in transit. Among all the processes in T after that point, let p_i be the process having the request message with the lowest timestamp. Then since p_i is blocked forever it must be because some other process p_j has not returned an *OK* message to it. Process p_j could only have deferred the *OK* because it was either

- in C: but since p_j eventually leaves C, it must eventually send the deferred *OK*.
- in T: in this case p_j deferred the *OK* because the timestamp of its request was smaller than p_i's. Since p_i's request has the smallest timestamp in T now, p_j must have completed, thus after exiting C it had to send the deferred *OK*.

In either case we get a contradiction.

Fairness: it can be shown that the algorithm has lockout-freedom.

Complexity: it is easy to see that there are $2(n-1)$ messages per request. The time complexity is left as an exercise.

22.1.2 Carvalho & Roucairol (1983)

This algorithm improves on Ricart-Agrawala by giving a different interpretation to the *OK* message. When some process p_i sends an *OK* to some other process p_j, not only does it approve p_j's current request, but it also gives p_j p_i's permission to re-enter C again and again until p_j sends an *OK* to p_i in response to a *try* from p_i.

This algorithm performs well under light load. When a single process is requesting again and again, with no other process interested, it can go critical with no messages sent! Under heavy load, however, it basically reduces to Ricart and Agrawala's algorithm.

This algorithm is closely related to (i.e., is generalized by) the Chandy-Misra Dining Philosophers algorithm.

22.2 General Resource Allocation

We now turn to consider more general resource allocation problems in asynchronous networks, as defined earlier.

22.2.1 Burns-Lynch Algorithm

This algorithm solves *conjunctive* resource problems. Recall the strategies studied in asynchronous shared memory setting, involving seeking forks in order, and waiting for resources one at a time. We can run the same strategies in asynchronous networks. The best way to do this is *not* via a general simulation of the shared memory model in the network model: recall that the algorithms involve busy-waiting on shared variables; if simulated, each busy-waiting step would involve sending messages. Instead, we have each resource represented by its own process (which must be located at some node). The user process sends a message to the resource process for the first resource it needs; the resource process puts the user index on its queue; the user process then waits. When the user index reaches the front of the queue, the resource process sends a message back to that user process, which then goes on to wait for its next resource, etc. When a process gets all its resources, it tells its external user to go critical; when exit occurs, the user process sends messages to all the resource processes to remove its index from the queue.

The analysis is similar to that done in the shared memory case, and depends only on local parameters. The general case has exponential dependence on the number of colors. There are some variations in the literature, to improve the running time. This is still a topic of current research: see, e.g., Styer-Peterson, Awerbuch-Saks, Choy-Singh.

22.2.2 Drinking Philosophers

This is a dynamic variant of the general (conjunctive) static resource allocation problem. In this setting we have the same kind of graph as before, where a process is connected to everyone with whom it *might* have a resource conflict. But now, when a try request arrives from the outside, it contains an extra parameter: a set B of resources, indicating which resources the process actually needs this time. This set must be a subset of the static set for that process — the twist is that now it needn't be the whole set.

As before, we want to get exclusion, but this time based on the *actual* resources being requested. We would like to have also deadlock-freedom and lockout-freedom. The concurrency condition is now modified as follows. Suppose that a request is invoked at node i, and during the interval of this request, there are no (actual) conflicting requests. Suppose the execution proceeds so that i and all its neighbors in the underlying graph continue to take steps fairly, and the connecting links continue to operate fairly (note that some processes might remain in C). Then eventually i reaches C.

Chandy and Misra gave a solution to this, which was based on a particular (inefficient) Dining Philosophers algorithm. Welch and Lynch noticed that the Chandy-Misra algorithm made almost-modular use of the underlying Dining Philosophers algorithm, and so they redid their result, with the implicit modularity made explicit. Then it was possible to substitute a more efficient Dining Philosophers algorithm for the original Chandy-Misra subroutine.

The proposed architecture is sketched in Figure 22.2. The Dining Philosophers subroutine executes using its own messages, as usual. In addition, there are new messages for the Drinking Philosophers algorithm.

We assume for simplicity here that each resource is shared between only two processes. The heart of the algorithm is the D_i automata. First we describe the algorithm informally, and then present the D_i automaton code.

When D_i enters its trying region needing a certain set of resources, it sends requests for those resources that it needs but lacks. A recipient D_j of a request satisfies the request unless D_j currently also wants the resource or is already using it. In these two cases, D_j *defers* the request so that it can satisfy it when it is finished using the resource.

In order to prevent drinkers from deadlocking, a Dining Philosophers algorithm is used as a subroutine. The "resources" manipulated by the Dining Philosophers subroutine are

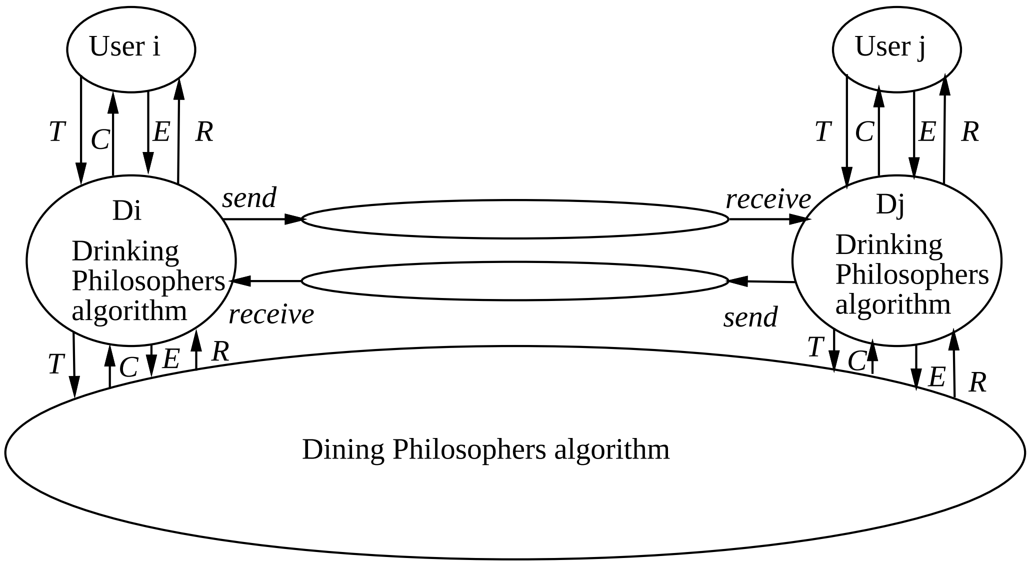

User i
User j
T
C
E
R
Di
Drinking Philosophers algorithm
Dj
Drinking Philosophers algorithm
send
receive
receive
send
T
C
E
R
Dining Philosophers algorithm

Figure 22.2: Architecture for the Drinking Philosophers problem.

priorities for the "real resources" (there is one dining resource for each drinking resource). As soon as D_i is able to do so in its drinking trying region, it enters its dining trying region, that is, it tries to gain priority for its maximum set of resources. If D_i ever enters its dining critical region while still in its drinking trying region, it sends *demands* for needed resources that are still missing. A recipient D_j of a demand must satisfy it even if D_j wants the resource, unless D_j is actually using the resource. In that case, D_j defers the demand and satisfies it when D_j is through using the resource.

Once D_i is in its dining critical region, we can show that it eventually receives all its needed resources and never gives them up. Then it may enter its drinking critical region. Once D_i enters its drinking critical region, it may relinquish its dining critical region, since the benefits of having the priorities are no longer needed. Doing so allows some extra concurrency: even if D_i stays in its drinking critical region forever, other drinkers can continue to make progress.

A couple of points about the code deserve explanation. We can show that when a request is received, the resource is always at the recipient; thus it is not necessary for the recipient to check that it has the resource before satisfying or deferring the request. On the other hand, it is possible for a demand for a missing resource to be received, so before satisfying or deferring a demand, the recipient must check that it has the resource.

Another point concerns the questions when the actions of the dining subroutine should be performed. Note that some drinkers could be locked out if D_i never relinquishes the dining critical region. The reason for this is that as long as D_i is in its critical region, it has priority for the resources. Thus, if we are not careful, D_i could cycle through its drinking critical region infinitely many times, while other drinkers wait. To avoid this situation, we keep track (using variable *current*) of whether the dining critical region was entered on behalf of the current drinking trying region (i.e., whether the latest C_i occurred after the latest $T_i(B)$). If it was, then D_i may enter its drinking critical region (assuming it has all the needed resources). Otherwise, $D(i)$ must wait until the current dining critical region has been relinquished before continuing.

The full code is given in Figures 22.3,22.4 and 22.5.

Correctness proof. Exclusion is straightforward, based on possession of the resources (since we use explicit tokens). Concurrency is also straightforward. To prove lockout-freedom we need to rely on the Dining-Philosophers subroutine. The proof is a bit subtle. First, we argue that the environment of the dining philosophers algorithm preserves dining-well-formedness (this is proved by a straightforward explicit check of the code). Therefore, every execution of the system is dining-well-formed (since the dining subroutine also preserves dining-well-formedness). Therefore the dining subroutine provides its guarantees: dining exclusion, and lockout-freedom for the dining subroutine in the context of the drinking algorithm.

Next, we show the following lemma.

Lemma 1 *In any fair drinking-well-formed execution, if drinking users always return the resources, then dining users always return the resources.*

Proof Sketch: Suppose that the dining resources are granted to D_i. Then, D_i sends demands. Consider any recipient D_j of a demand. If D_j has the resource and is not actually using it, then it gives it to i, because by the exclusion property of the dining subroutine, D_j cannot also be in its dining critical region. On the other hand, if D_j is using the resource, then by the assumption that no drinker is stuck in its critical region, D_j eventually finishes and satisfies the demand. Thus, eventually, D_i gets all the needed resources, and enters its drinking critical region, after which (by the code) it returns the dining resources. ∎

With this we can prove the following property.

Lemma 2 *In any fair drinking-well-formed execution, if drinking users always return the resources, then every drinking request is granted.*

Proof Sketch: Once a drinking request occurs, say at node i, then (unless the grant occurs immediately), a subsequent dining request occurs at i. Then by dining lockout-freedom and Lemma 1 eventually the dining subroutine grants at i. Again by Lemma 1, eventually i returns the dining resource. But note that i only returns the dining resource if in the interim it has done a drinking grant. The formal proof is by a contradiction, as usual. ∎

State:

- *drink-region*: equals T if the most recent drinking action was $T_i(B)$ (for some B), C if $C_i(B)$, etc. Initially R.
- *dine-region*: equals T if the most recent dining action was T_i, C if C_i, etc. Initially R.
- *need*: equals B, where the most recent drinking action had parameter B. Initially empty.
- *bottles*: set of tokens that have been received and not yet enqueued to be sent. Initially the token for a resource shared between D_i and D_j is in the *bottles* variable of exactly one of D_i and D_j, with the choice being made arbitrarily.
- *deferred*: set of tokens in the set *bottles* for which a request or demand has been received since the token was received. Initially empty.
- *current*: Boolean indicating whether current dining critical region is on behalf of current drinking trying region. Initially false.
- *msgs*$[j]$ for all $j \neq i$: FIFO queue of messages for D_j enqueued but not yet sent. Initially empty.

Actions for process i:

```
$try_i(B)$ for all $B \subseteq B_i$
    Effect: $drink\text{-}region \leftarrow T$
            $need \leftarrow B$
            for all $j \neq i$ and $b \in (need \cap B_j) - bottles$
                enqueue $request(b)$ in $msgs[j]$

$send_i(m, j)$ for all $j \neq i$, $m \in \{request(b), token(b), demand(b) : b \in B_i \cap B_j\}$
    Precondition: $m$ is at head of $msgs[j]$
    Effect: dequeue $m$ from $msgs[j]$

$try_i$
    Precondition: $dine\text{-}region = R$
                  $drink\text{-}region = T$
    Effect: $dine\text{-}region \leftarrow T$
```

Figure 22.3: Code for the modular drinking philosophers algorithm—part I.

Actions for process i (cont.):

```
$Receive_i(request(b), j)$ for all $j \neq i$, $b \in B_i \cap B_j$
  Effect: if $(b \in need)$ and $(drink\text{-}region \in \{T, C\})$ then
              $deferred \leftarrow deferred \cup \{b\}$
          else
              enqueue $token(b)$ in $msgs[j]$
              $bottles \leftarrow bottles - \{b\}$

$crit_i$
  Effect: $dine\text{-}region \leftarrow C$
          if $drink\text{-}region = T$ then
              for all $j \neq i$ and $b \in (need \cap B_j) - bottles$: enqueue $demand(b))$ in $msgs[j]$
              $current \leftarrow$ true

$Receive_i(demand(b), j)$ for all $j \neq i$, $b \in B_i \cap B_j$
  Effect: if $(b \in bottles)$ and $((b \notin need)$ or $(drink\text{-}region \neq C))$ then
              enqueue $token(b)$ in $msgs[j]$
              $bottles \leftarrow bottles - \{b\}$
              $deferred \leftarrow deferred - \{b\}$

$Receive_i(token(b), j)$ for all $j \neq i$, $b \in B_i \cap B_j$
  Effect: $bottles \leftarrow bottles \cup \{b\}$

$crit_i(B)$ for all $B \subseteq B_i$
  Precondition: $drink\text{-}region = T$
          $B = need$
          $need \subseteq bottles$
          if $dine\text{-}region = C$ then $current =$ true
  Effect: $drink\text{-}region \leftarrow C$
          $current \leftarrow$ false

$exit_i$
  Precondition: $dine\text{-}region = C$
          if $drink\text{-}region = T$ then $current =$ false

  Effect: $dine\text{-}region \leftarrow E$
```

Figure 22.4: Code for the modular drinking philosophers algorithm—part II.

Actions for process i (cont.):

```
$rem_i$
  Effect: $dine\text{-}region \leftarrow R$
$exit_i(B)$ for all $B \subseteq B_i$
  Effect: $drink\text{-}region \leftarrow E$
          for all $j \neq i$ and $b \in deferred \cap B_j$:
              enqueue $token(b)$ in $msgs[j]$
          $bottles \leftarrow bottles - deferred$
          $deferred \leftarrow \emptyset$

$rem_i(B)$ for all $B \subseteq B_i$
  Precondition: $drink\text{-}region = E$
          $B = need$
  Effect: $drink\text{-}region \leftarrow R$
```

Figure 22.5: Code for the modular drinking philosophers algorithm—part III.

We remark that the complexity bounds depend closely on the costs of the underlying dining algorithm.

22.3 Stable Property Detection

In this section we consider a new problem. Suppose we have some system of I/O automata as usual, and suppose we want to superimpose on this system another algorithm that "monitors" the given algorithm somehow. For instance, the monitoring algorithm can

- detect when the given system has terminated execution,
- check for violation of some global invariants,
- detect some kind of deadlock where processes are waiting for each other (in a cycle) to do something,
- compute some global quantity (e.g., the total amount of money).

We shall see some powerful techniques to solve such problems.

22.3.1 Termination for Diffusing Computations

First, we consider the termination problem alone. Dijkstra-Scholten gave an efficient algorithm, for the special case where the computation originates at one node. In this kind of computation, called *diffusing computation*, all nodes are originally *quiescent* (defined here to mean that they are not enabled to do any locally controlled actions). The algorithm starts with an input from the outside world at one of the nodes, and we assume that no further inputs occur. According to the IOA definitions, once a node receives such a message, it can change state in such a way that it now is enabled to do some locally-controlled actions, e.g., to send messages to other nodes. (The nodes can also do various outputs to the outside world.) When other nodes receive these messages, they in turn might also become enabled to do some locally-controlled actions, and so on.

The *termination detection problem* is formulated as follows.

> If the entire system ever reaches a *quiescent* state (i.e., all the processes are in quiescent states and there are no messages in transit), then eventually the originator node outputs a special message *done*.

Of course, the original algorithm does not have such an output. The new system constructed has to be an augmentation of the given system, where each node automaton is obtained from the old automaton in some allowed way.

Below we state informally what is allowed to solve this problem. The precise definition should be similar to the *superposition* definition described in Chandy and Misra's Unity work.

- We may add new state components, but the projection of the start states of the augmented machine must be exactly the start states of the original machine.
- The action set can be augmented with new inputs, outputs and internal actions. The old actions must remain with the same preconditions, but they may have new effects on the new components; they might also have additional information piggybacked on them, e.g., $send(m)$ now does $send(m, c)$. The old actions remain in the same classes.
- The new input actions affect the new components only.
- The new output and internal actions can be preconditioned on the entire state (old and new components), but they may only affect the new components. They get grouped into new fairness classes.

The idea in the Dijkstra-Scholten termination detection algorithm is to superimpose a kind of spanning-tree construction on the existing work of the underlying algorithm. Starting from the initial node, every message of the underlying algorithm gets a *tree* message piggybacked upon it. Just as in the asynchronous algorithm for constructing a spanning tree, each node records the first receipt of a *tree* message in its state, as its parent. Any subsequent receipts of *tree* messages are immediately acknowledged (they are not discarded as in the spanning tree protocol: explicit acknowledgment is sent back). Only the *first* one gets held and unacknowledged (for now). Also, if the initial node ever receives a *tree* message, it immediately acknowledges it. So as the messages circulate around the network, a spanning tree of the nodes involved in the protocol gets set up.

Now, we are going to let this tree converge back upon itself (as in convergecast), in order to report termination back to the initial node. Specifically, each node looks for when *both* of the following hold at the same time: (a) its underlying algorithm is in a quiescent state, and (b) all its outgoing *tree* messages have been acknowledged. When it finds this, it "closes up the shop" — it sends an acknowledgment to its parent, and forgets everything about this protocol. Note that this is different from what had to happen for ordinary broadcast-convergecast. There, we had to allow the nodes to remember that they had participated in the algorithm, marking themselves as "done", so they didn't participate another time. (If we hadn't, the nodes would just continue broadcasting to each other repeatedly.) This time, however, we *do* allow the nodes to participate any number of times; whether or not they do so is controlled by where the messages of the underlying algorithm get sent.

Example. Suppose the nodes are $0, 1, 2, 3$, and consider the following scenario (see Figure 22.6 for illustration). (i) Node 0 wakes up, sends a message to 1 and sends a message to 2. (ii) Nodes 1 and 2 receive the messages, and set their parent pointers to point to node 0. Then they both wake up and send messages to each other, but since each node already has a parent, it will just send an ack. Then both nodes send messages to 3. (iii) Suppose 1's message gets to node 3 first; then 1 is set to be the parent of 3, and therefore 3 acknowledges 2 immediately. The four nodes continue for a while sending messages to each other; they immediately acknowledge each message.

Now suppose the basic algorithm at 1 quiesces. Still, 1 does not close up because it has an unacknowledged message to 3. (iv) Suppose 3 now quiesces; it then is ready to acknowledge to 1, and it forgets it ever participated in the algorithm (actually, it forgets everything). When 1 receives this ack, and since 1 is also done, it sends ack to 0 and forgets everything. (v) Now suppose 2 sends out messages to 1 and 3. When they receive these messages, they wake up as at the beginning (continue carrying out the basic algorithm) and reset their parent pointers, this time to 2.

This execution can continue in this fashion indefinitely. But if all the nodes ever quiesce, and there are no messages of the basic algorithm in transit, then the nodes will all succeed in converging back to the initial node, in this case 0. When 0 is quiesced and has

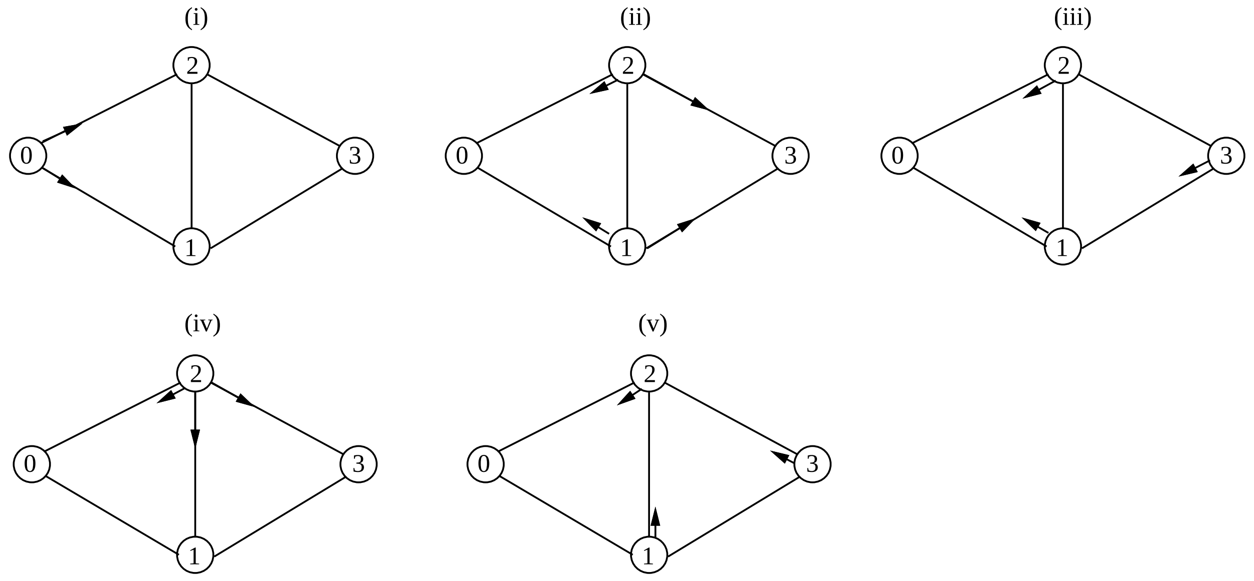


Figure 22.6: Example of an execution of the termination detection algorithm. The arrows on the edges indicate messages in transit, and the arrows parallel to the edges indicate parent pointers.

acknowledgments for all its outgoing messages, it can announce termination.

Suppose the underlying algorithm is A_i. In Figures 22.7 and 22.8 we give code for the superimposed version p_i.

We have argued roughly above that if the basic algorithm terminates, this algorithm eventually announces it. We can also argue that this algorithm never does announce termination unless the underlying system has actually terminated. We can prove this using the following invariant (cf. homework).

Invariant 9 *All non-idle nodes form a tree rooted at the node with status "leader", with the tree edges given by "parent" pointers.*

Francez-Shavit have generalized this algorithm to the case where the computation can initiate at several locations (the idea there is to use several trees, and wait for them all to terminate).

Complexity. The number of messages is proportional to the number of messages of underlying algorithm. A nice property of this algorithm is its "locality" — if the diffusing computation only operates for a short time in a small area of the network, the termination protocol only incurs proportionately small costs. On the other hand, if the algorithm works for a very long time before terminating, then the termination detection algorithm requires considerable overhead.

New state components:

- $status \in \{idle, leader, non\text{-}leader\}$, initially $idle$
- $parent$, a node or nil, initially nil
- for each neighbors j,
  $send(j)$, a queue of messages, initially empty
  $deficit(j)$, an integer, initially 0

Actions:

```
$\pi$, any input action of node $i$ excluding a $receive$)
  New Effect:
        $status \leftarrow leader$

$send_{i,j}(m)$
  New Effect: $deficit(j) \leftarrow deficit(j) + 1$

New action $receive_{j,i}(ack)$
  New Effect: $deficit(j) \leftarrow deficit(j) - 1$

$receive_{j,i}(m)$, where $m$ a message of $A_i$
  New Effect: if $status = idle$ then
          $status \leftarrow non\text{-}leader$
          $parent \leftarrow j$
        else add $ack$ to $send(j)$

New action $close\text{-}up\text{-}shop(i)$
  Precondition: $status = non\text{-}leader$
        $j = parent(i)$
        $send(k) = \emptyset$ for all $k$
        $deficit(k) = 0$ for all $k$
        state of $A_i$ is quiescent
  Effect: add $ack$ to $send_j$
        $status \leftarrow idle$
        $parent \leftarrow nil$
```

Figure 22.7: Code for Dijkstra-Scholten termination detection algorithm — part I.

Actions (cont.):

```
New action $send_{i,j}(ack)$
  Precondition: $ack$ first on $send(j)$
  Effect: remove first message from $send(j)$

New action $done$
  Precondition: $status = leader$
        $send(k) = \emptyset$ for all $k$
        $deficit(k) = 0$ for all $k$
        state of $A_i$ is quiescent
  Effect: $status \leftarrow idle$
```

Figure 22.8: Code for Dijkstra-Scholten termination detection algorithm — part II.

22.3.2 Snapshots

Problem Statement

The snapshot problem is to obtain a *consistent global state* of a running distributed algorithm. By "global state" we mean a state for each process and each FIFO channel. The "consistency", roughly speaking, requires that the output is a global state that "could have" occurred at a fixed moment in time.

We make the definition of the problem more precise using the notion of *logical time* discussed in the last lecture. Recall the definition of logical time: it is possible to assign a unique time to each event, in a way that is increasing at any given node, and such that *send*s precede *receive*s, and only finitely many events precede any particular event. Clearly, a given execution can get many logical time assignments. A consistent global state is one that arises from any particular logical time assignment and a particular real time t. For this assignment and this t, there is a well-defined notion of what has happened up through (and including) time t. We want to include exactly this information in the snapshot: states of nodes after the local events up through time t, and the states of channels at time t (i.e., those messages sent by time t and not received by time t, in the order of sending). See Figure 22.9 for an example.

We can stretch and shrink the execution to align corresponding times at different nodes, and the information in snapshot is just the states of nodes and channels at the t-cut.

Because we might have to shrink and stretch the execution, it is not necessarily true that the state which is found actually occurs at any point in the execution. But, in a sense, this

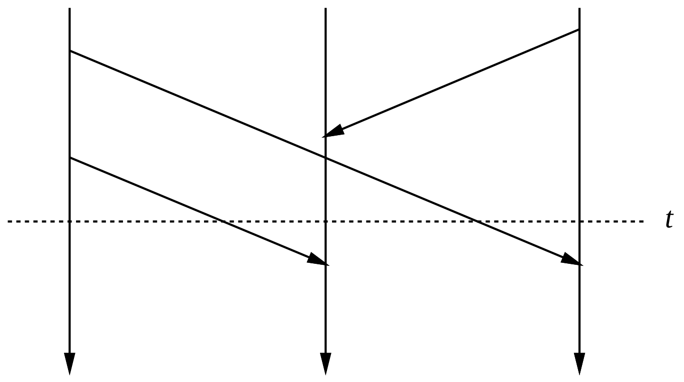


Figure 22.9: t is a logical time cut.

doesn't matter: it "could have" occurred, i.e., no process can tell locally that it didn't occur.

Actually, we would like more than just any consistent global state — we would like one that is fairly recent (e.g., we want to rule out the trivial solution that always returns the initial global state). For instance, one might want a state that reflects all the events that have actually occurred (in real time) before the invocation of the snapshot algorithm. So we suppose that the snapshot is initiated by the arrival of a *snap* input from the outside world at some nonzero number of the nodes (but only once at each node). (We do not impose the restriction that the snapshot must originate at a single node as in the diffusing computation problem. Also, the underlying computation is not necessarily diffusing.)

Applications

A simple example is a banking system, where the goal is to count all the money exactly once. More generally, we have the problem of *stable property* detection. Formally, we say that a property of global states is *stable* if it persists, i.e., if it holds in some state, then it holds in any subsequent state. For instance, termination, the property of all the nodes being in quiescent states and the channels being empty, is a stable property. Also, (unbreakable) deadlock, where a cycle of nodes are in states where they are waiting for each other, is a stable property.

The Algorithm

The architecture is similar to the one used above for the termination-detection algorithm – the underlying A_i is augmented to give p_i with certain additions that do not affect the underlying automaton's behavior.

We have already sketched one such algorithm, based on an explicit logical time assignment. We now give another solution, by Chandy and Lamport. The algorithm is very much like the snapshot based on logical time, only it does away with the explicit logical time.

The idea is fairly simple. Each node is responsible for snapping its own state, plus the states of the incoming channels. When a node first gets a *snap* input, it takes a snapshot of the state of the underlying algorithm. Then it immediately puts a *marker* message in each of its outgoing channels, and starts to record incoming messages in order to snap each of its incoming channels. This *marker* indicates the dividing line between messages that were sent out before the local snap and messages sent out after it. For example, if this is a banking system, and the messages are money, the money before the marker was *not* included in the local snap, but the money after the marker was.

A node that has already snapped just records these incoming messages, until it encounters the marker, in which case it stops recording. Thus, the node has recorded all the messages sent by the sender before the sender did its local snap. Returning to the banking system example, in this case the recipient node has counted all the money that was sent out before the sender did its snap and thus not counted by the sender.

There is one remaining situation to consider: suppose that a node receives a *marker* message before it has done its own local snap (because it has not received a *snap* input). Then immediately upon receiving the first *marker*, the recipient snaps its local state, sends out its markers, and begins recording the incoming channels. The channel upon which it has just received the marker is recorded as empty, however.

Modeling. It is not completely straightforward to model a snapshot system. The subtlety is in the atomicity of the snapping and marker sending steps. To do this much atomically, we seem to need to modify the underlying algorithm in slightly more complicated ways than for the Dijkstra-Scholten algorithm – possibly by blocking some messages from being sent by A_i while the markers are being put into the channels. This really means interfering with the underlying algorithm, but, roughly speaking, we do not interfere by "too much".

The code is given in Figure 22.10.

Correctness. We need to argue that the algorithm terminates, and that when it does, it gives a consistent global state (corresponding to some fixed logical time).

Termination. We need strong connectivity to prove termination. As soon as the *snap* input occurs, the recipient node does a local snapshot, and sends out markers. Every time anyone gets a marker, it snaps its state if it hasn't already. The markers thus eventually propagate everywhere, and everyone does a local snapshot. Also, eventually the nodes will finish collecting the messages on all channels (when they receive markers).

Consistency. We omit details. The idea is to assign a logical time to each event of the *underlying* system, by means of such an assignment for the complete system. We do this in

New state components:

- *status* $\in \{start, snapping, reported\}$, initially *start*
- *snap-state*, a state of A_i or *nil*, initially *nil*
- for each neighbor j: *channel-snapped*(j), a Boolean, initially *false*, and *send*(j), *snap-channel*(j), FIFO queues of messages, initially empty.

Actions:

```
New action $snap$
  Effect: if $status = start$ then
        $snap\text{-}state$ gets the state of $A_i$
        $status \leftarrow snapping$
        for all $j \in neighbors$, add $marker$ to $send(j)$

Replace $send_{i,j}(m)$ actions of $A_i$ with $internal\text{-}send$ actions, which put $m$ at the end of $send(j)$.

New action $send_{i,j}(m)$
  Precondition: $m$ is first on $send(j)$
  Effect: remove first element of $send(j)$

$receive_{j,i}(m)$, $m$ a message of $A_i$
  New Effect: if $status = snapping$ and $channel - snapped(j) = false$ then
        add $m$ to $snap\text{-}channel(j)$

New action $receive_{j,i}(marker)$
  Effect: if $status = start$ then
        $snap\text{-}state$ gets the state of $A_i$
        $status \leftarrow snapping$
        for all $j \in neighbors$, add $marker$ to $send(j)$
        $channel\text{-}snapped(j) \leftarrow true$

New action $report\text{-}snapshot(s, C)$, where $s$ is a node state, and $C$ is the states of incoming links
  Precondition: $status = snapping$
     for all $j \in neighbors$, $channel\text{-}snapped(j) = true$
     $s = snap\text{-}state$
     for all $j \in neighbors$, $C(j) = snap\text{-}channel(j)$
  Effect: $status \leftarrow reported$
```

Figure 22.10: The snapshot algorithm, code for process i.

such a way that the same time t (with ties broken by indices) is assigned to each event at which a node snaps its local state. The fact that we can do this depends on the fact that there is no message sent at one node after the local snap and arriving at another node before its local snap. (If a message is sent after a local snap, it follows the marker on the channel; then when the message arrives, the marker will have already arrived, and so the recipient will have already done its snapshot.) It is obvious that what is returned by each node is the state of the underlying algorithm up to time t. We must also check that the channel recordings give exactly the messages "in transit at logical time t", i.e., the messages sent after the sender's snap and received before the receiver's snap.

Some simple examples. We again use the bank setting. Consider a two-dollar, two-node banking system, where initially each node, i and j, has one dollar. The underlying algorithm has some program (it doesn't matter what it is) that determines when it sends out some money. Consider the following execution (see Figure 22.11).

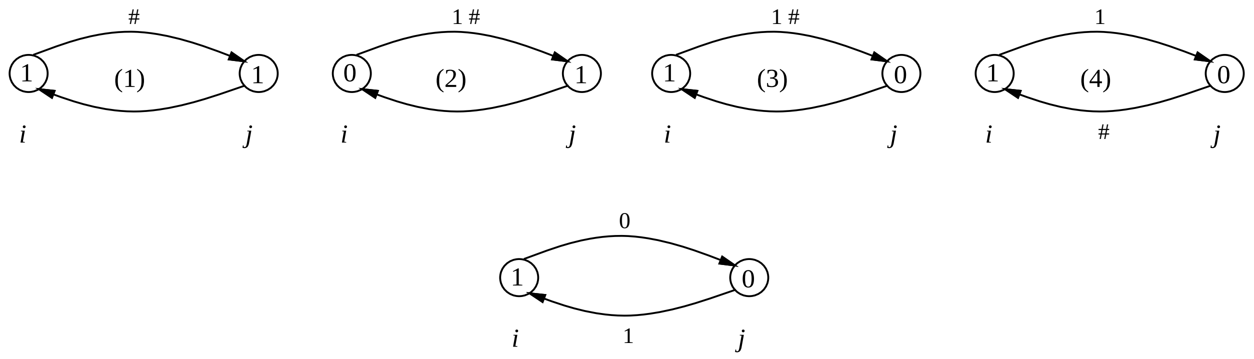


Figure 22.11: Execution of a two-node bank system. The final snapshot is depicted below the intermediate states.

1. $snap_i$ occurs, i records the state of the bank ($1), puts a *marker* in the buffer to send to j, and starts recording incoming messages

2. i puts its dollar in the buffer to send to j, behind the marker,

3. j sends its dollar to i, i receives it and records it in its location for recording incoming messages.

4. j receives the marker from i, records its local bank state ($0), puts a *marker* in the buffer to send to i, and snaps the state of the incoming channel as empty.

5. i receives the marker, snaps the state of the channel as the sequence consisting of one message, the dollar it received before the marker.

Put together, the global snapshot obtained is one dollar at i, one dollar in channel from j to i, and the other node and channel empty. This looks reasonable; in particular, the correct total ($2) is obtained. But note that this global state never actually arose during the computation. However, we can show a "reachability" relationship: Suppose that α is the actual execution of the underlying bank algorithm. Then α can be partitioned into $\alpha_1\alpha_2\alpha_3$, where α_1 indicates the part before the snapshot starts and α_3 indicates the part after the snapshot ends. Then the snapped state is reachable from the state at the end of α_1, and the state at the beginning of α_3 is reachable from the snapped state. So the state "could have happened", as far as anyone before or after the snapshot could tell.

In terms of Lamport's partial order, consider the actual execution of the underlying system depicted in Figure 22.12 (a). We can reorder it, while preserving the partial order, so that all the states recorded in the snapshot are aligned, as in Figure 22.12 (b).

