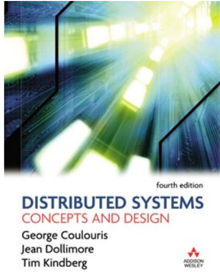


Slides for Chapter 15: Coordination and Agreement

From Coulouris, Dollimore and Kindberg
Distributed Systems:
Concepts and Design
Edition 5, © Pearson Education 2005

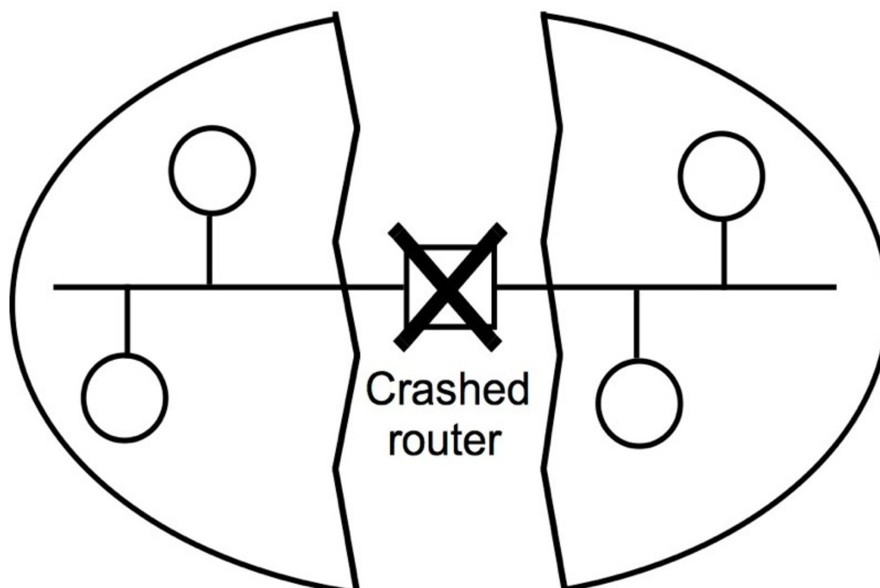
Coordination and agreement

- Given a set of distributed processes:
  - how to coordinate their actions?
  - how to reach an agreement on some value?
- E.g.: a cruise control, implemented by several redundant control processes: how to decide if the spaceship is functionally normally?
- We don't have master/slave relationships, in order to avoid single point of failures
- In distributed systems, both nodes and channels can fail
- (Surprising) negative results in presence of even benign failures

Failure assumptions

- Let us assume that network channels are reliable (but nodes are not)
  - Each channel **eventually** delivers a message to the recipient's input buffer
  - No time bound in asynchronous systems
- This covers also the case when a link is broken but then eventually repaired (e.g. a router crashes) - the messages are kept in queue
- During the crash, the network can be *partitioned*
  - intra-partition communication still possible
  - inter-partition communication is delayed (a lot)
  - processes may not be able to communicate at the same time
- In real network, connectivity may be *asymmetric* and *intransitive*

Figure 15.1
A network partition



Process failures

- Unless stated otherwise, a process fails by **crashing**
- More complex situations are arbitrary (Byzantine) failure (e.g. still working but malfunctioning, corrupted internal state, etc.)
- A **correct** process is one that exhibits no failures at any point of the execution
  - a process that suffers a failure is “non failed” before that point, not “correct”
- Problem: how to decide whether a process has failed (e.g. crashed)?
  - “ping” messages may be delayed

Failure detection

- A **failure detector** is a service that process queries about whether a particular process has failed
- Often implemented as a object local to the monitored process, that runs a failure-detection algorithm (watchdog, local failure detector)
- Failure detectors are usually inaccurate

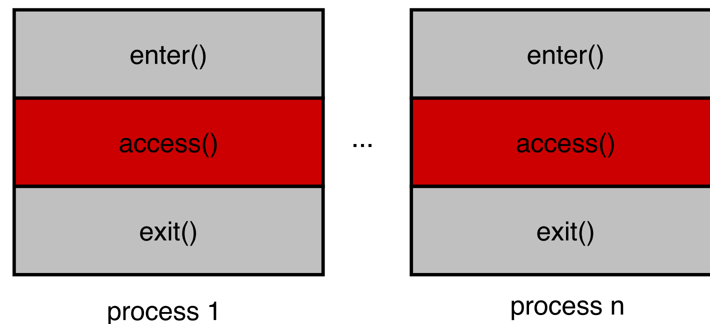
Failure detection

- An **unreliable** failure detector can answer
  - Suspected failure: something (apparently) is going wrong (but maybe it is not so)
  - Unsuspected failure: apparently it is still working (e.g. I have received a signal recently... but maybe it is failed after that.)
- A **reliable** failure detector can answer
  - Failed: the detector is sure that the process has crashed
  - Unsuspected failure: apparently it is still working (received a signal recently... but maybe it is failed after that)

Failure detection

- Failure detection is a knowledge local to a process
  - different processes may see different failures
- In asynchronous systems, can be implemented by “still-alive” messages (“ping”), each T seconds
  - the detectors claims “suspected failure” if it does not receive a message after T+D (D=network delay)
  - difficult to choose T: if too small: too many false failures; if too large: delays in discovering failures
- *Reliable failure detectors can be implemented only in synchronous systems* (where D is bounded)
  - Special channels (e.g. dedicated network links, serial ports)

Mutual exclusion



- n distributed processes, each with its own critical section
- processes may coordinate only by message passing
- aim: at most one process can execute its critical section

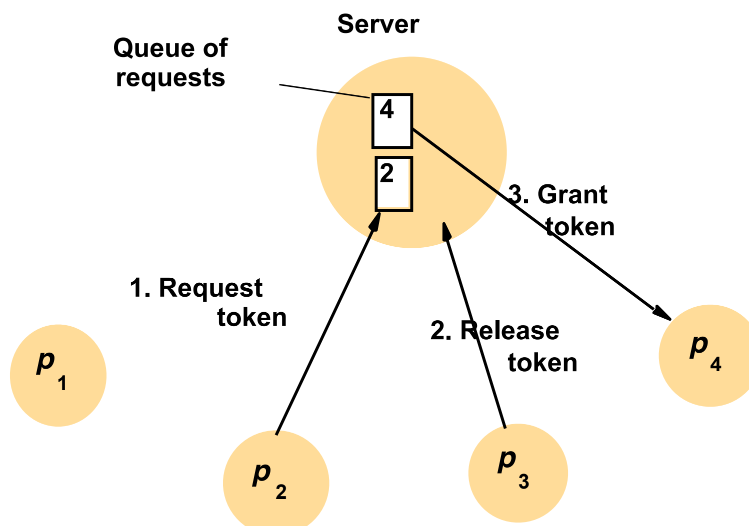
Requirements for Mutual Exclusion Algorithms in Message-Passing Based Distributed Systems

- ME1: at most one process may execute in the critical section at any given point in time (safety)
- ME2: requests to enter or exit the critical section will eventually succeed (liveness)
  - impossible for one process to enter critical section more than once while other processes are awaiting entry
- ME3: if one request to enter the critical section is issued before another request (as per the → relation), then the requests will be served in the same order

Mutual exclusion

- Performance criteria to be used in the assessment of mutual exclusion algorithms
  - bandwidth consumed (corresponds to number of messages sent)
  - client delay at each entry and exit
  - throughput: number of critical region accesses that the system allows
    - here: measured in terms of the synchronisation delay between one process exiting the critical section and the next process entering

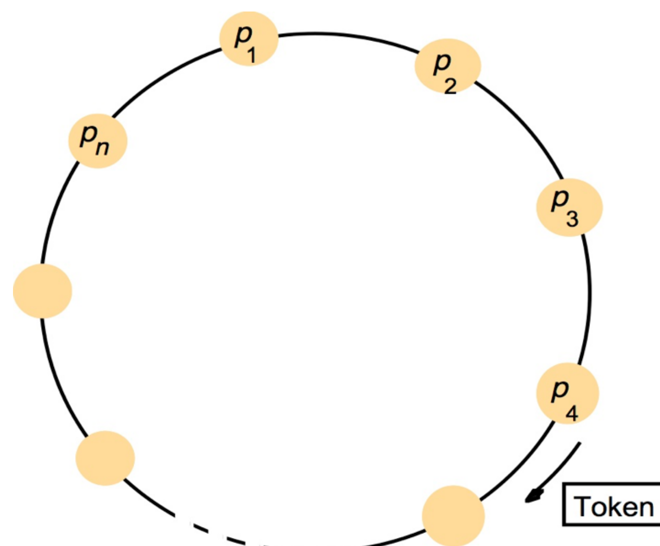
Figure 15.2
Server managing a mutual exclusion token for a set of processes



Central Server-based Algorithm

- central server receives access requests
  - if no process in critical section, request will be granted
  - if process in critical section, request will be queued
  - process leaving critical section
    - grant access to next process in queue, or wait for new requests if queue is empty
- Properties
  - satisfies ME1 and ME2, but not ME3 (network delays may reorder requests)
  - two messages per request, one per exit, exit does not delay process
  - performance and availability of server are the bottlenecks

Figure 15.3
A ring of processes transferring a mutual exclusion token



Ring-based Algorithm

- logical, not necessarily physical link: every process pi has connection to process p(i+1) mod N
- token passes in one direction through the ring
- token arrival
  - only process in possession of token may access critical region
  - if no request upon arrival of token, or when exiting critical region, pass token on to neighbor
- satisfies ME1 and ME2, but not ME3
- performance
  - constant bandwidth consumption
  - entry delay between 0 and N message transmission times
  - synchronization delay between 1 and N message transmission times

Algorithm by Ricart and Agrawala [1981]

- based on multicast
  - process requesting access multicasts request to all other processes
  - process may only enter critical section if all other processes return positive acknowledgement messages
- assumptions
  - all processes have communication channels to all other processes
  - all processes have distinct numeric ID and maintain logical clocks

Figure 15.4
Ricart and Agrawala's algorithm

On initialization
  state := RELEASED;
To enter the section
  state := WANTED;
  Multicast *request* to all processes;
  T := request's timestamp;
  Wait until (number of replies received = $(N - 1)$);
  state := HELD;

request processing deferred here

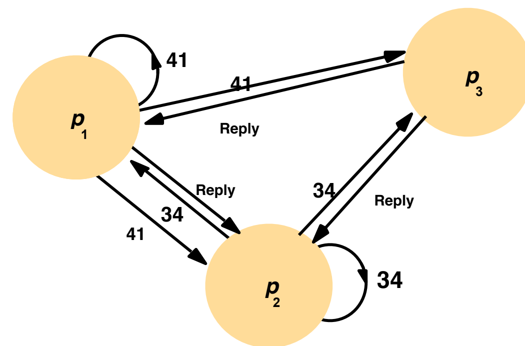
On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
  if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
  then
    queue *request* from p_i without replying;
  else
    reply immediately to p_i;
  end if
To exit the critical section
  state := RELEASED;
  reply to any queued requests;

Algorithm by Ricart and Agrawala

- if request is broadcast and state of all other processes is RELEASED, then all processes will reply immediately and requester will obtain entry
- if at least one process is in state HELD, that process will not reply until it has left critical section, hence mutual exclusion
- if two or more processes request at the same time, whichever process' request bears lower timestamp will be the first to get N-1 replies
- in case of equal timestamps, process with lower ID wins

Figure 15.5
Multicast synchronization

- p3 not attempting to enter, p1 and p2 request entry simultaneously
- p3 replies immediately
- p2 receives request from p1, timestamp(p2) < timestamp(p1), therefore p2 does not reply
- p1 sees its timestamp to be larger than that of the request from p2, hence it replies immediately and p2 is granted access
- p2 will reply to p1's request after exiting the critical section



Algorithm by Ricart and Agrawala

- algorithms satisfies ME1
  - two processes p_i and p_j can only access critical section at the same time in case they would have replied to each other
  - since pairs $<T_i, p_i>$ are totally ordered, this cannot happen
- algorithms also satisfies ME2 and ME3
- performance
  - getting access requires 2(N-1) messages per request
  - synchronization delay: just one round-trip time (previous algorithms: up to N)

Maekawa's Voting Algorithm

- observation
  - to get access, not all processes have to agree
  - suffices to split set of processes up into subsets ("voting sets") that overlap
  - suffices that there is consensus within every subset
- model:
  - processes p1, .., pN
  - voting sets V1, .., VN chosen such that ∀ i,k and for some integer M:
    - pi ∈ Vi
    - Vi ∩ Vk ≠ ∅ (some overlap in every voting set)
    - | Vi | = K (fairness: all voting sets have equal size)
    - each process pk, is contained in M voting sets
  - Optimal solution: M=K=sqrt(N)

Figure 15.6 Maekawa's algorithm

On initialization
  state := RELEASED;
  voted := FALSE;
For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;
On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;
On receipt of a release from p_i at p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

Maekawa's Algorithm

- to obtain entry to critical section, pi sends request messages to all K-1 members of voting set Vi
- cannot enter until K-1 replies received
- when leaving critical section, send release to all members of Vi
- when receiving request
  - if state = HELD or already replied (voted) since last request, then queue request else immediately send reply
- when receiving release
  - remove request at head of queue and send reply

Maekawa's Algorithm

- performance
  - bandwidth utilization: 2√N per entry, √N per exit, total 3√N
  - is better than Ricart-Agrawala for N>4
  - client delay: same as for Ricart-Agrawala
  - synchronization delay: round-trip time instead of single-message transmission time in Ricart-Agrawala
- satisfies ME1
  - if possible for two processes to enter critical section, then processes in the non-empty intersection of their voting sets would have both granted access
  - impossible, since all processes make at most one vote after receiving request

Maekawa’s Algorithm

- but deadlocks are possible
  - consider three processes with V1 = {p1, p2}, V2 = {p2, p3}, V3 = {p3, p1}
  - possible to construct cyclic wait graph
  - p1 replies to p2, but queues request from p3; p2 replies to p3, but queues request from p1; p3 replies to p1, but queues request from p2
- algorithm can be modified to ensure absence of deadlocks
  - use of logical clocks to tag requests
  - processes queue requests in happened-before order
  - means that ME3 is also satisfied

Mutual exclusion: Notes on Fault Tolerance

none of these algorithms tolerates message loss

- **Central-Server**: tolerates crash failure of node that has neither requested access nor is currently in the critical section
- **Ring algorithms** cannot tolerate single crash failure
- **Ricart-Agrawala** algorithm can be modified to tolerate crash failures by the assumption that a failed process grants all requests immediately
  - requires reliable failure detector
- **Maekawa’s** algorithm can tolerate some crash failure
  - if process is in a voting set not required, rest of the system not affected
  - using failure detector, failed systems can be removed from voting sets

Election algorithms

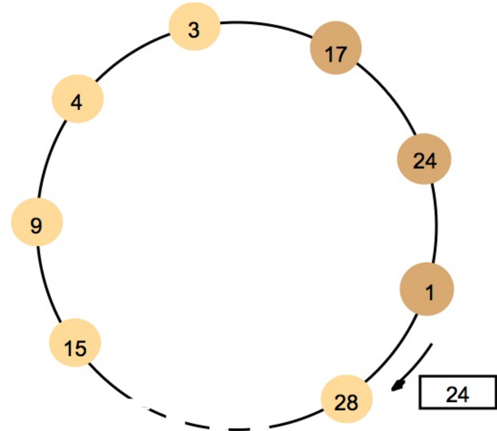
- algorithm designed to designate one unique process out of a set of processes with similar capabilities to take over certain functions in a distributes system
  - central server for mutual exclusion
  - ring master in token ring networks
  - bus master
- necessary when
  - system is booted
  - server fails
  - server retires

Election algorithms

- properties to be valid during any particular run of the system
  - E1 (safety) : a process p_i has $elected_i = \perp$ (undefined) or $elected_i = P$ for some non-crashed process P that will be chosen at the end of the run with the largest identifier
  - E2 (liveness): all processes p_i will eventually set $elected_i \neq \perp$
- performance
  - network bandwidth utilization (proportional to total number of messages sent)
  - turnaround time: number of serialized message transmission times between initiation and termination of a single run

Election algorithms - Ring-based Algorithm [Chang-Roberts 1979]

- assumptions
  - all nodes communicate on uni-directional ring structure
  - all processes have unique integer id
  - asynchronous, reliable system



The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

Election algorithms - Ring-based Algorithm [Chang-Roberts 1979]

- initially, all processes are marked “non-participant”
- to begin election, process place election message with own identifier on ring and marks itself “participant”
- upon receipt of election message, compare received identifier with own
  - if received id greater than own id, forward message to neighbor
  - if received id smaller than own id,
    - if own status is “non-participant”, then substitute own id in election message and forward on ring
    - otherwise, do not forward message (already “participant”)
  - if received id is identical to own id
    - this process’s id must be greatest and it becomes elected
    - marks own status as “non-participant”
    - sends out “elected” message with its own id
- upon any forwarding, mark own state as “participant”
- when receiving “elected” message and own state is “participant”
  - mark own status as “non-participant”
  - set $elected_i$ appropriately and forward elected message

Election algorithms - Ring-based Algorithm

- Ring-based Algorithm properties
  - E1 satisfied, since all identifiers are compared
  - E2 follows from reliable communication property
- performance
  - at worst 2N-1 messages for electing the left-hand neighbour
  - another N elected messages
  - worse turnaround time is 3N-1 messages
- failures
  - tolerates no failures
  - practically useful only with reliable failure detectors

Election algorithms - The Bully Algorithm

- works for **synchronous** networks
  - nodes can crash, and crashes will be detected reliably
- assumptions
  - each node knows identifiers of all other nodes
  - every node can communicate with every other node
- message types
  - *election*: announce an election
  - *answer*: reply to an election message
  - *coordinator*: announce identity of elected process

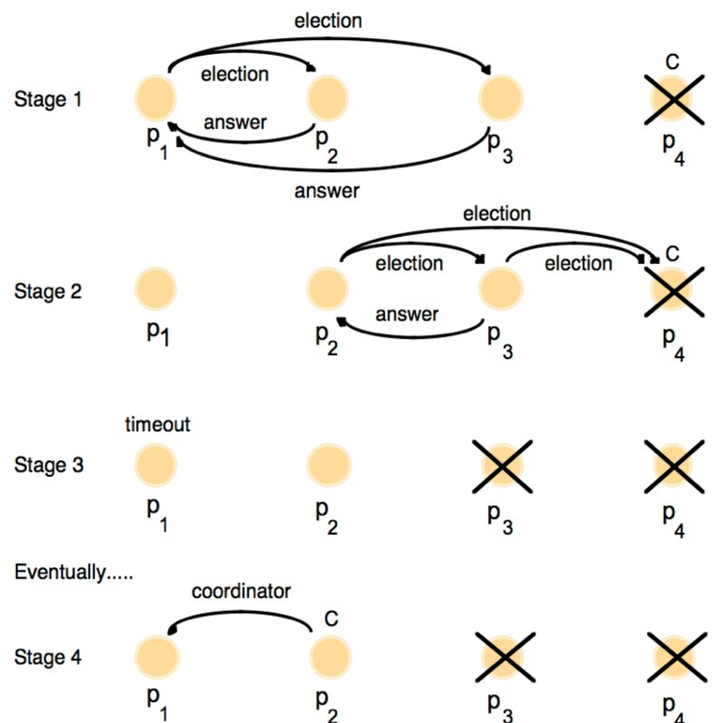
Election algorithms - The Bully Algorithm

- initiation of algorithm: a process detects that the current coordinator has failed (via a reliable failure detection)
  - a peer process failed if no answer to request within $T = 2T_{trans} + T_{process}$
- process can decide whether to become coordinator by comparing own id with all other ids (highest wins)
  - announce by sending *coordinator* message to all other nodes with lower id
- process with lower id can bid to become coordinator by sending election message to all processes with higher ID
  - if no response within T, considers itself elected coordinator, sends *coordinator* message to all processes with lower id
  - otherwise, wait for another T' time units for a *coordinator* message to arrive from new coordinator
    - if no response, then begin another election process

Election algorithms - The Bully Algorithm

- process receiving election message sets variable $election_i$ to the id of the coordinator received in the election message
- if process receives election message, sends back an answer message and begins another election - unless one was already initiated
- new process replacing crashed process
  - starts a new election
  - if highest id, will immediately send coordinator message and "bully" current coordinator to resign

Figure 15.8
The bully algorithm



The current coordinator is p4, and p1 has detected its failure. So it starts a new election, which leads to the election of p2, after the failure of p3.