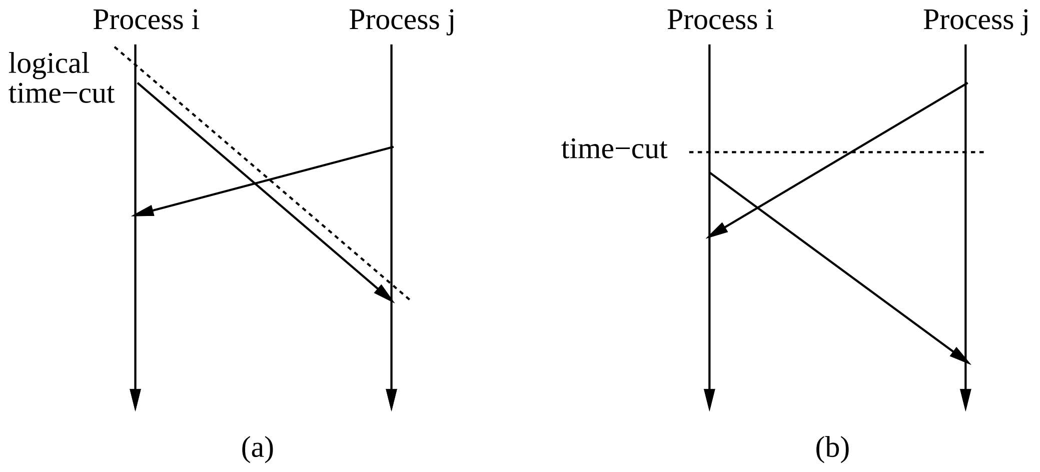


Figure 22.12: (a) is the actual execution of Figure 22.11, and (b) is a possible orderings of it, where the logical time cut of (a) is a real time cut.

Applications revisited

Stable property detection.

A stable property is one that, if it is true of any state s of the underlying system, is also true in all states reachable from s. Thus, if it ever becomes true, it stays true. The following is a simple algorithm to detect the occurrence of a stable property P. First, the algorithm does a global snapshot, and then either it collects the information somewhere and returns $P(result)$, or else it does a distributed algorithm on the *static* information recorded from the snapshot, to determine $P(result)$. The correctness conditions of the snapshot (the reachability version of those conditions, that is) imply the following. If the output is *true*, then P is true in the real state of the underlying algorithm at the end of the snapshot, and if the output is *false*, then P is false in the real state of the underlying algorithm at the beginning of the snapshot. (There is some uncertainty about the interim.)

Example: Termination detection. Assume a system of basic processes with no external inputs, but the processes don't necessarily begin in quiescent states. This will execute on its own for a while, and then might quiesce (terminate). More precisely, quiescence refers to all nodes being quiescent and no messages in transit anywhere. We want to be able to detect this situation, since it means that the system will perform no further action (since there are no inputs). Quiescence of global states is a stable property (assuming no inputs). So here is a simple algorithm to detect it: do a snapshot, send the states somewhere and check if anything is enabled or in transit. Actually, we don't need to send the whole state to one process: we can have everyone check termination locally, and fan in (i.e., convergecast) the results to some leader along a spanning tree. (We only need to convergecast a bit saying if the computation is done in the subtree.) It may be interesting to contrast this algorithm to the Dijkstra-Scholten termination detection: the snapshot algorithm involves the whole network, i.e., it is not local at all. But on the other hand, if it only has to be executed once, then the overhead is bounded regardless of how long the underlying algorithm has gone on for.

Example: Deadlock detection. Now the basic algorithm involves processes that are "waiting for" other processes, e.g., it can be determined from the state of process A_i that it is waiting for process A_j (say to release a resource). The exact formulation varies in different situations, e.g., a process might be waiting for resources owned by the other processes, etc. In all formulations, however, deadlock essentially amounts to a waiting cycle. If we suppose that every process that is waiting is stopped and while it is stopped it will never "release resources" to allow anyone else to stop waiting, then deadlock is stable. Thus, we can detect it by taking snapshots, sending the information to one place, then doing an ordinary centralized cycle-detection algorithm. Alternatively, after taking the snapshot, we can perform some distributed algorithm to detect cycles, on the static data resulting from the snapshot.

Lecturer: George Varghese

Lecture 23

Designing network protocols is complicated by three issues: parallelism, asynchrony, and fault-tolerance. This course has already covered techniques for making protocols robust against Byzantine faults. The Byzantine fault model is attractive because it is general – the model allows a faulty process to *continue* to behave arbitrarily. In this lecture, we will study another general fault model – the *self-stabilization* model – which is attractive from both a theoretical and practical viewpoint.

We will compare the two models a little later but for now let's just say that self-stabilization allows an *arbitrary* number of faults that *stop* while Byzantine models allow a *limited* number of faults that *continue*. Stabilizing solutions are also typically much cheaper and have found their way into real networks. In this lecture we will describe self-stabilization using two network models – an elegant shared memory model and a more practical network model. We will also describe three general techniques for making protocols stabilizing – *local checking and correction, counter flushing,* and *timer flushing*.

23.1 The Concept of Self-Stabilization

23.1.1 Door Closing and Domain Restriction

Today we will focus on the ability of network protocols to stabilize to "correct behavior" after *arbitrary* initial perturbation. This property was called *self-stabilization* by Dijkstra [Dijkstra74]. The "self" emphasizes the ability of the system to stabilize by *itself* without manual intervention.

A story illustrates the basic idea. Imagine that you live in a house in Alaska in the middle of winter. You establish the following protocol (set of rules) for people who enter and leave your house. Anybody who leaves or enters the house must shut the door after them. If the door is initially shut, and nobody makes a mistake, then the door will eventually return to the closed position. Suppose, however, that the door is initially open or that somebody forgets to shut the door after they leave. Then the door will stay open until somebody passes through again. This can be a problem if heating bills are expensive and if several hours can go by before another person goes through the door. It is often a good idea to make the door

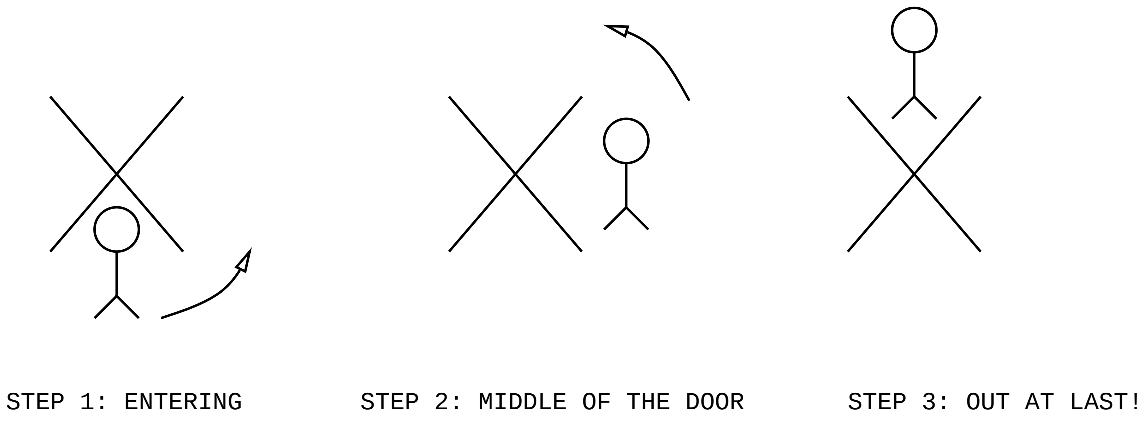


Figure 23.1: Exiting through a revolving door.

closing protocol self-stabilizing. This can be done by adding a spring (or automatic door closer) that constantly restores the door to the closed position.

In some cases, it is possible to trivialize the problem by hardwiring relationships between variables to avoid illegal states – such a technique is actually used in a revolving door! A person enters the revolving door Figure 23.1, gets into the middle of the door, and finally leaves. It is physically impossible to leave the door open and yet there is a way to exit through the door.

This technique, which we will call *domain restriction*, is often a simple but powerful method for removing illegal states in computer systems that contain a single shared memory. Consider two processes A and B that have access to a common memory as shown in the first part of Figure 23.2. Suppose we want to implement mutual exclusion by passing a token between A and B.

One way to implement this is to use two boolean variables $token_A$ and $token_B$. To pass the token, Process A sets $token_A$ to *false* and sets $token_B$ to *true*. In a self-stabilizing setting, however, this is not a good implementation. For instance, the system will deadlock if $token_A = token_B = false$ in the initial state. The problem is that we have some extra and useless states. The natural solution is to restrict the domain to a single bit called *turn*, such that $turn = 1$ when A has the token and $turn = 0$ when B has the token. By using domain restriction, [13] we ensure that *any possible state is also a legal state.*

It is often feasible to use domain restriction to avoid illegal states *within a single node* of a computer network. Domain restriction can be implemented in many ways. The most

[13]In this example, we are really *changing* the domain. However, we prefer the term domain restriction.

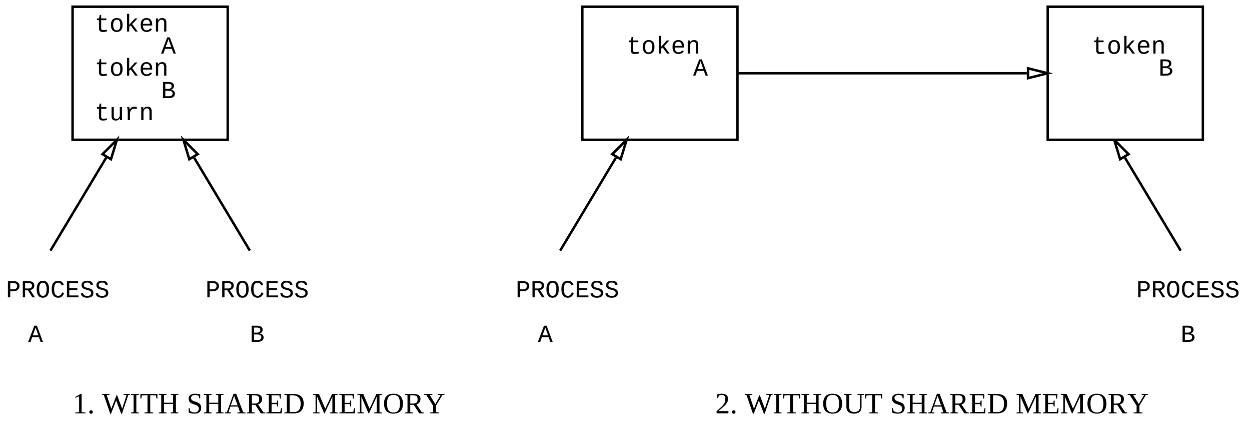


Figure 23.2: Token passing among two processes

natural way is by restricting the number of bits allocated to a set of variables so that every possible value assigned to the bits corresponds to a legal assignment of values to each of the variables in the set. Another possibility is to modify the code that reads variables so that only values within the specified domain are read.

Unfortunately, domain restriction cannot solve all problems. Consider the same two processes A and B that wish to achieve mutual exclusion. This time, however, (see Figure 23.2, Part 2) A and B are at two different nodes of a computer network. The only way they can communicate is by sending *token* messages to each other. Thus we cannot use a single *turn* variable that can be read by both processes. In fact, A must have at least two states: a state in which A has the token, and a state in which A does not have the token. B must also have two such states. Thus we need at least four combined states, of which two are illegal.

Thus domain restriction at each node *cannot prevent illegal combinations across nodes.* We need other techniques to detect and correct illegal states of a network. It should be no surprise that the title of Dijkstra's pioneering paper on self-stabilization [Dijkstra74] was "Self-Stabilization in spite of Distributed Control."

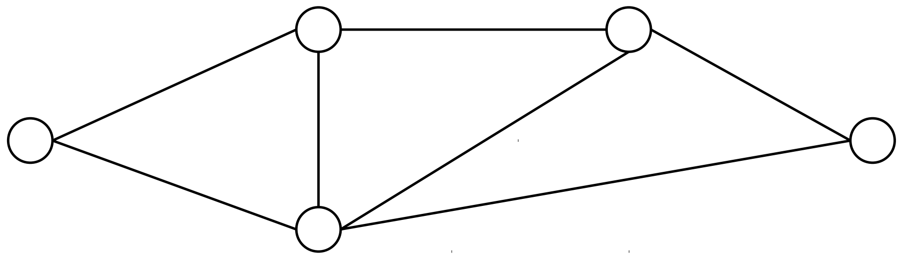


Figure 23.3: A typical mesh Network

23.1.2 Self-Stabilization is attractive for Networks

We will explore self-stabilization properties for computer networks and network protocols. A computer network consists of nodes that are interconnected by communication channels. The network topology (see Figure 23.3) is described by a graph. The vertices of the graph represent the nodes and the edges represent the channels. Nodes communicate with their neighbors by sending messages along channels. Many real networks such as the ARPANET, DECNET and SNA can be modeled in this way.

A network protocol consists of a program for each network node. Each program consists of code and inputs as well as local state. The global state of the network consists of the local state of each node as well as the messages on network links. We define a *catastrophic* fault as a fault that arbitrarily corrupts the global network state, but not the program code or the inputs from outside the network.

Self-stabilization formalizes the following intuitive goal for networks: *despite a history of catastrophic failures, once catastrophic failures stop, the system should stabilize to correct behavior without manual intervention.* Thus self-stabilization is an abstraction of a strong fault-tolerance property for networks. It is an important property of real networks because:

- **Catastrophic faults occur:** Most network protocols are resilient to common failures such as nodes and link crashes but not to *memory corruption.* But memory corruption does happen from time to time. It is also hard to prevent a malfunctioning device from sending out an incorrect message.

- **Manual intervention has a high cost:** In a large decentralized network, restoring the network manually after a failure requires considerable coordination and expense. Thus even if catastrophic faults occur rarely, (say once a year) there is considerable incentive to make network protocols self-stabilizing. A reasonable guideline is that *the network should stabilize preferably before the user notices and at least before the user logs a service call!*

These issues are illustrated by the crash of the original ARPANET protocol ([Rosen81] [Perlman83]). The protocol was carefully designed never to enter a state that contained three conflicting updates a, b, and c. Unfortunately, a malfunctioning node injected three such updates into the network and crashed. After this the network cycled continuously between the three updates. It took days of detective work [Rosen81] before the problem was diagnosed. With hindsight, the problem could have been avoided by making the protocol self-stabilizing.

Self-stabilization is also attractive because a self-stabilizing program does not require initialization. The concept of an initial state makes perfect sense for a single sequential program. However, for a distributed program an initial state seems to be an artificial concept. How was the distributed program placed in such an initial state? Did this require another distributed program? Self-stabilization avoids these questions by eliminating the need for distributed initialization.

23.1.3 Criticisms of Self-Stabilization

Despite the claims of the previous section, there are peculiar features of the self-stabilization model that need justification.

- The model allows network state to be corrupted but not program code. However, program code can be protected against arbitrary corruption of memory by redundancy since code is rarely modified. On the other hand, the state of a program is constantly being updated and it is not clear how one can prevent illegal operations on the memory by using checksums. It is even harder to prevent a malfunctioning node from sending out incorrect messages.

- The model only deals with catastrophic faults that *stop*. Byzantine models deal with continuous faults. However, in Byzantine models, only a fraction of nodes are allowed to exhibit arbitrary behavior. In the self-stabilization model, *all* nodes are permitted to start with arbitrary initial states. Thus, the two models are orthogonal and can even be combined.

- A self-stabilizing program P is is allowed to make initial mistakes. However, the important stabilizing protocols that we know of are used for routing, scheduling, and resource allocation tasks. For such tasks, initial errors only result in a temporary loss of service.

23.2 Definitions of Stabilization

23.2.1 Execution Definitions

All the existing definitions of stabilization are in terms of the states and executions of a system. We will begin with a definition of stabilization that corresponds to the standard definitions (for example, that of Katz and Perry [KP90]). Next, we will describe another definition of stabilization in terms of external behaviors. We believe that the definition of behavior stabilization is appropriate for large systems that require modular proofs. However, the definition of execution stabilization given below is essential in order to *prove* results about behavior stabilization.

Suppose we define the correctness of an automaton in terms of a set C of legal executions. For example, for a token passing system, we can define the legal executions to be those in which there is exactly one token in every state, and in which every process periodically receives a token.

What do we mean when we say that an automaton A stabilizes to the executions in set C. Intuitively, we mean that eventually all executions of A begin to "look like" an execution in set C. For example, suppose C is the set of legal executions of a token passing system. Then in the initial state of A there may be zero or more tokens. However, the definition requires that eventually there is some suffix of any execution of A in which there is exactly one token in any state.

Formally:

Definition 1 *Let C be a set of executions. We say that automaton A stabilizes to the executions in C if for every execution α of A there is some suffix of execution α that is in C.*

We can extend this definition in the natural way to define what it means for an automaton A to stabilize to the executions of another automaton B.

23.2.2 Definitions of Stabilization based on External Behavior

In the I/O Automaton model, the correctness of an automaton is specified in terms of its external behaviors. Thus we specify the correctness of a token passing system without any reference to the state of the system. We can do so by specifying the ways in which token delivery and token return actions (to and from some external users of the token system) can be interleaved

Thus it natural to look for a definition of stabilization in terms of external behaviors. We would also hope that such a definition would allow us to modularly "compose" results about the stabilization of parts of a system to yield stabilization results about the whole system.

Now an IOA A is said to *solve* a problem P if the behaviors of A are contained in P. For stabilization, however, it is reasonable to weaken this definition and ask only that an IOA *eventually* exhibit correct behavior. Formally:

Definition 2 *Let P be a problem (i.e, a set of behaviors). An IOA A stabilizes to the behaviors in P if for every behavior β of A there is a suffix of β that is in P.*

Similarly, we can specify that A stabilizes to the behaviors of some other automaton B in the same way. The behavior stabilization definition used in this section and [V92] is a special case of a definition suggested by Nancy Lynch.

Finally, we have a simple lemma that ties together the execution and behavior stabilization definitions. It states that execution stabilization implies behavior stabilization. In fact, the only method we know to *prove* a behavior stabilization result is to first prove a corresponding execution stabilization result, and then use this lemma. Thus the behavior and execution stabilization definitions complement each other in this thesis: the former is typically used for *specification* and the latter is often used for *proofs*.

Lemma 1 *If IOA A stabilizes to the executions of IOA B then IOA A stabilizes to the behaviors of B.*

23.2.3 Discussion on the Stabilization Definitions

First, notice that we have defined what it means for an arbitrary IOA to stabilize to some target set or automaton. Typically, we will be interested in proving stabilization properties only for a special kind of automata: unrestricted automata. An unrestricted IOA (UIOA) is one in which all states of the IOA are also start states. Such an IOA models a system that has been placed in an arbitrary initial state by an arbitrary initial fault.

A possible modification (which is sometimes needed) is to only require (in Definitions 2 and 1) that the suffix of a behavior (execution) be a *suffix* of a behavior (execution) of the target set. One problem with this modified definition is that we know of no good proof technique to prove that the behaviors (executions) of an automaton are *suffixes* of a specified set of behaviors (executions).By contrast, it is much easier to prove that every behavior of an automaton has a suffix that is in a specified set. Thus we prefer to use the simpler definitions for what follows.

23.3 Examples from Dijkstra's Shared Memory Model

In Dijkstra's [Dijkstra74] model, a network protocol is modeled using a graph of finite state machines. In a single move, a *single* node is allowed to read the state of its neighbors, compute, and then possibly change its state. In a real distributed system such atomic communication is impossible. Typically communication has to proceed through channels. While Dijkstra's original model is not very realistic, it is probably the simplest model of an asynchronous distributed system. This simple model provided an ideal vehicle for *introducing* [Dijkstra74] the concept of stabilization without undue complexity. For this section, we will use Dijkstra's original model to *introduce* two important and general techniques for self-stabilization: *local checking and correction* and *counter flushing*. In the next section, we will introduce a more realistic message passing model.

23.3.1 Dijkstra's Shared Memory Model and IOA

How can we map Dijkstra's model into the IOA model? Suppose each node in Dijkstra's model is a separate automaton. Then in the Input/Output automata model, it is not possible to model the simultaneous reading of the state of neighboring nodes. The solution we use is to dispense with modularity and model *the entire network as a single automaton.* All actions, such as reading the state of neighbors and computing, are *internal actions.* The asynchrony in the system, which Dijkstra modeled using a "demon", is naturally a part of the IOA model. Also, we will describe the correctness of Dijkstra's systems in terms of *executions* of the automaton.

Formally:

A *shared memory network automaton* $\mathcal{N}$ for graph $G = (E, V)$ is an automaton in which:

- The state of $\mathcal{N}$ is the cross-product of a set of node states, $S_u(\mathcal{N})$, one for each node $u \in V$. For any state s of $\mathcal{N}$, we use $s|u$ to denote s projected onto S_u. This is also read as the state of node u in global state s.

- All actions of $\mathcal{N}$ are internal actions and are partitioned into sets, $A_u(\mathcal{N})$, one for each node $u \in V$

- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Consider any state s' of $\mathcal{N}$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u. Then there is some transition $(s', \pi, \tilde{s}')$ of $\mathcal{N}$ such that $\tilde{s}'|v = \tilde{s}|v$ for u and all u's neighbors in G.

- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Then $s|v = \tilde{s}|v$ for all $v \neq u$.

Informally, the third condition requires that the transitions of a node $u \in V$ only depend on the state of node u and the states of of the neighbors of u in G. The fourth condition requires that the effect of a transition assigned to node $u \in V$ can only be to change the state of u.

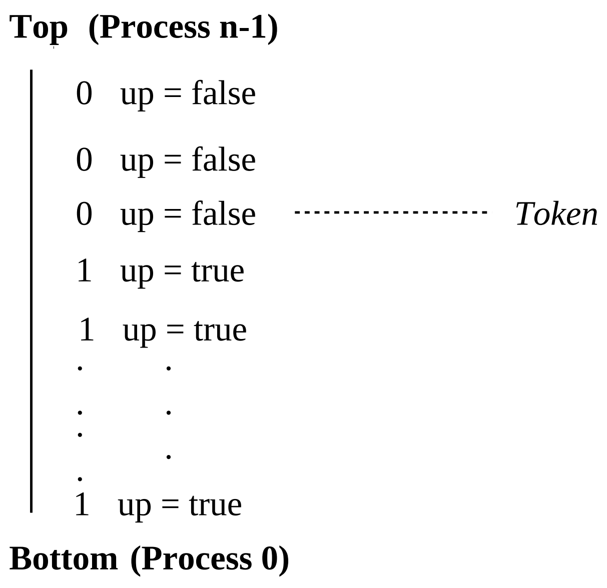


Figure 23.4: Dijktra's protocol for token passing on a line

23.3.2 Dijkstra's Second Example as Local Checking and Correction

In this section, we will begin by reconsidering the second example in [Dijkstra74]. The derivation here is based on some unpublished work I did with Anish Arora and Mohamed Gouda at the University of Texas. This protocol is essentially a token passing protocol on a line of nodes with process indices ranging from 0 to $n-1$. Imagine that the line is drawn vertically so that process 0 is at the bottom of the line (and hence is called "bottom") and Process $n-1$ is at the top of the line (and called "top"). This is shown in Figure 23.4. The down neighbor of Process i is Process $i-1$ and the up neighbor is Process $i+1$. Process $n-1$ and Process 0 are not connected.

Dijkstra observed that it is impossible (without randomization) to solve mutual exclusion in a stabilizing fashion if all processes have identical code. To break symmetry, he made the code for the "top" and "bottom" processes different from the code for the others.

Dijkstra's second example is modeled by the automaton $D2$ shown in Figure 23.5. Each process i has a boolean variable up_i, and a bit x_i. Roughly, up_i is a pointer at node i that points in the direction of the token, and x_i is a bit that is used to implement token passing. Figure 23.4 shows a state of this protocol when it is working correctly. First, there can be at most two consecutive nodes whose up pointers differ in value and the token is at one of these two nodes. If the two bits at the two nodes are different (as in the figure) then the token is at the upper node; else the token is at the lower node.

The state of the system consists of a boolean variable
up_i and a bit x_i, one for every process in the line.
We will assume that $up_0 = true$ and $up_{n-1} = false$ by definition

In the initial state $x_i = 0$ for $i = 0 \dots n-1$ and $up_i = false$ for $i = 1 \dots n-1$

```
MOVE_UP$_0$ (*action for the bottom process only to move token up*)
    Precondition: $x_0 = x_1$ and $up_1 = false$
    Effect: $x_0 := \sim x_0$

MOVE_DOWN$_{n-1}$ (*action for top process only to move token down*)
    Precondition: $x_{n-2} \neq x_{n-1}$
    Effects:
        $x_{n-1} := x_{n-2}$;

MOVE_UP$_i, 1 \leq i \leq n-2$ (*action for other processes to move token up*)
    Precondition: $x_i \neq x_{i-1}$
    Effects:
        $x_i := x_{i-1}$;
        $up_i := true$; (*point upwards in direction token was passed*)

MOVE_DOWN$_i, 1 \leq i \leq n-2$ (*action for other processes to move token down*)
    Precondition: $x_i = x_{i+1}$ and $up_i = true$ and $up_{i+1} = false$
    Effect: $up_i := false$; (*point downwards in direction token was passed*)
```

All actions are in a separate class.

Figure 23.5: Automaton $D2$: a version of Dijkstra's second example with initial states. The protocol does token passing on a line using nodes with at most 4 states.

For the present, assume that all processes start with $x_i = 0$. Also, initially assume that $up_i = false$ for all processes other than process 0. We will remove the need for such initialization below. We start by understanding the correct executions of this protocol when it has been correctly initialized.

A process i is said to have the token when any action at Process i is enabled. As usual the system is correct when there is at most one token in the system. Now, it is easy to see that in the initial state only MOVE_UP_0 is enabled. Once node 0 makes a move, then MOVE_UP_1 is enabled followed by MOVE_UP_2 and so on as the "token" travels up the line. Finally the token reaches node $n-1$, and we reach a state s in which $x_i = x_{i+1}$ for $i = 0 \ldots n-3$ and $x_{n-1} \neq x_{n-2}$. Also in state s, $up_i = true$ for $i = 0 \ldots n-2$ and $up_{n-1} = false$. Thus in state s, MOVE_DOWN_{n-1} is enabled and the token begins to move down the line by executing MOVE_DOWN_{n-2} followed by MOVE_DOWN_{n-3} and so on until we reach the initial state again. Then the cycle continues. Thus in correct executions, the "token" is passed up and down the line.

We describe these "good states" of $D2$ (that occur in correct executions) in terms of local predicates. In the shared memory model, a local predicate is any predicate that only refers to the state variables of a pair of neighbors. Thus in a good state of $D2$, two properties are true for any Process i other than 0:

- If $up_{i-1} = up_i$ then $x_{i-1} = x_i$.
- If $up_i = true$ then $up_{i-1} = true$.

First, we prove that if these two local predicates hold for all $i = 1 \ldots n-1$, then there is exactly one action enabled. Intuitively, since $up_{n-1} = false$ and $up_0 = true$, we can start with process $n-1$ and go down the line until we find a pair of nodes i and $i-1$ such that $up_i = false$ and $up_{i-1} = true$. Consider the first such pair. Then the second predicate guarantees us that there is exactly one such pair. The first predicate then guarantees that all nodes $j < i-1$ have $x_j = x_{i-1}$ and all nodes $k > i$ have $x_k = x_i$. Thus only one action is enabled. If $x_i = x_{i-1}$ and $i-1 \neq 0$ then only MOVE_DOWN_{i-1} is enabled. If $x_i = x_{i-1}$ and $i-1 = 0$ then only MOVE_UP_0 is enabled. If $x_i \neq x_{i-1}$ and $i \neq n-1$ then only MOVE_UP_i is enabled. If $x_i \neq x_{i-1}$ and $i = n-1$ then only MOVE_DOWN_{n-1} is enabled.

Similarly we can show that if exactly one action is enabled then all local predicates hold. Thus if the system is in a bad state, some pair of neighbors will be able to detect this fact. Hence we conclude that $D2$ is *locally checkable.* However, we would like to go even further and be able to correct the system to a good state by adding extra correction actions to each node. If we can add correction actions so that all link predicates will eventually become true, then we say that the $D2$ is also *locally correctable.* Adding local correction actions is tricky because the correction actions of nodes may "interfere" and result in a form of "thrashing".

However, a line is a special case of a tree with (say) 0 as the root; each node i other than 0, can consider $i-1$ to be its parent. The tree topology suggests a simple correction strategy. For each node $i \neq 0$, we can add a new action CORRECT_CHILD_i (which is a separate class for each i). Basically, CORRECT_CHILD_i checks whether the link predicate on the link between i and its parent is true. If not, i changes its state such that the predicate becomes true. Notice that CORRECT_CHILD_i leaves the state of i's parent unchanged. Suppose j is the parent of i and k is the parent of j. Then CORRECT_CHILD_i will leave the local predicate on link (j, k) true if it was true in the previous state.

Thus we have an important stability property: correcting a link does not affect the correctness of links above it in the tree. Using this it is easy to see that eventually all links will be in a good state and so the system is in a good state. We will ignore the details of this proof but the basic idea should be clear. Dijkstra's actual code does not use an explicit correction action. However, we feel that this way of understanding his example is clearer and illustrates a general method. The general method of local checking and correction was first introduced in [APV-91].

23.3.3 Dijkstra's first example as Counter Flushing

The counter flushing paradigm described in this section is taken from [V92].

Dijkstra's first example is modeled by the automaton $D2$ shown in Figure 23.6. As in the previous example, the nodes (once again numbered from 0 to $n-1$) are arranged such that node 1 has node 0 and node 2 as its neighbors and so on. However, in this case we also assume that Process 0 and $n-1$ are neighbors. In other words, by making 0 and $n-1$ adjacent we have made the line into a ring. For process i, let us call Process $i-1$ (we assume that all arithmetic on indices and counters is mod n) the counter-clockwise neighbor of i and $i+1$ the clockwise neighbor of i.

Each node has a counter $count_i$ in the range $0, \ldots n$ that is incremented mod $n+1$. Once again the easiest way to understand this protocol is to understand what happens when it is properly initialized. Thus assume that initially Process 0 has its counter set to 1 while all other processes have their counter set to 0. Processes other than 0 are only allowed to "move" (see Figure 23.6) when their counter differs in value from that of their counter-clockwise neighbor; in this case, the process is allowed to make a move by setting its counter to equal that of its counter-clockwise neighbor. Thus initially, only Process 1 can make a move after which Process 1 has its counter equal to 1; next, only Process 2 can move, after which Process 2 sets its counter equal to 1; and so on, until the value 1 moves clockwise

```
    The state of the system consists of an integer variable
    $count_i \in \{0, \dots n\}$, one for every process in the ring.
    We assume that Process 0 and $n-1$ are neighbors

    In the initial state $count_i = 0$ for $i = 1 \dots n-1$ and $count_1 = 1$

MOVE$_0$ (*action for Process 0 only *)
    Precondition: $count_0 = count_{n-1}$ (*equal to counter-clockwise neighbor?*)
    Effect: $count_0 := (count_0 + 1) \bmod (n+1)$ (*increment counter*)

MOVE$_i$, $1 \le i \le n-1$ (*action for other processes*)
    Precondition: $count_i \ne count_{i-1}$ (*not equal to counter-clockwise neighbor?*)
    Effects:
        $count_i := count_{i-1}$;(*set equal to counter-clockwise neighbor*)

All actions are in a separate class
```

Figure 23.6: Automaton $D1$: a version of Dijkstra's first example with initial states. The protocol does token passing on a ring using nodes with n states.

around the ring until all processes have their counter equal to 1.

Process 0 on the other hand cannot make a move until Process $n-1$ has the same counter value as Process 0. Thus until Process 1 sets its counter to 1, Process 0 cannot make a move. However, when this happens, Process 1 increments its counter mod $n+1$. Then the cycle repeats as now the value 2 begins to move across the ring (assuming $n > 2$) and so on. Thus after proper initialization, this system does perform a form of token passing on a ring; each node is again considered to have the token, when the system is in a state in which the node can take a move.

It is easy to see that the system is in a good state iff the following local predicates are true.

- For $i = 1 \dots n-1$, either $count_{i-1} = count_i$ or $count_{i-1} = count_i + 1$.
- Either $count_0 = count_{n-1}$ or $count_0 = count_{n-1} + 1$.

The system is locally checkable but it does not appear to be locally correctable. However, it does stabilize using a paradigm that we can call *counter flushing*. Even if the counter values are arbitrarily initialized (in the range $0, \dots, n$) the system will eventually begin executing as some suffix of a properly initialized execution. We will prove this informally using three claims:

- **In any execution, Process 0 will eventually increment its counter.** Suppose not. Then since Process 0 is the only process that can "produce" new counter values, the number of distinct counter values cannot increase. If there are two or more distinct counter values, then moves by Processes other 0 will reduce the number of distinct counter values to 1, after which Process 0 will increment its counter.

- **In any execution, Process 0 will eventually reach a "fresh" counter value that is not equal to the counter values of any other process.** To see this, note that that in the initial state there are at most n distinct counter values. Thus there is some counter value say m that is not present in the initial state. Since, process 0 keeps incrementing its counter, Process 0 will eventually reach m and in the interim no other process can set their counter value to m.

- *Any state in which Process 0 has a fresh counter value m is eventually followed by a state in which all processes have counter value m.* It is easy to see that the value m moves clockwise around the ring "flushing" any other counter values, while Process 0 remains at m. This is why I call this paradigm counter flushing.

The net effect is that any execution of $D1$ eventually reaches a good state in which it remains. The general counter flushing paradigm can be stated roughly as follows:

- A sender (in our case, Process 0) periodically sends a message to a set of responders; after all responders have received the message the sender sends the next message (this corresponds to one cycle around the ring in our case).

- To make the protocol stabilizing, the sender numbers each message with a counter. The size of the counter must be greater than the maximum number of distinct counter values that can be present in the network in any state. The sender increments its counter after the message has reached all responders.

- The protocol must ensure that the sender counter value will always increment and so eventually reach a "fresh" value not present in the network. A freshly numbered message m from the sender must guarantee that all old counter values will be "flushed" from the network before the sender sends the next message.

In Dijkstra's first example, the flushing property is guaranteed because the topology consists of a ring of unidirectional links. Similar forms of counter flushing can be used to implement Data Links ([AfekB89]) and token passing [DolevIM91:unbound]) between a pair of nodes. Counter flushing is, however, not limited to rings or pair of nodes. Katz and Perry

[KatzP90] extend the use of counter flushing to arbitrary networks in an ingenious way. The stabilizing end-to-end protocol ([APV-91]) is obtained by first applying local correction to the Slide protocol [AGR92:slide] and then applying a variant of counter flushing to the Majority protocol of [AGR92:slide].

23.4 Message Passing Model

Having understood the techniques of counter flushing and local checking and correction in a shared memory model, we move to a more realistic message passing model. The work in the next three sections is based on [V92], which formalizes the ideas introduced in [APV-91]. To model a network protocol, we will model the network topology, the links between nodes, and the nodes themselves. Our model is essentially the standard asynchronous message passing model except for two major differences:

- The major difference is that links are restricted to store at most one packet at a time.
- We assume that for every pair of neighbors, there is some *a priori* way of assigning one of the two nodes as the "leader" of the pair.

We will argue that even with these differences our model can be implemented in real networks.

23.4.1 Modeling the Topology

We will call a directed graph (V, E) *symmetric* if for every edge $(u, v) \in E$ there is an edge $(v, u) \in E$.