Election algorithms - The Bully Algorithm

- properties
  - E1 satisfied (if no process replaced and timeout T estimate accurate)
  - E2 satisfied (synchronous network, reliable transmission)
  - E1 not satisfied if crashed process replaced at the same time while another process has announced that it is the new coordinator
- performance
  - best case: process with the second highest identifier detects coordinators failure
    - elects itself coordinator and sends N-2 coordinator messages
  - worst case: when lowest id detects failure, requires $O(N^2)$ messages
    - N-1 processes with higher IDs start election

Multicast

- group communication
  - sending and delivery of messages to more than one recipient
    - receiving of message: queueing of arriving message in network interface buffer
    - delivery of message: passing message from network interface buffer to to target application
  - membership in recipient group transparent to sender
    - one send operation to one address without having to send individual messages to all receipients

Multicast

- issues
  - addressing
  - coordination
    - guarantees that messages are received by a group of recipients
    - delivery ordering amongst group members
- uses of multicast
  - Computer Supported Collaborative Work (CSCW)
  - communication with replicated servers (to achieve fault-tolerance)
  - event notification in networks
  - discovery services in spontaneous networking

IP-based multicast

- only implemented by some IP routers
- available for UDP transport service
- addressing: multicast address and port number
- IP multicast group
  - class D IP address for which first 4 bits are 1110 in IPv4
  - membership is dynamic
  - computer belongs to multicast group if one or more processes have sockets that belong to a multicast group

IP-based multicast

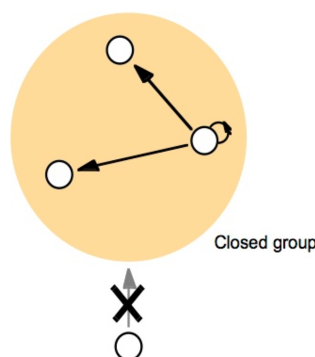
- implementation of multicast IP routers
  - on local area networks, use LAN's multicast capabilities (e.g., Ethernet)
    - use locally valid multicast address, set Time To Live (TTL) counter in IP header to 1 so that packet will never get routed outside LAN
  - in the Internet, router forwards messages to all other routers that have members in the multicast group, which in turn forward the datagrams to group members
- no guarantees whatsoever
  - message loss, reordering, duplication, etc.

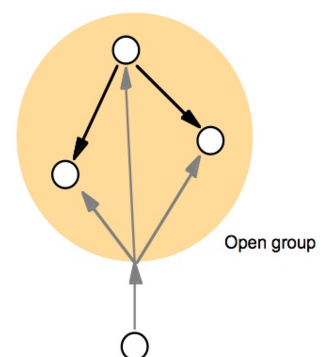
Properties of multicast

- achieves not only transparency, but also enables stronger guarantees than "delivery by hand"
- efficient use of network hardware
  - router sends individual messages
  - uses tree-like distribution structure if avialabe
    - two UDP-datagrams to same subnet: two IP packets, the first delays the second
    - with IP multicast capabilities, only one IP datagram transmitted iuse of LAN-based multicast capabilities, if available
  - delivery guarantees

System model

- message m: contains ID of sender and of destination group
  - multicast(g, m): multicast message m to group g
  - deliver(m): delivery of a message at recipient
    - called by recipient
- multicast group is
  - closed, if multicast only within
  - open, if processes not member of the group may send to it





Basic Multicast

- guaranteed delivery, unless multicaster crashes
- primitives and implementation
  - B-multicast(g, m): for each process p ∈ g, send(p, m)
  - B-deliver(m) at p: when receive(m) at p, for all p
  - problem in using concurrent one-to-one send(p, m) operations
- ack-implosion:
  - all recipients acknowledge receipt at about same time
  - buffer overflow leads to dropping of ack messages
  - retransmits, even more ack messages

Reliable Multicast

- primitives
  - R-multicast(m, g)
  - R-deliver(m)
- desired properties
  - **integrity:** a correct process p delivers a message at most once, and the delivered message is identical to the message sent in the multicast send operation (safety)
  - **validity:** if a correct process multicasts message m, then it will eventually deliver m (liveness)
  - **agreement:** if a correct process delivers a message m, then all other correct processes in the target group of message m will also deliver message m
  - (additionally) **uniform agreement:** if a process, no matter whether it is correct or fails, delivers a message m, then all correct processes in the group will deliver m as well

Reliable multicast: notes

- validity is expressed in terms of self-delivery, for simplicity reasons
- validity and agreement amount to overall liveness requirement: if one process (the sender) delivers a message m, then m will eventually be delivered to all the group's correct members
- agreement is similar to "atomicity": all-or-nothing semantics

Reliable multicast: implementation

- B-multicast to processes in group
- R-deliver
- properties
  - **validity:** a correct process will eventually B-deliver to itself
  - **integrity:** based on underlying communication medium
  - **agreement:** B-multicast to all other processes after B-deliver
- inefficient, since each message is sent |g| times to each process

Figure 15.9
Reliable multicast algorithm

On initialization
  Received := {};

For process p to R-multicast message m to group g
  B-multicast(*g*, *m*);   // $p \in g$ is included as a destination

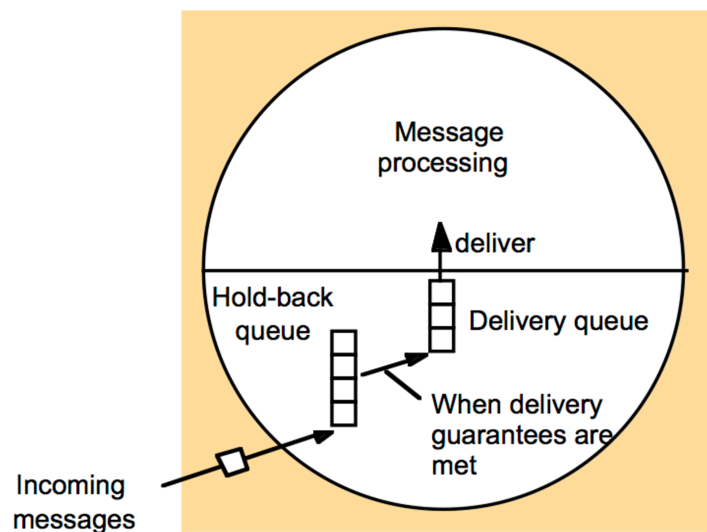
On B-deliver(*m*) *at process q with g = group*(*m*)
  if ($m \notin$ *Received*)
  then
    Received := *Received* $\cup$ {*m*};
    if ($q \neq p$) *then B-multicast*(*g*, *m*); *end if*
    R-deliver m;
  end if

Reliable Multicast over IP Multicast

- R-IP-multicast based on observation, that multicast successful in most cases
  - use negative acknowledgement to indicate non-delivery
- Basic idea
  - closed multicast groups
  - S_g^p: sequence number for group g that process p belongs to
  - R_g^p: sequence number of latest message that a process has delivered from process p and that was sent to group g
  - p R-multicasts message to group g
    - piggy back onto message:
      - S_g^p
      - acknowledgements $\langle q, R_g^q \rangle$ for all q
  - IP-multicast message and piggy back information
  - increment S_g^p by one

Figure 15.11
The hold-back queue for arriving multicast messages

- R-deliver message from p
  - only if received sequence number $S = R_g^p+1$
  - then increment R_g^p by 1
  - retain any message that cannot yet be delivered in hold-back-queue



Reliable multicast over IP - basic idea

- R-deliver message from p
  - if $S \leq R_g^p$, then message is already delivered, discard
  - if $S > R_g^p$ or $R > R_g^p$ for any enclosed acknowledgement <q, R>, then receiver has missed one or more messages, requests retransmit through negative acknowledgement
- properties
  - integrity: follows from detection of duplicates and properties of IP multicast (e.g., checksum to detect message corruption)
  - validity: message loss can only be detected when a successor message is eventually transmitted
  - requires processes to multicast messages indefinitely
- agreement
  - requires unbounded history for broadcast messages so that retransmit is always possible
- there exist practical variants that ensure validity and agreement

Ordered multicast

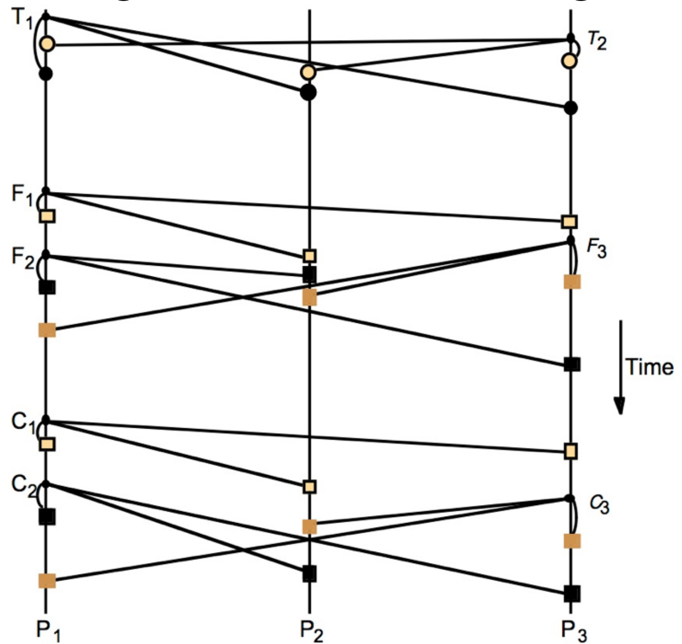
- assume: every process belongs to at most one group
- properties
  - **FIFO ordering:** if a correct process issues a multicast(g, m) and then multicast(g, m'), then every correct process that delivers m' will deliver m before m'
  - **causal ordering:** if multicast(g, m) → multicast(g, m'), where → is induced by message passing only, then every correct process that delivers m' will deliver m before m'
  - **total ordering:** if a correct process delivers m before it delivers m', then any other correct process that delivers m' will deliver m before m'

Ordered multicast

- notes
  - causal ordering implies FIFO ordering
  - FIFO ordering and causal ordering are partial (pre-)orders
  - total order allows arbitrary ordering of deliver events relative to multicast events, as long as this order is identical in all correct processes
  - atomic multicast: reliable, totally ordered multicast

Figure 15.12
Total, FIFO and causal ordering of multicast messages

Notice the consistent ordering of totally ordered messages T_1 and T_2, the FIFO-related messages F_1 and F_2 and the causally related messages C_1 and C_3 – and the otherwise arbitrary delivery ordering of messages.



Ordered Multicast: implementing FIFO ordering

- S_g^p: sequence number for group g that process p belongs to
- R_g^p: sequence number of latest message that a process has delivered from process p and that was sent to group g
- assumption: non-overlapping groups
- FO-multicast(m, g)
  - **B-multicast(m, g, $< S_g^p >$)**
  - **increment S_g^p by 1**
- upon receipt of a message at q with sequence number S
  - **if $S = R_g^p+1$, then this is the next message**
    - **therefore FO-deliver(m) and $R_g^p := S$**
  - **if $S > R_g^p+1$, then place message on hold-back queue until intervening messages have been delivered and $S = R_g^p+1$**

Ordered Multicast: implementing total ordering

- idea: assign totally ordered identifiers to multicast messages so that every process makes the same delivery decision based on these identifiers
- delivery similar to FIFO delivery, only that group-specific sequence numbers rather than process-specific sequence numbers are used
- assumption: non-overlapping groups
- two main methods for the assignment of identifiers
  - sequencer
  - collective agreement on the assignment of message identifiers

Ordered Multicast: implementing total ordering

- sequencer:
  - process wishing to TO-broadcast attaches a unique identifier id(m) to the message
  - message is sent to sequencer as well as all members of g
  - sequencer maintains group-specific sequence number s_g which it uses to assign increasing and consecutive sequence numbers to the messages it B-delivers
  - announces the order in which members of g have to deliver these messages using a B-multicasted order message

Figure 15.14
Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
B-multicast($g \cup \{sequencer(g)\}$, <*m, i*>);

On B-deliver(<*m, i*>) *with* $g = group(m)$
Place <*m, i*> in hold-back queue;

On B-deliver(m_{order} = <"order", *i, S*>) *with* $g = group(m_{order})$
wait until <*m, i*> in hold-back queue and $S = r_g$;
TO-deliver m; // (after deleting it from the hold-back queue)
$r_g = S + 1$;

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver(<*m, i*>) *with* $g = group(m)$
B-multicast(*g*, <"order", *i*, s_g>);
$s_g := s_g + 1$;

Ordered Multicast: implementing total ordering

- sequencer is bottleneck (performance and/or reliability)
- alternative: collective agreement on the assignment of message identifiers
  - implemented in the ISIS toolkit
  - groups may be open or closed
  - receiving processes bounce proposed sequence numbers to sender
  - sender returns agreed sequence numbers
  - each process q in group g maintains
    - A_g^q: the largest agreed sequence number it has observed so far for group g
    - P_g^q: own largest proposed sequence number

Ordered Multicast: implementing total ordering

- algorithm for collective agreement on the assignment of message identifiers
- p B-multicasts <m, i> to g, where i is unique identifier for m
- each recipient q replies to g with proposal for agreed sequence number
  - $P_g^q := \max(A_g^q, P_g^q) + 1$
  - each process q provisionally assigns own proposed sequence number to message and queues message in hold back queue, ordered according to proposed sequence number
- p chooses largest proposed number as sequence number a
- p B-multicasts <i, a> to g
- each process q in group
  - sets $A_g^q := \max(A_g^q, a)$
  - reorders received message in hold-back queue if received sequence number differs from proposed number
- only when message at head of hold-back queue is assigned an agreed sequence number, it will be queued in delivery queue

Figure 15.15
The ISIS algorithm for total ordering [Birman 87]

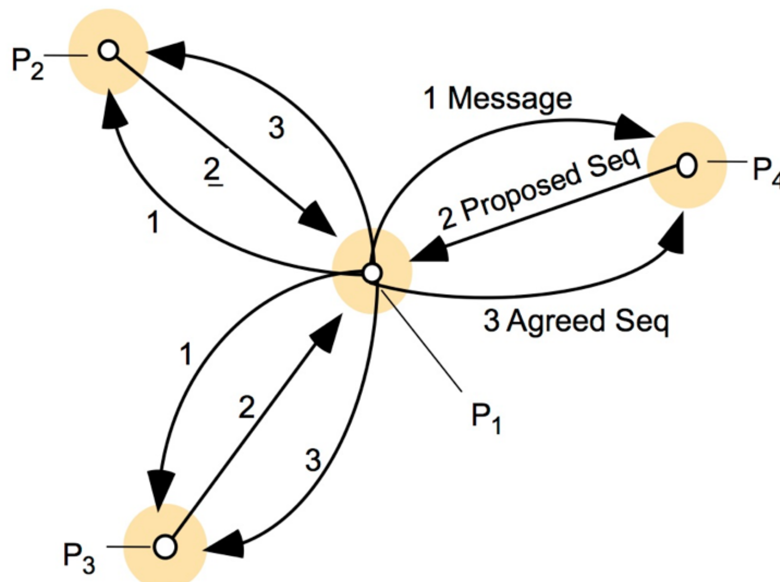


Figure 15.16
Causal ordering using vector timestamps

Algorithm for group member p_i ($i = 1, 2 \ldots, N$)

On initialization
$V_i^g[j] := 0$ ($j = 1, 2 \ldots, N$);

To CO-multicast message m to group g
$V_i^g[i] := V_i^g[i] + 1$;
B-multicast(g, $< V_i^g, m >$);

On B-deliver($< V_j^g, m >$) *from* p_j, *with* $g = group(m)$
place $< V_j^g, m >$ in hold-back queue;
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k]$ ($k \ne j$);
CO-deliver m;   // after removing it from the hold-back queue
$V_i^g[j] := V_i^g[j] + 1$;

Consensus and related problems

- all correct computers controlling a spaceship should decide to proceed with landing, or all of them should decide to abort (after each has proposed one action or the other)
- in an electronic money transfer transaction, all involved processes must consistently agree on whether to perform the transaction (debit and credit), or not
- in mutual exclusion, processes need to agree on which process enters critical section
- in election, processes need to agree on elected process
- in totally ordered multicast, processes need to agree on a consistent message delivery order

Recall process failure models

- crash failures: processes stop (fail), but remain silent
- byzantine failures: processes fail, but may still respond to environment with arbitrary, erratic behavior (e.g., send false acknowledgements, etc.)

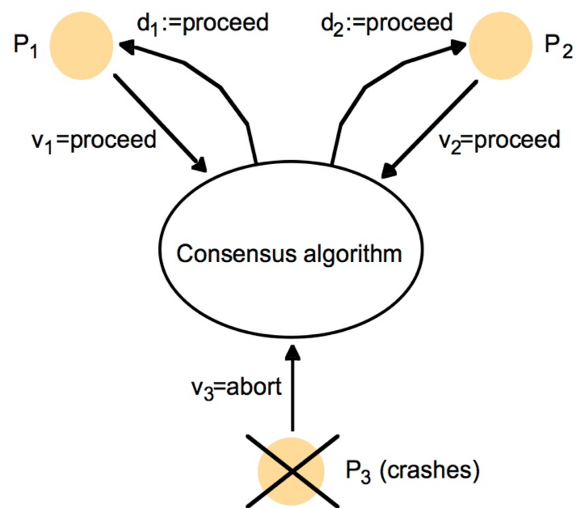
Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send*, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Consensus

- Factors threatening consensus
  - failures
    - communication link or process failures
    - crash failures (fail-silent) or byzantine failures (arbitrary)
  - network characteristics (synchronous or asynchronous)
  - failure detectors (reliable or unreliable)
  - are messages authenticated (digitally signed) or not
    - can a process lie about the content of message that it received from a correct process?
    - can adversary claim to send message under a false expedient's id?
- Model
  - processes communicating by message passing
  - aim: reaching consensus even in the presence of faults
    - assumption: communication is reliable, but processes may fail

The Consensus Problem (C)

- agreement in the value of a decision variable amongst all correct processes
  - p_i is in state undecided and proposes a single value v_i
  - next, processes communicate with each other to exchange values
  - in doing so, p_i sets decision variable d_i and enters the decided state after which the value of d_i remains unchanged



Properties of a consensus algorithm

- **termination**: eventually, each correct process sets its decision variable
- **agreement**: for all correct p_i and p_k such that state(p_i) = state(p_k)= decided: $d_i = d_k$
- **integrity**: if the correct processes all proposed the same value, then all correct process has chosen that value in the decided state
  - variation: ... then *some* correct process has chosen that value in the decided state

Solve consensus in a failure-free environment

- each process reliably multicasts proposed values
- after receiving response, solves consensus function
  majority(v1,.., vN)
  which returns most often proposed value, or undefined if no majority exists
  - [remark: other problem-specific functions possible]
- properties
  - termination guaranteed by reliability of multicast
  - agreement, integrity: definition of majority, and integrity of reliable multicast (all processes solve same function on same data)

Solve consensus with failures?