Definition 3 *A* topology graph *$G = (V, E, l)$ is a symmetric, directed graph (V, E) together with a leader function l such that for every edge $(u, v) \in E$, $l(u, v) = l(v, u)$ and either $l(u, v) = u$ or $l(u, v) = v$.*

We use $E(G)$ and $V(G)$ to denote the set of edges and nodes in G. If it is clear what graph we mean we sometimes simply say E and V. As usual, if $(u, v) \in E$ we will call v a neighbor of u.

23.4.2 Modeling Links

Traditional models of a data link have used what we call *Unbounded Storage* Data Links that can store an unbounded number of packets. Now, real physical links do have bounds on the

Each p belongs to the packet alphabet P defined above.

The state of the automaton consists of a single variable $Q_{u,v} \in P \cup nil$.

```
SEND_{u,v}(p) (*input action*)
    Effect:
        If $Q_{u,v} = nil$ then $Q_{u,v} := p$;

FREE_{u,v} (*output action*)
    Precondition: $Q_{u,v} = nil$
    Effect: None

RECEIVE_{u,v}(p) (*output action*)
    Precondition: $p = Q_{u,v} \neq nil$
    Effect: $Q_{u,v} := nil$;
```

The FREE and RECEIVE actions are in separate classes

Figure 23.7: Unit Storage Data Link automaton

number of stored packets. However, the unbounded storage model is a useful abstraction in a non-stabilizing context.

Unfortunately, this is no longer true in a stabilizing setting. *If the link can store an unbounded number of packets, it can have an unbounded number of "bad" packets in the initial state.* It has been shown [DolevIM91:unbound] that almost any non-trivial task is impossible in such a setting. Thus in a stabilizing setting it is necessary to define Data Links that have bounded storage.

A *network automaton* for topology graph G consists of a node automaton for every vertex in G and one channel automaton for every edge in G. We will restrict ourselves to a special type of channel automaton, a unit storage data link or UDL for short. Intuitively, a UDL can only store at most one packet at any instant. Node automata communicate by sending packets to the UDLs that connect them. In the next section, we will argue that a UDL can be implemented over real physical channels.

We fix a packet alphabet P. We assume that $P = P_{data} \cup P_{control}$ consists of two disjoint packet alphabets. These correspond to what we call data packets and control packets. The specification for a UDL will allow both data and control packets to be sent on a UDL.

Definition 4 *We say that $C_{u,v}$ is the UDL corresponding to ordered pair (u, v) if $C_{u,v}$ is the IOA defined in Figure 23.7.*

$C_{u,v}$ is a UIOA since we have not defined any start states for $C_{u,v}$. The external interface to $C_{u,v}$ includes an action to send a packet at node u ($\textsc{Send}_{u,v}(p)$), an action to receive a packet at node v ($\textsc{Receive}_{u,v}(p)$), and an action $\textsc{Free}_{u,v}$ to tell the sender that the link is ready to accept a new packet. The state of $C_{u,v}$ is simply a single variable $Q_{u,v}$ that stores a packet or has the default value of *nil*.

Notice two points about the specification of a UDL. The first is that if the UDL has a packet stored, then any new packet sent will be dropped. Second, the FREE action is enabled continuously whenever the UDL does not contain a packet.

23.4.3 Modeling Network Nodes and Networks

Next we specify node automata. We do so using a set that contains a node automaton for every node in the topology graph. For every edge incident to a node u, a node automaton N_u must have interfaces to send and receive packets on the channels corresponding to that edge. However, we will go further and require that nodes obey a certain stylized convention in order to receive feedback from and send packets on links.

In the specification for a UDL if a packet p is sent when the UDL already has a packet stored, then the new packet p is dropped. We will prevent packets from being dropped by requiring that the sending node keep a corresponding *free* variable for the link that records whether or not the link is free to accept new packets. The sender sets the *free* variable to *true* whenever it receives a FREE action from the link. We require that the sender only send packets on the link when the *free* variable is *true*. Finally, whenever the sender sends a packet on the link, the sender sets its *free* variable to *false*.

We wish the interface to a UDL to stabilize to "good behavior" even when the sender and link begin in arbitrary states. Suppose the sender and the link begin in arbitrary states. Then we can have two possible problems. First, if $\mathit{free} = \mathit{true}$ but the UDL contains a packet, then the first packet sent by the sender can be dropped. However, it is easy to see that all subsequent packets will be accepted and delivered by the link. This is because after the first packet is sent, the sender will never set *free* to *true* unless it receives a FREE notification from the link. But a FREE notification is delivered to the sender only when the link is empty. The second possible problem is deadlock. Suppose that initially $\mathit{free} = \mathit{false}$ but the channel does not contain a packet. To avoid deadlock, the UDL specification ensures that the FREE action is enabled continuously whenever the link does not contain a packet.

A formal description of node automata can be found in [V92]. We omit it here. Finally a network automaton is the composition of a set of node and channel automata.

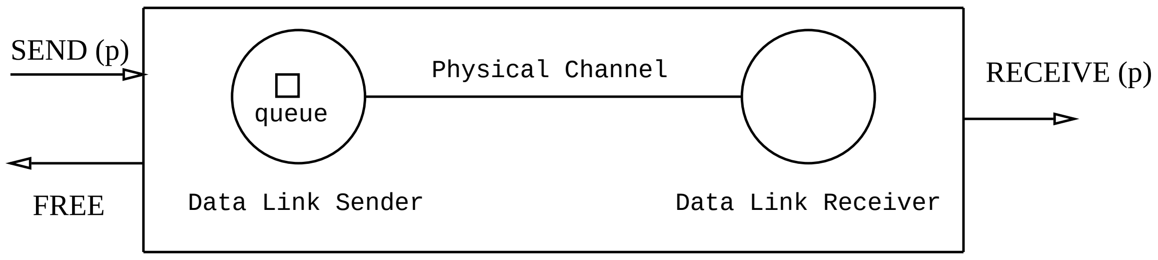


Figure 23.8: Implementing a UDL over a physical channel

23.4.4 Implementing the Model in Real Networks

In a real network implementation, the physical channel connecting any two neighboring nodes would typically not be a UDL. For example, a telephone line connecting two nodes can often store more than one packet. The physical channel may also not deliver a free signal. Instead, an implementation can *construct* a Data Link protocol on top of the physical channel such that the resulting Data Link protocol *stabilizes* to the behaviors of a UDL (e.g. [AfekB89], [S-89]).

Figure 23.8 shows the structure of such a Data Link protocol over a physical link. The sender end of the Data Link protocol has a queue that can contain a single packet. When the queue is empty, the FREE signal is enabled. When a SEND(p) arrives and the queue is empty, p is placed on the queue; if the queue is full, p is dropped. If there is a packet on the queue, the sender end constantly attempts to send the packet. When the receiving end of the Data Link receives a packet, the receiver sends an ack to the sender. When the sender receives an ack for the packet currently in the queue, the sender removes the packet from the queue.

If the physical channel is initially empty and the physical channel is FIFO (i.e., does not permute the order of packets), then a standard stop and wait or alternating bit protocol [BSW-69] will implement a UDL. However, if the physical channel can initially store packets, then the alternating bit protocol is not stabilizing [S-89]. There are two approaches to creating a stabilizing stop and wait protocol. Suppose the physical channel can store at most X packets in both directions. Then [AfekB89] suggest numbering packets using a counter that has at least $X + 1$ values. Suppose instead that no packet can remain on the physical channel for more than a bounded amount of time. [S-89] exploits such a time bound to build a stabilizing Data Link protocol. The main idea is to use either numbered packets (counter flushing) or timers (timer flushing) to "flush" the physical channel of stale packets.

A stop and wait protocol is not very efficient over physical channels that have a high transmission speed and/or high latency. It is easy to generalize a UDL to a *Bounded Storage Data Link* or BDL that can store more than one packet.

23.5 Local Checking and Correction in our Message Passing Model

In this section, we introduce formal definitions of local checkability and local correctability. The definitions for a message passing model will be slightly more complex than corresponding ones for shared memory model because of the presence of channels between nodes.

23.5.1 Link Subsystems and Local Predicates

Consider a network automaton with graph G. Roughly speaking, a property is said to be local to a subgraph G' of G if the truth of the property can be ascertained by examining only the components specified by G'. We will concentrate on *link subsystems* that consist of a pair of neighboring nodes u and v and the channels between them. It is possible to generalize our methods to arbitrary subsystems.

In the following definitions, we fix a network automaton $\mathcal{N} = Net(G, N)$.

Definition 5 *We define the (u, v) link subsystem of $\mathcal{N}$ as the composition of N_u, $C_{u,v}$, $C_{v,u}$, and N_v.*

For any state s of $\mathcal{N}$: $s|u$ denotes s projected on to node N_u and $s|(u, v)$ denotes s projected onto $C_{u,v}$. Thus when $\mathcal{N}$ is in state s, the state of the (u, v) subsystem is the 4-tuple: $(s|u, s|(u, v), s|(v, u), s|v)$.

A predicate L of $\mathcal{N}$ is a subset of the states of $\mathcal{N}$. Let (u, v) be some edge in graph G of $\mathcal{N}$. A *local predicate* $L_{u,v}$ of $\mathcal{N}$ for edge (u, v) is a subset of the states of the (u, v) subsystem in $\mathcal{N}$. We use the word "local" because $L_{u,v}$ is defined in terms of the (u, v) subsystem.

The following definition provides a useful abbreviation. It describes what it means for a local property to hold in a state s of the entire automaton.

Definition 6 *We say that a state s of $\mathcal{N}$* satisfies *a local predicate $L_{u,v}$ of $\mathcal{N}$ iff*

$$(s|u, s|(u, v), s|(v, u), s|v) \in L_{u,v} \ .$$

We will make frequent use of the concept of a closed predicate. Intuitively, a property is closed if it remains true once it becomes true. In terms of local predicates:

Definition 7 *A local predicate $L_{u,v}$ of network automaton $\mathcal{N}$ is closed if for all transitions $(s, \pi, \tilde{s})$ of $\mathcal{N}$, if s satisfies $L_{u,v}$ then so does $\tilde{s}$.*

The following definitions provide two more useful abbreviations. The first gives a name to a collection of local predicates, one for each edge in the graph. The second, the conjunction of a collection of "local properties", is the property that is true when all local properties hold at the same time. We will require that the conjunction of the local properties is non-trivial – i.e., there is some global state that satisfies all the local properties.

Definition 8 *$\mathcal{L}$ is a* link predicate set *for $\mathcal{N} = Net(G, N)$ if for each $(u, v) \in G$ there is some $L_{u,v}$ such that:*

- *If $(a, b, c, d) \in L_{u,v}$ then $(d, c, b, a) \in L_{v,u}$. (i.e., $L_{u,v}$ and $L_{v,u}$ are identical except for the way the states are written down.)*
- $\mathcal{L} = \{L_{u,v}, (u, v) \in G\}$
- *There is at least one state s of $\mathcal{N}$ such that s satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}$.*

Definition 9 *The conjunction of a link predicate set $\mathcal{L}$ is the predicate $\{s : s$ satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}\}$. We use $Conj(\mathcal{L})$ to denote the conjunction of $\mathcal{L}$.*

Note that $Conj(\mathcal{L})$ cannot be the null set by the definition of a link predicate set.

23.5.2 Local Checkability

Suppose we wish a network automaton $\mathcal{N}$ to satisfy some property. An example would be the property "all nodes have the same color". We can often specify a property of $\mathcal{N}$ formally using a predicate L of $\mathcal{N}$. Intuitively, $\mathcal{N}$ can be locally checked for L if we can ascertain whether L holds by checking all link subsystems of $\mathcal{N}$. The motivation for introducing this notion is performance: in a distributed system we can check all link subsystems in parallel in constant time. We formalize the intuitive notion of a locally checkable property as follows.

Definition 10 *A network automaton $\mathcal{N}$ is locally checkable for predicate L using link predicate set $\mathcal{L}$ if:*

- *$\mathcal{L}$ is a link predicate set for $\mathcal{N}$ and $L \supseteq Conj(\mathcal{L})$.*
- *Each $L_{u,v} \in \mathcal{L}$ is closed.*

The first item in the definition requires that L holds if a collection of local properties all hold. The second item is perhaps more surprising. It requires that each local property also be closed.

We add this extra requirement because in an asynchronous distributed system it appears to be impossible to check whether an arbitrary local predicate holds *all* the time. What we can do is to "sample" the local subsystem periodically to see whether the local property holds. Suppose the network automaton consists of three nodes u, v and w and such that v is the neighbor of both u and w. Suppose the property L that we wish to check is the conjunction of two local predicates $L_{u,v}$ and $L_{v,w}$. Suppose further that exactly one of the two predicates is always false, and the predicate that is false is constantly changing. Then whenever we "check" the (u, v) subsystem we might find $L_{u,v}$ true. Similarly whenever we "check" the (v, w) subsystem we might find $L_{v,w}$ true. Then we may never detect the fact that L does not hold in this execution. We avoid this problem by requiring that $L_{u,v}$ and $L_{v,w}$ be closed.

23.5.3 Local Correctability

The motivation behind local checking was to efficiently ensure that some property L holds for network automaton $\mathcal{N}$. We would also like to *efficiently* correct $\mathcal{N}$ to make the property true. We have already set up some plausible conditions for local checking. Can we find some plausible conditions under which $\mathcal{N}$ can be *locally corrected*?

To this end we define a local reset function f. This is a function with three arguments: the first argument is a node say u, the second argument is any state of node automaton N_u, and the second argument is a neighbor v of u. The function produces a state of the node automaton corresponding to the first argument. Let s be a state of $\mathcal{N}$; recall that $s|u$ is the state of N_u. Then $f(u, s|u, v)$ is the state of N_u obtained by applying the local reset function at u with respect to neighbor v. We will abuse notation by omitting the first argument when it is clear what the first argument is. Thus we prefer to write $f(s|u, v)$ instead of the more cumbersome $f(u, s|u, v)$.

We will insist that f meet two requirements so that f can be used for local correction (Definition 11).

Assume that the property L holds if a local property $L_{u,v}$ holds for every edge (u, v). The first requirement is that if any (u, v) subsystem does not satisfy $L_{u,v}$, then applying f to both u and v should result in making $L_{u,v}$ hold. More precisely, let us assume that by some magic we have the ability to simultaneously:

- Apply f to N_u with respect to v;
- Apply f to N_v with respect to u;
- Remove any packets stored in channels $C_{u,v}$ and $C_{v,u}$.

Then the resulting state of the (u, v) subsystem should satisfy $L_{u,v}$. Of course, in a real distributed system such simultaneous actions are clearly impossible. However, we will achieve essentially the same effect by applying a so-called "reset" protocol to the (u, v) subsystem. We will describe a stabilizing local reset protocol for this purpose in the next section.

The first requirement allows nodes u and v to correct the (u, v) subsystem if $L_{u,v}$ does not hold. But other subsystems may be correcting at the same time! Since subsystems overlap, correction of one subsystem may invalidate the correctness of an overlapping subsystem. For example, the (u, v) and (v, w) subsystems overlap at v. If correcting the (u, v) subsystem causes the (v, w) subsystem to be incorrect, then the correction process can "thrash". To prevent thrashing, we add a second requirement. In its simplest form, we might require that correction of the (u, v) subsystem leaves the (v, w) subsystem correct *if* the (v, w) subsystem was correct in the first place.

However, there is a more general definition of a reset function f that turns out to be useful. Recall that we wanted to avoid thrashing that could be caused if correcting a subsystem causes an adjacent subsystem to be incorrect. Informally, let us say that the (u, v) subsystem depends on the (v, w) subsystem if correcting the (v, w) subsystem can invalidate the (u, v) subsystem. If this dependency relation is cyclic, then thrashing can occur. On the other hand if the dependency relation is acyclic then the correction process will eventually stabilize. Such an acyclic dependency relation can be formalized using a partial order $<$ on *unordered* pairs of nodes: informally, the (u, v) subsystem depends on the (v, w) subsystem if $\{v, w\} < \{u, v\}$.

Using this notion of a partial order, we present the formal definition of a local reset function:

Definition 11 *We say f is a local reset function for network automaton $\mathcal{N} = net(G, N)$ with respect to link predicate set $\mathcal{L} = \{L_{u,v}\}$ and partial order $<$, if for any state s of $\mathcal{N}$ and any edge (u, v) of G:*

- **Correction:** $(f(s|u, v), nil, nil, f(s|v, u)) \in L_{u,v}$.
- **Stability:** *For any neighbor w of v,*

  If $(s|u, s|(u, v), s|(v, u), s|v) \in L_{u,v}$ and $\{v, w\} \not< \{u, v\}$ then
  $(s|u, s|(u, v), s|(v, u), f(s|v, w)) \in L_{u,v}$.

Notice that in the special case where all the link subsystems are independent, no edge is "less" than any other edge in the partial order.

Using the definition of a reset function, we can define what it means to be locally correctable.

Definition 12 *A network automaton $\mathcal{N}$ is* locally correctable *to L using link predicate set $\mathcal{L}$, local reset function f, and partial order $<$ if:*

- *$\mathcal{N}$ is locally checkable for L using $\mathcal{L}$.*

- *f is a local reset function for $\mathcal{N}$ with respect to $\mathcal{L}$ and $<$.*

Intuitively, if we have a reset function f with partial order $<$ we can expect the local correction to stabilize in time proportional to the maximum chain length in the partial order. Recall that a chain is a sequence $a_1 < a_2 < a_3 \ldots, < a_n$. Thus the following piece of notation is useful.

Definition 13 *For any partial order $<$, height($<$) is the length of the maximum length chain in $<$.*

23.6 Local Correction Theorem

23.6.1 Theorem Statement

In the previous section, we set up plausible conditions under which a network automaton can be locally corrected to achieve a property L. We claimed that these conditions could be exploited to yield local correction. In this section we make these claims precise.

Intuitively, the result states that if $\mathcal{N}$ is locally correctable to L using local reset function f and partial order $<$, then we can transform $\mathcal{N}$ into a new *augmented automaton* $\mathcal{N}^+$ such that $\mathcal{N}^+$ satisfies the following property: eventually every behavior of $\mathcal{N}^+$ will "look like" a behavior of $\mathcal{N}$ in which L holds. We use the notation $\mathcal{N}|L$ to denote an automaton identical to $\mathcal{N}$ except that the initial states of $\mathcal{N}|L$ are restricted to lie in set L.

Theorem 2 Local Correction: *Consider any network automaton $\mathcal{N} = Net(G, N)$ that is* locally correctable *to L using link predicate set $\mathcal{L}$, local reset function f, and partial order $<$. Then there exists some $\mathcal{N}^+$ that is a UIOA and a network automaton such that $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(c)|L$.*

So far we have ignored time complexity. However, time complexity is of crucial importance to a protocol designer because nobody would be interested in a protocol that took years to stabilize. Thus, [V92], uses a version of the timed automata model throughout and uses this model to give a formal definition of stabilization time. However, intuitively, we can measure time complexity as follows. We assume that it takes 1 time unit to send any message on a link and to deliver a free action; we assume that node processing takes time 0. With this assumption, we can define the stabilization time of a behavior to be time it takes before the behavior has a suffix that belongs to the target set. We define the overall stabilization time as the worst case stabilization time across all behaviors.

Using this intuitive definition of stabilization time we can state a more refined version of the Local Correction Theorem. The refinement we add is that $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(c)|L$ in time proportional to *height*($<$), the height of the partial order.

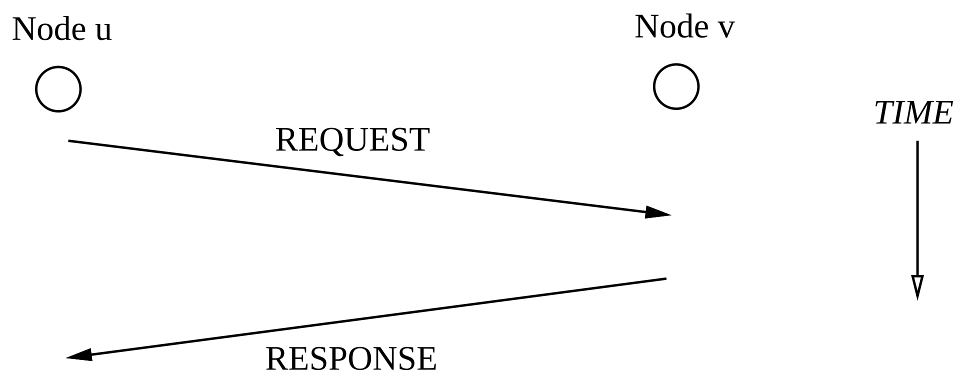


Figure 23.9: The structure of a single phase of the local snapshot/reset protocol

23.6.2 Overview of the Transformation Code

To transform $\mathcal{N}$ into $\mathcal{N}^+$ we will add actions and states to $\mathcal{N}$. These actions will be used to send and receive *snapshot packets* (that will be used to do local checking on each link subsystem) and *reset* packets (that will be used to do local correction on each link subsystem). For every link (u, v), the leader $l(u, v)$ initiates the checking and correction of the (u, v) subsystem. The idea is, of course, that the leader of each (u, v) subsystem will periodically do a *local snapshot* to check if the (u, v) subsystem satisfies its predicates; if the leader detects a violation, it tries to make the local predicate true by doing a *local reset* of the (u, v) subsystem.

The structure of our local snapshot protocol is slightly different from the Chandy-Lamport snapshot protocol [ChandyL85] studied earlier in the course. It is possible to show [V92] that the Chandy-Lamport scheme cannot be used without modifications over unit storage links. (Prove this: it will help you understand the Chandy-Lamport scheme better.)

Our local snapshot/reset protocol works roughly as follows. Consider a (u, v) subsystem. Assume that $l(u, v) = u$ – i.e., u is the leader on link (u, v). A single snapshot or reset phase has the structure shown in Fig 23.9.

A single phase of either a snapshot or reset procedure consists of u sending a request that is received by v, followed by v sending a response that is received by u. During a phase, node u sets a flag ($check_u[v]$) to indicate that it is checking/correcting the (u, v) subsystem. While this flag is set, *no packets other than request packets can be sent on link* $C_{u,v}$. Since a phase completes in constant time, this does not delay the data packets by more than a constant factor.

In what follows, we will use the basic state at a node u to mean the part of the state at u "corresponding" to automaton N_u. To do a snapshot, node u sends a snapshot request to

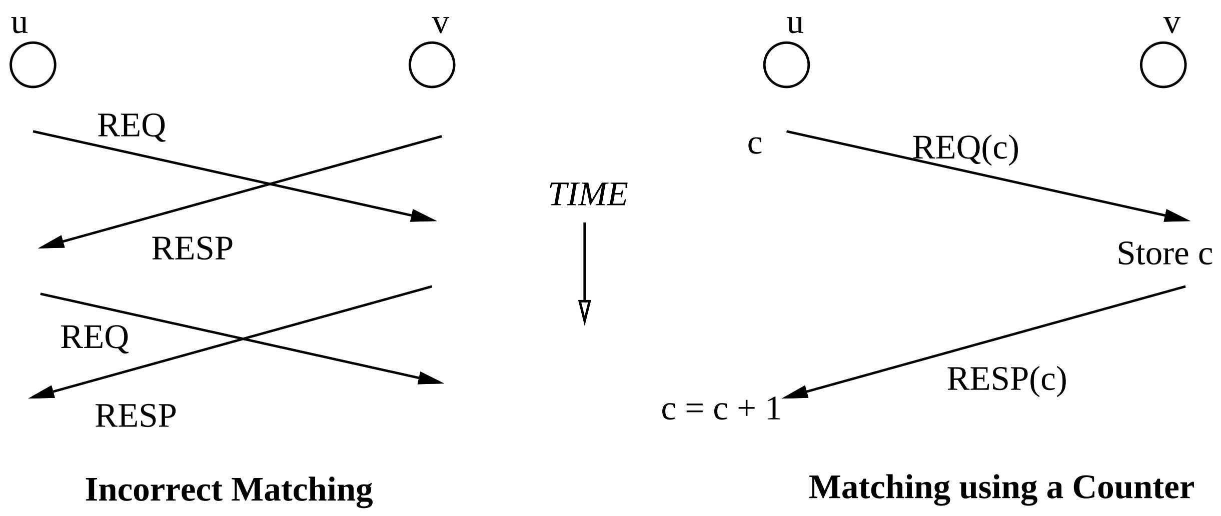


Figure 23.10: Using counter flushing to ensure that request-response matching will work correctly within a small number of phases.

v. A snapshot request is identified by a *mode* variable in the request packet that carries a *mode* of *snapshot*. If v receives a request with a *mode* of *snapshot*, Node v then records its basic state (say s) and sends s in a response to u.

When u receives the response, it records its basic state (say r). Node u then records the state of the (u, v) subsystem as $x = (r, nil, nil, s)$. If $x \notin L_{u,v}$ (i.e., local property $L_{u,v}$ does not hold) then u initiates a reset.

To do a reset, node u sends a reset request to v. A reset request is identified by a *mode* variable in the request packet that carries a *mode* of *reset*. Recall that f denotes the local reset function. After v receives the request, v changes its basic state to $f(v, s, u)$, where s is the previous value of v's basic state. Node v then sends a response to u. When u receives the response, u changes its basic state to $f(u, r, v)$, where r is the previous value of u's basic state.

Of course, the local snapshot and reset protocol must also be stabilizing. However, the protocol we just described informally may fail if requests and responses are not properly matched. This can happen, for instance, if there are spurious packets in the initial state of $\mathcal{N}^+$. It is easy to see that both the local snapshot and reset procedures will only work correctly if the response from v is sent following the receipt of the request at u. The diagram on the left of Figure 23.10 shows a scenario in which requests are matched incorrectly to "old" responses that were sent in previous phases. Thus we need each phase to eventually follow the structure shown in the right of Figure 23.10.

To make the snapshot and reset protocols stabilizing, we use counter flushing. Thus we number all request and response packets. Thus each request and response packet carries a number *count*. Also, the leader u keeps a variable $count_u[v]$ that u uses to number all requests sent to v within a phase. At the end of the phase, u increments $count_u[v]$. Similarly, the responder v keeps a variable $count_v[u]$ in which v stores the number of the last request it has received from u. Node v weeds out duplicates by only accepting requests whose number is not equal $count_u[v]$.

Clearly the *count* values can be arbitrary in the initial state and the first few phases may not work correctly. However, counter flushing guarantees that in constant time a response will be properly matched to the correct request. Because the links are unit storage, there can be at most 3 distinct counter values (one in each link and one at the receiver) stored in the link subsystem. Thus a space of 4 numbers is sufficient to guarantee that eventually (more precisely within 5 phases) requests and responses are correctly matched.

Besides properly matching requests and responses, we must also avoid deadlock when the local snapshot/reset protocol begins in an arbitrary state. To do so, when $check_u[v]$ is *true* (i.e., u is in the middle of a phase), u continuously sends requests. Since v weeds out duplicates this does no harm and also prevents deadlock. Similarly, v continuously sends responses to the last request v has received. Once the responses begin to be properly matched to requests, this does no harm, because u discards such duplicate responses.

23.7 Intuitive Proof of Local Correction Theorem

A formal proof of the Local Correction theorem can be found in [V92]. In this section, we provide the main intuition. We have already described how to transform a locally correctable automaton $\mathcal{N}$ into an augmented automaton $\mathcal{N}^+$ We have to prove that in time proportional to the height of the partial order every behavior of $\mathcal{N}^+$ is a behavior of $\mathcal{N}$ in which all local predicates hold.

We have already described the intuition behind the use of a counter to ensure proper request-response matching. We now describe the intuition behind the local snapshot and reset procedures. The intuition behind the snapshot is very similar to the intuition behind the snapshot procedure studied earlier in the course.

23.7.1 Intuition Behind Local Snapshots

The diagram on the left of Figure 23.11 shows why a snapshot works correctly if the response from v is sent following the receipt of the request at u. Let a' and b be the state of nodes u and v respectively just *before* the response is *sent*. Let a and b' be the state of nodes u and v respectively just *after* the response is delivered. This is sketched in Figure 23.11.

The snapshot protocol must guarantee that node u does not send any data packets to v during a phase. Also v cannot send another data packet to u from the time the response is sent until the response is delivered. This is because the link from v to u is a UDL that will not give a free indication until the response is received. Recall that *nil* denotes the absence of any packet on a link. Thus the state of the (u, v) subsystem just before the response is sent is (a', nil, nil, b). Similarly, the state of the (u, v) subsystem just after the response is delivered is (a, nil, nil, b').

We claim that it is *possible* to construct some other execution of the (u, v) subsystem which starts in state (a', nil, nil, b), has an intermediate state equal to (a, nil, nil, b) and has a final state equal to (a, nil, nil, b'). This is because we could have first applied all the actions that changed the state of node u from a' to a, which would cause the (u, v) subsystem to reach the intermediate state. Next, we could apply all the actions that changed the state of node v from b to b', which will cause the (u, v) subsystem to reach the final state. Note that this construction is only possible because u and v do not send data packets to each other between the time the response is sent and until the time the response is delivered.

Thus the state (a, nil, nil, b) recorded by the snapshot is a *possible* successor of the state of (u, v) subsystem when the response is sent. The recorded state is also a a *possible* predecessor of the state of (u, v) subsystem when the response is delivered. But $L_{u,v}$ is a closed predicate – it remains true once it is true. Thus if $L_{u,v}$ was true just before the response was sent, then the state recorded by the snapshot must also satisfy $L_{u,v}$. Similarly, if $L_{u,v}$ is false just after the response is delivered, then the state recorded by the snapshot *does not* satisfy $L_{u,v}$. Thus the snapshot detection mechanism *will not produce false alarms* if the local predicate holds at the start of the phase. Also the snapshot mechanism *will detect a violation* if the the local predicate does not hold at the end of the phase.

23.7.2 Intuition Behind Local Resets

The diagram on the right of Figure 23.11 shows why a local reset works correctly if the response from v is sent following the receipt of the request at u. Let b be the state of node v just *before* the response is *sent*. Let a and b' be the state of nodes u and v respectively just *before* the response is delivered. This is sketched in Figure 23.11.

The code for an augmented automaton will ensure that just after the response is sent, node v will locally reset its state to $f(b, u)$. Similarly, immediately after it receives the response, node u will locally reset its state to $f(a, v)$. Using similar arguments to the ones used for a snapshot, we can show that there is some execution of the (u, v) subsystem which begins in the state $(f(a, v), nil, nil, f(b, u))$ and ends in the state $(f(a, v), nil, nil, b')$. But the latter state is the state of the (u, v) subsystem immediately after the response

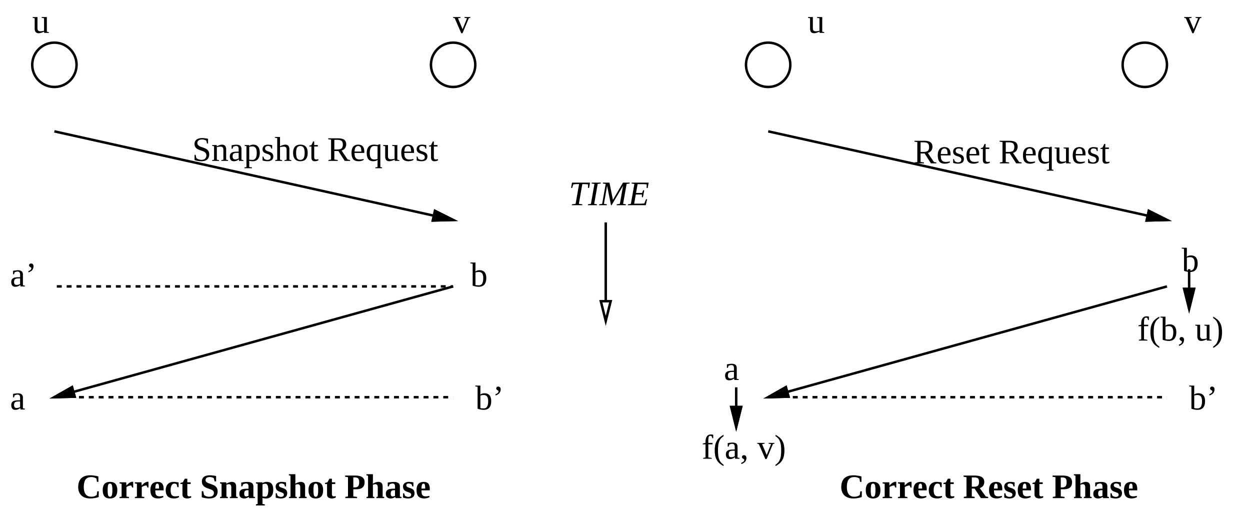


Figure 23.11: Local Snapshots and Resets work correctly if requests and responses are properly matched.

is delivered. But we know, from the correction property of a local reset function, that $(f(a,v), nil, nil, f(b,u))$ satisfies $L_{u,v}$. Since $L_{u,v}$ is a closed predicate, we conclude that $L_{u,v}$ holds at the end of the reset phase.

23.7.3 Intuition Behind Local Correction Theorem

We can now see intuitively why the augmented automaton will ensure that all local predicates hold in time proportional to the height of the partial order. Consider a (u,v) subsystem where $\{u,v\} \not< \{w,x\}$ for any pair of neighbors w, x – i.e., $\{u,v\}$ is a minimal element in the partial order. Then, within 5 phases of the (u,v) subsystem the request-response matching will begin to work correctly. If the sixth phase of the (u,v) subsystem is a snapshot phase, then either $L_{u,v}$ will hold at the end of the phase or the snapshot will detect a violation. But in the latter case, the seventh phase will be a reset phase which will cause $L_{u,v}$ to hold at the end of the seventh phase.

But once $L_{u,v}$ remains true, it remains true. This is because $L_{u,v}$ is a closed predicate of the original automaton $\mathcal{N}$ and the only extra actions we have added to $\mathcal{N}^+$ that can affect $L_{u,v}$ are actions to locally reset a node using the reset function f. But by the stability property of a local reset function, any applications of f at u with respect to some neighbor other than v cannot affect $L_{u,v}$. Similarly, any applications of f at v with respect to some neighbor other than u cannot affect $L_{u,v}$. Thus in constant time, the local predicates – corresponding to link subsystems that are minimal elements in the partial order – will become and remain true.

Now suppose that the local predicates for all subsystems with height $\leq i$ hold from some state s_i onward. By similar arguments, we can show that in constant time after s_i, the local predicates for all subsystems with height $i+1$ become and remain true. Once again, the argument depends crucially on the stability property of a local reset function. The intuition is that applications of the local reset function to subsystems with height $\leq i$ do not occur after state s_i. But these are the only actions that can falsify the local predicates for subsystems with height $i+1$. The net result is that all local predicates become and remain true within time proportional to the height of the partial order $<$.

23.8 Implementing Local Checking in Real Networks: Timer Flushing

In the previous section, we made the snapshot/reset protocol stabilizing by numbering snapshot requests and responses. We also relied on the fact that each link was a UDL. In practice, however, there is an even simpler way of making the snapshot/reset protocol stabilizing. This can be done using timers.

Suppose there is a known bound on the length of time a packet can be stored in a link and a known bound on the length of time between the delivery of a request packet and the sending of a matching response packet. Then by controlling the interval between successive snapshot/reset phases it is easily possible to obtain a stabilizing snapshot protocol. The interval is chosen to be large enough such that all packets from the previous phase will have disappeared at the end of the interval [APV91:sigcomm]. We call the general method timer flushing.