- when crashes occur
  - how to detect failure?
  - will algorithm terminate?
- when byzantine failures occur
  - processes communicate random values
  - evaluation of consensus function may be inconsistent
- malevolent processes may deliberately propose false or inconsistent values

The Byzantine Generals Problem (BG)

- three or more generals are to agree on an attack or retreat
- commander issues order
  - others (lieutenants to the commander) have to decide to attack or retreat
  - one of the generals may be treacherous
- if commander is treacherous, it proposes attacking to one general and retreating to the other
- if lieutenants are treacherous, they tell one of their peers that commander ordered to attack, and others that commander ordered to retreat

The Byzantine Generals Problem (BG)

- difference to consensus problem: one process supplies a value that others have to agree on
- properties
  - **termination**: eventually each correct process sets it decision variable
  - **agreement**: the decision value of all correct processes is the same
  - **integrity**: if the commander is correct, then all processes decide on the value that the commander proposes
    - note: implies agreement only if the commander is correct, but commander need not be correct (see above)

Interactive Consistency (IC)

- each process suggests one value
- goal: all correct processes agree on a vector of values, each component corresponding to one processes' agreed value
  - example: agreement about each processes' local state
- requirements
  - **termination:** eventually each correct process sets it decision variable
  - **agreement:** the decision vector of all correct processes is the same
  - **integrity:** if pi is correct, then all correct processes decide on vi as the i-th component of their vector

Relationship of Consensus to Other Problems

- assume that the previous problems could be solved, yielding the following decision variables
  - $C(v_1,.., v_N)$ returns the decision value of p_i
  - $BG_i(k, v)$ returns the decision value of p_i where p_k is the commander which proposes value v
  - $IC_i(v_1,.., v_N)[k]$ returns the k-th value in the decision vector of p_i where $v_1,.., v_N$ are the values that the processes propose
- possibilities to derive solutions from these problem solutions
  - IC from BG: run BG N times, once with each pi acting as commander $IC_i(v_1,.., v_N)[k] = BG_i(k, v_k)$

Relationship of Consensus to Other Problems

- C from IC:
  - run IC to produce a vector of values at each process
  - apply an appropriate function on the vector's values to derive a single value
    $C_i(v_1,.., v_N) = majority(IC_i(v_1,.., v_N)[1],.., IC_i(v_1,.., v_N)[N])$
- BG from C
  - commander pk sends its proposed value v to itself and each of the remaining processes
  - all processes run C with the values $v_1,.., v_N$ that they receive
  - $BG_i(k, v) = C_i(v_1,.., v_N)$
- termination, agreement and integrity preserved in each case

Relationship of Consensus to Other Problems

- solving consensus equivalent to solving reliable, totally ordered multicast (RTO)
  - implementing consensus with RTO-multicast
    - collect all processes in one group
    - each p_i performs RTO-multicast(g, v_i)
    - each p_i chooses $d_i = m_i$, where mi is the first value that the RTO-multicast delivers
    - properties
      * termination follows from reliability of multicast
      * agreement and integrity follow from reliability and total ordering
  - implementing RTO-multicast from consensus can be shown as well

Consensus in a synchronous system

- assumption: no more than f of the N processes crash
- Dolev-Strong algorithm proceeds in f+1 rounds
  - processes B-multicast values between them
  - at the end of f+1 rounds, all surviving processes are in a position to agree

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization
  $Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f + 1)$
  B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent
  $Values_i^{r+1} := Values_i^r$;
  while (in round r)
  {
    On B-deliver(V_j) *from some* p_j
      $Values_i^{r+1} := Values_i^{r+1} \cup V_j$;
  }

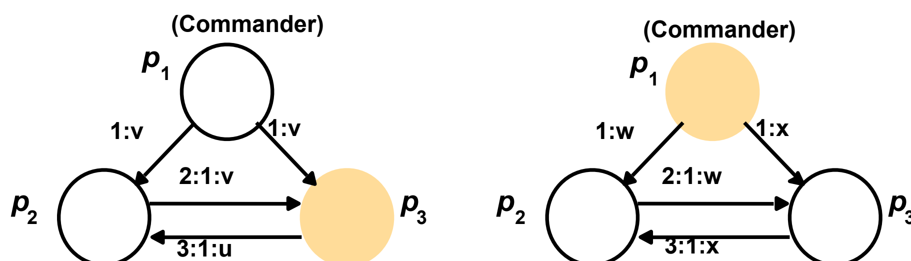
After $(f+1)$ *rounds*
  Assign $d_i = minimum(Values_i^{f+1})$;

- Dolev-Strong terminates (due to timeouts)
- and it is correct: after f+1 rounds, surviving processes reach the same set of values
  - $Values_i^r$ holds the set of proposed values known to p_i at the beginning of round r
- it can be shown that in synchronous systems, any algorithm to reach consensus, tolerating up to f crash or byzantine failures, requires at least f+1 rounds

Byzantine Generals Problem in Synchronous Network

- allow arbitrary (byzantine) failures
- up to f faulty processes
- correct processes can detect the absence of a message through timeout, but cannot conclude that sender has crashed, since it may be silent for some time and then start sending messages again
- assume private communication channels
  - **fourth process cannot detect if one process sends messages with different content to two peers**
  - **no faulty process can inject messages into channels connecting correct processes**
- assume that messages are not digitally signed (authenticated and verifiable)
- general result (Lamport, Shostak and Pease, ACM TOPLAS 1982)
- – no solution if $N \leq 3f$
  – give an algorithm for $N \geq 3f+1$

Figure 15.19
Three byzantine generals



Faulty processes are shown coloured

Notation: ":" reads "says".
E.g.: "3:1:u" is the message "3 says 1 says u"

Byzantine Generals Problem in Synchronous Network: impossibility for N = 3 processes

- both scenarios show two rounds of messages
- left: all p2 knows is that it has received two different values
- right: same situation, even though now commander is faulty
- assume a solution existed
  - p2 would have to decide on value v, by integrity condition of BG
- assume that no algorithm can decide locally for p2 between the two scenarios
  - then p2 would need to decide on w (value sent by commander) in right hand scenario
- same reasoning for p3
  - will have to decide for commander's value, which is a violation of agreement in right hand scenario, hence contradiction

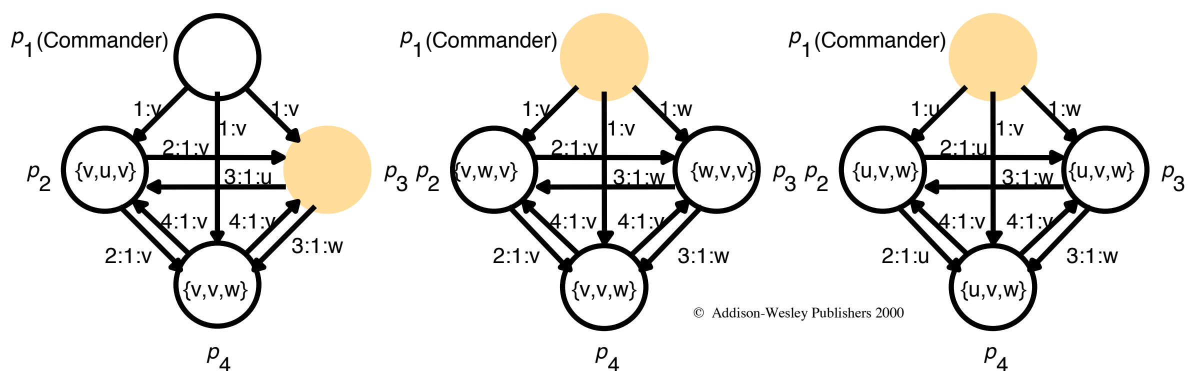
Byzantine Generals Problem in Synchronous Network: solution for N ≥ 3f+1 processes

- general solution by Pease, Shostak and Lamport too complex to present here
- we will present solution for N = 4, f = 1
- correct generals reach agreement in two rounds:
  - first, commander sends value to each lieutenant
  - second, each lieutenant sends value it received to all peers
- lieutenant receives
  - value from commander
  - N-2 values from peers

Byzantine Generals Problem in Synchronous Network: solution for N ≥ 3f+1 processes

- if commander faulty, then all lieutenants correct, each will have gathered exactly the set of values that the commander sent out
- if one lieutenant faulty, each of its peers receives N-2 copies of the value the commander sent out, plus the faulty lieutenant value
- to reach agreement, simple majority function suffices
  - **since N ≥ 4, N-2 ≥ 2, majority function will ignore value of faulty lieutenant, and produce value of commander if commander is correct (will produce ⊥ if commander incorrect)**
  - **other functions can be used; e.g., median ($v_1,\ldots,v_n$) if these values come from an ordered set**
- note: BG requires agreement only if commander correct