The main idea in timer flushing is to bound the lifetime of "old" state information in the network. This is done by using node clocks that run at approximately the same rate and by enforcing a maximum packet lifetime over every link. State information that is not periodically refreshed is "timed out" by the nodes. Timer flushing has been used in the OSI Network Layer Protocol [Perlman83] and the IEEE 802.1 Spanning Tree protocol [Perlman85]. Spinelli [S-88] uses timer flushing to build (practical) stabilizing Data Link and virtual circuit protocols.

In most real networks, each node sends "keep-alive" packets periodically on every link in order to detect failures of adjacent links. If no keep-alive packet arrives before a local timer expires, the link is assumed to have failed. Thus, it is common practice to assume time bounds for the delivery and processing of packets. Note also that the snapshot and reset packets used for local checking can be "piggy-backed" on these keep-alive packets without any appreciable loss in efficiency.

23.9 Summary

Self-stabilization was introduced by Dijkstra in a seminal paper [Dijkstra74]. Dijkstra's shared memory model is enchanting in its simplicity and makes an ideal vehicle to describe non-trivial examples of self-stabilization. We showed how to simulate Dijkstra's model as one big IOA with a few restrictions. We also used Dijkstra's first two examples to introduce two powerful methods for self-stabilization: counter flushing and local checking and correction.

Next, we moved to a more realistic message passing model. We introduced a reasonable model for bounded storage links. Using this model, we formally defined the notions of local checkability and correctability and described the local correction theorem. This theorem shows that any locally correctable protocol can be stabilized in time proportional to the height of a partial order that underlies the definition of local correctability. Our proof of this theorem was constructive: we showed how to do augment the original protocol with snapshot and reset actions. We made the local snapshot and reset protocols stabilizing using counter flushing. While counter flushing relies on bounding the number of packets that can be stored on a link, we observed that a practical implementation would be based on bounding the time that a message could stay on a link – we called this technique timer flushing.

Together counter flushing, timer flushing, and local checking (with and without local correction) can be used to explain a number of self-stabilizing protocols and can be used to design new protocols. While these techniques cannot be used to solve every problem, there are a large number of useful protocols that can be efficiently stabilized using these notions. In the next lecture, Boaz will provide an example of a reset protocol that – somewhat surprisingly - is locally correctable using the trivial partial order. In other places we have used local checking to provide solutions for mutual exclusion, the end-to-end problem, stabilizing synchronizers, and spanning tree protocols. More detailed references can be be found in [APV-91], [V92] and AVarghese91.

The messages used for local checking can be piggybacked on keepalive traffic in real networks without any appreciable loss of efficiency. As a side benefit, local checking also provides a useful debugging tool. Any violations of local checking can be logged for further examination. In a trial implementation of our reset procedure on the Autonet [Auto90], local checking discovered bugs in the protocol code. In the same vein, local checking can provide a record of catastrophic, transient faults that are otherwise hard to detect.

Lecturer: Boaz Patt-Shamir

Lecture 24

24.1 Self-Stabilization and Termination

Can a self-stabilizing protocol terminate? This question has several possible answers. First, consider the regular IO automata model, and the behavior stabilization definition. In this setting, if the problem spec allows for finite behaviors, then the trivial automaton that does nothing is a self-stabilizing solution. This is clearly unsatisfactory. We shall use a different formulation of the model to see that no "interesting" problem admits a terminating self-stabilizing protocol.

Consider the *input/output relation* specification of problems, defined as follows. In each process, there are special input and output registers; the problem is defined in terms of a *relation* that specifies for each input assignment what are the possible output assignments.[14] We also assume that some of the states are designated as "terminating", i.e., no action is enabled in them. This is just another formulation of a *non-reactive* distributed task. We saw quite a few such tasks, e.g., MST, shortest-paths, leader election, etc.

In this setting, it is easy to see that a self-stabilizing protocol can have no terminating states, if it satisfies the following *non-locality* condition:

> There exist an input value v, an output value v', a node j, and two input assignments I and I' such that v is assigned to j in both I and I', v' is a possible output value for I and is not a possible output value for I'.

Intuitively, the non-locality condition means that node j cannot tell locally that having v as input and v' as output is correct or wrong. We remark that all interesting distributed tasks have this property: any other task amounts to a collection of unrelated processes.

The moral of this obvious fact is that self-stabilizing algorithms must constantly verify whether their output (or, more generally, their state) is "updated" with respect to the rest of the system, and therefore they cannot terminate.

[14]This can be formalized in the IOA model by assuming that the input value is input to the system infinitely often, and that the output action that carries the output value is continuously enabled.

24.2 Self-Stabilization and Finite State

Let us consider an issue which may seem unrelated to self-stabilization at first glance, namely *infinite-state protocols*. In particular, consider the idea of unbounded registers. We went through a lot of difficulty (conceptually) and paid a significant price (in terms of performance) to get bounded-registers protocols (e.g., applying CTS to Lamport's bakery).

This approach is easy to criticize from the practical viewpoint: to be concrete, suppose that the registers contain 64 bits. There is no system today that can hit the bound of 2^{64} in any practical application. However, note that this argument depends crucially on the assumption that the registers are *initialized properly*. For instance, consider the case where, as a result of a transient fault, the contents of the registers may be corrupted. Then no matter how big we let the registers be, they may hit their bound very quickly. On the other hand, any feasible system can have only finitely many states, and in many cases it is desirable to be able to withstand transient faults.

In short, unbounded registers are usually a convenient abstraction, given that the system is initialized properly, while self-stabilizing protocols are useful only when they have finitely many states. This observation gives rise to two possible approaches. The first is to develop only finite-state self-stabilizing protocols, tailored to the specific applications. The second is to have some kind of a general transformation that enables us to design abstract unbounded-state protocols, using real-world, bounded-size registers, and whenever the physical limit is hit, to *reset* the system.

The main tool required for the second approach is a *self-stabilizing reset protocol.* This will be the focus of our lecture today.

24.3 Self-stabilizing Reset Protocol

24.3.1 Problem statement

Informally, the goal of the reset procedure is that whenever it is invoked, to create a fresh "version" of the protocol, which should be free of any possible corruption in the past. Formally, we require the following.

We are given a *user* at each node (we sometimes identify the user with its node). The user has input action $\mathit{Receive}(m, e)$ and output action $\mathit{Send}(m, e)$, where m is drawn from some message alphabet Σ, and e is an incident link.[15] The meaning of these actions is the natural one: m is received from e, and m is to be sent over e, respectively. In addition, the

[15]We distinguish between the actual link alphabet Σ' and the user message alphabet Σ. We assume that Σ' comprises of Σ and the distinct Reset protocol messages.

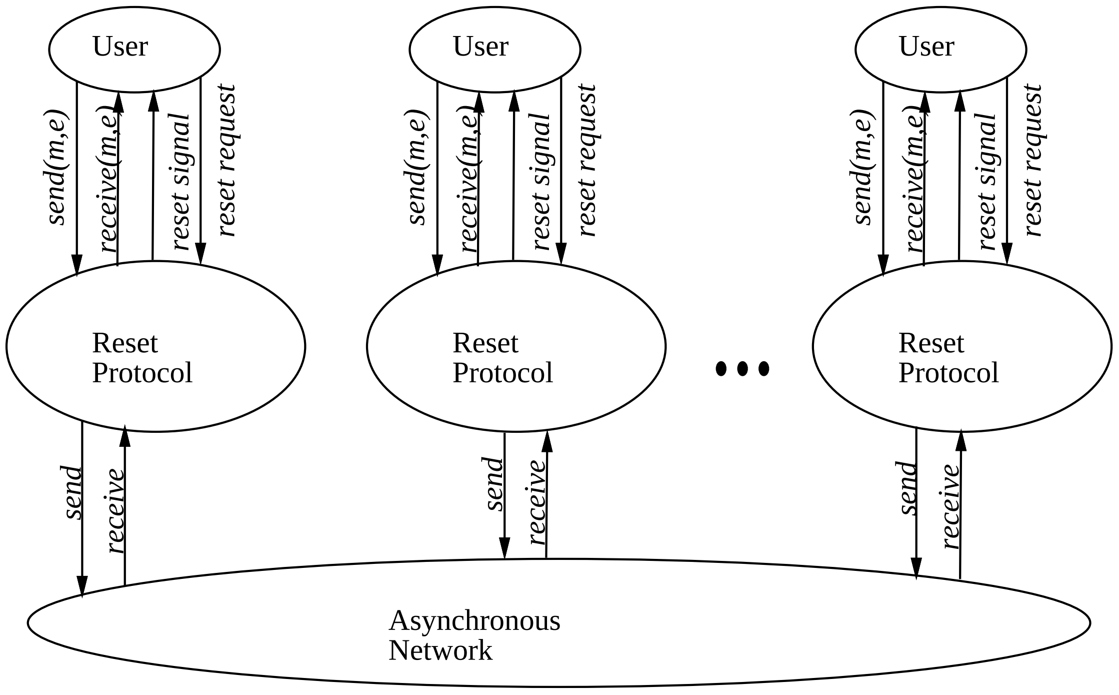


Figure 24.1: *Interface specification for reset service.*

user also has output action *reset request* and input action *reset signal.* The problem is to design a protocol ("reset service"), such that if one of the nodes makes a reset request and no node makes infinitely many requests, then

1. In finite time all the nodes in the connected component of a requesting node receive a reset signal.

2. No node receives infinitely many reset signals.

3. Let $e = (u, v)$ be any link in the final topology of the network. Then the sequence of $Send(m, e)$ input at u after the last reset signal at u is identical to the sequence of $Receive_{\overline{e}}(m)$ output at v after the last reset signal at v.

Informally, (1) and (2) guarantee that every node gets a *last* reset signal, and (3) stipulates that the last reset signal provides a consistent reference time-point for the nodes.

In order that the consistency condition (3) will be satisfiable, it is assumed that all Σ-messages from the user to the network and vice-versa are controlled by the reset protocol (see Figure 24.1).

24.3.2 Unbounded-registers Reset

The original problem requires us to run a new version of the user's protocol. Consider the following simple solution.

1. The reset protocol maintains a "version number".
2. All outgoing messages are stamped with the version number, and the numbers are stripped before delivery of received messages.
3. Whenever a request occurs, version counter is incremented, and signal is output immediately (all messages will be stamped with new value).
4. Whenever a higher value is seen, it is adopted, and signal is output.
5. Messages with low values are ignored.

This protocol works fine (for the nodes participating in the protocol). Stabilization time is $O(diam)$ which is clearly optimal. What makes this protocol uninteresting, as discussed above, is the need of unbounded version counter.

24.3.3 Example: Link Reset

Recall the local correction theorem. The basic mechanism there was applying some "local correction" action which reset the nodes to some pre-specified state, and reset the links to empty. This matches the above spec for reset. This link reset can be implemented fairly efficiently — the details are a bit messy, however. (It requires the use identifiers at nodes, and a flushing mechanism based on either a time bound on message delivery, or a bound on the maximal capacity of the link.)

Our goal, in the reset protocol, is to provide a network-wide reset. We could do it using similar flushing mechanisms (e.g., global timeout used in DECNET). However, global bounds are usually very bad, as they need to accommodate simultaneous worst cases at all the system. Thus we're interested in enhancing the self-stabilization properties of the links (as described by the local correction theorem) to all the network, without incurring the cost of, say, a global timeout.

24.3.4 Reset Protocol

Let us first describe how the reset protocol should have worked (if it was initialized properly). In the so-called *Ready* mode, the protocol relays, using buffers, Σ-messages between the network and the user. When a reset request is made at some node, its mode is changed to

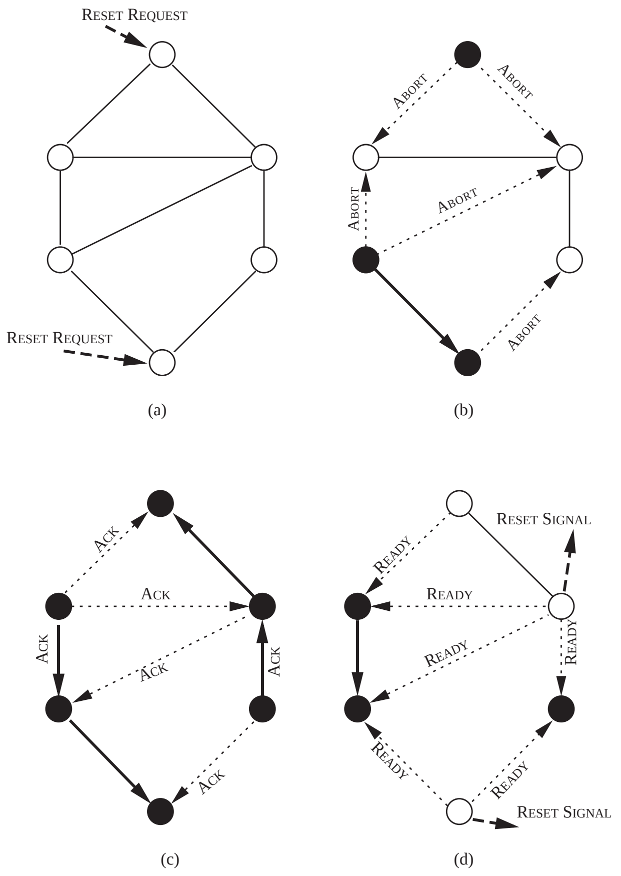


Figure 24.2: *An example of the run of the reset protocol. In (a), two nodes get reset request from the user. In (b), the nodes in Abort mode are colored black. The heavy arrows represent* `parent` *pointers, and the dashed arrows represent messages in transit. In (c), none of the nodes is in Ready mode. In (d), some of the nodes are in Ready mode, and some reset signals are output.*

Abort, it broadcasts ABORT messages to all its neighbors, sets the `Ack_Pend` bits to TRUE for all incident links, and waits until all the neighboring nodes send back ACK. If a node receives ABORT message while in *Ready* mode, it marks the link from which the message arrived as its `parent`, broadcasts ABORT, and waits for ACKs to be received from all its neighbors. If ABORT message is received by a node not in a *Ready* mode, ACK is sent back immediately. When a node receives ACK it sets the corresponding `Ack_Pend` bit to FALSE. The action of a node when it receives the last anticipated ACK depends on the value of its `parent`. If `parent` $\neq$ NIL, its mode is changed to *Converge*, and ACK is sent on `parent`.

If `parent` = NIL, the mode is changed to *Ready*, the node broadcasts READY messages, and outputs a reset signal. A node that gets a READY message on its `parent` link while in *Converge* mode, changes its mode to *Ready*, makes its `parent` NIL, outputs a reset signal, and broadcasts READY.

While the mode is not *Ready*, Σ-messages input by the user are discarded, and all Σ-messages from a link e are queued on `Buffer`$[e]$. When ABORT arrives on link e, `Buffer`$[e]$ is flushed. While the mode is *Ready*, Σ-messages from the user to the network are put on the output queue, and Σ-messages in `Buffer` are forwarded to the user. The size of `Buffer` depends on the assumptions we make on the user.

The actual code is given in Figures 24.3–24.6. A few remarks are in order.

1. In the self-stabilization model, we can deal with the assumption that the topology isn't fixed in advance. This is done by assuming that links may be up or down, and the nodes are constantly informed of the status of their incident links. As usual, we require correctness only after the system has stabilized. We model this by assuming that the maximal degree d is known (e.g., the number of ports), and that the state of the links is maintained updated in the `Edges` array.

2. In the self-stabilization model, we tend to have as little variables as possible: less things can get corrupted. See, for example, the definition for $mode(u)$ in Figure 24.3.

The first step in making the above protocol self-stabilizing is to make it locally checkable. A clear problem with the existing protocol is that it will deadlock if in the initial state some `parent` edges form a cycle. We mend this flaw by maintaining `distance` variable at each node, such that a node's `distance` is one greater than that of its `parent`. Specifically, `distance` is initialized to 0 upon reset request, and its accumulated value is appended to the ABORT messages. However, since all we care about is acyclicity, there is no need to update `distance` when a link goes down.

Next, we list all the link predicates that are necessary to ensure correct operation of the Reset protocol. It turns out that all the required link predicates are independently stable, and hence the stabilization time would be one time unit (each subsystem needs to be corrected at most once, independently)

Our last step is to design a local correction action for links (the f of the local correction theorem).

The main difficulty about designing a correcting strategy is making it local, i.e., to ensure that when we correct a link we do not violate predicates of incident link-subsystems. For the case of dynamic-topology network the solution is simple: emulate a link failure and recovery. Of course, care must be exerted when writing the code for these events (see code for local reset in Figure 24.3).

In the code, the detection and correction mechanism is written explicitly (Figure 24.5). For every link, we have the ID of the node at the other endpoint; the node with the higher

State of process u:

`Ack_Pend`:	array of d Boolean flags
`parent`:	pointer, ranging over $[1..d] \cup \{\text{NIL}\}$
`distance`:	ranges over $[0..N]$, where N is a bound on the number of processes
`Edges`:	set of operational incident links (maintained by the links protocol)
`Queue`:	array of d send buffers
`Buffer`:	array of d receive buffers
`IDs`:	array of d node identifiers
`Timers`:	array of d timers
`Check`:	array of d Boolean flags
`Snap_Mode`:	array of d Boolean flags

Shorthand for the code of process u:

$$mode(u) = \begin{cases} \textit{Abort}, & \text{if } \exists e \text{ such that } \texttt{Ack\_Pend}[e] = \text{TRUE} \\ \textit{Converge}, & \text{if } \texttt{parent} \neq \text{NIL} \textbf{ and } \forall e\,(\texttt{Ack\_Pend}[e] = \text{FALSE}) \\ \textit{Ready}, & \text{if } \texttt{parent} = \text{NIL} \textbf{ and } \forall e\,(\texttt{Ack\_Pend}[e] = \text{FALSE}) \end{cases}$$

```
$Propagate(e, dist)$  $\equiv$
    if $dist \neq 0$ then parent $\leftarrow e$ else parent $\leftarrow$ NIL
    distance $\leftarrow dist$
    for all edges $e' \in$ Edges do
      Ack_Pend[$e'$] $\leftarrow$ TRUE
      enqueue ABORT(distance $+ 1$) on Queue[$e'$]

$Local\_Reset(e)$  $\equiv$
    Buffer[$e$] $\leftarrow \emptyset$
    Queue[$e$] $\leftarrow \emptyset$
D1: if $\exists e'(e' \neq e$ and Ack_Pend[$e'$] $=$ TRUE$)$ or $(e \neq$ parent and parent $\neq$ NIL$)$ then
      if parent $= e$ then parent $\leftarrow$ NIL
      if Ack_Pend[$e$] $=$ TRUE then
        Ack_Pend[$e$] $\leftarrow$ FALSE
        if $mode(u) = Converge$ then enqueue ACK in Queue[parent]
D2: else if $mode(u) \neq Ready$
      Ack_Pend[$e$] $\leftarrow$ FALSE
      parent $\leftarrow$ NIL
      output reset signal
      for all $e' \in$ Edges do enqueue READY in Queue[$e'$]
```

Figure 24.3: *Self-stabilizing reset protocol — part I.*

Code for process u:

```
Whenever reset request  and $mode(u) = Ready$
$Q$:  $Propagate(\text{NIL}, 0)$

Whenever $\text{ABORT}(dist)$ on link $e$
$A1$:  if $mode(u) = Ready$ then
          $Propagate(e, dist)$
$A2$:  if $(dist = 0)$ or $(mode \neq Ready)$ enqueue ACK in Queue[$e$]

Whenever ACK on link $e$  and Ack_Pend[$e$] = TRUE
       Ack_Pend[$e$] $\leftarrow$ FALSE
$K1$:  if $mode(u) = Converge$ then
          enqueue ACK in Queue[parent]
$K2$:  else if $mode(u) = Ready$ then
          output reset signal
          for all $e' \in$ Edges do
             enqueue READY in Queue[$e'$]

Whenever READY on link $e$  and parent = $e$
$R$:   if $mode(u) = Converge$ then
          parent $\leftarrow$ NIL
          output reset signal
          for all edges $e' \in$ Edges do
             enqueue READY in Queue[$e'$]

Whenever $Send(m, e)$
       if $mode(u) = Ready$ then
          enqueue $m$ in Queue[$e$]

Whenever Queue[$e$] $\neq \emptyset$
       send $head$(Queue[$e$]) over $e$
       delete $head$(Queue[$e$])
```

Figure 24.4: *Self-stabilizing reset protocol — part II.*

Code for process u (cont.):

```
Whenever $m \in \Sigma$ received from link $e$
    **if** Check[$e$] = TRUE **then**
        **if** Snap_Mode[$e$] = SNAP **then**
            record $m$ as part of snapshot state on link $e$
            enqueue $m$ in Buffer[$e$]
    **else** enqueue $m$ in Buffer[$e$]

Whenever Buffer[$e$] $\neq \emptyset$ **and** $mode(u) = Ready$
    output $Receive(head($Buffer[$e$]$), e)$
    delete $head($Buffer[$e$]$)$

Whenever ABORT on $e$:
    Buffer[$e$] $\leftarrow \emptyset$

Whenever $Down$ on $e$:
    execute $Local\_Reset(e)$

Whenever Timers[$e$] expires for some link $e$:
    **if** MY_ID < IDs[$e$] **then**
        Check[$e$] $\leftarrow$ TRUE
        **if** Snap_Mode[$e$] = SNAP **then**
            record local state
        **else** execute $Local\_Reset(e)$
    send $Start\_Snap($Snap_Mode[$e$], MY_ID$)$ on $e$

Whenever $Start\_Snap(b, id)$ on link $e$
    IDs[$e$] $\leftarrow id$
    **if** $id <$ MY_ID **then**
        **if** $b$ = RESET **then**
            execute $Local\_Reset(e)$
        send $Response\_Snap$(local_state)

Whenever $Response\_Snap(S)$ on link $e$
    Check[$e$] $\leftarrow$ FALSE
    **if** Snap_Mode[$e$] = SNAP and any invariant does not hold **then**
        Snap_Mode[$e$] $\leftarrow$ RESET
    **else** execute $Local\_Reset(e)$
```

Figure 24.5: *Self-stabilizing reset protocol — part III.*

$\mathcal{A}$: $\texttt{Ack\_Pend}_u[e] = \text{TRUE}$ iff one of the following holds.

$\text{ABORT}(\texttt{distance}_u + 1) \in \texttt{Queue}_u[e]$

$\mathit{mode}(v) = \mathit{Abort}$ **and** $\texttt{parent}_v = \overline{e}$

$\text{ACK} \in \texttt{Queue}_v[\overline{e}]$

$\mathcal{B}$: At most one of the following hold.

$\text{ABORT}(\texttt{distance}_u + 1) \in \texttt{Queue}_u[e]$

$\mathit{mode}(v) = \mathit{Abort}_v$ **and** $\texttt{parent}_v = \overline{e}$

$\text{ACK} \in \texttt{Queue}_v[\overline{e}]$

$\mathcal{C}$: $\texttt{parent}_u = e$ implies that $\mathit{mode}(u) = \mathit{Converge}$ iff one of the following holds.

$\text{ACK} \in \texttt{Queue}_u[e]$

$\texttt{Ack\_Pend}_v[\overline{e}] = \text{FALSE}$ **and** $\mathit{mode}(v) \neq \mathit{Ready}$

$\text{READY} \in \texttt{Queue}_v[\overline{e}]$

$\mathcal{D}$: $\texttt{parent}_u = e$ implies that at most one of the following holds.

$\text{ACK} \in \texttt{Queue}_u[e]$

$\texttt{Ack\_Pend}_v[\overline{e}] = \text{FALSE}$ **and** $\mathit{mode}(v) \neq \mathit{Ready}$

$\text{READY} \in \texttt{Queue}_v[\overline{e}]$

$\mathcal{R}$: $\mathit{tail}(\texttt{Queue}_u[e]) \in \{\text{READY}\} \cup \Sigma$ implies

$\mathit{mode}(u) = \mathit{Ready}$

$\mathcal{P}$: $\texttt{parent}_u = e$ implies

one of the following holds.

$\texttt{distance}_u = \texttt{distance}_v + 1$

$\text{READY} \in \texttt{Queue}_v[\overline{e}]$

$\mathcal{E}$: $e \notin \texttt{Edges}_u$ implies all the following.

$\texttt{Ack\_Pend}_u[e] = \text{FALSE}$

$\texttt{parent}_u \neq e$

$\texttt{Queue}_u[e] = \emptyset$

$\texttt{Buffer}_u[e] = \emptyset$

$\mathcal{Q}$: $\texttt{Queue}[e]$ is a subsequence of the following.

$\langle \text{ABORT}, \text{ACK} \rangle$

$\langle \text{ACK}, \text{READY}, \text{ABORT} \rangle$

$\langle \text{ACK}, \text{READY}, \Sigma^* \rangle$

Figure 24.6: *Reset protocol part IV: link invariants for link $e = (u, v)$. The link (v, u) is denoted $\overline{e}$.*

ID is assumed to be "responsible" for the correctness of the subsystem. This is done as follows. Whenever the timer dedicated to that link expires, the node checks whether it is "in charge" of that link; if so, it sends out a snapshot message that returns with the state of the other node. Based on this state, its own state, and the messages received, it can find out whether any of the invariants (Figure 24.6) is violated. If this is the case, local reset is executed in a coordinated way that resembles the way snapshots of the link are taken.

The invariants of Figure 24.6 express simple properties that turn out to be sufficient to ensure correct operation of the protocol. Invariants $\mathcal{A}$ and $\mathcal{B}$ capture the desired behavior af abort messages; $\mathcal{C}$ and $\mathcal{D}$ deal with the converge messages analogously The ready message is simpler (since we don't expect anything from a node in ready mode), and is described by invariant $\mathcal{R}$. Invariant $\mathcal{P}$ is crucial to ensure stabilization: it guarantees that no cycles of the parent pointers can survive checking and correction. Invariant $\mathcal{E}$ describes what is expected from a non-existing edge. (The goal of local reset action brings the subsystem to that state.) The last invariant, $\mathcal{Q}$, is a technicality: we need to consider the messages in the send queue also.

24.3.5 Analysis

We'll sketch the outlines of the proofs, just to show the ideas. Details are a bit messy.

We start with the statement of stabilization.

Theorem 1 *Suppose that the links are bounded-delay links with delay time D, and suppose that invariants are verified by the manager at least every P time units. Then any execution of the Reset protocol, regardless of the initial state, in all states from time $O(D+P)$ onwards, all the invariants hold for all the links.*

This theorem is proven by showing that the invariants are stable. The argument is induction on the actions: assuming that a given state is legal with respect to a given link subsystem, we show that applying any action will not break any invariant in this subsystem. Thus the proofs are basically a straightforward (somewhat tedious) case-analysis. Using the local correction theorem, we can deduce that the protocol is self-stabilizing.

It is more interesting to analyze the executions of the algorithm. We need a few definitions.

Definition 1 *Let $(s_k, a_k, s_{k+1}, \ldots)$ be an execution fragment. An interval $[i, j]$ is a* ready interval *at node u if*

1. *$mode(u) = Ready$ in all states s_l for all $i \leq l \leq j$.*
2. *If $i > k$ then $mode(u) \neq Ready$ in s_{i-1}, and if $j < \infty$ then $mode(u) \neq Ready$ in s_{j+1}.*

Abort interval *and* converge interval *are defined analogously.*

For these mode intervals, we have the following lemma.

Lemma 2 *Let $(s_n, a_n, s_{n+1} \ldots)$ be an execution fragment and let u be a node. Then at u,*

1. *A ready interval may be followed only by an abort interval.*

2. *A converge interval may be followed only by a ready interval.*

The proof is by straightforward inspection of the actions. Next we define the concept of parent graph.

Definition 2 *Let s be a state. The* parent graph *$G(s)$ is defined by the* `parent` *and the* `Queue` *variables as follows. The nodes of $G(s)$ are the nodes of the network, and $e = (u, v)$ is an edge in $G(s)$ iff* $\texttt{parent}_u = e$ *and* $\textsc{Ready} \notin \texttt{Queue}_v[\overline{e}]$ *in s.*

From the distances invariant, we know that the parent graph is actually a forest in legal states. We also define a special kind of abort intervals. An abort interval $[i, j]$ at node u is a *root interval* if for some $i \leq k \leq j$, $\texttt{parent}_u = \textsc{nil}$ in s_k.

The next step of the proof is to show that for any abort interval there exists a root interval that contains it. This is done by induction on the depth of the node in the parent graph. Using this fact, we bound the duration of abort intervals.

Lemma 3 *Let s be a legal state, let u be a node, and suppose that no topological changes occur after s. If $mode(u) = Abort$ in s and $depth(u) = k$ in $G(s)$, then there exists a state s' in the following $2(n - k)$ time units such that $mode(u) \neq Abort$ in s'.*

This lemma is proven by showing that no `Ack_Pend` bit is continuously $\textsc{true}$ for more than $2(n - k)$ time units. This implies the lemma, since no `Ack_Pend` bit is set to $\textsc{true}$ while the node is in a non-ready mode.

Similarly, we can prove by induction on *depth*(u) the following bound on the duration of the converge intervals.

Lemma 4 *Let s be a legal state at time t, let u be a node, and suppose that no topological changes occur after time t. If $mode(u) = Abort$ in s, and $depth(u) = k$ in $G(s)$, then by time $t + 2n + depth(u)$, an action occurs, in which $mode(u)$ is changed to Ready, and* $\textsc{Ready}$ *messages are sent by u to all its neighbors.*

It is easy to derive the following conclusion.

Theorem 5 *Suppose that the links (including the link invariants) stabilize in C time units. If the number of reset requests and topological changes is finite, then in $C + 3n$ time units after the last reset request/topological change, $mode(u) = Ready$ at all nodes u.*

Proof: Consider the root intervals. Note that by the code, each such root interval can be associated with an input request made at this node, or with a topological change. Since each request can be associated with at most one root interval, and since each topological change can be associated with at most two root intervals, the number of root intervals is finite. Furthermore, by Lemma 4, $2n$ time units after the last reset request all root intervals terminate, and READY will be propagated. Since any non-root abort interval is contained in some root interval, after $2n$ time *all* abort intervals have terminated, and hence, $3n$ time units after the last reset request, the mode of all the nodes is *Ready*. ∎

The following theorem states the liveness property of the Reset protocol.

Theorem 6 *Suppose that the execution of the Reset protocol begins at time 0 at arbitrary state, and suppose that the links (including the link invariants) stabilize in C time units. If the last reset request/topological change occurs at time $t > C + 3n$, then a RESET signal is output at all the nodes by time $t + 3n$.*

Proof Sketch: The key property we need is that the time between successive ready intervals is at most $3n$ time units.

First, notice that it suffices to show that for any node v, there is a time in the interval $[C, t + n]$ in which $mode(v) \neq Ready$.

Suppose now, for contradiction, that at time $t > C + 3n$ occurs a reset request at some node w, and that there exists a node u whose mode is *Ready* in times $[C, t + n]$. Let v be any node in the network. We show, by induction on the distance i between u and v, that $mode(i) = Ready$ in the time interval $[C + 3n, t + n - i]$. This will complete the proof, since the hypothesis implies that $mode(w) \neq Ready$ at time t.

The base case follows from the assumption that $mode(u) = Ready$ in the time interval $[C, t+n]$. The inductive step follows from the fact that if for some node v', in time $t' > C+3n$ we have $mode(v') \neq Ready$, then there occurs an action in the time interval $[t' - 3n, t']$ in which the mode of v' changed from *Ready* to *Abort*, and ABORT messages were sent to all the neighbors of v'. Hence, for each neighbor v of v', there is a state in the time interval $[t' - 3n, t' - 3n + 1]$ in which $mode(v) \neq Ready$. ∎

Next, we prove the consistency of the Reset protocol. This property follows from the simple `Buffer` mechanism. There is a buffer dedicated to each link, such that all Σ-messages arriving from link e are stored in `Buffer`$[e]$. `Buffer`$[e]$ is flushed whenever a ABORT message arrives from e.

Theorem 7 *Suppose no topological changes occurs after time C, and that the last reset signals occur at two adjacent nodes u and v after time $C+3n$. Then the sequence Send(m, e) input by the user at u following the last reset signal at u, is identical to the sequence of Receive$(m, \overline{e})$ output to the user at v after the last reset signal at v.*

Proof: Consider the last non-ready intervals in u and v. Suppose that the last states in which *mode*$(u) \neq$ *Ready* and *mode*$(v) \neq$ *Ready* are s_{j_u} and s_{j_v}, respectively, and let s_{i_u} and s_{i_v} be the first states in the last non-ready intervals, respectively (i.e., for all $i_u \leq k \leq j_u$, in s_k *mode*$(u) \neq$ *Ready*, and in s_{i_u-1}, *mode*$(u) =$ *Ready*). Note that since the last reset signal is output after time $C + 3n$, it follows that the states s_{i_u-1} and s_{i_u-1} indeed exist. Note further that the change of mode to *Abort* is accompanied by broadcasting ABORT messages (by *Propagate*).

Consider *Send*(m, e) input at u after s_{j_u}. We argue that m is delivered at v after s_{i_v}: suppose not. Then the mode of v is changed to *Abort* after s_{j_u}, and v broadcasts ABORT to u. Hence there must be a state s_k, $k > j_u$, such that *mode*$(u) \neq$ *Ready* in s_k, contradicting the assumption that $_{j_u}$ is the last state in the last non-ready interval. Recalling that the manager buffers Σ-messages until the mode is *Ready*, we conclude that the corresponding *Receive*$(m, \overline{e})$ is output to the user at v after s_{j_v}.

Consider *Send*(m, e) input at u before s_{j_u}. Since the manager discards all Σ-messages input while in non-ready mode, we need only to consider *Send*(m, e) input at u before s_{i_u}. Again, we argue that *Receive*$(m, \overline{e})$ is not output at v after s_{j_v}. This follows from the fact that there must be ABORT message sent to u from v after m, that implies that the mode of u is not *Ready* after s_{j_u}, a contradiction. ▮

24.3.6 Comments

The reset protocol is a powerful tool, but sometimes it's an overkill: it doesn't seem reasonable to reset the whole system whenever a minor but frequent fault occur (e.g., a new node joins, a link fails). It seems that one of the best applications of reset is when it is combined with unbounded-register protocols: the effect of the reset signal in this case can usually be defined easily (e.g., set counters to 0).

Finally, we note that the time complexity actually is bounded by the length of the longest simple path in the network (which is a more refined measure than just the number of nodes). If the network is a tree, for instance, the protocol works in diameter time, which is clearly optimal. The space complexity is logarithmic in the bound N. As typical for the local correction method, the communication bandwidth required for stabilization is proportional to the space, which is in our case logarithmic.

24.4 Application: Network Synchronization

Recall the network synchronization problem. The basic property there was that if a message is sent in (local) round i, then no messages of rounds $i-1$ or less will ever be received in that node. We abstract the synchronizer as a service module whose task is to provide the user with pulse numbers at all times. If we associate with each node v its round/pulse number $P(v)$, it can be easily seen that a state is legal only if no two adjacent nodes pulses differ by more than 1, i.e., for all nodes v

$$u \in \mathcal{N}(v) \Rightarrow |P(u) - P(v)| \leq 1 \tag{24.9}$$

($\mathcal{N}(v)$ denotes the set of nodes adjacent to v.) Call the states satisfying Eq. (24.9) *legal*. Of course, we also want to keep progressing. This can be stated as asserting that for each configuration, there is some pulse number K, such that all nodes get all pulses $K, K+1, \ldots$

If the system initialized in a legal state, we can use the following rule.

$$P(v) \leftarrow \min_{u \in \mathcal{N}(v)} \{P(u) + 1\} \tag{24.10}$$

The idea is that whenever the pulse number is changed, the node sends out all the messages of previous rounds which haven't been sent yet.