Figure 15.20
Four byzantine generals



© Addison-Wesley Publishers 2000

p_2: majority({v,u,v}) = v
p_3: majority({v,v,w}) = v

p_2: majority({v,w,v}) = v
p_3: majority({v,v,w}) = v
p_4: majority({w,v,v}) = v

p_2, p_3, p_4:
majority({v,u,w}) = ⊥

General Pease-Shostak-Lamport algorithm for Byzantine Generals

- We define a family of algorithms BG_f, where f=number of faulty processes
- BG_0: (in this case, no faulty processes)
  1. The commander sends a value *v* to all lieutenants
  2. Each lieutenants uses the *v* received from commander
- BG_{n+1}:
  1. The commander sends a value to each lieutenant
  2. For each i, let v_i the value that process p_i receives from commander. Process p_i uses this values to execute BG_n with all the other n-1 lieutenants
  3. For each i, for each j≠i, let v_j the value that p_i received from p_j at step 2 (using BG_n). Then, p_i chooses the value majority(v_1,…,v_n)

Byzantine Generals Problem in Asynchronous Network: impossibility of agreement

- previous algorithms: synchrony assumption
  - **message exchanges in rounds**
  - **timeouts**
- in asynchronous systems, no algorithm can guarantee reaching consensus, even with just one process crash failure (Fischer, Lynch and Paterson, 1985)
- proof idea
  - **show that there is always some continuation of the process's execution that avoids consensus being reached**

Impossibility of Agreement in Asynchronous Systems

- consequences
  - in asynchronous systems, no solution to BG, IC, RTO-multicast
  - of course, in practice consensus can often be reached, but a residual probability that consensus cannot be reached remains
- possible approaches to reaching consensus by weakening system assumptions
  - partial synchrony
  - masking faults
  - modified failure detectors
  - randomized algorithms
  - Satoshi agreement (used in blockchain)

Bitcoin and the Blockchain

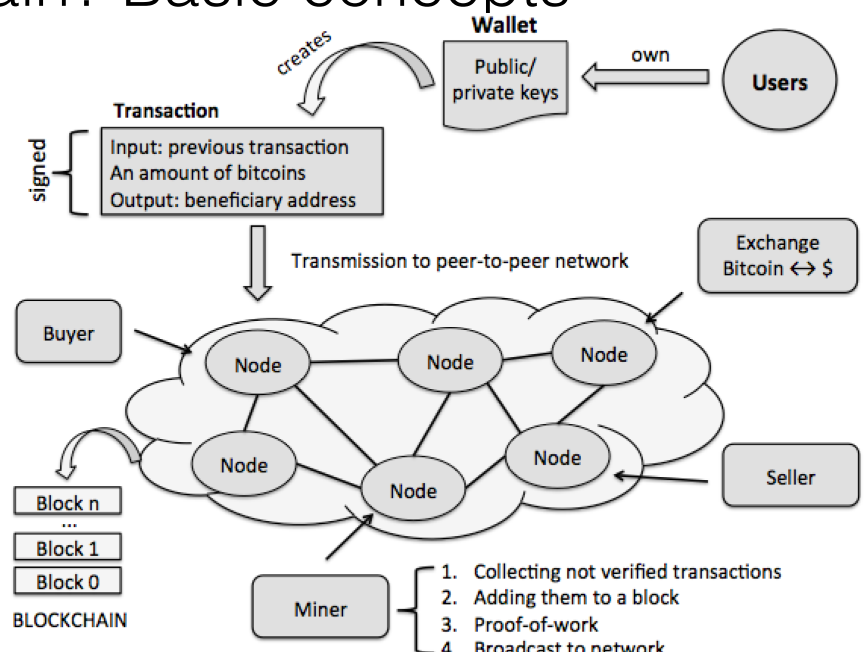
- Bitcoin is a cryptocurrency
  - "Bitcoin" can refer to:
    - Bitcoin (uppercase) - the protocol, software, and community
    - bitcoins (conventionally lowercase) - the unit
- "blockchain" - umbrella term for the datastructure shared among the nodes (the "public ledger")
- Nodes *asymptotically* reach an agreement on the content of the blockchain
  - Inconsistency ("forks", i.e. different evolutions) may happen, but are resolved by a "majority vote", where majority is in terms of computational power (Satoshi agreement)

Bitcoin: The Protocol

- **Distributed public ledger** of **transactions**
- shared with peer-to-peer technology
- Transactions specify the **ownership transfer** of a native digital scriptural asset
- a “digital token” that can be exchanged, but not duplicated
- keeps records of each and every transaction forever
- It could replace any processing central authority with decentralized peer-to-peer cryptographically secure equivalent

What is Blockchain? Basic concepts

- **Transactions:** Transfers of bitcoin from input **addresses** to output **addresses**
- **Blocks:** Timestamped collection of transactions.
- **Miner:** Agent which validates transactions and puts them into blocks
- **Blockchain:** The entire series of blocks 'chained' together
  - Miners compete to add blocks, the “winner” is compensated with bitcoins



Basic Concepts - Identity in Bitcoin

- Bitcoin is really **pseudonymous**, not anonymous
- More similar to «anonymous bank accounts» in fiscal paradises…
- …but whose transactions are visibile to everyone!
- **All transactions are transparent to everybody's inspection.**
- The bitcoin address does not provide direct information about the bitcoin owner
- **Perfect persistent public account history**: the public ledger is forever
- Exchange must identify customers, so if you lost anonymity, they can track all transactions, from the beginning.

Basic Concepts - Identity in Bitcoin

- The pseudonym is generated randomly directly by the user (not issued by any central authority) in "wallets" using **Asymmetric Cryptography**
- Two mathematically linked keys perform opposite digital signature functions:
  - The private (secret) key, used to generate the signature
  - The public key, used by anyone to verify the signature
- The private key cannot be derived from the public one
- A bitcoin address is derived from a public key, but the public key cannot be derived from the address

  Private key -> public key -> bitcoin address
- The corresponding private key allows spending from that address

Basic Concepts - Identity in Bitcoin

- A user can generate as many keys (and addreses) he wants
- Could two users generate the same keys (and addresses)?
- No, in practice, because the amount of keys is HUGE
  - Example Address: 3EnQkjmt3Pv2Uyk8gG736xYKen9efED5LQ
  - 2^160 possible addresses (1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,976 addresses)
  - Grains of sand on earth: 2^63
  - If we «see a World in a Grain of Sand»: 2^126 is actually only 0.0000000058% of 2^160

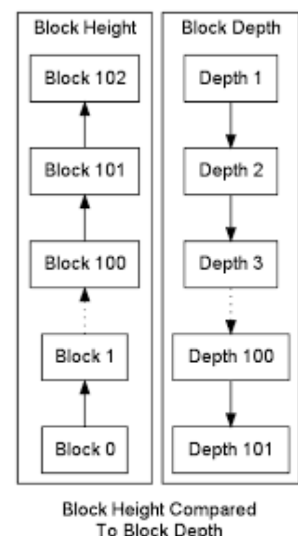
Basic Concepts - Blocks + Blockchain

Blocks

- Contains an ordered bunch of transactions
  - Timestamps the transactions, are immutable
- Each block references a previous block
- Each block has height and depth (confirmations)
  - Currently 558k blocks…and counting

Blockchain

- The entire series of blocks 'chained' together
- All nodes participating to the blockchain have a complete copy of the blockchain



Block Height Compared To Block Depth

A Bitcoin Transaction - Basic Version

- Bitcoin exists as software
  - Transactions are conducted through wallet software
  - Wallet creation generates a Bitcoin address
  - This is my Bitcoin address (in case you wanna give me tip): 3EnQkjmt3Pv2Uyk8gG736xYKen9efED5LQ
- To receive money, you share your address
  - Sender specifies address and amount
- The transaction is broadcast to the network, where "miners" verify it and add it to the transaction history
- Once validated, the transaction is stored foreve
- Possibly a note can be added to the transaction (and stored as well)

Send Funds

Recipient

Email or bitcoin address

Amount

0.00 BTC

My Wallet 0.8635703 BTC

Note

Write an optional message

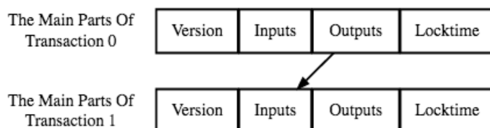
Send Funds

Coinbase interface

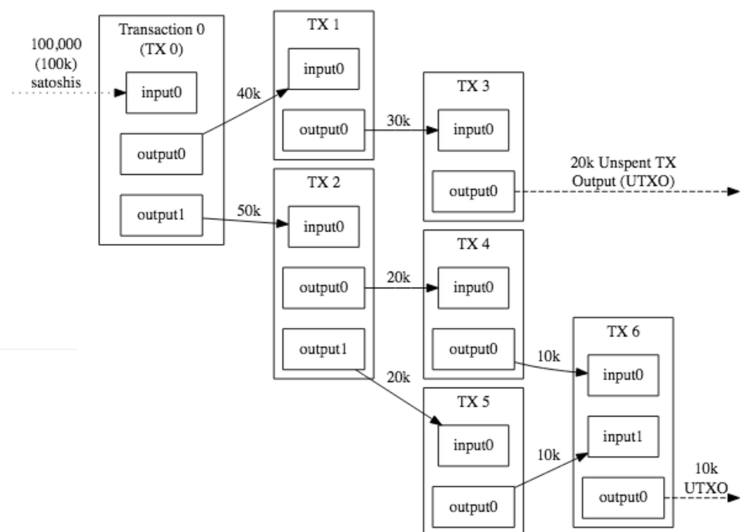
Basic Concepts - Transactions

- Maps inputs addresses to output addresses
  - Outputs can only be spent once
- Typical tx: one input, two outputs
- Fees are implicit

Each input spends a previous output



Each output waits as an Unspent TX Output (UTXO) until a later input spends it



Triple-Entry Bookkeeping (Transaction-To-Transaction Payments) As Used By Bitcoin

BLOCKCHAIN info | Home | Charts | Stats | Markets | API | Wallet | Search | English

Transaction View information about a bitcoin transaction

447cb6623db32b5f28c94ac10551802075f053208fe995204a145197e2904bb9

3Nxwenay9Z8Lc9JBiywExpnEFiLp6Afp8v (44,000 BTC - Output)

Output	Amount
3LrLWTSdd69oZVVQ6dtWaAAaBLn7N3rRjz - (Spent)	333.33328889 BTC
3QkXtcSWJA9w77eCujnMBKDWFe7F7zwxTg - (Spent)	333.33328889 BTC
3Qd7hXZoZ1iyXZznrbdUwUQBxHMmujdqhJ - (Spent)	333.33328889 BTC
3ECJwvx9VgfotcUuEJMVNvmWnTGVMk179L - (Spent)	333.33328889 BTC
3BuQmbmdce3e31GEovq5SgowLdfMgJzLDE - (Spent)	333.33328889 BTC
3NwKLjJjzXSnBFQWokXRgBG3JeuF3bsnfE - (Spent)	333.33328889 BTC
3GEaT8ZRXELcjMSFvGro6eZcC5S1LSLZuN - (Spent)	333.33328889 BTC
35DVAzDtZDKAU94kFT9sxoscnuLCTxgwYc - (Spent)	333.33328889 BTC
3Nxwenay9Z8Lc9JBiywExpnEFiLp6Afp8v - (Unspent)	38,000 BTC
35mwqShnStDro6uEB4bmsgbyBo8en6Byfm - (Spent)	333.33328889 BTC
39pvSqfNcUosc8RGVWxyzKM3ny96a3uSkW - (Spent)	333.33328889 BTC
39QNJSgQg5JnBXAtbF8ezkDn72VqWdPZPJ - (Spent)	333.33328889 BTC
3L9qAGBQLbXkFAB2GpijnJXPScSVjuiJio - (Spent)	333.33328889 BTC
37WSkANPVUQ8uuktf8hv671CejRtBtQ4tJ - (Spent)	333.33328887 BTC
3EEwPZZ6pYRJSotCz9RBoVYPRnoWyGWEka - (Spent)	333.33328889 BTC
3C4ABC7iPcAAKBh6SJXfvUSDBew3abCtw3 - (Spent)	333.33328889 BTC
3HpQozfTzoXAsHf87m2mwJXUQ14LVtLgK4 - (Spent)	333.33328889 BTC
337RfngTLRTpU7RT9sKWQWDdmfcdmWnugi - (Spent)	333.33328889 BTC
3P2eoKr3vAeZhJcTzon3VFkv5r7DqSXW9G - (Spent)	333.33328889 BTC
	43,999.9992 BTC

Summary	
Size	1055 (bytes)
Received Time	2016-08-30 11:45:03
Included In Blocks	427512 (2016-08-30 11:51:09 + 6 minutes)
Confirmations	854 Confirmations
Relayed by IP	5.39.93.85 (whois)
Visualize	View Tree Chart

Inputs and Outputs	
Total Input	44,000 BTC
Total Output	43,999.9992 BTC
Fees	0.0008 BTC
Estimated BTC Transacted	333.33328887 BTC
Scripts	Hide scripts & coinbase

Basic Concepts - UTXO analogy

UTXOs stands for "Unspent Transaction Outputs"

- Global set of unspent bitcoins
- "I'm spending THIS bitcoin," not "I'm spending A bitcoin."

Analogous to real estate (land), or Rai stones of the Yap Islands

- Rai stones are carved and placed somewhere, then never moved
- Instead: common and shared agreement on change of ownership
- The same for land lots: a land lot does not move only the ownership changes (Cf. *Grundbuch)*

The Innovation of Satoshi Nakamoto

Bitcoin was created by Satoshi Nakamoto in 2009

- First ever decentralized, trustless system for transactions
  - A low cost financial system that only requires an internet connection
- Nakamoto solved the Double Spending problem
  - Prevent someone from spending the same asset twice
  - Solution? The blockchain + Proof of Work

Avoiding double spending: centralised ledger

In a centralized solution, a central bank manages transactions and balances

* We have to trust the bank

* Bottleneck and SPOF

Avoiding double spending: decentralised ledger

Decentralization: Making everyone the bank

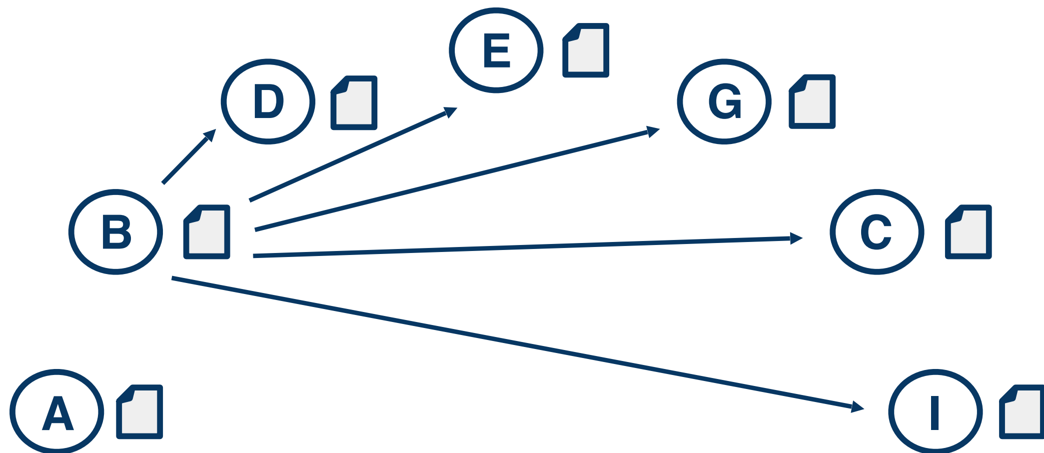
Every node has a complete copy of transactions

Avoiding double spending: decentralised ledger

Alice sends her transaction to Bob

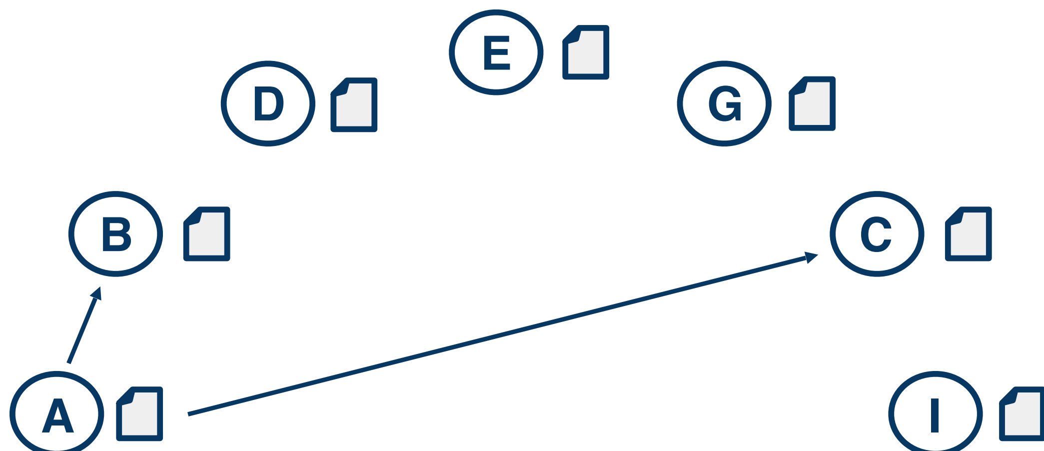
Avoiding double spending: decentralised ledger

Bob announces the transaction to the world
Every node updates its copy of the ledger



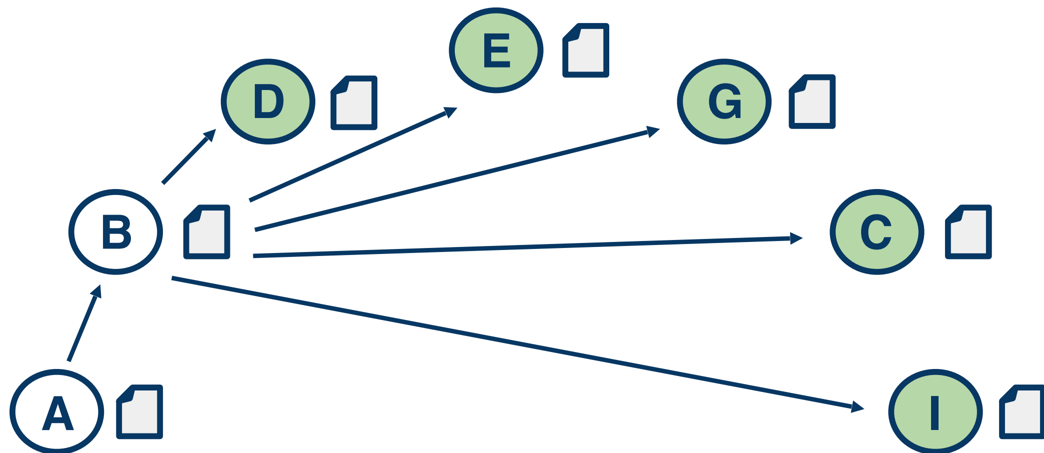
Avoiding double spending: decentralised ledger

But what if Alice double spends the same bitcoin on Bob and Charlie?