The rule above is *stable*, i.e., if the configuration is legal, then applying the rule arbitrarily can yield only legal configuration. Notice however, that if the state is not legal, then applying the rule may cause pulse numbers to drop. This is a reason to be worried, since the regular course of the algorithm requires pulse numbers only to grow. And indeed, the rule is not self-stabilizing.

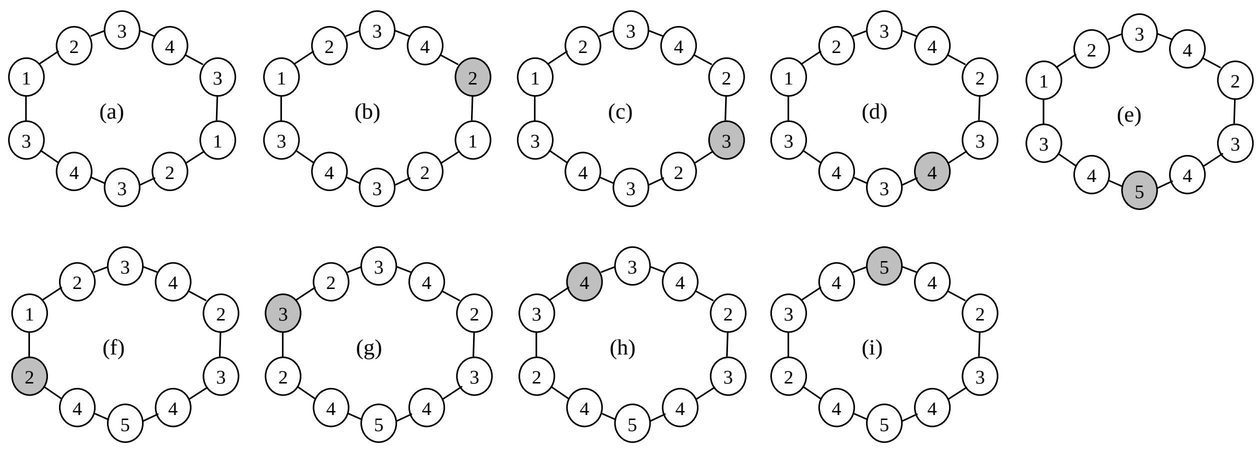


Figure 24.7: *An execution using rule 24.10. The node that moved in each step in marked.*

One idea that can pop into mind to try to repair the above flaw is to never let pulse numbers go down. Formally, the rule is the following.

$$P(v) \leftarrow \max \left\{ P(v), \min_{u \in \mathcal{N}(v)} \{P(u) + 1\} \right\} \tag{24.11}$$

This rule can be proven to be self-stabilizing. However, it suffers from a serious drawback regarding its stabilization time. Consider the configuration depicted below.

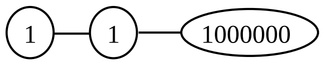


Figure 24.8: *A pulse assignment for rule 24.11.*

A quick thought should suffice to convince the reader that the stabilization time here is in the order of 1000000 time units, which seems to be unsatisfactory for such a small network.

The next idea is to have a combination of rules: if the neighborhood is legal, then the problem specification requires the min+1 rule is used. But if the neighborhood is not legal, another rule can be used. The first idea we consider is the following.

$$P(v) \leftarrow \begin{cases} \min_{u \in \mathcal{N}(v)} \{P(u)+1\} \ , & \text{if } \forall u \in \mathcal{N}(v) \ : \ |P(v)-P(u)| \leq 1 \\ \max_{u \in \mathcal{N}(v)} \{P(u)\} \ , & \text{otherwise} \end{cases} \tag{24.12}$$

It is fairly straightforward to show that if an atomic action consists of reading one's neighbors and setting its own value (in particular, no neighbor changes its value in the meanwhile), then the max rule above indeed converges to a legal configuration. Unfortunately, this model, traditionally called *central demon* model is not adequate for a truly distributed system, which is based on loosely coordinated asynchronous processes. And as one might suspect, the max rule does not work in a truly distributed system.

So here is a solution.

$$P(v) \leftarrow \begin{cases} \min_{u \in \mathcal{N}(v)} \{P(u)+1\} \ , & \text{if } \forall u \in \mathcal{N}(v) \ : \ |P(u)-P(v)| \leq 1 \\ \max_{u \in \mathcal{N}(v)} \{P(u)-1\} \ , & \text{if } \exists u \in \mathcal{N}(v) \ : \ P(u)-P(v) > 1 \\ P(v), & \text{otherwise} \end{cases} \tag{24.13}$$

In words, the rule is to apply "minimum plus one" when the neighborhood seems to be in a legal configuration, and if the neighborhood seems to be illegal, to apply "maximum minus one".

The intuition behind the modification is that if nodes change their pulse numbers to be the *maximum* of their neighbors, then "race condition" might evolve, where nodes with high pulses can "run away" from nodes with low pulses. If the correction action takes the pulse number to be one less than the maximum, then the high nodes are "locked", in the sense that they cannot increment their pulse counters until all their neighborhood have reached their pulse number. This "locking" spreads automatically in all the "infected" area of the network.

We shall now prove that this rule indeed stabilizes in time proportional to the diameter of the network. For this we shall need a few definitions.

Definition 3 *Let v be a node in the graph. The* potential *of v is denoted by $\phi(v)$ and is defined by*

$$\phi(v) = \max_{u \in V} \{P(u) - P(v) - dist(u, v)\} \ .$$

Intuitively, $\phi(v)$ is a measure of how much is v *not* synchronized, or alternatively the size of the largest skew in the synchronization of v, corrected by the distance. Pictorially, one can think that every node u is a point on a plane where the x-coordinate represents the distance of u from v, and the y coordinate represents the pulse numbers (see Figure 24.9 for an example). In this representation, v is the only node on the y-axis, and $\phi(v)$ is the maximal vertical distance of any point (i.e., node) above the 45-degree line going through $(0, P(v))$.

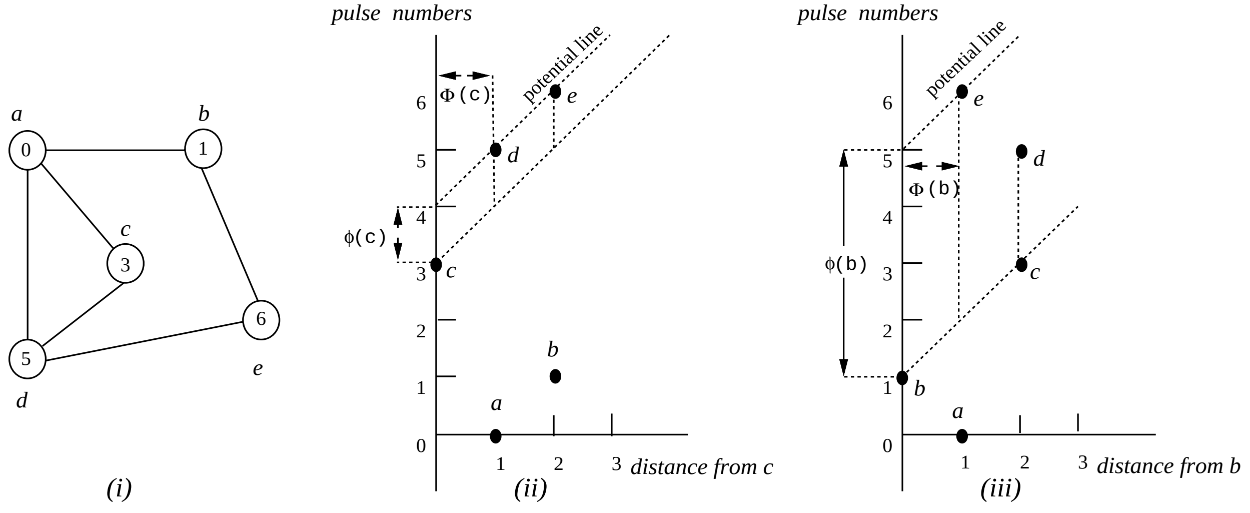


Figure 24.9: *On the left is an example of a graph with pulse assignment (i). Geometrical representations of this configuration are shown in (ii) and (iii). The plane corresponding to node c is in the middle (ii), and the plane corresponding to node b is on the right (iii). As can be readily seen, $\phi(c) = 1$, and $\phi(b) = 4$. Also, $\Phi(c) = 1$, and $\Phi(b) = 1$ (see Definition 4).*

Let us start with some properties of ϕ that follow immediately from the definition.

Lemma 8 *For all nodes $v \in V$, $\phi(v) \geq 0$.*

Lemma 9 *A configuration of the system is legal if and only if for all $v \in V$, $\phi(v) = 0$.*

We now show the key property of the new rule, namely that the potential of the nodes never increases under this rule.

Lemma 10 *Let P be any pulse assignment, and suppose that node u changes its pulse number by applying the rule. Denote the new pulse number of u by $P'(u)$, and the potential of the nodes in the new configuration by ϕ'. Then for all nodes $v \in V$, $\phi'(v) \leq \phi(v)$.*

Proof: The first easy case to consider is the potential of u itself. Since $P'(u) > P(u)$, we have

$$\begin{aligned} \phi'(u) &= \max_{w \in V} \{P(w) - P'(u) - dist(w,u)\} \\ &\leq \max_{w \in V} \{P(w) - P(u) - dist(w,u)\} \qquad (24.14) \\ &= \phi(u) \ . \end{aligned}$$

(Note, for later reference, that the inequality in (24.14) is strict if $\phi(u) > 0$.) Now consider $v \neq u$. The only value that was changed in the set

$$\{P(w) - P(v) - dist(w,v) \ | \ w \in V\}$$

is $P(u) - P(v) - dist(u,v)$. There are two cases to examine. If u changed its pulse by applying the "min plus one" part of the rule, then there must be a node w which is a neighbor of u, and is closer to v, i.e., $dist(u,v) = dist(w,v) + 1$. Also, since "min plus one" was applied, we have $P'(u) \leq P(w) + 1$. Now,

$$\begin{aligned} P'(u) - P(v) - dist(u,v) &\leq (P(w)+1) - P(v) - (dist(w,v)+1) \\ &= P(w) - P(v) - dist(w,v) \end{aligned}$$

and hence the $\phi(v)$ does not increase in this case. The second case to consider is when u has changed its value by applying the "max minus one" part of the rule. The reasoning in this case is similar: let w be a neighbor of u with $P(w) = P'(u) + 1$. Clearly, $dist(w,v) \leq dist(u,v) + 1$. This implies that

$$\begin{aligned} P'(u) - P(v) - dist(u,v) &\leq (P(w)-1) - P(v) - (dist(w,v)-1) \\ &= P(w) - P(v) - dist(w,v) \end{aligned}$$

and we are done. ▮

As noted above, the inequality in (24.14) is strict if $\phi(u) > 0$. In other words, each time a node with positive potential changes its pulse number, its potential decreases. This fact, when combined with Lemmas 8 and 9, immediately implies eventual stabilization. However, using this argument leads to a proof that the stabilization time is bounded by the total potential of the configuration, which in turn depends on the initial pulse assignment. We need a stronger argument in order to prove a bound on the stabilization time that depends only on the topology. Toward this end, we define the notion of "wavefront".

Definition 4 *Let v be any node in the graph with potential $\phi(v)$. The* wavefront *of v, denoted $\Phi(v)$, is defined by*

$$\Phi(v) = \min_{u \in V} \{ dist(u,v) \mid P(u) - P(v) - dist(u,v) = \phi(v) \} .$$

In the graphical representation, the wavefront of a node is simply the distance to the closest node of on the "potential line" (see Figure 24.9 for an example). Intuitively, one can think of $\Phi(v)$ as the distance to the "closest largest trouble" of v. The importance of the wavefront becomes apparent in Lemma 12 below, but let us first state an immediate property it has.

Lemma 11 *Let $v \in V$. Then $\Phi(v) = 0$ if and only if $\phi(v) = 0$.*

Lemma 12 *Let v be any node with $\Phi(v) > 0$, and let $\Phi'(v)$ be the wavefront of v after one time unit. Then $\Phi'(v) \le \Phi(v) - 1$.*

Proof: Suppose $\Phi(v) = f > 0$ at some state. Let u be any node such that $P(u) - P(v) - dist(u,v) = \phi(v)$, and $dist(u,v) = f$. Consider a neighbor w of u which is closer to v, i.e., $dist(w,v) = f - 1$ (it may be the case the $w = v$). From the definition of $\Phi(v)$, it follows that $P(w) < P(u) - 1$. Now consider the next time in which w applies Rule 24.13. If at that time $\Phi(v) < f$, we are done. Otherwise, w must assign $P(w) \leftarrow P(u) - 1$. No greater value can be assigned, or otherwise Lemma 10 would be violated. At this time, $P(w) - P(v) - dist(w,v) = \phi(v)$ also, and hence $\Phi(v) \le f - 1$. ▮

Corollary 13 *Let v be any node. Then after $\Phi(v)$ time units, $\phi(v) = 0$.*

We can now prove the main theorem.

Theorem 14 *Let $G = (V,E)$ be a graph with diameter d, and let $P : V \to \mathbf{N}$ be a pulse assignment. Applying Rule 24.13 above results in a legal configuration in d time units.*

Proof: By Lemma 9, it suffices to show that after d time units, $\phi(v) = 0$ for all $v \in V$. From Corollary 13 above, we actually know that a slightly stronger fact holds: for all node $v \in V$, after $\Phi(v)$ time units, $\phi(v) = 0$. The theorem follows from the facts that for all $v \in V$, $\Phi(v) \le d$, and by the fact that $\phi(v)$ never increases, by Lemma 10. ▮

24.4.1 Implementation with Bounded Pulse Numbers

The optimal rule from above works only when the pulse numbers may grow unboundedly. However, we can use the reset protocol to make is work with bounded-size registers. Suppose that the registers dedicated to the pulse numbers may hold the values $0 \ldots B$, for some bound B. Then the protocol would work by proceeding according to the unbounded rule (Eq. 24.13), and whenever a value should be incremented above B, reset is invoked. Notice that we must require that B is sufficiently large, to enable propper operation of the protocol between resets: if B is less than the diameter of the network, for instance, then there is a possible scenario in which the far nodes never get to participate in the protocol.

Lecturer: Nancy Lynch

Lecture 25

This is the last lecture of the course: we didn't have time to cover a lot of material we intended to. Some of the main topics missing are fault-tolerant network algorithms and timing-based computing. We'll have several additional lectures in January to cover some of this, for those who are interested. Today, we'll do one major result in fault-tolerant network algorithms, namely the impossibility of fault-tolerant consensus in asynchronous networks. We'll also do two other results on ways to get around this limitation.

25.1 Fischer-Lynch-Paterson Impossibility Result

Recall the consensus problem from the shared memory work. The interface is defined by the input actions $init_i(v)$, and output actions $decide_i(v)$, where i is a process and v is a value. Here we shall consider the same *init-decide* interface, but the implementation is now a distributed network algorithm. The message delivery is FIFO, and completely reliable. In fact, we will assume that the nodes have reliable broadcast actions, so that when a node sends a message, it is sent simultaneously to all the others, and because of the reliability of message transmission, it will eventually get there. We consider solving this problem in the face of processor faults; since we are giving an impossibility result, we consider only a very simple kind of faulty behavior — at most one stopping failure. (Using such a weak kind of fault makes the impossibility result stronger.)

The impossibility of solving consensus in such a friendly environment is rather surprising. The story is that Fischer, Lynch and Paterson worked on this problem for some while, trying to get a positive result (i.e., an algorithm). Eventually, they realized that there was a very fundamental limitation getting in the way. They discovered an inherent problem with reaching agreement in an asynchronous system, in the presence of *any* faulty process.

The Model. As in any impossibility result, it is very important to define precisely the assumptions made about the underlying model of computation. Here we assume that the network graph is complete, and that the message system consists of fair FIFO queues. Each process (node) i is an I/O automaton. There are inputs and outputs $init_i(v)$ and $decide_i(v)$, respectively, where i is a node and v is a value. There are also outputs $broadcast_i(v)$ and inputs $receive_{j,i}(v)$, where i and j are nodes and v is a value. The effect of a $broadcast_i(v)$ action is to put (atomically) a v message in the channels leading to all the nodes (other than i) in the system. With the *broadcast* action we actually don't need individual *send* actions.

We assume without loss of generality that each process has only one fairness class (i.e., the processes are *sequential*), and that the processes are deterministic, in the sense that (1) in each process state, there is at most one enabled locally controlled action, and (2) for each state s and action π, there is at most one new state s' such that (s, π, s') is a transition of the protocol. In fact, we can assume furthermore (without loss of generality) that in each state there is *exactly* one enabled locally controlled action (since we can always add in dummy steps).

Correctness Requirement. For the consensus problem we require that in any execution such that exactly one $init_i$ occurs for each node i, we have the following conditions satisfied.

Agreement: There are no two different decision values.

Validity: If all the initial values are the same value v, then v is the only possible decision value.

In addition, we need some termination conditions. First, we require that if there is exactly one $init_i$ for each i and if the entire system (processes and queues) executes fairly, then all processes should eventually decide. This condition is easy to satisfy (e.g., just exchange values and take majority). But we require more. Define 1-*fair* executions to be those in which all but at most 1 process execute fairly, and all channels execute fairly. We assume that a process that fails still does its input steps, but it can stop performing locally-controlled actions even if such actions remain enabled. We require that in all 1-fair executions, all processes that don't stop eventually decide.

It seems as though it ought to be possible to solve the problem for 1-*fair* executions also, especially in view of the powerful broadcast primitive (you should try to design an algorithm!). Our main result in this section, however, is the following theorem.

Theorem 1 *There is no 1-resilient consensus protocol.*

Henceforth, we shall restrict our attention to the binary problem, where the input values are only 0 and 1 — this is sufficient to prove impossibility.

Terminology. Define A to be a 0-*resilient consensus protocol* (0-RCP) provided that it satisfies agreement, validity, and termination in all its fair executions. Define A to be a

1-*resilient consensus protocol* (1-RCP) provided it is a 0-resilient consensus protocol and if, in addition, all non-stopped processes eventually decide in 1-fair executions.

We call a finite execution α 0-*valent* if 0 is the only decision value in all extensions of α. (Note: the decision can occur anywhere in the extension, including the initial segment α.) Similarly, we define 1-*valent* finite executions. We say that α is *univalent* if it is either 0-valent or 1-valent, and we say that α is *bivalent* if it is not univalent (i.e., α is bivalent if there is one extension of α in which someone decides 0, and another extension in which someone decides 1).

These "valency" concepts capture the idea that a decision may be determined, although the actual decision action might not yet have occurred. As in the shared memory case, we will restrict attention in the proof to *input-first executions*, i.e., executions in which all inputs arrive at the beginning, in round robin order of *init* events, for process $1, 2, \ldots, n$. Define an *initial execution* to be an input-first execution of length exactly n; that is, an initial execution just provides the initial values to all the processes.

We can now start proving the impossibility result. We begin with a statement about the initial executions.

Lemma 2 *In any* 1*-RCP there is a bivalent initial execution.*

Proof: The idea is the same as in the shared memory setting. Assume that the lemma is false, i.e., that all initial executions are univalent. By validity, the all-0 input must lead to decision of 0, and all 1's to decision of 1. By assumption, any vector of 0's and 1's leads to a unique decision. Hence there exist two input vectors with Hamming distance 1, with corresponding initial executions such that one is 0-valent, and the other is 1-valent; let α_0 and α_1 be these two respective initial executions. Let i be the index of the unique process in whose initial value they differ.

The idea is that the rest of the system can't tell which of the vectors was input if i never does anything. More formally, consider a 1-fair execution α_0' that extends initial execution α_0, in which i fails right at the beginning, i.e., it takes no locally-controlled steps. In α_0' some process j eventually decides 0, by the assumptions that α_0 is 0-valent and that the protocol is 1-RCP.

But then we claim that there is another execution α_1' that extends α_1, that also leads to $decide_j(0)$. This is because the only difference between the states at the end of α_0 and α_1 is in the state of process i; now allow the same processes to take the same steps in α_1' as in α_0', the same messages to get delivered, etc. (see Figure 25.1). The only difference in how α_0' and α_1' will unfold is in the state changes caused by the receipt of the same message by i in the two executions – those states can be different, but since i never performs any locally controlled action in either, the rest of the system cannot tell the difference.

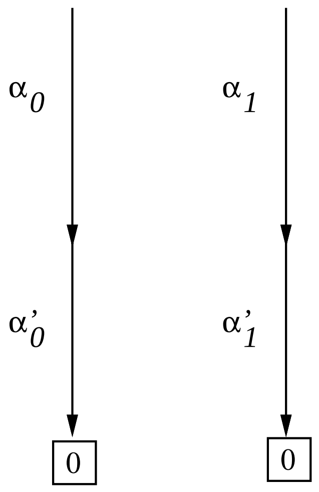


Figure 25.1: In both extensions for α_0 and α_1 the same processes take the same actions, and therefore α_1 cannot be 1-valent.

Hence, process j must decide 0 in α'_1 too, contradicting the 1-valence of α_1. ▮

Examples. The majority protocol is a 0-RCP; it has no bivalent initial execution, since an initial execution can only lead to a decision of v in case the value v is the initial value of a majority of the processes.

Another example of an 0-RCP is a leader-based protocol, where everyone sends their values to process 1, and 1 decides on the first value received and informs everyone else about the decision. This does have bivalent initial executions – anything in which there are two processes other than p_1 having different initial values.

We now proceed with the definition of a *decider* (which is defined somewhat differently from the way it was defined for the shared memory setting).

Definition 1 *A* decider *for an algorithm A consists of an input-first execution α of A and a process index i such that the following conditions hold (see Figure 25.2).*

1. *α is bivalent.*
2. *There exists a 0-valent extension α_0 of α such that the suffix after α consists of steps of process i only. (These can be input, output or internal steps.)*
3. *There exists a 1-valent extension α_1 of α such that the suffix after α consists of steps of process i only.*

Note that the flexibility in obtaining two different executions here arises because of the possible different orders of interleaving between locally controlled steps and message-receipt

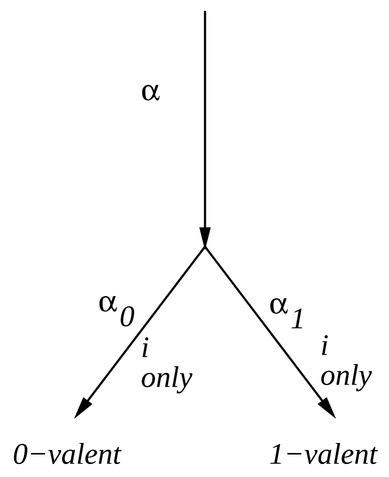


Figure 25.2: schematic diagram of a decider (α, i).

steps on the various channels. Also, messages could arrive in different orders in two different executions.

Let us now state a lemma about the existence of a decider.

Lemma 3 *Let A be any* 0*-RCP with a bivalent initial execution. Then there exists a decider for A.*

We shall prove this lemma later. Let us first complete the impossibility proof, given Lemma 3.

Proof: *(of Theorem 1).* Suppose A is a 1-RCP. Since A is also a 0-RCP, and since it has a bivalent initial prefix, we can obtain, by Lemma 3, a decider (α, i). We now argue a contradiction in a way that is similar to the argument for the initial case.

Consider a 1-fair extension α_2 of α in which i takes no locally-controlled steps (i.e., i fails right after α). Eventually in α_2, some process j must decide; suppose without loss of generality that the decision value is 0. Also, we can suppose without loss of generality that there are no message deliveries to i in the suffix of α_2 after α (if there are any, then omitting them still leads to the same decision).

Now we claim that the there is another execution α_2' that extends α_1, and that also leads to $decide_j(0)$ (see Figure 25.3).

This is because the only differences between the states at the end of α and α_1 are in (a) the state of process i, (b) the *last* elements in the channels leaving process i (there can be some extras after α_1), and (c) the *first* elements in the channels entering process i can be different (there can be some messages missing after α_1). Then we allow the same processes to take the same steps in the suffix after α_1 as in α_2, the same messages to get delivered, etc. (Nothing that i has done will interfere with this; note that we never deliver any of the extra

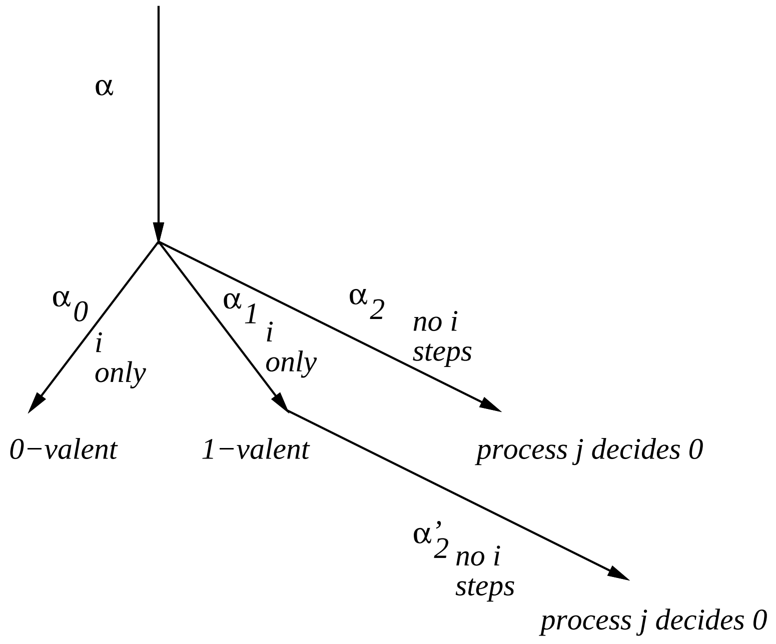


Figure 25.3: Contradiction derived from the existence of a decider α.

messages). As before, the only difference in what happens is in the state changes caused by the receipt of the same message by i in the two executions: those states can be different, but since i never performs any locally controlled action in either, the rest of the system cannot tell the difference.

Hence, process j decides 0 by the end of α'_2, contradicting the 1-valence of α_1. ▮

It remains now to prove Lemma 3.

Proof: *(of Lemma 3).* Suppose there is no decider. Then starting from a bivalent initial execution, we construct a fair execution (no failures at all) in which successive prefixes are all bivalent, thus violating the termination requirement.

We work in a round-robin fashion: at each phase, we either let one specified process take a locally controlled step or else we deliver the first message in some channel. Visiting all processes and all channels in the round robin order yields a fair execution (if a channel is empty, we can bypass it in the order).

We have now reduced the proof to doing the following. We need to consider one step at a time, where each step starts with a particular bivalent input-first execution and we need to either

1. let a particular process i take a turn, or

2. deliver the oldest message in some channel.

We must do this while keeping the result bivalent. We proceed by considering the possible cases of the step at hand.

Case 1: Deliver message m from j to i, where m is the first message in the channel from i to j in the state after α. Consider the tree of all possible finite extensions of α in which m is delivered to i just at the end (in the interim, arbitrary other steps can occur). If any of these "leaves" is bivalent, we are done with this stage. So assume all are univalent. As in Loui-AbuAmara, assume without loss of generality that delivering the message immediately after α leads to a 0-valent state. Consider a finite extension of α that contains a decision of 1; this is possible because α is bivalent (see Figure 25.4).

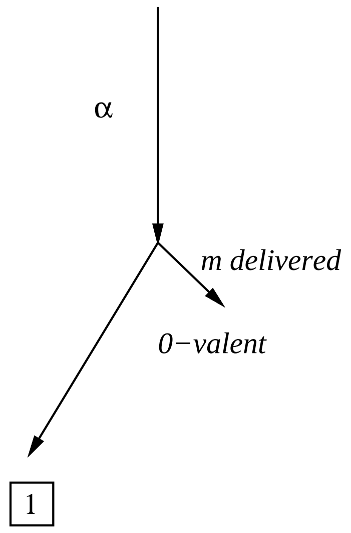


Figure 25.4: the delivery of m leads to a 0-valent state.

We now use this to get a configuration involving one step, such that if m is delivered just before the step the result is a 0-valent execution, whereas if it is delivered just after the step, the result is a 1-valent execution (see Figure 25.5). We show this must happen by considering cases as follows.

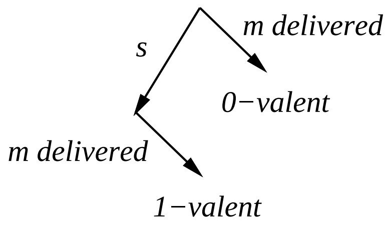
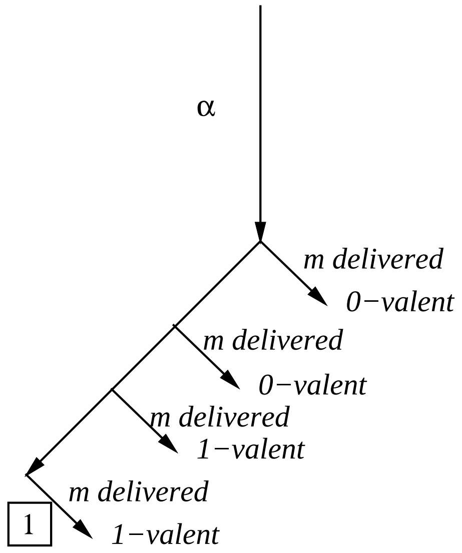


Figure 25.5: the delivery of m before step s leads to a 0-valent state, and the delivery of m after step s leads to a 1-valent state.

If the extension leading to the decision of 1 does not contain a delivery of m, then delivering m right at the end leads to a 1-valent state; then since attaching it anywhere gives univalence, there must be 2 consecutive positions along the chain where one gives 0-valence and one gives 1-valence (see Figure 25.6).



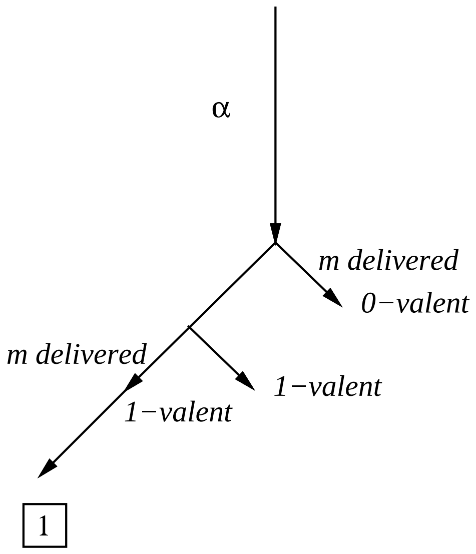


Figure 25.6: Left: the delivery of m does not appear in the extension to a 1-valent state. Right: m is delivered in the extension of α.

On the other hand, if this extension does contain a delivery event for m, then consider the place where this delivery occurs. Immediately after the delivery, we must have univalence, which for consistency with the later decision in this execution must be 1-valence. Then we use the same argument between this position and the top.

Now, if the intervening step (step s in Figure 25.5) involves some process other than i, then it is also possible to perform the same action after the delivery of m, and the resulting global state is the same after either order of these two events (see Figure 25.7).

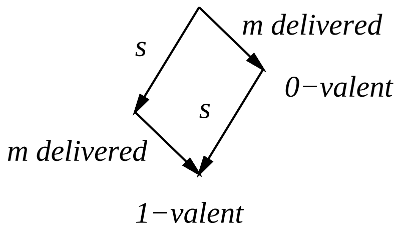


Figure 25.7: if s does not involve i, then performing s before or after m contradicts valency.

But this is a contradiction, since one execution is supposed to be 0-valent and the other

1-valent. So it must be that the intervening step involves process i.

But then this yields a decider, a contradiction to the assumption.

Case 2: Process i take a locally-controlled step.

Essentially the same argument works in this case; we only sketch it. Consider the tree of finite extensions in which anything happens not including a locally-controlled step of i, and then i does a locally-controlled step. (Here we use the assumption that locally-controlled steps are always possible.) Note that the tree can include message deliveries to i, but not locally-controlled steps of i, except at the leaves.

This time, we get a configuration involving one step, such that if i takes a locally-controlled step just before this step, the result is a 0-valent execution, whereas if i takes its locally-controlled step just after the step, the result is a 1-valent execution. Again, we get this by cases. If the extension doesn't contain a locally-controlled step of i, we can attach it at the end and proceed as in Case 1. If it does, then choose the first such in the extension and proceed as in Case 1. Then we complete the proof is as in Case 1, by showing that either commutativity holds or a decider exists. ▮

25.2 Ben-Or Randomized Protocol for Consensus

Fischer, Lynch and Paterson show that consensus is impossible in an asynchronous system even for simple stopping faults. However, this is only for deterministic algorithms. It turns out that the problem can be solved in an asynchronous environment using randomization, with probability 1 of eventually terminating. In fact, an algorithm can even be designed to tolerate stronger types of process faults — Byzantine faults, where processes can move to arbitrary states, and send arbitrary messages at any time.

The algorithm uses a large number of processes relative to the number of faults being tolerated, e.g., $n \geq 7f + 1$. (This can be reduced, but this version is easiest to see.) The algorithm proceeds as follows. Each process starts with its initial value in variable x. The algorithm works in asynchronous "phases", where each phase consists of two rounds. Each process sends messages of the form $\mathit{first}(r, v)$ and $\mathit{second}(r, v)$ where r is the phase number and v is a value in the domain being considered. The code is given in Figure 25.8 for binary values. We assume for simplicity that the nodes continue performing this algorithm forever, even though they only decide once.

25.2.1 Correctness

Validity. If all the processes start with the same initial value v, then it is easy to see that all nonfaulty processes decide on v in the first phase: In the first round of phase 1, all the

```
Initially, $x$ is $i$'s initial value.
 For each phase $r$ do:
 Round 1:            broadcast $\mathit{first}(r, x)$
                     wait for $(n - f)$ messages of the form $\mathit{first}(r, *)$
                     if at least $(n - 2f)$ messages in this set have same value $v$
                     then $x \leftarrow v$
                     else $x \leftarrow \mathit{nil}$

 Round 2:            broadcast $\mathit{second}(r, x)$
                     wait for $(n - f)$ messages of the form $\mathit{second}(r, *)$
                     Let $v$ be the value occurring most often (break ties arbitrarily),
                        and let $m$ be the number of occurrences of $v$
                     if $m \geq (n - 2f)$
                     then (DECIDE $v$; $x \leftarrow v$)
                     else if $m \geq (n - 4f)$
                          then $x \leftarrow v$
                          else $x \leftarrow \mathit{random}$
```

Figure 25.8: Randomized Consensus Algorithm for Asynchronous Network: code for process i.

nonfaulty processes broadcast $\mathit{first}(1, v)$ and receive $\mathit{first}(1, v)$ messages from at least $n - 2f$ processes. Then in round 2 of phase 1, all nonfaulty processes broadcast $\mathit{second}(1, v)$ and again receive at least $n - 2f$ $\mathit{second}(1, v)$ messages.

Agreement. We show that the nonfaulty processes cannot disagree. We will also see that all nonfaulty processes terminate quickly once one process decides. Suppose that process i decides v at phase r. This can happen only if i receives at least $n - 2f$ $\mathit{second}(r, v)$ messages for a particular r and v, which guarantees that each other nonfaulty process j receives at least $n - 4f$ $\mathit{second}(r, v)$ messages. This is true since although there can be some difference in which processes' messages are received by i and j, process j must receive messages from at least $n - 3f$ of the $n - 2f$ senders of the $\mathit{second}(r, v)$ messages received by i. Among those senders, at least $n - 4f$ must be nonfaulty, so they send the same message to j as to i. A similar counting argument shows that v must be the value occurring most often in p_j's set of messages (since $n > 7f$). Therefore, process j cannot decide on a different value at phase r. Moreover, j will set $x \leftarrow v$ at phase r. Since this holds for all nonfaulty processes j, it follows (as in the argument for validity) that all nonfaulty processes that have not yet decided will decide v in phase $r + 1$.

Termination. Now it remains to argue that the probability that someone eventually terminates is 1. So consider any phase r. We will argue that, regardless of what has happened at prior rounds, with probability at least 2^{-n} (which is quite small, but nevertheless positive), all nonfaulty processes will end up with the same value of x at the end of round r. In this case, by the argument for agreement, all nonfaulty processes will decide by round $r+1$. For this phase r, consider the first time any nonfaulty process, say i, reaches the point of having received at least $n-f$ $\mathit{first}(r,*)$ messages. Let M be this set of messages, and let v be the majority value of the set (define v arbitrarily if there is no majority value). We now claim that for any $\mathit{second}(r,w)$ message sent by a nonfaulty process, we must have $w=v$ or $w=\mathit{nil}$. To see that this is true, suppose $w \neq \mathit{nil}$. Now, if $\mathit{second}(r,w)$ is sent by a nonfaulty process j, then j receives at least $n-2f$ $\mathit{first}(r,w)$ messages. Since (by counting) at least $n-4f$ $\mathit{first}(r,w)$ messages appear in M, and since $n \geq 7f+1$, this is a majority, and so we have $w=v$.

Next, we claim that at any phase r, the only value that can be "forced upon" any nonfaulty process as its choice is v. To see this, note that for a process to be forced to choose w, the process must receive at least $n-4f$ $\mathit{second}(r,w)$ messages. Since at least one of these messages is from a nonfaulty process, it follows (from the previous claim) that $w=v$.

Now, note that the value v is determined at the first point where a nonfaulty process has received at least $n-f$ $\mathit{first}(r,*)$ messages. Thus, (by the last claim) the only forcible value for round r is determined at that point. But this point is before any nonfaulty process tosses a coin for round r, and hence these coin tosses are *independent of the choice* of forcible value for round r. Therefore, with probability at least 2^{-n}, all processes tossing coins will choose v, agreeing with all those that do not toss coins. This gives the claimed decision for round r.

The main interest in this result is that it shows a significant difference between the randomized and non-randomized models, in that the consensus problem can be solved in the randomized model. The algorithm is not practical, however, because its expected termination time is very high. We remark that cryptographic assumptions can be used to improve the probability, but the algorithms and analysis are quite difficult. (See [Feldman90].)

25.3 Parliament of Paxos

In this section we'll see Lamport's recent consensus algorithm, for deterministic, asynchronous systems. Its advantage is that *in practice*, it is very tolerant to faults of the following types: node stopping and recovery; lost, out-of-order, or duplicated messages; link failure and recovery. However, it's a little hard to state exactly what fault-tolerance properties it has, so instead we'll present the algorithm and then describe its properties informally. We remark that the protocol always satisfies agreement and validity. The only issue is termination – under what circumstances nonfaulty processes are guaranteed to terminate.

The Protocol. All or some of the participating nodes keep trying to conduct *ballots*. Each ballot has an identifier, which is a $(timestamp, index)$ pair, where the timestamp needn't be a logical time, but can just be an increasing number at each node. The originator of each ballot tries (if it doesn't fail) to (1) associate a *value* with the ballot, and (2) get the ballot accepted by a majority of the nodes. If it succeeds in doing so, the originator decides on the value in the ballot and informs everyone else of the decision.

The protocol proceeds in five rounds as follows.

1. The originator sends the identifier (t, i) to all processes (including itself). If a node receives (t, i), it commits itself to voting "NO" in all ballots with identifier smaller than (t, i) on which it has not yet voted.

2. Each process sends back to the originator of (t, i) the set of all pairs $((t', i'), v)$, indicating those ballots with identifier smaller than (t, i) on which the node has previously voted "YES"; it also encloses the final values associated with those ballots. If the originator receives this information from a *majority*, it chooses as the final value of the ballot the value v associated with the largest $(t', i') < (t, i)$ that arrives in any of these messages. If there is no such value reported by anyone in the majority, the originator chooses its own initial value as the value of the ballot.

3. The originator sends out $((t, i), v)$, where v is the chosen value from round 2.

4. Each process that hasn't already voted "NO" on ballot (t, i) votes "YES" and sends its vote back to the originator. If the originator receives "YES" votes from a majority of the nodes, it decides on the value v.

5. The originator sends out the decision value to all the nodes. Any node that receives it also decides.

We first claim that this protocol satisfies agreement and validity. Termination will need some more discussion.

Validity. Obvious, since a node can only decide on a value that is some node's initial value.

Agreement. We prove agreement by contradiction: suppose there is disagreement. Let b be the smallest ballot on which some process decides, and say that the associated value is v. By the protocol, some majority, say M, of the processes vote "YES" on b. We first claim that v is the only value that gets associated with any ballot strictly greater than b. For suppose not, and let b' be the smallest ballot greater than b that has a different value, say $v' \neq v$. In order to fix the value of ballot b', the originator has to hear from a majority, say M', of the processes. M' must contain at least one process, say i, that is also in M, i.e., that votes "YES" on b. Now, it cannot be the case that i sends its information (round 2) for b' before it votes "YES" (round 4) on b: for at the time it sends this information, it commits itself to voting "NO" on all smaller ballots. Therefore, i votes "YES" on b before sending its information for b'. This in turn means that (b, v) is included in the information i sends for b'. So the originator of b' sees a "YES" for ballot b. By the choice of b', the originator cannot see any value other than v for any ballot between b and b'. So it must be the case that the originator chooses v, a contradiction.

Termination. Note that it is possible for two processes to keep starting ballots and prevent each other from finishing. Termination can be guaranteed if there exists a "sufficiently long time" during which there is only one originator trying to start ballots (in practice, processes could drop out if they see someone with a smaller index originating ballots, but this is problematic in a purely asynchronous system), and during which time a majority of the processes and the intervening links all remain operative.

The reason for the Paxos name is that the entire algorithm is couched in terms of a fictitious Greek parliament, in which legislators keep trying to pass bills. Probably the cutest part of the paper is the names of the legislators.

Homework Assignment 1

Due: Thursday, September 17

Reading

Lamport-Lynch survey paper.

Exercises

1. Carry out careful inductive proofs of correctness for the LeLann-Chang-Roberts (LCR) algorithm outlined in class.
2. For the LCR algorithm,
   (a) Give a UID assignment for which $\Omega(n^2)$ messages are sent.
   (b) Give a UID assignment for which only $O(n)$ messages are sent.
   (c) Show that the average number of messages sent is $O(n \log n)$, where this average is taken over all the possible orderings of the elements on the ring, each assumed to be equally likely.
3. Write "code" for a state machine to express the Hirschberg-Sinclair algorithm, in the same style as the code given in class for the LCR algorithm.
4. Show that the Frederickson-Lynch counterexample algorithm doesn't necessarily have the desired $O(n)$ message complexity if processors can wake up at different times. Briefly describe a modification to the algorithm that would restore this bound.
5. Prove the best *lower* bound you can for the number of rounds required, in the worst case, to elect leader in a ring of size n. Be sure to state your assumptions carefully.

Homework Assignment 2

Due: Thursday, September 24

Reading

The Byzantine generals problem, by Lamport, Shostak and Pease.

Exercises

1. Prove the best *lower* bound you can for the number of rounds required, in the worst case, to elect leader in a ring of size n. Be sure to state your assumptions carefully.

2. Recall the proof of the lower bound on the number of messages for electing a reader in a synchronous ring. Prove that the bit-reversal ring described in class for $n = 2^k$ for any positive integer k, is $\frac{1}{2}$-symmetric.

3. Consider a synchronous bidirectional ring of unknown size n, in which processes have UID's. Give upper and lower bounds on the number of messages required for all the processors to compute $n \bmod 2$ (output is via a special message).

4. Write pseudocode for the simple algorithm discussed in class, for determining shortest paths from a single source i_0, in a weighted graph. Give an invariant assertion proof of correctness.

5. Design an algorithm for electing a leader in an arbitrary strongly connected directed graph network, assuming that the processes have UID's but that they have no knowledge of the size or shape of the network. Analyze its time and message complexity.

6. Prove the following lemma: If all the edges of a connected weighted graph have distinct weights, then the minimum spanning tree is unique.

7. Consider the following variant of the generals problem. Assume that the network is a complete graph of $n > 2$ participants, and that the system is deterministic (i.e., non-randomized). The validity requirement is the same as described in class. However, suppose the agreement requirement is weakened to say that "if any general decides 1

then there are at least two that decide 1". That is, we want to rule out the case where one general attacks alone.

Is this problem solvable or unsolvable? Prove.

Homework Assignment 3

Due: Thursday, October 1

Exercises

1. Show that the generals problem (with link failures) is unsolvable in any nontrivial connected graph.

2. Consider the generals problem with link failures for two processes, in a setting where each message has an independent probability p of getting delivered successfully. (Assume that each process can send only one message per round.) For this setting, design an algorithm that guarantees termination, has "low probability" ϵ of disagreement (for any pair of inputs), and has "high probability" L of attacking in case both inputs are 1 and there are no failures. Determine ϵ and L in terms of p for your algorithm.

3. In the proofs of bounds for randomized generals algorithms, we used two definitions of information level: $level_A(i,k)$, and $mlevel_A(i,k)$. Prove that for any A, i and k, these two are always within 1 of each other.

4. *Be the Adversary!*

   (a) Consider the algorithm given in class for consensus in the presence of processor stopping faults (based on a labeled tree). Suppose that instead of running for $f+1$ rounds, the algorithm only runs for f rounds, with the same decision rule. Describe a particular execution in which the correctness requirements are violated. (*Hint:* a process may stop "in the middle" of a round, before completing sending all its messages).

   (b) Now consider the algorithm given for consensus in the presence of Byzantine faults. Construct an execution to show that the algorithm gives a wrong result if it is run with either (i) 7 nodes, 2 faults, and 2 rounds, or (ii) 6 nodes, 2 faults, and 3 rounds.

5. Using the stopping fault consensus algorithm as a starting point, modify the correctness conditions, algorithm and proof as necessary to obtain a consensus algorithm that is resilient to Byzantine faults, assuming that authentication of messages is available.

6. (**optional**) Consider the optimized version of the stopping fault consensus algorithm, where only the first two messages containing distinct values are relayed. In this algorithm it isn't really necessary to keep all the tree information around. Can you reduce the information kept in the state while still preserving the correctness of the algorithm? You should sketch a proof of why your "optimized" algorithm works.

Homework Assignment 4

Due: Thursday, October 8

Reading

Lynch and Tuttle, *An Introduction to Input/Output Automata*, CWI-Quarterly, 2(3), 1989.

Exercises

1. Consider the network graph G depicted in Figure 25.9.

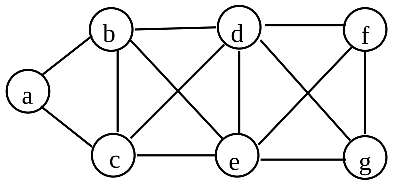


Figure 25.9: the graph G is a 7-node, 2-connected communication graph

   Show directly (i.e., not by quoting the connectivity bound stated in class, but by a proof specific to this graph) that Byzantine agreement cannot be solved in G when 2 processors are faulty.

2. In class, a recursive argument was used to show impossibility of f-round consensus in the presence of f stopping failures. If the recursion is unwound, the argument essentially constructs a long chain of runs, where each two consecutive runs in the chain look the same to some process. How long is the chain of runs?
   Optional: Can you shorten this chain using Byzantine faults rather than stopping faults?

3. Write "code" for a correct 3-phase commit protocol.

4. Fill in more details in the inductive proof of mutual exclusion, for Dijkstra's algorithm.

5. Describe an execution of Dijkstra's algorithm in which a particular process is locked out forever.

Homework Assignment 5

Due: Thursday, October 22

Exercises

1. Consider the Burns mutual exclusion algorithm.

   (a) Exhibit a fair execution in which some process is locked out.

   (b) Do a time analysis for deadlock-freedom. That is, assume that each step outside the remainder region takes at most 1 time unit, and give upper and lower bounds on the length of the time interval in which there is some process in T and no process in C ("upper bound" means analysis that applies to all schedules; "lower bound" means a particular schedule of long duration).

2. Why does the Lamport bakery algorithm fail if the integers are replaced by the integers mod B, for some very large value of B? Describe a specific scenario.

3. Design a mutual exclusion algorithm for a slightly different model in which there is one extra process, a *supervisor process*, that can always take steps. The model should use single-writer-multi-reader shared variables. Analyze its complexity.

4. Design an atomic read-modify-write object, based on lower-level multi-writer multi-reader read-write memory. The object should support one operation, *apply*(f), where f is a function. In the serial specification, the effect of *apply*(f) on an object with value v is that the value of the object becomes $f(v)$, and the original value v is returned to the user. The executions can assumed to be fair.

5. Consider the following variant of the unbounded-registers atomic snapshot object. In this variant, the sequence number is also incremented each time a *snap* is completed (rather than only in *update*). Does this variant preserve the correctness of the original algorithm?

Homework Assignment 6

Due: Thursday, October 29

Exercises

1. Why does the Lamport bakery algorithm fail if the integers are replaced by the integers mod B, for some very large value of B? Describe a specific scenario.

2. Programmers at the Flaky Computer Corporation designed the following protocol for n-process mutual exclusion with deadlock-freedom.

Shared variables: x, y. Initially $y = 0$.

Code for p_i:

```
  ** Remainder Region **
$try_i$
$L$ :
$x := i$
**if** $y \neq 0$ **then goto** $L$
$y := 1$
$x := i$
**if** $x \neq i$ **then goto** $L$
$crit_i$
  ** Critical Region **
$exit_i$
$y := 0$
$rem_i$
```

Does this protocol satisfy the two claimed properties? Sketch why it does, or show that it doesn't. (If you quote an impossibility result to say that it doesn't satisfy the properties, then you must still go further and actually exhibit executions in which the properties are violated.)

3. Reconsider the consensus problem using read-write shared memory. This time suppose that the types of faults being considered are more constrained than general stopping failures, in particular, that the only kind of failure is stopping right at the beginning (never performing any non-input steps). Is the consensus problem solvable? Give an algorithm or an impossibility proof.

4. Let A be any I/O automaton, and suppose that α is a finite execution of A. (a) Prove that there is a fair execution $\alpha\alpha'$ of A (i.e., an extension of α). (b) If β is any finite or infinite sequence of input actions of A, prove that there is a fair execution $\alpha\alpha''$ of A, such that the sequnce of input actions of $beh(\alpha'')$ is identical to β.

Homework Assignment 7

Due: Thursday, November 5

Exercises

1. Let A be any I/O automaton. Show that there is another I/O automaton B with only a single partition class that "implements" or "solves" A, in the sense that $fairbehs(B) \subseteq fairbehs(A)$.

2. Rewrite the Lamport bakery algorithm as an I/O automaton in precondition-effects notation. While doing this, generalize the algorithm slightly by introducing as much nondeterminism in the order of execution of actions as you can. (The precondition-effects notation makes it easier to express nondeterministic order of actions than does the usual flow-of-control notation.)

3. (Warning: We haven't entirely worked this one out.) Consider a *sequential timestamp object* based on the sequential timestamp system discussed informally in class (the one with the 3^{n-1} values). This is an I/O automaton that has a big shared variable $\overline{label}$ containing the latest labels (you can ignore the values) for all processes, and that has two indivisible actions involving the $\overline{label}$ vector: (1) An action that indivisibly looks at the entire $\overline{label}$ vector, chooses a new label according to the rule based on full components, and writes it in the $\overline{label}$ vector. This is all done indivisibly. (2) An action that just snaps the entire $\overline{label}$ vector.

   (a) Write this as an IOA.

   (b) Prove the following invariant: For any cycle, at any level, at most two of three components are occupied.

   (c) Describe a similar sequential timestamp system based on unbounded real number timestamps.

   (d) Use a possibilities mapping to show that your algorithm of (a) implements your algorithm of (c).

   (Plenty of partial credit will be given for good attempts.)

Homework Assignment 8

Due: Thursday, November 12

Exercises

1. Give example executions to show that the multiwriter multireader register implementation given in class (based on a concurrent timestamp object) is *not* correctly serialized by serialization points placed as follows.

   (a) For a *read*: at the point of the embedded *snap* operation. For a *write*: at the point of the embedded *update* operation.

   (b) For all operations: at the point of the embedded *snap* operation.

   Note that the embedded *snap* and *update* operations are embedded inside the atomic snapshot out of which the concurrent timestamp object is built.

2. Consider the following simple "majority-voting" algorithm for implementing a multiwriter multireader register, in *majority snapshot* shared memory model. In this model, the memory consists of a single vector object, one position per writer process, with each position containing a pair consisting of a value v and a positive integer timestamp t. Each *read* operation *instantaneously* reads any majority of the memory locations, and returns the value associated with the largest timestamp. Each *write* operation *instantaneously* reads any majority of the memory locations, determines the largest timestamp t, and writes the new value to any majority of the memory locations, accompanied by timestamp $t + 1$.

   Use the lemma proved in class (Lecture 15) to sketch a proof that this is a correct implementation of an atomic multiwriter multireader register.

3. Show that every exclusion problem can be expressed as a resource problem.

4. Show that Lamport's Construction 1, when applied to atomic registers, does not necessarily give rise to an atomic register.

Homework Assignment 9

Due: Thursday, November 19

Exercises

1. Give an explicit I/O automaton with a single-writer multi-reader user interface, that preserves user well-formedness (alternation of requests and responses), whose fair well-formed behaviors contain responses for all invocations, and whose well-formed external behaviors are exactly those that are legal for a *regular* read/write register. Do this also for a *safe* read/write register.

2. Extend the Burns lower bound on the number of messages needed to elect a leader in an asynchronous ring so that it applies to rings whose sizes are not powers of two.

3. Give an explicit algorithm that allows a distinguished leader node i_0 in an arbitrary connected network graph G to calculate the exact total number of nodes in G. Sketch a proof that it works correctly.

4. Consider a "banking system" in which each node of a network keeps a number indicating an amount of money. We assume, for simplicity, that there are no external deposits or withdrawals, but messages travel between nodes at arbitrary times, containing money that is being "transferred" from one node to another. The channels preserve FIFO order. Design a distributed network algorithm that allows each node to decide on (i.e., output) its own balance, so that the total of all the balances is the correct amount of money in the system. To rule out trivial cases, we suppose that the initial state of the system is not necessarily "quiescent", i.e., there can be messages in transit initially.

   The algorithm should not halt or delay transfers "unnecessarily". Give a convincing argument that your algorithm works.

Homework Assignment 10

Due: Thursday, December 3

Exercises

1. (This question is open ended, and we haven't worked it out entirely.) Compare the operation of the Gallager-Humblet-Spira spanning tree algorithm to that of the synchronous version discussed earlier in the course. E.g., is there any relationship between the fragments produced in the two cases? (Perhaps such a connection can be used in a formal proof.)

2. Design (outline) an asynchronous algorithm for a network with a leader node i_0, that allows i_0 to discover and output the maximum distance d from itself to the furthest node in the network. The algorithm should work in time $O(d)$. What is the message complexity?

3. Consider a square grid graph G, consisting of $n \times n$ nodes. Consider a partition P_k into k^2 equal-sized clusters, obtained by dividing each side into k equal intervals. In terms of n and k, what are the time and message complexities of synchronizer γ based on partition P_k?

4. Obtain the best upper bound you can for an asynchronous solution to the k-session problem.

Homework Assignment 11

Due: Thursday, December 10

Exercises

1. Develop a way of simulating single-writer multi-reader shared memory algorithms in an asynchronous network; your method should be based on *caching*, where readers keep local copies of all the variables and just read their local copies (if possible). The writer, however, needs to carry out a more complicated protocol in order to write. Describe appropriate protocols for the writer (and also for the reader, if the local copy is not available), in order to simulate instantaneous read/write shared memory. Be sure to guarantee that each process' invocations eventually terminate.

   Outline why your algorithm works correctly.

2. "Optimize" the mutual exclusion algorithm described in class, based on Lamport's state machine simulation strategy, in order not to keep all the messages ever received from all other processes.

3. Prove the following invariant for the Dijkstra-Scholten termination detection algorithm.

   All non-idle nodes form a tree rooted at the node with status leader, where the tree edges are given by parent pointers.

4. In the shortest paths problem, we are given a graph $G = (V, E)$, with weights $w(e) > 0$ for all $e \in E$, and a distiguished node $i_0 \in V$. The objective is to compute, at each node, its weighted distance from i_0. Consider the Bellman-Ford algorithm for synchronous networks defined as follows. Each node i maintains a non-negative $distance_i$ variable. The algorithm proceeds in rounds as follows. At each round, all the nodes send to their neighbors the value of their *distance* variable. The distinguished node i_0 has $distance = 0$ fixed, and all other nodes i assign, at each round,

   $$distance_i \leftarrow \min_{(j,i)\in E} \{distance_j + w(j,i)\} \ .$$

   Assume that the state of a node consists only of its *distance* variable. Prove that this algorithm is self-stabilizing.

Appendix

Notes for extra lectures

6.852J/18.437J Distributed Algorithms January 5, 1993
Lecturer: Nancy Lynch

Lecture 26

A.1 Implementing Reliable Point-to-Point Communication in terms of Unreliable Channels

In our treatment of asynchronous networks, we have mostly dealt with
the fault-free case.
Exceptions: self-stabilization (single arbitrary failure)
processor stopping faults and Byzantine faults, for consensus
Now emphasize failures, but in a very special case – a two-node
network.
The problem to be considered is reliable communication between users
at two nodes.

Messages are sent by some user at one node, and are supposed to be
delivered to a user at the other.
Generally want this delivery to be reliable – FIFO, exactly once each.
The sender and receiver nodes are I/O automata.
The two channels inside are generally not our usual reliable channels,
however.
They may have some unreliability properties, e.g.,
loss of packets
duplication
reordering.
However, we don't normally allow worse faults such as manufacturing of
completely bogus messages.

We also don't allow total packet loss – e.g., have a liveness
assumption that says something like:
"If infinitely many packets are sent, then infinitely many of
them are delivered."
Have to be careful to state this right – since we are considering
duplication of packets, it's not enough just to say that infinitely
many sends imply infinitely many receives – we want infinitely many
of the different send *events* to have "corresponding" receive events.
(We don't want infinitely many receives of some early
message, while infinitely many new messages keep being inserted.)

Can usually formalize the channels as I/O automata, (using IOA fairness
to express the liveness condition) but it may be more natural
just to specify their allowable external behavior
(in the form of a "cause" function from packet receipt events to
packet send events satisfying certain properties).

Later, we will want to generalize the conditions to utilize a
"port structure", and say that the channel exhibits FIFO behavior
and liveness with respect to each port separately. Amounts to a
composition of channels, one for each pair of matching ports.

A.1.1 Stenning's Protocol

The sender places incoming messages in a buffer, $buffer_s$.
It tags the messages in its buffer with successively larger integers
(no gaps), starting from 1.
The receiver assembles a buffer, $buffer_r$, also containing
messages with tags starting from 1.
As the buffer gets filled in, the receiver can deliver the
initial messages in order.

(Alternative modeling choice: eliminate the sender's buffer in favor of a
handshake protocol with the user telling it when it can submit the next
message.)

Other than the basic liveness condition described earlier, this
protocol can tolerate a lot of unreliable behavior on the

part of the channel: loss, duplication and reordering.

Note that only one direction of underlying channel is required, strictly speaking, although in this case, the sender never knows to stop sending any particular message.

If we would like to eventually stop sending each message after it has been delivered, need to also add an acknowledgement protocol in the receiver-to-sender direction.

The protocol can send lots of messages concurrently, but just to be specific (and simple), we describe a "sequential" version of the protocol, in which the sender waits to know that the previous message has been delivered to the receiver before starting to send the next version.

Code: (see Figures A.1 and A.2).

Correctness proof:

Can give a high-level spec that is basically a queue, MQ, with single state component *mqueue*.

A send-message action puts a message on the end and a receive-message removes it from the front.

Same as our earlier channel spec.

The liveness condition is then the same as we used before for individual channels in our reliable network model – eventual delivery of the first message (easy spec as an IOA).

The implementation consists of IOA's for the sender and receiver, but it is natural to use a more general model for the underlying channels.

However, I don't want to just use axioms for external behavior.

To do a mapping proof,
it is still convenient to have a state machine, even though the fairness condition needed isn't so easily specified using IOA fairness.

State machine has as its state a set, send puts element into the set, can deliver a packet any number of times if it's in the set (never gets removed from the set).

Interface:

Input:

$send\_msg(m), m \in M$

$receive\_pkt^{rt}(i), i$ a nonnegative integer

Output:

$send\_pkt^{tr}(m,i), m \in M, i$ a nonnegative integer

State:

buffer, a finite queue of elements of M, initially empty, and

integer, a nonnegative integer, initially 1.

Steps:

```
$send\_msg(m), m \in M$
    Effect:
        add $m$ to $buffer$.

$send\_pkt^{tr}(m,i), m \in M, i$ a nonnegative integer
    Precondition:
        $m$ is first on $buffer$.
        $i = integer$
    Effect:
        None.

$receive\_pkt^{rt}(i), i$ a nonnegative integer
    Effect:
        if $i = integer$ then
            [remove first element (if any) from $buffer$;
            $integer := integer + 1$]
```

Partition:

all $send\_pkt^{tr}$ actions are in one class.

Figure A.1: Stenning Sender A^s

Interface:

Input:

$receive\_pkt^{tr}(m, i), n \in M, i$ a nonnegative integer

Output:

$receive\_msg(m), m \in M$

$send\_pkt^{rt}(i), i$ a nonnegative integer

State:

buffer, a finite queue of elements of M, initially empty, and

integer, a nonnegative integer, initially 0.

Steps:

```
$receive\_msg(m), m \in M$
    Precondition:
        $m$ is first on $buffer$.
    Effect:
        remove first element from $buffer$.

$receive\_pkt^{tr}(m, i), m \in M, i$ a nonnegative integer
    Effect:
        if $i = integer + 1$ then
            [add $m$ to $buffer$;
            $integer := integer + 1$]

$send\_pkt^{rt}(i), i$ a nonnegative integer
    Precondition:
        $i = integer$
    Effect:
        None.
```

Partition:

all $send\_pkt^{rt}$ actions

all $receive\_msg$ actions

Figure A.2: Stenning Receiver A^r

Liveness – i.e., the infinite-send-infinite-receive condition –
is specified as an additional condition (a "liveness predicate").

In order to prove that this works, it is useful to have some invariants:

Lemma 1 *The following are true about every reachable state of Stenning's protocol.*

1. *$integer_r \le integer_s \le integer_r + 1$.*
2. *All packets in $queue_{sr}$ have integer tag at most equal to $integer_s$.*
3. *All packets in $queue_{sr}$ with integer tag equal to $integer_s$ have their message components equal to the first message of $buffer_s$.*
4. *All packets in $queue_{rs}$ have integer tag at most equal to $integer_r$.*
5. *If $integer_s$ appears anywhere other than the sender's state, then $buffer_s$ is nonempty.*

To show the safety property, show that every behavior of the implementation is also a behavior of the spec.
Do this by using a forward simulation — actually, a refinement.
Specifically,
If s and u are states of the implementation and specification respectively, then we define $u = r(s)$ provided that $u.queue$ is determined as follows:

1. If $s.integer_s = s.integer_r$, then $u.queue$ is obtained by first removing the first element of $s.buffer_s$ and then concatenating $s.buffer_r$ and the "reduced" $buffer_s$.
   (Invariant implies that $s.buffer_s$ is nonempty.)
2. Otherwise, $u.queue$ is just the concatenation of $s.buffer_r$ and $s.buffer_s$.

Now we show that r is a refinement.
The correspondence of initial states is easy because all queues are empty.

Inductive step:

Given (s, π, s'), and $u = r(s)$, we consider the following cases.

1. $\mathit{send\_msg}(m)$:
   Let $u' = r(s')$.
   We must show that (u, π, u') is a step of MQ.
   This is true because the *send_msg* event modifies both queues in the same way.

$\mathit{receive\_msg}(m)$:
   Let $u' = r(s')$. We must show (u, π, u') is a step of MQ. Since
   π is enabled in s, m is first on $s.\mathit{buffer}_r$, so m
   is also first on $u.\mathit{queue}_r$, so π is also enabled in u.
   Both queues are modified in the same way.

3. $\mathit{send\_pkt}$:
   Then we must show that $u = r(s')$.
   This is true because the action doesn't change the virtual queue.

4. $\mathit{receive\_pkt}^{tr}(m, i)$:
   We show that $u = r(s')$.
   The only case of interest is if the packet is accepted.
   This means $i = s.\mathit{integer}_r + 1$, so by Lemma
   1, in state s, $\mathit{integer}_s = \mathit{integer}_r + 1$,
   so $r(s) = u$ is the concatenation of $s.\mathit{buffer}_r$ and
   $s.\mathit{buffer}_s$.

   Then in s', the integers are equal, and m is added to the end of
   buffer_r.
   So $r(s')$ is determined by removing the first element of
   $s'.\mathit{buffer}_s$
   and then concatenating with $s'.\mathit{buffer}_r$.
   Then in order to see that $r(s')$ is the same as u, it suffices to note
   that m is the same as the first element in $s.\mathit{buffer}_s$.
   But Lemma 1 implies that $i = s.\mathit{integer}_s$,
   and that m is the same as the first element in $s.\mathit{buffer}_s$, as needed.

4. $\mathit{receive\_pkt}^{rt}(i)$:
   Again we show that $u = r(s')$.
   The only case of interest is if the packet is accepted.
   This means $i = s.\mathit{integer}_s$, which implies by Lemma

1 that $s.integer_s = s.integer_r$.
This means that $r(s)$ is the concatenation of
$s.buffer_r$ and $s.buffer_s$, with the first element of
$s.buffer_s$ removed.
Then $s'.integer_s = s'.integer_r + 1$, so $r(s')$ is just
the concatenation of $s'.buffer_s$ and $s'.buffer_r$.
But $s'.buffer_s$ is obtained from $s.buffer_s$ by removing
the first element, so that $r(s') = u$.

Since r is a refinement, all behaviors of Stenning's protocol are
also behaviors of the specification automaton, and so satisfy the
safety property that the sequence of messages in *receive_msg* events is
consistent with the sequence of messages in *send_msg* events.
In particular, the fair behaviors of Stenning's protocol satisfy this safety
property.

Liveness:
The basic idea is that we first show a correspondence between an
execution of the impl and of the spec MQ.
We can get this by noticing that forward simulation actually yields a
stronger relationship than just behavior inclusion – execution
correspondence lemma gives a correspondence between states and
external actions throughout the execution.
Then suppose that the liveness condition of the specification is
violated.
That is, the abstract *queue* is nonempty from some point in the
execution, but no receive-message events ever occur.
We consider what this means in the implementation execution.
If the $buffer_r$ is ever nonempty, then eventually deliver
message, contradiction, so can assume that $buffer_r$ is always empty.
Then $buffer_s$ must be nonempty throughout the execution;
sender keeps sending packets for the first message on $buffer_s$,
gets delivered to receiver by channel fairness, then put
in $buffer_r$, which is a contradiction.

A.1.2 Alternating Bit Protocol

Optimized version of Stenning's protocol that doesn't use sequence numbers, but only Boolean values.

These get associated with successive messages in an alternating fashion.

This only works for channels with stronger reliability conditions: assume channels exhibit loss and duplication, but no reordering.

Liveness again says that infinitely many sends lead to receives of infinitely many of the sent packets.

Sender keeps trying to send a message with a bit attached.

When it gets an ack with the same bit, it switches to the opposite bit to send the next message.

Receiver is also always looking for a particular bit.

Code: (see Figures A.3,A.4).

Correctness Proof:

Model the sender and receiver as IOA's, just as before.

Again, the channels are not quite IOA's, because of the fairness conditions being different from usual IOA fairness.

But it is again convenient to have a state machine with a liveness restriction.

This time, the channel state machine has as its state a queue.

A send puts a copy of the packet at the end of the queue, and a receive can access *any element* of the queue; when it does, it throws away all the *preceding* elements, but not the element that is delivered.

This allows for message loss and duplication.

Again, need a liveness condition to say that any packet that is sent infinitely many times is also delivered infinitely many times.

A convenient correctness proof can be based on a forward simulation (this one is multi-valued) to a restricted version of Stenning's protocol, which we call FIFO-Stenning.

This restricted version is correct because the less restricted version is.

Interface:

Input:

$send\_msg(m), m \in M$

$receive\_pkt^{rt}(b), b$ a Boolean

Output:

$send\_pkt^{tr}(m, b), m \in M, b$ a Boolean

The state consists of the following components:

buffer, a finite queue of elements of M, initially empty, and

flag, a Boolean, initially 1.

The steps are:

```
$send\_msg(m), m \in M$
    Effect:
        add $m$ to $buffer$.

$send\_pkt^{tr}(m, b), m \in M, b$ a Boolean
    Precondition:
        $m$ is first on $buffer$.
        $b = flag$
    Effect:
        None.

$receive\_pkt^{rt}(b), b$ a Boolean
    Effect:
        if $b = flag$ then
            [remove first element (if any) from $buffer$;
            $flag := flag + 1 \, mod \, 2$]
```

Partition:

all $send\_pkt^{tr}$ actions are in one class.

Figure A.3: ABP Sender A^s

Interface:

Input:

  $receive\_pkt^{tr}(m,b), n \in M, b$ a Boolean

Output:

  $receive\_msg(m), m \in M$

  $send\_pkt^{rt}(b), b$ a Boolean

State:

  buffer, a finite queue of elements of M, initially empty, and

  flag, a Boolean, initially 0.

Steps:

```
$receive\_msg(m), m \in M$
    Precondition:
        $m$ is first on $buffer$.
    Effect:
        remove first element from $buffer$.

$receive\_pkt^{tr}(m,b), m \in M, b$ a Boolean
    Effect:
        if $b \neq flag$ then
            [add $m$ to $buffer$;
            $flag := flag + 1 \, mod \, 2$]

$send\_pkt^{rt}(b), b$ a Boolean
    Precondition:
        $b = flag$
    Effect:
        None.
```

Partition:

  all $send\_pkt^{rt}$ actions are in one class, and

  all *receive_msg* actions are in another class.

Figure A.4: ABP Receiver A^r

The restricted version is based on the FIFO channels described above instead of arbitrary channels.

Use an additional invariant about this restricted version of the algorithm:

Lemma 2 *The following is true about every reachable state of the FIFO-Stenning protocol.*
Consider the sequence consisting of the indices in $queue_{rs}$
(in order from first to last on the queue),
followed by $integer_r$*,*
followed by the indices in $queue_{sr}$*,*
followed by $integer_s$*. (In other words, the sequence starting at* $queue_{rs}$*, and tracing the indices all the way back to the sender automaton.)*
The indices in this sequence are nondecreasing; furthermore, the difference between the first and last index in this sequence is at most 1.

Proof: The proof proceeds by induction on the number of steps in the finite execution leading to the given reachable state.
The base case is where there are no steps, which means we have to show this to be true in the initial state.
In the initial state, the channels are empty, $integer_s = 1$ and $integer_r = 0$.
Thus, the specified sequence is $0, 1$, which has the required properties.

For the inductive step, suppose that the condition is true in state s, and consider a step (s, π, s') of the algorithm.
We consider cases, based on π.

1. π is a *send_msg* or *receive_msg* event.
   Then none of the components involved in the stated condition is changed by the step, so the condition is true after the step.

2. π is a $send\_pkt^{tr}(m, i)$ event, for some m.
   Then $queue_{sr}$ is the only one of the four relevant components of the global state that can change.
   We have $s'.queue_{sr}$ equal to $s.queue_{sr}$ with the addition of (m, i).
   But $i = s.integer_s$ by the preconditions of the action.

Since the new i is placed consecutively in the concatenated sequence with $s'.integer_s = i$, the properties are all preserved.

3. π is a $receive\_pkt^{rt}(i)$ event.
   If $i \neq s.integer_s$ then the only change is to remove some elements in the concatenated sequence, so all properties are preserved.
   On the other hand, if $i = s.integer_s$ then the inductive hypothesis implies that the entire concatenated sequence in state s must consist of i's.
   The only changes are to remove some elements from the beginning of the sequence and add one $i+1$ to the end (since $s'.integer_s = i+1$, by the effect of the action). Thus, the new sequence consists of all i's followed by one $i+1$, so the property is satisfied.

4. π is a $send\_pkt^{rt}$ event.
   Similar to the case for $send\_pkt^{tr}$.

5. π is a $receive\_pkt^{tr}(m,i)$ event.
   If $i \neq s.integer_r + 1$ then the only change is to remove some elements from the concatenated sequence, so all properties are preserved.
   On the other hand, if $i = s.integer_r + 1$ then the inductive hypothesis implies that the entire concatenated sequence in state s must consist of $i-1$'s up to and including $s.integer_r$, followed entirely by i's.
   The only changes are to remove some elements from the sequence and change the value of $integer_r$ from $i-1$ to i, by the effect of the action; this still has the required properties.

∎

Now we will show that the ABP is correct by demonstrating a mapping from ABP to FIFO-Stenning.
This mapping will be a multivalued "possibilities mapping", i.e., a forward simulation.
In particular, it will fill in the integer values of tags only working from bits.

More precisely, we say that a state u of FIFO-Stenning is in $f(s)$ for state s of ABP provided that the following conditions hold.

1. $s.buffer_s = u.buffer_s$
   and $s.buffer_r = u.buffer_r$.

2. $s.flag_s = u.integer_s \bmod 2$
   and $s.flag_r = u.integer_r \bmod 2$.

3. $queue_{sr}$ has the same number of packets in s and u.
   Moreover, for any j, if (m, i) is the j^{th} packet in
   $queuesr$ in u,
   then $(m, i \bmod 2)$ is the j^{th} packet in $queue_{sr}$ in s.
   Also, $queue_{rs}$ has the same number of packets in s and u.
   Moreover, for any j, if i is the j^{th} packet in
   $queue_{rs}$ in u,
   then $i \bmod 2$ is the j^{th} packet in $queue_{rs}$ in s.

Theorem 3 *f above is a forward simulation.*

Proof: By induction.
For the base, let s be the start state of ABP and u the start state of FIFO-Stenning.
First, all the buffers are empty.
Second, $s.flag_s = 1 = u.integer_s\ mod 2$
and $s.flag_r = 0$ and $u.integer_r = 0$, which is as needed.
Third, both channels are empty.
This suffices.

Now show the inductive step.
Suppose (s, π, s') is a step of ABP and $u \in f(s)$.
We consider cases based on π.
1. $\pi = send\_msg(m)$

Choose u' to be the unique state such that (u, π, u') is a step of FIFO-Stenning.
We must show that $u' \in f(s')$.
The only condition that is affected by the step is the first, for the $buffer_s$ component.
However, the action affects both $s.buffer_s$ and $u.buffer_s$ in the same way, so the correspondence holds.

2. $\pi = \mathit{receive\_msg}(m)$

   Since π is enabled in s, m is the first value on $s.\mathit{buffer}_r$.
   Since $u \in f(s)$, m is also the first value on $u.\mathit{buffer}_r$,
   which implies that π is enabled in u.
   Now choose u' to be the unique state such that (u, π, u') is a step of FIFO-Stenning.
   All conditions are unaffected except for the first for buffer_r, and buffer_r is changed
   in the same way in both algorithms, so the correspondence holds.

3. $\pi = \mathit{send\_pkt}^{tr}(m, b)$

   Since π is enabled in s,
   $b = s.\mathit{flag}_s$ and m is the first element on $s.\mathit{buffer}_s$.
   Let $i = u.\mathit{integer}_s$.
   Since $u \in f(s)$, m is also the first element on $u.\mathit{buffer}_s$.
   It follows that $\bar{\pi} = \mathit{send\_pkt}^{tr}(m, i)$ is enabled in u.

   Now choose u' so that $(u, \bar{\pi}, u')$ is a step of FIFO-Stenning.
   We must show that $u' \in f(s')$.
   The only interesting condition is the third, for C^{sr}.
   By inductive hypothesis and the fact that the two steps each insert one packet, it is easy to see that
   C^{sr} has the same number of packets in s' and u'.
   Moreover, the new packet gets added with tag i in state u' and with tag b in state s';
   since $u \in f(s)$, we have $s.\mathit{flag}_s = u.\mathit{integer}_s \bmod 2$,
   i.e., $b = i \bmod 2$,
   which implies the result.

4. $\pi = \mathit{receive\_pkt}^{tr}(m, b)$

   Since π is enabled in s, (m, b) appears in C^{sr} in s.
   Since $u \in g(s)$, (m, i) appears in the corresponding position in
   C^{sr} in u,
   for some integer i with $b = i \bmod 2$.
   Let $\bar{\pi} = \mathit{receive\_pkt}^{tr}(m, i)$; then $\bar{\pi}$ is enabled in u.
   Let u' be the state such that $(u, \bar{\pi}, u')$ is a step of

FIFO-Stenning, and such that the same number of packets is removed from the queue.
We must show that $u' \in f(s')$.

It is easy to see that the third condition is preserved, since each of π and $\bar{\pi}$ removes the same number of packets from C^{sr}.

Suppose first that $b = s.\mathit{flag}_r$.
Then the effects of π imply that the receiver state in s' is identical to that in s.
Now, since $u \in f(s)$, $s.\mathit{flag}_r = u.\mathit{integer}^r \bmod 2$;
since $b = i \bmod 2$, this case must have
$i \neq u.\mathit{integer}_r + 1$.
Then the effects of $\bar{\pi}$ imply that the receiver state in u' is identical to that in u.
It is immediate that the first and second conditions hold.

So now suppose that $b \neq s.\mathit{flag}_r$.
The invariant above for FIFO-Stenning implies that either $i = u.\mathit{integer}_r$ or $i = u.\mathit{integer}_r + 1$.
Since $b = i \bmod 2$ and
(since $u \in f(s)$) $s.\mathit{flag}_r = u.\mathit{integer}_r \bmod 2$,
this case must have $i = u.\mathit{integer}_r + 1$.
Then by the effect of the action, $u'.\mathit{integer}_r = u.\mathit{integer}_r + 1$ and $s'.\mathit{flag}_r = 1 - s.\mathit{flag}_r$,
preserving the second condition.
Also, buffer_r is modified in both cases by adding the entry m at the end; therefore, the first condition is preserved.

5. $\pi = \mathit{send\_pkt}^{rt}(b)$

   Similar to $\mathit{send\_pkt}^{tr}(m, b)$.

6. $\pi = \mathit{receive\_pkt}^{rt}(b)$

   Since π is enabled in s, b is in C^{rs} in s.
   Since $u \in f(s)$, i is the corresponding position in C^{rs} in u, for some integer i with $b = i \bmod 2$.

Let $\bar{\pi} = \mathit{receive\_pkt}^{rt}(i)$; then $\bar{\pi}$ is enabled in u.
Let u' be the state such that $(u, \bar{\pi}, u')$ is a step of
FIFO-Stenning, and such that the same number of packets are removed.
We must show that $u' \in f(s')$.

It is easy to see that the third condition is preserved,
since each of π and
$\bar{\pi}$ removes the same number of messages from C^{rs}.

Suppose first that $b \neq s.\mathit{flag}_s$.
Then the effects of π imply that the sender state in s'
is identical to that in s.
Now, since $u \in f(s)$, $s.\mathit{flag}_s = u.\mathit{integer}_s \bmod 2$;
since $b = i \bmod 2$, this case must have
$i \neq u.\mathit{integer}_s$.
Then the effects of $\bar{\pi}$ imply that the sender state in u' is
identical to that in u.
It is immediate that the first and second conditions hold for this situation.

So now suppose that $b = s.\mathit{flag}_s$.
The invariant above for FIFO-Stenning implies that either $i = u.\mathit{integer}_s - 1$
or $i = u.\mathit{integer}_s$.
Since $b = i \bmod 2$ and
(since $u \in f(s)$) $s.\mathit{flag}_s = u.\mathit{integer}_s \bmod 2$,
this case must have $i = u.\mathit{integer}_s$.
Then the effect of the action implies that
$u'.\mathit{integer}_s = u.\mathit{integer}_s + 1$
and $s'.\mathit{flag}_s = 1 - s.\mathit{flag}_s$, preserving the second condition.
Also, buffer_s is modified in the same way in both cases,
so the first condition is preserved.

∎

Remark:

Consider the structure of the forward simulation f of this example.
In going from FIFO-Stenning to ABP,
integer tags are condensed to their low-order bits.
The multiple values of the mapping f essentially "replace" this
information.

In this example, the correspondence between ABP and FIFO-Stenning can be described
in terms of a mapping in the opposite direction - a (single-valued) projection from the state of FIFO-Stenning to that of ABP that removes information.
Then f maps a state s of ABP to
the set of states of FIFO-Stenning whose projections are equal to s.
While this formulation suffices to describe many interesting examples,
it does not always work.
(Consider garbage collection examples.)

Machine assistance should be useful in verifying (and maybe producing)
 such proofs.

Liveness:
As before, the forward simulation yields a close correspondence
 between executions of the two systems.
Given any live execution α of ABP, construct a "corresponding"
 execution α' of FIFO-Stenning (needs formalizing).
Show that α' is a live execution of FIFO-Stenning:
 Includes conditions of IOA fairness for the sender and receiver.
 If not live in this sense, then some IOA class is enabled forever but no
  action in that class occurs – easily yields the same situation in
  α (if the correspondence is stated strongly enough), which
  contradicts the fairness of the sender and receiver in α.
 Also includes the channel liveness conditions – infinitely many
  sends imply infinitely many receives.
 Again, this carries over from ABP.

So far, we see that we can tolerate all types of channel faults using
 Stenning, with unbounded "headers" for messages.
With "bounded headers" as in ABP, can tolerate loss and duplication,
 but not reordering.

A.1.3 Bounded-Header Protocols Tolerating Reordering

This leads us to the question of whether we can use bounded headers and tolerate any reordering.

Consider what goes wrong with the ABP, for example, if we attempt to use it with channels that reorder packets.

The recipient can get fooled into accepting an old packet with the same bit as the one expected.

This could cause duplicate acceptance of the same message.

Here we consider the question of whether there exists a protocol that uses bounded headers, and tolerates reordering.

Wang-Zuck 89 prove that there is no bounded-header protocol that tolerates duplication and reordering, and consequently none that tolerates all three types of faulty behavior.

In contrast, AAFFLMWZ produce a bounded-header protocol that tolerates loss and reordering, but not duplication.

That paper also contains an impossibility result for "efficient" bounded-header protocols tolerating loss and reordering.

Formalize the notion of bounded-header by assuming that the external message alphabet is finite and requiring that the packet alphabet also be finite.

Wang-Zuck Reo-Dup Impossibility

Suppose there is an implementation.

Starting from an initial state, run the system to send, systematically, copies of as many different packets as possible, yielding execution α.

That is, there is no extension of α in which any packet is sent that isn't also sent in α.

Suppose there are n send-message events in α.

Let α' be a live extension of α that contains exactly one additional send-message event.

By the correcteness condition, all messages in α' get delivered.

Now we construct a finite execution α''
that looks like α' to the receiver, just up to the point
where the $n+1$st message delivery to the user occurs,

but in which the additional send-message event never occurs.
Namely, delay the sender immediately after α (also don't allow the new send-message event).
But let the receiver do the same thing that it does in α' – receive the same packets, send the same packets, deliver the same messages to the user.
It can receive the same packets, even though the sender doesn't send them, because the earlier packets in α can be duplicated – nothing new can be sent that doesn't appear already in α.
Then as in α', it delivers $n + 1$ messages to the user, even though only n send-message events have occurred.
To complete the contradiction, extend α'' to a live execution of the system, without any new send-message events.
This has more receive-message events than send-message events, a contradiction.

AAFFLMWZ Bounded-Header Protocol Tolerating Loss and Reordering

It turns out that it is possible to tolerate reordering and also loss of messages, with bounded headers.
This assumes that duplication does not occur.
It is not at all obvious how to do this, however; it was a conjecture for a while that this was impossible – that tolerating reordering and loss would require unbounded headers.
This algorithm is NOT a practical algorithm – just a counterexample algorithm to show that an impossibility result cannot be proved.

The algorithm can most easily be presented in two pieces.
The underlying channels are at least guaranteed not to duplicate anything.
So first, use the no-dup channels to implement channels that do not reorder messages (but can lose or duplicate them).
Then, use the resulting FIFO channels to implement reliable communication.
Note that pasting these two together requires each underlying channel to be used for two separate purposes – to simulate channels in two

different protocols, one yielding FIFO communication in either direction.

This means that the underlying channels need to be fair to each protocol separately – need the port structure mentioned earlier.

The second task can easily be done using bounded headers - e.g., use the ABP.

The first is much less obvious.

The sender sends a value to the receiver only in response to an explicit *query* packet from the receiver.

The value that it sends is always the most recent message that was given to it, saved in *latest*.

To ensure that it answers each *query* exactly once, the sender keeps a variable *unanswered* which is incremented whenever a new *query* is received, and decremented whenever a value is sent.

The receiver continuously sends *query*s to the sender, keeping track, in *pending*, of the number of unanswered queries.

The receiver counts, in $count[m]$, the number of copies of each value m received since the last time it delivered a message (or since the beginning of the run if no message has yet been put on that buffer).

At the beginning, and whenever a new message is delivered, the receiver sets *old* to *pending*.

When $count[m] > old$, the receiver knows that m was the value of *latest* at some time after the receiver performed its last receive event.

It can therefore safely output m.

The finiteness of the message alphabet, the fact that the sender will always respond to *query* messages, and the liveness of the channels, together imply that the receiver will output infinitely many values (unless there is no send event at all).

Code:

Sender:

```
Variables:
    $latest$, an element of $M$ or $nil$, initially $nil$
    $unanswered$, a nonnegative integer, initially 0

$send(m)$
Effect:
    $latest := m$

$rcv-pkt(query)$
Effect:
    $unanswered := unanswered + 1$

$send-pkt(m)$
Precondition:
    $unanswered > 0$
    $m = latest \neq nil$
Effect:
    $unanswered := unanswered - 1$

Receiver:
Variables:
    $pending$, a nonnegative integer, initially 0,
    $old$, a nonnegative integer, initially 0,
    for each $m \in M$, $count[m]$, a nonnegative integer,
      initially 0

$receive(m)$
Precondition:
    $count[m] > old$
Effect:
    $count[n] := 0$ for all $n \in M$
    $old := pending$

$send-pkt(query)$
Effect:
    $pending := pending + 1$

$rcv-pkt(m)$
Effect:
```

```
$pending := pending - 1$
$count[m] := count[m] + 1$
```

Lecture 27

B.1 Reliable Communication Using Unreliable Channels

B.1.1 Bounded-Header Protocols Tolerating Reordering

We began considering bounded-header protocols tolerating reordering.
We gave a simple impossibility result for reordering in combination
with duplication, and then showed the AAFFLMWZ protocol that
tolerates reordering in combination with loss (but no duplication).
Today we start with an impossibility result saying that it's not
possible to tolerate reordering and loss with an efficient protocol.

AAFFLMWZ Impossibility Result

Again consider the case of channels C^{sr} and C^{rs}
that cannot duplicate messages but can lose and reorder them.
Again, if infinitely many packets are sent, infinitely many of these packets
are required to be delivered.
(The impossibility proof will still work if we generalize this condition to a
corresponding port-based condition.)
As above, we can get a protocol for this setting;
that protocol was very inefficient in the sense that more and more
packets are required to send individual messages.
Now we will show that this is inherent: such a protocol cannot be
efficient in that it *must* send more and more messages.
Again assume finite message alphabet and finite packet alphabet.

To capture the inefficiency, need a definition to capture the number of packets needed to send a message.

First, a finite execution is *valid* if the number of *send-msg* actions is equal to the number of *rec-msg* actions. (This means that the datalink protocol succeeded in sending all the messages m provided by the user i.e.,all the messages m for which an action *send-msg*(m) occurred.)

The following definition expresses the notion that, in order to successfully deliver any message the datalink protocol only needs (in the best case) to send a *bounded* number of packets over the channels.

Definition 1 *If α is a valid execution, an extension $\alpha\beta$ is a k-extension if:*

1. *In β, the user actions are exactly the two actions* send-msg(m) *and* rec-msg(m) *for some given message m. (This means that exactly one message has been sent successfully by the protocol.)*

2. *All packets received in β are sent in β (i.e.,no old packets are received).*

3. *The number of* rec-pkt^{sr} *actions in β is less than or equal to k.*

*A protocol is k-*bounded *if there is a k-extension of α for every message m and every valid execution α.*

Theorem 1 *There is no k-bounded datalink protocol.*

Assume there is such a protocol and look for a contradiction.

Suppose that there is a multiset T of packets, a finite execution α, and a k-extension $\alpha\beta$ such that:

- every packet in T is in transit from the s to r

  after the execution α (i.e.,T is a multiset of "old" packets).

- the multiset of packets received by the receiver in β is a

  submultiset of T.

We could then derive a contradiction:

Consider an alternative execution that begins similarly with α, but that contains no *send-msg* event.

All the packets in T cause the receiver to behave as in the k-extension $\alpha\beta$ and hence to generate an incorrect *rec-msg* action.

In other words, the receiver is confused by the presence of old packets in the channel, which were left in transit in the channel in α and are equivalent to those sent in β.

At the end of the alternative execution, a message has been received without its being sent, and the algorithm fails.

(Note that we are here assuming a *universal channel* – one that can actually exhibit all the behavior that is theoretically allowed by the channel spec. Such universal channels exist.)

So it remains to manufacture this bad situation.

We need one further definition:

Definition 2 *$T \underset{k}{<} T'$ if*

- $T \subseteq T'$ *(This inclusion is among* multisets *of packets.)*

- *$\exists$ packet p s.t. $mult(p, T) < mult(p, T') \leq k$)*

  ($mult(p, T)$ denotes the multiplicity of p within the multiset T).

Lemma 2 *If α is valid, and T is a multiset of packets in transit after α, then either*

1. *there exists a k-extension $\alpha\beta$ such that the multiset of packets received by A^r in β is a submultiset of T, or*
2. *there exists a valid execution $\alpha' = \alpha\beta$ such that*

(a) all packets received in β are sent in β, and

(b) $\exists$ a new multiset T' of packets in transit after α' such that $T \underset{k}{<} T'$.

Suppose that this lemma is true.
We then show that there is some valid α_∞, and a multiset T_∞ of packets in transit after α_∞ such that case 1 holds.
As we already argued, the existence of such α_∞ and T_∞ leads to the desired contradiction.

For this we define two sequences α_i and T_i with α_0 equal to the empty sequence and T_0 empty.
If condition 1 does not hold for α_0 and T_0 (i.e.,for $i = 0$) we are in the situation of case 2.
We then set $\alpha_1 = \alpha'$ and $T_1 = T'$.
In general, assuming that case 1 does not hold for α_i and T_i, we are then in case 2 and derive a valid extension α_{i+1} of α_i and a multiset T_{i+1} of packets in transit after α_{i+1} $(T_i \underset{k}{<} T_{i+1})$.
But, by definition of the $\underset{k}{<}$ relation, the sequence $T_0 \underset{k}{<} T_1 \underset{k}{<} \dots \underset{k}{<} T_i \underset{k}{<} \dots$ can have at most $k|P|$ terms.
Its last term is the T_∞ we are looking for.
($|P|$ is the number of different possible packets, which is finite because we assumed that the packet alphabet and the size of the headers are bounded.)

Proof: (of Lemma 2)

Pick any m, and get a k-extension $\alpha\beta$ for m, by
the k-boundedness condition.

If the multiset of packets received by A_r is included in T we
are in case (a).

Otherwise there is some packet p for which the multiplicity
of *rec-pkt*(p) actions in β is bigger then
mult(p, T).

We then set $T' = T \cup \{p\}$.

Since the extension is k-bounded, the number of
of these *rec-pkt*(p) is at most k so that
mult$(p, T') \leq k$.

We now want to get a *valid* extension α' of α
leaving the packets from T' in transit.

We know that there is a *send-pkt*(p) action in β
(since all packets received in β were sent in β).

Consider then a prefix $\alpha\gamma$ of $\alpha\beta$ ending with the
first such *send-pkt*(p). (See Figure B.1.)

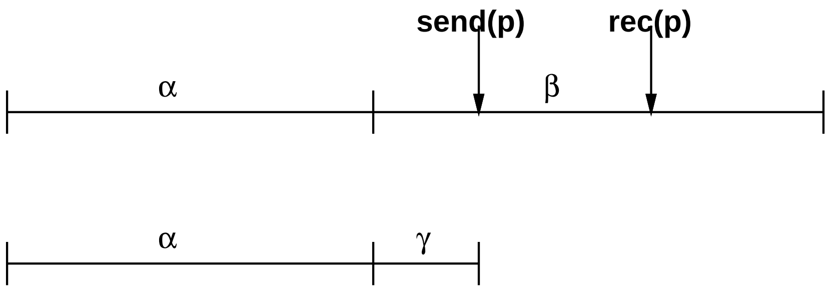


Figure B.1: An extension of α with a $send - pkt(p)$ action

After $\alpha\gamma$ all packets from T' are still in transit.

We want to extend $\alpha\gamma$ to a valid execution without
delivering any packet from T'.

There are three cases.

First, suppose that both the send-message and receive-message events
from β appear in γ; then $\alpha\gamma$ itself suffices.

Second, suppose that only the send-message, but not the receive-message event appears in γ.
To get validity we need to deliver m, which is the only undelivered message, without delivering a packet from T'.
We can achieve this because of a basic property of live executions in this architecture: for any finite execution, there is a live extension that contains no new send-message events and that does not deliver any old packets.
Because of the correctness condition, this must eventually deliver m.
The needed sequence stops just after the receive-message event.

Third and finally, suppose that neither the send-message nor the receive-message event appears in γ.
Then add a new send-message event just after $\alpha\gamma$, and extend this just as in the previous case to get the needed valid extension.

∎

B.1.2 Tolerating Node Crashes

The results above settle pretty much all the cases for reliable nodes.
Now we add consideration of faulty nodes.
This time, we consider node crashes that lose information.
We give two impossibility results, then show a practical algorithm used in the internet.

Impossibility Results

Consider the case where each node has an input action *crash*, and a corresponding output *recover*.
When a *crash* occurs, a special crash flag just gets set, and no locally controlled actions other than *recover* can be enabled.
At some later time, a corresponding *recover* action is supposed to occur; when this happens, the state of the node goes back to an arbitrary initial state.
Thus, the crash causes all information to be lost.

In such a model, it is not hard to see that we can't solve the
reliable communication problem, even if the underlying channels are
completely reliable (FIFO, no loss or duplication).

Proof:

Consider any live execution α in which a single
send(m) event occurs but no crashes.

Then correctness implies that there is a later *receive*(m).

Let α' be the prefix ending just before this *receive*(m).

Now consider α' followed by the events $crash_r recover_r$.

By basic properties of the model,
this can be extended to a live execution α_1, in which no
further send-message events or crashes occur.

Since α_1 must also satisfy the correctness conditions, it must
be that α_1 contains a *receive*(m) event corresponding
to the given *send*(m) event, sometime after the given crash.

Now take the portion of α_1 after the crash-recover,
and splice it at the end of the finite execution
$\alpha' receive(m) crash_r recover_r$.

Claim that the result is another live execution.

But this execution has two *receive*(m) events, a contradiction to the
correctness conditions.

Thus, the key idea is that the system cannot tell whether a receiver crash
occurred before or after the delivery of a message.

Note that a key fact that makes this result work is the separation
between the *receive*(m) event and any other effect such as
sending an acknowledgement.

We could argue that the problem statement is too strong for the case
of crashes.

Now we weaken the problem statement quite a lot, but still obtain an
impossibility result (though it's harder to prove).

We continue to forbid duplication.

We allow reordering, however.

And, most importantly, we now allow loss of messages sent before the
last recover action.

Messages sent after the last recover must be delivered, however.

We describe the underlying channels in as strong a way as possible.
Thus, we insist that they are FIFO and don't duplicate.
All they can do is lose messages.
The only constraint on losing messages is the liveness constraint:
infinitely many sends lead to infinitely many receives.
(or a port version of this).

Now it makes sense to talk about a *sequence* of packets being
"in transit" on a channel.
By this we mean that the sequence is a subsequence of all the packets
in the channel (sent after the sending of the last packet delivered).

Notation:
If α is an execution, $x \in \{s, r\}$, $0 \leq k \leq |\alpha|$,
then define:
$in(\alpha, x, k)$ to be the sequence of packets received by A^x
during the first k steps of α,
$out(\alpha, x, k)$ to be the sequence of packets sent by A^x
during the first k steps of α,
$state(\alpha, x, k)$ to be the state of A^x after exactly k
steps of α, and
$ext(\alpha, x, k)$ to be the sequence of external actions of A^x
occurring during the first k steps of α.

Define $\bar{x}$ to be the opposite node to x.

Lemma 3 *Let α be any crash-free finite execution, of length n. Let x be either node, and let $0 \leq k \leq n$. Suppose that either $k = 0$ or else the k^{th} action in α is an action of A^x. Then there is a finite execution α' (possibly containing crashes) that ends in a state in which:*

1. *the state of x is $state(\alpha, x, k)$,*
2. *the state of $\bar{x}$ is $state(\alpha, \bar{x}, k)$, and*
3. *the sequence $out(\alpha, x, k)$ is in transit from x.*

Proof: Proceed by induction on k.
$k = 0$ is immediate – use the initial configuration only.

Inductive step:

Suppose true for all values smaller than $k > 0$ and prove for k.

Case 1: The first k steps in α are all steps of A^x.
Then the first k steps of α suffice.

Case 2: The first k steps in α include at least one step of $A^{\bar{x}}$.
Then let j be the greatest integer $\leq k$ such that the k^{th} step is a step of $A^{\bar{x}}$.
Note that in fact $j < k$ since we have assumed the k^{th} step is a step of A^x.
Then $in(\alpha, x, k)$ is a subsequence of
$out(\alpha, \bar{x}, j)$, and
$state(\alpha, \bar{x}, k) = state(\alpha, \bar{x}, j)$.
By inductive hypothesis, we get an execution α_1 that leads the two nodes to $state(\alpha, x, j)$ and
$state(\alpha, \bar{x}, j)$, respectively, and that has
$out(\alpha, \bar{x}, j)$ in transit from $\bar{x}$ to x.
This already has $\bar{x}$ in the needed state.
Now run A^x alone – let it first crash and recover, going back to its initial state in α.
Now let it run on its own, using the messages in
$in(\alpha, x, k)$, which are available in the incoming channel since they are a subsequence of $out(\alpha, \bar{x}, j)$.
This brings x to the needed state and puts the needed messages in the outgoing channel.

∎

Now we finish the proof.
Choose α (length n) to be a crash-free execution containing one *send*(m) and its corresponding *receive*(m) (must exist).
Now use α to construct an execution whose final node states are those of α,
and that has a *send* as the last external action.

How to do this:
Let k be the last step of α occurring at the receiver (must exist one, since there is a *receive* event in α.

Then Lemma yields an execution α' ending in the node states
after k steps, and with $out(\alpha, r, k)$ in transit.
Note that $state(\alpha, r, k) = state(\alpha, r, n)$.
Now crash and recover the sender, and then have it run again from
the beginning, using the inputs in $in(\alpha, s, n)$, which are a
subsequence of the available sequence $out(\alpha, r, k)$.
This lets it reach $state(\alpha, s, n)$.
Also, there is a *send* step, but no other external step, in
this suffix.
This is as needed.

Now we get a contradiction.
Let α_1 be the execution obtained above, whose final node
states are those of α,
and that has a *send* as the last external action.
Now lose all the messages in transit in both channels, then extend
α_1 to a live execution containing no further send or crash
events.
By the correctness, this extension must eventually deliver the last
message.
Now claim we can attach this suffix at the end of α, and it
will again deliver the message.
But α alread had an equal number of sends and receives (one of
each), so this extra receive violates correctness.
QED

This says that we can't even solve a weak version of the problem, when
we have to contend with crashes that lose all state information.

B.2 5-packet Handshake Internet Protocol

But note that it is important in practice to have a message delivery
protocol that can tolerate node crashes.
Here we give one important example, the 5-packet handshake protocol of
Belsnes (used in the Internet).
This does tolerates node crashes.

It uses a small amount of stable storage in the form of unique ID's – think of this as stable storage because even after a crash, the system "remembers" not to reuse any ID.

Underlying channels can reorder, duplicate (but only finitely many times), and lose.

Yields no reordering, no duplication (ever), and always delivers messages sent after the last recovery.

The five-packet handshake protocol of Belsnes is one of a batch he designed, getting successively more reliability out of successively more messages. Used in practice.

For each message that the sender wishes to send, there is an initial exchange of packets between the sender and receiver to establish a commonly-agreed-upon message identifier.

That is, the sender sends a new unique identifier (called jd) to the receiver, which the receiver then pairs with another new unique identifier (called id).

It sends this pair back to the sender, who knows that the new id is recent because it was paired with its current jd.

The identifier jd isn't needed any longer, but id is used for the actual message communication.

The sender then associates this identifier id with the message.

The receiver uses the associated identifier to decide whether or not to accept a received message – it will accept a message provided the associated identifier id is equal to the receiver's current id.

The receiver also sends an acknowledgement.

Additional packets are required in order to tell the receiver when it can throw away a current identifier.

Variable		**Type**	**Initially**	**Description**
$mode_s$		`acked`, `needid`, `send`, `rec`	`acked`	Sender's mode.
buf_s		M list	empty	Sender's list of messages to send.
jd_s		JD or NIL	NIL	jd chosen for current message by sender.
$jd-used_s$	S	JD set	empty	Includes all jds ever used by sender.
id_s		$\mathcal{D}$ or NIL	NIL	id received from receiver.
$current-msg_s$		M or NIL	NIL	Message about to be sent to receiver.
$current-ack_s$		Boolean	*false*	Ack from receiver.
$acked-buf_s$		$\mathcal{D}$ list	empty	List of ids for which the sender will issue `acked` message.
$mode_r$		`idle`, `accept`, `rcvd`, `ack`, `rec`	`idle`	Receiver's mode.
buf_r		M list	empty	Receiver's list of messages to deliver.
jd_r		JD or NIL	NIL	jd received from sender.
id_r		$\mathcal{D}$ or NIL	NIL	id chosen for received jd by receiver.
$last_r$		$\mathcal{D}$ or NIL	NIL	Last id the receiver remembers accepting.
$issued_r$	S	$\mathcal{D}$ set	empty	Includes all ids ever issued by receiver.
$nack-buf_r$		$\mathcal{D}$ list	empty	List of ids for which receiver will issue negative acks.

Sender Actions:

Input:

$send\_msg(m)$, $m \in M$
$receive\_pkt_{rs}(p)$, $p \in P$
$crash_s$

Output:

$\textsc{Ack}(b)$, b a Boolean
$send\_pkt_{sr}(p)$, $p \in P$
$recover_s$

Internal:

$choose\_jd$
$grow-jd-used_s$

```
Input:
    $receive\_pkt_{sr}(p)$, $p \in P$
    $crash_r$
Output:
    $receive\_msg(m)$, $m \in M$
    $send\_pkt_{rs}(p)$, $p \in P$
    $recover_r$
Internal:
    $grow - issued_r$
```

```
$send\_msg(m)$
    Effect:
        if $mode_s \neq \texttt{rec}$ then
            add $m$ to $buf_s$

$choose\_jd$
    Precondition:
        $mode_s = \texttt{acked}$
        $buf_s$ nonempty
        $jd \notin jd - used_s$
    Effect:
        $mode_s := \texttt{needid}$
        $current - msg_s := head(buf_s)$
        remove $head(buf_s)$
        $jd_s := jd$
        add $jd$ to $jd - used_s$

$send\_pkt_{sr}(\texttt{needid}, jd)$
    Precondition:
        $mode_s = \texttt{needid}$
        $jd_s = jd$
    Effect:
        none

$receive\_pkt_{sr}(\texttt{needid}, jd)$
    Effect:
        if $mode_s = \texttt{idle}$ then
            $mode_r := \texttt{accept}$
            choose an $id$ not in $issued_r$
            $jd_r := jd$
            $id_r := id$
            add $id$ to $issued_r$

$receive\_pkt_{rs}(\texttt{accept}, jd, id)$
    Effect:
        if $mode_s \neq \texttt{rec}$ then
            if $mode_s = \texttt{needid}$
                and $jd_s = jd$ then
                $mode_s := \texttt{send}$
                $id_s := id$
            else if $id_s \neq id$ then
                add $id$ to $acked - buf_s$

$send\_pkt_{rs}(\texttt{accept}, jd, id)$
    Precondition:
        $mode_r = \texttt{accept}$
        $jd_r = jd$,
        $id_r = id$
    Effect:
        none
```

```
$send\_pkt_{sr}(\texttt{send}, m, id)$
    Precondition:
        $mode_s = \texttt{send}$
        $current - msg_s = m$
        $id_s = id$
    Effect:
        none
```

```
$receive\_pkt_{sr}(\texttt{send}, m, id)$
    Effect:
        if $mode_r \neq \texttt{rec}$ then
            if $mode_r = \texttt{accept}$
                and $id = id_r$ then
                $mode_r := \texttt{rcvd}$
                add $m$ to $buf_r$
                $last_r := id$
            else if $id \neq last_r$ then
                add $id$ to $nack - buf_r$
```

```
$receive\_msg(m)$
    Precondition:
        $mode_r = \texttt{rcvd}$,
        $m$ first on $buf_r$
    Effect:
        remove $head(buf_r)$
        if $buf_r$ is empty then
            $mode_r := \texttt{ack}$
```

```
$receive\_pkt_{rs}(\texttt{ack}, id, b)$
    Effect:
        if $mode_s \neq \texttt{rec}$ then
            if $mode_s = \texttt{send}$ and
                $id = id_s$ then
                $mode_s := \texttt{acked}$
                $current - ack_s := b$
                $jd_s := \text{NIL}$
                $id_s := \text{NIL}$
                $current - msg_s := \text{NIL}$
            if $b = true$ then
                add $id$ to $acked - buf_s$
```

```
$send\_pkt_{rs}(\texttt{ack}, id, true)$
    Precondition:
        $mode_r = \texttt{ack}$
        $last_r = id$
    Effect:
        none
```

```
$send\_pkt_{rs}(\texttt{ack}, id, false)$
    Precondition:
        $mode_r \neq \texttt{rec}$
        $id$ first on $nack - buf_r$
    Effect:
        remove $head(nack - buf_r)$
```

```
$send\_pkt_{sr}(\texttt{acked}, id, \text{NIL})$
    Precondition:
        $mode_s \neq \texttt{rec}$,
        $id$ first on $acked - buf_s$
    Effect:
        remove $head(acked - buf_s)$
```

```
$receive\_pkt_{sr}(\texttt{acked}, id, \text{NIL})$
    Effect:
        if $(mode_r = \texttt{accept}$
            and $id = id_r)$ or
            $(mode_r = \texttt{ack}$
            and $id = last_r)$ then
                $mode_r := \texttt{idle}$
                $jd_r := \text{NIL}$
                $id_r := \text{NIL}$
                $last_r := \text{NIL}$
```

```
ACK(b)
    Precondition:
        $mode_s =$ acked
        $buf_s$ is empty
        $b = current-ack_s$
    Effect:
        none
```

```
$crash_s$
    Effect:
        $mode_s :=$ rec
$recover_s$
    Precondition:
        $mode_s =$ rec
    Effect:
        $mode_s :=$ acked
        $jd_s :=$ NIL
        $id_s :=$ NIL
        empty $buf_s$
        $current-msg_s :=$ NIL
        $current-ack_s := false$
        empty $acked-buf_s$
```

```
$grow-jd-used_s$
    Precondition:
        none
    Effect:
        add some $JD$s to $jd-used_s$
```

```
$crash_r$
    Effect:
        $mode_r :=$ rec
$recover_r$
    Precondition:
        $mode_r =$ rec
    Effect:
        $mode_r :=$ idle
        $jd_r :=$ NIL
        $id_r :=$ NIL
        $last_r :=$ NIL
        empty $buf_r$
        empty $nack-buf_r$
```

```
$grow-issued_r$
    Precondition:
        none
    Effect:
        add some $\mathcal{D}$s to $issued_r$
```

Explain the algorithm from my talk – overview, discuss why it works informally,

Note the two places where msgs get added to the acked-buffer, once normally, once in reponse to information about an old message.

The trickiest part of this code turns out to be the liveness argument – how do we know that this algorithm does continue to make progress?

There are some places where only single responses are sent, e.g., a single nack, but that could get lost.

However, in this case, the node at the other end is continuing to resend, so eventually another nack will be triggered.

Lecturer: Nancy Lynch

Lecture 28

This is the final lecture!

It's on timing-based computing.

Recall that we started this course out studying synchronous distributed algorithms, spent the rest of the time doing asynchronous algorithms.

There is an interesting model that is in between these two – really, a class of models.

Call these *partially synchronous.*

The idea is that the processes can presume some information about time, though the information might not be exact.

For example, we might have bounds on process step time, or message delivery time.

Note that it doesn't change things much just to include upper bounds, because the same set of executions results.

In fact, we have been associating upper bounds with events throughout the course, when we analyzed time complexity.

Rather, we will consider both lower and upper bounds on the time for events.

Now we do get some extra power, in the sense that we are now restricting the possible interleavings.

C.1 MMT Model Definition

A natural model to use is the one of Merritt, Modugno and Tuttle.

Here I'm describing the version that's used in Lynch, Attiya paper.

This is the same as I/O automata, but now each class doesn't just get

treated fairly;
rather, have a *boundmap*, which associates lower and upper
bounds with each class.
The lower bound is any nonnegative real number, while the upper bound
is either a positive real or ∞.
The intuition is that the successive actions in each class aren't
supposed to occur any sooner (after the class is enabled or after
the last action in that class) than indicated by the lower bound.
Also, no later than the upper bound.

Executions are described as sequences of alternating states and
(action,time) pairs,
Admissibility.

I've edited the following from the Lynch, Attiya model paper, Section 2.

C.1.1 Timed Automata

In this subsection, we augment the I/O automaton model to allow discussion of timing properties. The treatment here is similar to the one described in [AttiyaL89] and is a special case of the definitions proposed in [MerrittMT90]. A *boundmap* for an I/O automaton A is a a mapping that associates a closed subinterval of $[0, \infty]$ with each class in $part(A)$, where the lower bound of each interval is not ∞ and the upper bound is nonzero.[16] Intuitively, the interval associated with a class C by the boundmap represents the range of possible lengths of time between successive times when C "gets a chance" to perform an action. We sometimes use the notation $b_\ell(C)$ to denote the lower bound assigned by boundmap b to class C, and $b_u(C)$ for the corresponding upper bound. A *timed automaton* is a pair (A, b), where A is an I/O automaton and b is a boundmap for A.

We require notions of "timed execution", "timed schedule" and "timed behavior" for timed automata, corresponding to executions, schedules and behaviors for ordinary I/O automata. These will all include time information. We begin by defining the basic type of sequence that underlies the definition of a timed execution.

[16]In [MerrittMT90], the model is defined in a more general manner, to allow boundmaps to yield open or semi-open intervals as well as closed intervals. This restriction is not crucial in this paper, but allows us to avoid considering extra cases in some of the technical arguments.

Definition 1 *A* timed sequence *(for an I/O automaton A) is a (finite or infinite) sequence of alternating states and (action,time) pairs,*

$$s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \dots ,$$

satisfying the following conditions.

1. *The states s_0, s_1, ... are in states(A).*
2. *The actions π_1, π_2,... are in acts(A).*
3. *The times t_1, t_2,... are successively nondecreasing nonnegative real numbers.*
4. *If the sequence is finite, then it ends in a state s_i.*
5. *If the sequence is infinite then the times are unbounded.*

For a given timed sequence, we use the convention that $t_0 = 0$. For any finite timed sequence α, we define $t_{end}(\alpha)$ to be the time of the last event in α, if α contains any (action,time) pairs, or 0, if α contains no such pairs. Also, we define $s_{end}(\alpha)$ to be the last state in α. We denote by $ord(\alpha)$ (the "ordinary" part of α) the sequence

$$s_0, \pi_1, s_1, \pi_2, \dots ,$$

i.e., α with time information removed.

If i is a nonnegative integer and $C \in part(A)$, we say that i is an *initial index* for C in α if $s_i \in enabled(A, C)$ and either $i = 0$ or $s_{i-1} \in disabled(A, C)$ or $\pi_i \in C$. Thus, an initial index for class C is the index of an event at which C becomes enabled; it indicates a point in α from which we will begin measuring upper and lower time bounds.

Definition 2 *Suppose (A, b) is a timed automaton. Then a timed sequence α is a* timed execution *of (A, b) provided that $ord(\alpha)$ is an execution of A and α satisfies the following conditions, for each class $C \in part(A)$ and every initial index i for C in α.*

1. *If $b_u(C) < \infty$ then there exists $j > i$ with $t_j \leq t_i + b_u(C)$ such that either $\pi_j \in C$ or $s_j \in disabled(A, C)$.*
2. *There does not exist $j > i$ with $t_j < t_i + b_\ell(C)$ and π_j in C.*

The first condition says that, starting from an initial index for C, within time $b_u(C)$ either some action in C occurs or there is a point at which no such action is enabled. Note that if $b_u(C) = \infty$, no upper bound requirement is imposed. The second condition says that, again starting from an initial index for C, no action in C can occur before time $b_\ell(C)$

has elapsed. Note in particular that if a class C becomes disabled and then enabled once again, the lower bound calculation gets "restarted" at the point where the class becomes re-enabled.

The *timed schedule* of a timed execution of a timed automaton (A, b) is the subsequence consisting of the (action,time) pairs, and the *timed behavior* is the subsequence consisting of the (action,time) pairs for which the action is external. The *timed schedules* and *timed behaviors* of (A, b) are just those of the timed executions of (A, b).

We model each timing-dependent concurrent system as a single timed automaton (A, b), where A is a composition of ordinary I/O automata (possibly with some output actions hidden).[17] We also model problem specifications, including timing properties, in terms of timed automata.

We note that the definition we use for timed automata may not be sufficiently general to capture all interesting systems and timing requirements. It does capture many, however.

The MMT paper contains a generalization of this model, in which we have bounds that can be open or closed. Also, upper bound of 0 allowed (?). Finally, don't restrict to only finitely many processes. Do results about composition similar to I/O automata.

C.2 Simple Mutual Exclusion Example

Just as an illustrative example, I'll describe a very basic problem we
considered within this model – to get simple upper and lower bound
results for a basic problem.

Started with mutual exclusion.

When we did this, we did not have a clear idea of which parameters,
etc., would turn out to be the most important, so we considered
everything.

C.3 Basic Proof Methods

Algorithms such as the ones in this paper quickly lead to the need for
proof methods for verifying correctness of timing-based algorithms.

[17]An equivalent way of looking at each system is as a composition of timed automata. An appropriate definition for a composition of timed automata is developed in [MerrittMT90], together with theorems showing the equivalence of the two viewpoints.

In the untimed setting, we make heavy use of invariants and simulations, so it seems most reasonable to try to extend these methods to the timed setting.

However, it is not instantly obvious how to do this.

In an asynchronous system, everything important about the system's state (i.e., everything that can affect the system's future behavior) is summarized in the ordinary states of the processes and channels, i.e., in the values of the local variables and the multiset of messages in transit on each link.

In a timing-based system, that is no longer the case.

Now if a message is in transit, it is important to know whether it has just been placed into a channel or has been there a long time (and is therefore about to be delivered).

Similar remarks hold for process steps.

Therefore, it is convenient to augment the states with some extra predictive timing information, giving bounds on how long it will be before certain events may or must occur.

C.4 Consensus

Now we can also revisit the problem of distributed consensus in the timed setting.

This time, for simplicity, we ignore the separate clocks and just suppose we have processes with upper and lower bounds of $[c_1, c_2]$ on step time.

The questions is how much time it takes to reach consensus, if there are f faulty processes.

C.5 More Mutual Exclusion

Lynch-Shavit mutual exclusion with proofs.

C.6 Clock Synchronization

Work from Lamport, Melliar-Smith;
Lundelius, Lynch
Fischer, Lynch, Merritt

6.852 Course Reading List

General References

[1] N. Lynch and I. Saias. Distributed Algorithms. Fall 1990 Lecture Notes for 6.852. MIT/LCS/RSS 16, Massachusetts Institute of Technology, February 1992.

[2] N. Lynch and K. Goldman. Distributed Algorithms. MIT/LCS/RSS 5, Massachusetts Institute of Technology, 1989. Lecture Notes for 6.852.

[3] M. Raynal. *Algorithms for Mutual Exclusion.* M.I.T. Press, 1986.

[4] M. Raynal. *Networks and Distributed Computation.* M.I.T. Press, 1988.

[5] J. M. Hèlary and M. Raynal. *Synchronization and Control of Distributed Systems and Programs.* John Wiley & Sons, 1990.

[6] K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, 1988.

[7] Butler Lampson, William Weihl, and Eric Brewer. 6.826 Principles of computer systems, Fall 1991. MIT/LCS/RSS 19, Massachusetts Institute of Technology, 1992. Lecture Notes and Handouts.

D.1 Introduction

[1] L. Lamport and N. Lynch. Distributed computing. Chapter of Handbook on Theoretical Computer Science. Also, appeared as Technical Memo MIT/LCS/TM-384, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1989.

D.2 Models and Proof Methods

D.2.1 Automata

[1] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, Mook, The Netherlands, 1992. Springer-Verlag. Also, MIT/LCS/TM 458.

D.2.2 Invariants

[1] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[2] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[3] Butler Lampson, William Weihl, and Eric Brewer. 6.826 Principles of computer systems, Fall 1991. MIT/LCS/RSS 19, Massachusetts Institute of Technology, 1992. Lecture notes and Handouts.

D.2.3 Mapping Methods

[1] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, Mook, The Netherlands, 1992. Springer-Verlag. Also, MIT/LCS/TM 458.

[2] Màrtin Abadi and Leslie Lamport. The existence of refinement mapping. *Theoretical Computer Science*, 2(82):253–284, 1991.

[3] Nils Klarlund and Fred B. Schneider. Proving nondeterministically specified safety properties using progress measures, May 1991. To appear in *Information and Computation* Spring, 1993.

D.2.4 Shared Memory Models

[1] N. Lynch and M. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[2] Kenneth Goldman, Nancy Lynch, and Kathy Yelick. Modelling shared state in a shared action model, April 1992. Manuscript. Also, earlier verison in *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science,* pages 450-463, Philadelphia, PA, June 1990.

D.2.5 I/O Automata

[1] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1989.

[2] Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, MIT Dept. of Electrical Engineering and Computer Science, April 1987.

[3] Nick Reingold, Da-Wei Wang and Lenore Zuck. Games I/O automata play, September 1991. Technical Report YALE/DCS/TR–857.

[4] Da-Wei Wang, Lenore Zuck, and Nick Reingold. The power of I/O automata. Manuscript, May, 1991.

D.2.6 Temporal Logic

[1] Z. Manna and A.Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[2] K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, 1988.

[3] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Systems Research Center, December 25 1991.

D.2.7 Timed Models

[1] F. Modugno, M. Merritt, and M. Tuttle. Time constrained automata. In *CONCUR'91 Proceedings Workshop on Theories of Concurrency: Unification and Extension*, Amsterdam, August 1991.

[2] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, Mook, The Netherlands, 1992. Springer-Verlag. Also, MIT/LCS/TM 458.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. To appear in *Theoretical Computer Science.*

[4] F.W. Vaandrager and N.A. Lynch. Action transducers and timed automata. In *Proceedings of CONCUR '92, 3rd International Conference on Concurrency Theory*, Lecture Notes in Computer Science, Stony Brook, NY, August 1992. Springer Verlag. To appear.

[5] N. Lynch and H. Attiya. Using mappings to prove timing properties. *Distrib. Comput.*, 6(2), 1992. To appear.

[6] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for real-time systems, September 1991. Submitted for publication. An abbreviated version in [HMP91].

[7] T. A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 353–366, January 1991.

D.2.8 Algebra

[1] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[2] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

D.3 Synchronous Message-Passing Systems

D.3.1 Computing in a Ring

[1] G. LeLann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, Toronto, 1977.

[2] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, May 1979.

[3] D. Hirschberg and J. Sinclair. Decentralized extrema-finding in circular configuarations of processes. *Communications of the ACM*, 23:627–628, November 1980.

[4] G.L. Peterson. An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4:758–762, October 1982.

[5] G. Frederickson and N. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34(1):98–115, January 1987. Also, MIT/LCS/TM-277, July 1985.

[6] H. Attiya, M. Snir, and M. Warmuth. Computing in an anonymous ring. *Journal of the ACM*, 35(4):845–876, October 1988.

D.3.2 Network Protocols

[1] Baruch Awerbuch. Course notes.

[2] B. Awerbuch. Reducing complexities of distributed maximum flow and breadth-first search algorithms by means of network synchronization. *Networks*, 15:425–437, 1985.

[3] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.*, 15(4):1036–1053, November, 1985.

D.3.3 Consensus

Basic Results

[1] J. Gray. Notes on data base operating systems. Technical Report IBM Report RJ2183(30001), IBM, February 1978. (Also in Operating Systems: An Advanced Course, Springer-Verlag Lecture Notes in Computer Science #60.).

[2] G. Varghese and N. Lynch. A Tradeoff Between Safety and Liveness for Randomized Coordinated Attack Protocols. *11th Symposium on Principles of Distributed Systems*, Vancouver, B.C., Canada, 241–250, August, 1992.

[3] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[4] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[5] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. *Information and Computation*, 97(2):205–233, 1992.

[6] D. Dolev and H. Strong. Authenticated algorithms for Byzantine agreement. *SIAM J. Computing*, 12(4):656–666, November 1983.

[7] T. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

[8] R. Turpin and B. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18(2):73–76, 1984. Also, Technical Report MIT/LCS/TR-315, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, April 1984. Also, revised in B. Coan, *Achieving consensus in fault-tolerant distributed computing systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology,1987.

Number of Processes

[1] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3:14–30, 1982.

[2] M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[3] L. Lamport. The weak Byzantine generals problem. *Journal of the ACM*, 30(3):669–676, 1983.

Time Bounds

[1] M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

[2] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October, 1990.

[3] Y. Moses and M. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:249–259, 1988.

Communication

[1] B.A. Coan. *Achieving Consensus in Fault-Tolerant Distributed Computer Systems: Protocols, Lower Bounds, and Simulations.* PhD thesis, Massachusetts Institute of Technology, June 1987.

[2] Yoram Moses and O. Waarts. Coordinated traversal: $(t+1)$-round Byzantine agreement in polynomial time. In *Proceedings of 29th Symposium on Foundations of Computer Science*, pages 246–255, October 1988. To appear in the special issue of the Journal of Algorithms dedicated to selected papers from the 29th IEEE Symposium on Foundations of Computer Science. Extended version available as Waarts' M.Sc. thesis, Weizmann Institute, August 1988.

[3] Piotr Berman and Juan Garay. Cloture voting: n/4-resilient distributed consensus in t+1 rounds. To appear in *Mathematical Systems Theory—An International Journal on Mathematical Computing Theory*, special issue dedicated to distributed agreement. Preliminary version appeared in *Proc. 30th Symp. on Foundations of Computer Science*, pp. 410-415, 1989.

Approximate Agreement

[1] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):449–516, 1986.

Firing Squad

[1] J.E. Burns and N.A Lynch. The Byzantine firing squad problem. *Advances in Computing Research*, 4:147–161, 1987.

[2] B. Coan, D. Dolev, C. Dwork, and L. Stockmeyer. The distributed firing squad problem. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 335–345, May 1985.

Commit

[1] C. Dwork and D. Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, August 1983.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1986.

[3] B. A. Coan and J. L. Welch. Transaction commit in a realistic timing model. *Distributed Computing*, 4:87–103, 1990.

The Knowledge Approach

[1] J.Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 50–61, 1984. Revised as IBM Research Report, IBM-RJ-4421.

[2] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October, 1990.

D.4 Asynchronous Shared Memory Systems

D.4.1 Mutual Exclusion

[1] M. Raynal. *Algorithms for Mutual Exclusion.* M.I.T. Press, 1986.

[2] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[3] D.E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[4] J.G. DeBruijn. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 10(3):137–138, 1967.

[5] M. Eisenberg and M. McGuire. Further comments on Dijkstra's concurrent programming control. *Communications of the ACM*, 15(11):999, 1972.

[6] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. To appear in *Info. Comput.*

[7] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[8] L. Lamport. The mutual exclusion problem. *Journal of the ACM*, 33(2):313–326, 327–348, 1986.

[9] G. Peterson and M. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of 9th ACM Symposium on Theory of Computing*, pages 91–97, May 1977.

[10] J. Burns, M. Fischer, P. Jackson, N. Lynch, and G. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, 1982.

[11] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. AT&T Technical Memorandum, May 1991. Also, submitted for publication.

D.4.2 Resource Allocation

[1] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proceedings of 20th IEEE Symosium on Foundations of Computer Science*, pages 234–254, October 1979.

[2] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. Prog. Lang. and Syst.*, 11(1):90–114, January 1989.

[3] D. Dolev, E. Gafni, and N. Shavit. Toward a non-atomic era: l-exclusion as a test case. In *Proceedings of 20th ACM Symposium on Theory of Computing*, pages 78–92, May 1988.

[4] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, pages 115–138, 1971.

D.4.3 Atomic Registers

[1] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[2] L. Lamport. On interprocess communication. *Distributed Computing*, 1(1):77–85, 86–101, 1986. *Digital Systems Research* TM-8.

[3] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register, June 1992. Submitted for publication.

[4] Bard Bloom. Constructing two-writer registers. *IEEE Trans. Computers*, 37(12):1506–1514, December 1988.

[5] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings 27th Annual IEEE Symposium on Theory of Computing*, pages 233–243, Toronto, Ontario, Canada, May 1986. Also, MIT/LCS/TM-314, Laboratory

for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., 1986. Corrigenda in *Proceedings of 28th Annual IEEE Symposium on Theory of Computing*, page 487, 1987.

[6] R. Schaffer. On the correctness of atomic multi-writer registers. Bachelor's Thesis, June 1988, Massachusetts Institute Technology. Also, Technical Memo MIT/LCS/TM-364, Lab for Computer Science, MIT, June 1988 and Revised Technical Memo MIT/LCS/TM-364, January 1989.

[7] Soma Chaudhuri and Jennifer Welch. Bounds on the costs of register implementations. Manuscript, 12/91. Also, Technical Report TR90-025, University of North Carolina at Chapel Hill, June 1990.

D.4.4 Snapshots

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Proceedings of the* 9^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 1–13, Quebec, Canada, August 1990. Also, Technical Memo MIT/LCS/TM-429, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1990. To appear in *Journal of the ACM*.

[2] Cynthia Dwork, Maurice Herlihy, Serge Plotkin, and Orli Waarts. Time-lapse snapshots. In *In Israel Symposium on Theory of Computing and Systems (ISTCS)*, 1992. Also, Stanford Technical Report STAN-CS-92-1423, April 1992, and Digital Equipment Corporation Technical Report CRL 92/5.

D.4.5 Timestamp Systems

[1] Amos Israeli and Ming Li. Bounded time-stamps. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 371–382, October 1987.

[2] R. Gawlick, N. Lynch, and N. Shavit. Concurrent time-stamping made simple. In *In Israel Symposium on Theory of Computing and Systems (ISTCS)*, pages 171–185. Springer-Verlag, May, 1992.

[3] Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! In *In Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC)*, pages 655–666, Victoria, B.C., Canada, May 1992. Preliminary version appears in IBM Research Report RJ 8425, October 1991.

D.4.6 Consensus

[1] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[2] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[3] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? Submitted for publication. Earlier versions in MIT/LCS/TM-442 and *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science,* 1:55-64, October, 1990.

[4] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the* 7^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 291–302, Toronto, Canada, August 1988.

D.4.7 Shared Memory for Multiprocessors

[1] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

[2] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–529, September 1982.

[3] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[4] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[5] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *TOPLAS*, October 1992. To appear. Also, a preliminary version appeared in SPAA'89.

[6] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. AT&T Bell Laboratories, 11211-910509-09TM and 11261-910509-6TM, May 1991.

[7] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories. AT&T Bell Laboratories, BL011211-920528-07TM and BL011261-920528-17TM, May 1992.

[8] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. Unpublished Manuscript, November 1991.

[9] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Department of Computer Science, Princeton University, CS-TR-180-88, September 1988.

[10] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. College of Computing, Georgia Institute of Technology, GIT-CC-91/42, September 1991.

[11] Hagit Attiya and Jennifer Welch. Sequential consistency versus linearizability. In *Proceedings of the 3rd ACM Symp. on Parallel Algorithms and Architectures (SPAA3), 1991.* Also, Technion- Israel Institute Technology, Dept. of C.S., TR 694.

D.5 Asynchronous Message-Passage Systems

D.5.1 Computing in Static Networks

Rings

[1] G. LeLann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, Toronto, 1977.

[2] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, May 1979.

[3] D. Hirschberg and J. Sinclair. Decentralized extrema-finding in circular configuarations of processes. *Communications of the ACM*, 23:627–628, November 1980.

[4] G.L. Peterson. An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4:758–762, October 1982.

[5] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. Research Report RJ3185, IBM, July 1981. *J. Algorithms*, 3:245–260, 1982.

[6] J. Burns. A formal model for message passing systems. Technical Report TR-91, Computer Science Dept., Indiana University, May 1980.

[7] K. Abrahamson, A. Adler, L. Higham, and D. Kirkpatrick. Tight lower bounds for probabilistic solitude verification on anonymous rings. Submitted for publication.

Complete Graphs

[1] E. Korach, S. Moran, and S. Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *In Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 199–207, 1984.

[2] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proceedings of 4th ACM Symposium on Principles of Distributed Computing*, pages 186–195, Minaki, Ontario, August 1985.

General Networks

[1] D. Angluin. Local and global properties in networks of processors. In *Proceedings of 12th ACM Symposium on Theory of Computing*, pages 82–93, 1980.

[2] R. Gallager, P. Humblet, and P. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[3] P. Humblet. A distributed algorithm for minimum weight directed spanning trees. *IEEE Transactions on Computers*, COM-31(6):756–762, 1983. MIT-LIDS-P-1149.

[4] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, pages 503–513, St. Louis, Missouri, October 1990.

D.5.2 Network Synchronization

[1] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(1):558–565, July 1978.

[2] J. M. Hèlary and M. Raynal. *Synchronization and Control of Distributed Systems and Programs.* John Wiley & Sons, 1990.

[3] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985. Also, Technical Memo MIT/LCS/TM-268, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, January 1985.

[4] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*. IEEE, October 1988.

[5] E. Arjomandi, M. Fischer, and N. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[6] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *toplas*, 6(2):254–280, apr 1984.

[7] Leslie Lamport. The part-time parliament. Technical Memo 49, Digital Systems Research Center, September 1989.

[8] J.L. Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–171, August 1987.

D.5.3 Simulating Asynchronous Shared Memory

[1] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 363–376, 1990. Revised version to appear in *J. ACM*.

D.5.4 Resource Allocation

[1] M. Raynal. *Algorithms for Mutual Exclusion*. M.I.T. Press, 1986.

[2] K. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[3] N. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal Of Computer And Systems Sciences*, 23(2):254–278, October 1981.

[4] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th ACM Symposium on Principles of Programming Languages*, pages 133–138, 1981.

[5] Baruch Awerbuch and Michael Saks. A dining philosophers algorithm with polynomial response time. In *Proceedings of the 31st IEEE Symp. on Foundations of Computer Science*, pages 65–75, St. Louis, Missouri, October 1990.

[6] Jennifer Welch and Nancy Lynch. A modular drinking philosophers algorithm, July 1992. Earlier version appeared as MIT/LCS/TM-417.

D.5.5 Delecting Stable Properties

Termination Detection

[1] E. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), August 1980.

[2] N. Francez and N. Shavit. A new approach to detection of locally indicative stability. In *Proceedings of the 13th International Colloquium on Automata Languages and Programming (ICALP)*, pages 344–358, Rennes, France, July 1986. Springer-Verlag.

Global Snapshots

[1] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[2] M. Fischer, N. Griffeth, and N. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, May 1982. Also, in *Proceedings of IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1981, 33-38.

D.5.6 Deadlock Detection

[1] D. Menasce and R. Muntz. Locking and deadlock detection in distributed databases. *IEEE Transactions on Software Engineering*, SE-5(3):195–202, May 1979.

[2] V. Gligor and S. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, SE-6(5):435–439, September 1980.

[3] R. Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, June 1982.

[4] G. Ho and C. Ramamoorthy. Protocols for deadlock detection in distributed database systems. *IEEE Transactions on Software Engineering*, SE-8(6):554–557, November 1982.

[5] K. Chandy, J. Misra, and L. Haas. Distributed deadlock detection. *ACM Transactions on Programming Languages and Systems*, 1(2):144–156, May 1983.

[6] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. *Distributed Computing*, 2:127–138, 1987.

[7] D. Mitchell and M. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 282–284, Vancouver, B.C., Canada, August 1984.

D.5.7 Consensus

[1] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one family faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[2] S. Moran and Y Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26:145–151, 1987.

[3] M. Bridgland and R. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, pages 52–63, August 1987. Revision in progress.

[4] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *Journal of Algorithms*, 11(3):420–440, September 1990. Earlier version in *Proceedings of 7th ACM Symposium on Principles of Distributed Computing,* pages 263-275, August 1988.

[5] Gadi Taubenfeld, Shmuel Katz, and Shlomo Moran. Impossibility results in the presence of multiple faulty processes. In C.E. Veni Madhavan, editor, *Proc. of the 9th FCT-TCS Conference*, volume 405 of *Lecture Notes in Computer Science*, pages 109–120, Bangalore, India, 1989. Springer Verlag. To appear in *Information and Computation.*

[6] Gadi Taubenfeld. On the nonexistence of resilient consensus protocols. *Information Processing Letters*, 37:285–289, March 1991.

[7] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3), July 1990.

[8] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[9] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings 10th ACM Symposium on Principles of Distributed Computing*, pages 257–272, Montreal, Canada, August 1991. Also, Cornell University, Department of Computer Science, Ithaca, New York TR 91-1225.

[10] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *Proceedings 11th ACM Symposium on Principles of Distributed Computing*, pages 147–158, Vancouver, Canada, August 1992. Also, Cornell University, Department of Computer Science, Ithaca, New York TR 92-1293.

D.5.8 Datalink

[1] A. Aho, J. Ullman, A. Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications*, 8(3):205–214, 1982.

[2] J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. *J. ACM*, 39(3):449–478, July 1992. Earlier version exists in *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, August, 1987.

[3] N. Lynch, Y. Mansour, and A. Fekete. The data link layer: Two impossibility results. In *Proceedings of 7th ACM Symposium on Principles of Distributed Computation*, pages 149–170, Toronto, Canada, August 1988. Also, Technical Memo MIT/LCS/TM-355, May 1988.

[4] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck. Reliable communication over unreliable channels. Technical Memo MIT/LCS/TM-447, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, 02139, October 1992.

[5] A. Fekete, N. Lynch, Y. Mansour, and J Spinelli. The data link layer: The impossibility of implementation in face of crashes. Technical Memo MIT/LCS/TM-355.b, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1989. Submitted for publication.

[6] A. Fekete and N. Lynch. The need for headers: an impossibility result for communication over unreliable channels. In G. Goos and J. Hartmanis, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 199–216. Springer-Verlag, August 1990. Also, Technical Memo MIT/LCS/TM-428, May, 1990.

[7] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck. Reliable communication over an unreliable channel, 1991. Submitted for publication.

[8] Da-Wei Wang and Lenore Zuck. Tight bounds for the sequence transmission problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 73–83, August 1989.

[9] Ewan Tempero and Richard Ladner. Tight bounds for weakly bounded protocols. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 205–212, Quebec, Canada, August 1990.

[10] Ewan Tempero and Richard Ladner. Recoverable sequence transmission protocols, June 1991. Manuscript.

D.5.9 Special-Purpose Network Building Blocks

[1] J. M. Hèlary and M. Raynal. *Synchronization and Control of Distributed Systems and Programs*. John Wiley & Sons, 1990.

[2] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish. A tradeoff between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, April 1990.

[3] Yehuda Afek, Eli Gafni, and Adi Rosén. The slide mechanism with applications in dynamic networks. In *Proceedings of the Eleventh Annual Symp. on Principles of Distributed Computing*, pages 35–46, Vancouver, British Columbia, Canada, August 1992.

D.5.10 Self-Stabilization

[1] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, November 1974.

[2] Edsger W. Dijksra. A Belated Proof of Self-Stabilization. *Distributed Computing*, 1(1):5-6, January, 1986.

[3] Geoffery M. Brown and Mohamed G. Gouda and Chuan-Lin Wu. Token Systems that Self-Stabilize. *IEEE Trans. Computers*, 38(6):845-852, June, 1989.

[4] James E. Burns and Jan Pachl. Uniform Self-Stabilizing Ring. *ACM Trans. Prog. Lang. and Syst.*, 11(2):330-344, April, 1989.

[5] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems, March 1992. To appear in *Distrib. Comput.* Also, earlier version in *Proceedings*

of the 9^{th} *Annual ACM Symposium on Principles of Distributed Computing*, August, 1990.

[6] Yehuda Afek and Geoffrey Brown. Self-stabilization of the alternating bit protocol. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 80–83, 1989.

[7] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self stabilizing message driven protocols. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, August 1991.

[8] Baruch Awerbuch and Boaz Patt-Shamir and George Varghese Self-Stabilization by Local Checking and Correction. *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, 268-277, October, 1991.

[9] Varghese G. *Dealing with Failure in Distributed Systems.* PhD thesis, MIT Dept. of Electrical Engineering and Computer Science, 1992. In progress.

D.5.11 Knowledge

[1] K. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

D.6 Timing-Based Systems

D.6.1 Resource Allocation

[1] H. Attiya and N. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. *Inf. Comput.*, 1993. To appear.

[2] N. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1992. To appear.

[3] Rajeev Alur and Gadi Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1992. To appear.

D.6.2 Shared Memory for Multiprocessors

[1] M. Mavronicolas. Timing-based distributed computation: Algorithms and impossibility results. Harvard University, Cambridge, MA, TR-13-92.

D.6.3 Consensus

[1] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[2] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *J.ACM*, 1992. To appear. Also, a preliminary version appeared in 23rd STOC, 1991.

[3] S. Ponzio. The real-time cost of timing uncertainty: Consensus and failure detection. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, June 1991. Also, MIT/LCS/TR 518.

[4] Stephen Ponzio. Consensus in the presence of timing uncertainty: Omission and byzantine failures. In *10th ACM Symposium on Principles of Distributed Computing*, pages 125–138, Montreal, Canada, August 1991.

D.6.4 Communication

[1] Da-Wei Wang and Lenore D. Zuck. Real-time STP. In *Proceedings of Tenth ACM Symp. on Principles of Distributed Computing*, August 1991. This is also Technical Report YALE/DCS/TR–856.

[2] S. Ponzio. Bounds on the time to detect failures using bounded-capacity message links. In *Proceedings of the 13th Real-time Systems Symposium*, Phoenix, Arizona, December 1992. To appear.

[3] S. Ponzio. The real-time cost of timing uncertainty: Consensus and failure detection. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, June 1991. Also, MIT/LCS/TR 518.

[4] Amir Herzberg and Shay Kutten. Fast isolation of arbitrary forwarding-faults. *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, August, 1989.

D.6.5 Clock Synchronization

[1] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, January, 1985.

[2] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77:1–36, 1988.

[3] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2-3):190–204, August/September 1984.

[4] Danny Dolev, Joseph Y. Halpern, Barbara Simons, and Ray Strong. Dynamic fault-tolerant clock synchronization, January 1992. IBM Research Report, RJ 8576 (77355). Also, earlier version appeared in *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, August, 1984.

[5] D. Dolev, J. Halpern, and R. Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of 16th Symposium on Theory of Computing*, pages 504–510, May 1984. Later version in *Journal of Computer and Systems Sciences*, 32(2):230-250, April, 1986.

[6] S. Mahaney and F. Schneider. Inexact agreement: accuracy, precision, and graceful degradation. In *Proceedings of 4th ACM Symposium on Principles of Distributed Computing*, pages 237–249, August 1985.

[7] M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

D.6.6 Applications of Synchronized Clocks

[1] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–9, Montreal, Canada, August 1991.

[2] Gil Neiger and Sam Toueg. Simulating synchronized clocks and common knowledge in distributed systems, July 1991. To appear in *J. ACM*. Earlier version exists in *Proceedings 6th ACM Symposium on Principles of Distributed Computing*, August 1987, Vancouver, Canada, 281-293.