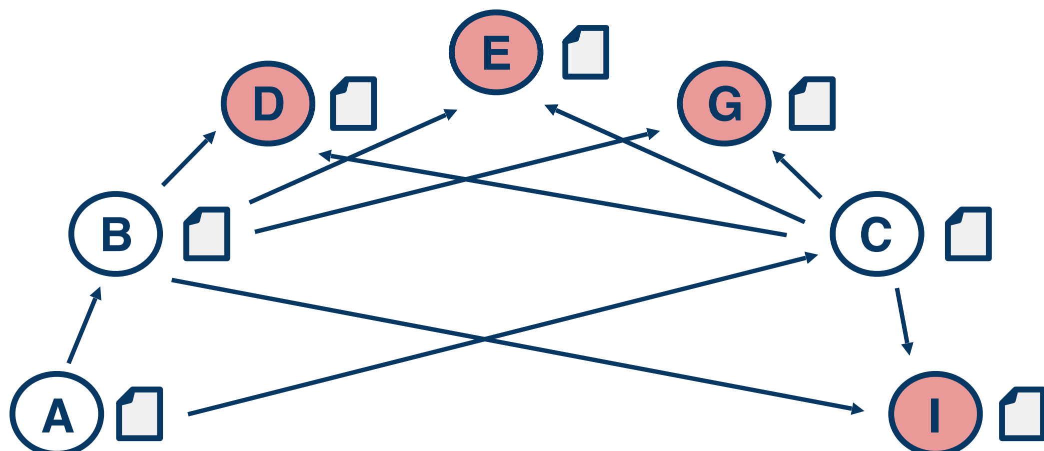
Avoiding double spending: decentralised ledger

Everyone verifies transactions: the first spending is accepted…



Avoiding double spending: decentralised ledger

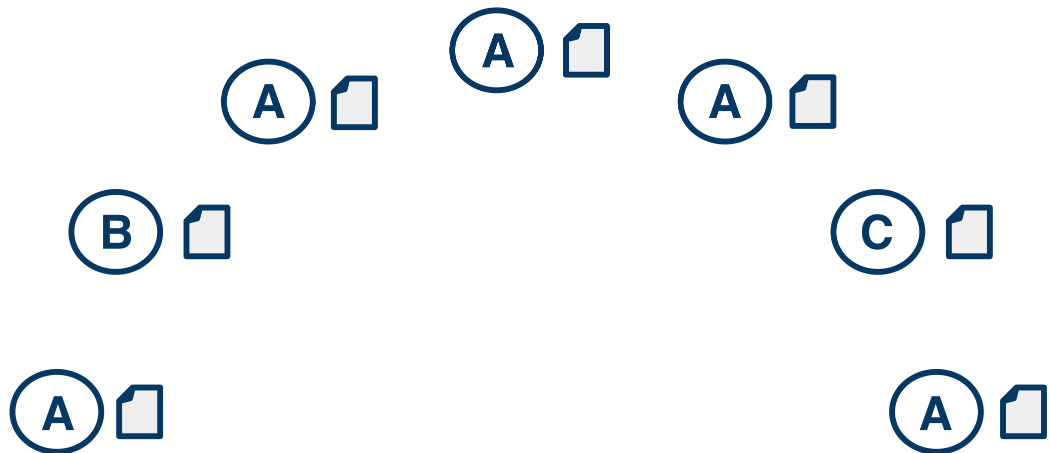
…but not the second, so Alice is prevented from double spending



Decentralised ledger is not enough…

"Sybil" attack:

Alice sets up multiple identities

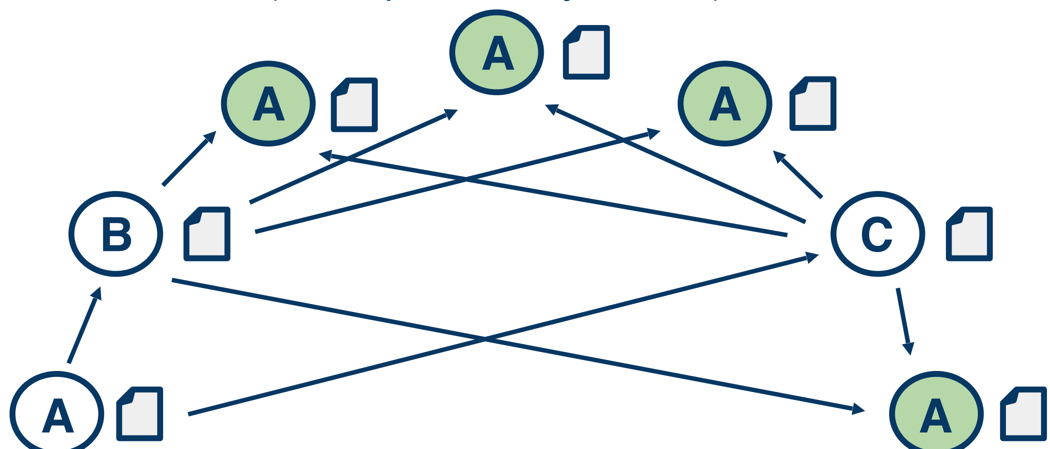


… if many nodes are byzantine

Alice double spends with her multiple identities

The fake identities confirm both spendings, so B and C are fooled!

(A's copies are byzantine)



Satoshi’s solution: Proof of Work for verifying transactions

The problem is that Alice can **too easily** generate incoherent confirmations.
Satoshi’s idea: **make verification hard**.
Let us suppose that a node, David, wants to validate and announce received a bunch of transactions to validate. To this end, he proceeds as follows.

1. David has to check his copy of the block chain to make sure the transactions are legitimate.
2. His computer has to use resources to solve a **hard mathematical puzzle**.
3. Only after the solution has been found, he can to announce the block of transactions to the network.
4. Every nodes can check (easily) that the transactions are legitimate and David has actually found a solution for the mathematical puzzle
5. If everything is fine, every node updates its blockchain (and David gains a reward)

Proof of work is a competition among nodes

- David has to solve the mathematical puzzle in order to avoid double spending
- Proof-of-work as a competition to verify transactions.
    - In Bitcoin, this is called ***mining.***
- If your computer solves the math puzzle before the others, you will verify the pending transactions (and receive some bitcoin as a reward).
- Proof-of-work prevents bad actors like Alice from double spending because it puts them in competition with everyone else trying to verify transactions.
- So as long as most of the computing power on the network is controlled by honest people, *more likely some honest node will win the race before Alice*
- Hence bad actors like Alice will have a hard time doing dishonest things.

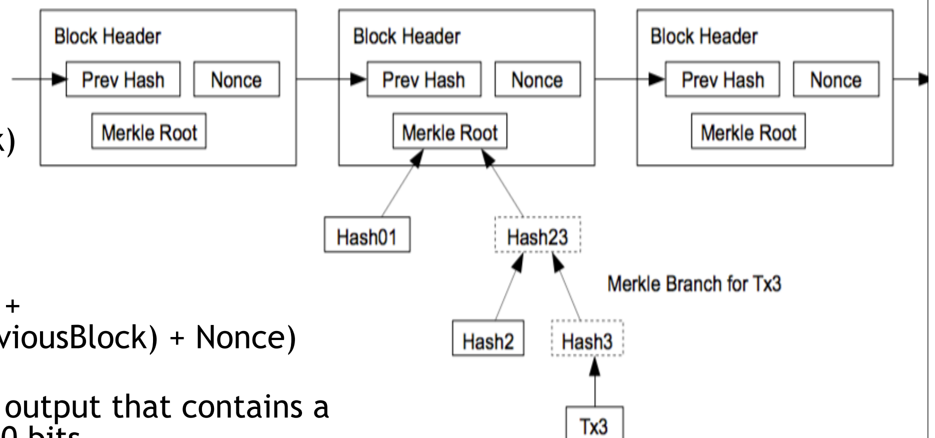
Sketch of Bitcoin Mining - The Mining Problem

Components hashed together:

- Merkle Root ('summary' of the transactions in the block)
- Hash of previous block
- Nonce

Formally:
Output = SHA-256(Merkle Root + SHA-256(PreviousBlock) + Nonce)

- Solution (Proof-of-work): an output that contains a requisite number of leading 0 bits
  - The number of 0 bits is the **difficulty**
  - Difficulty adjusts every every 2016 blocks (2 weeks) to maintain 1 block creation / 10 minutes

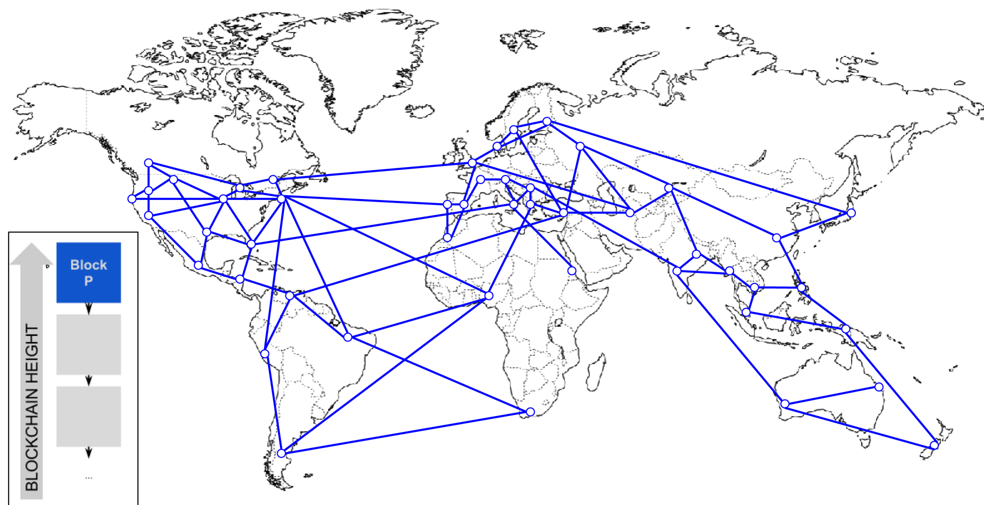


Sketch of Bitcoin Mining - Finding blocks

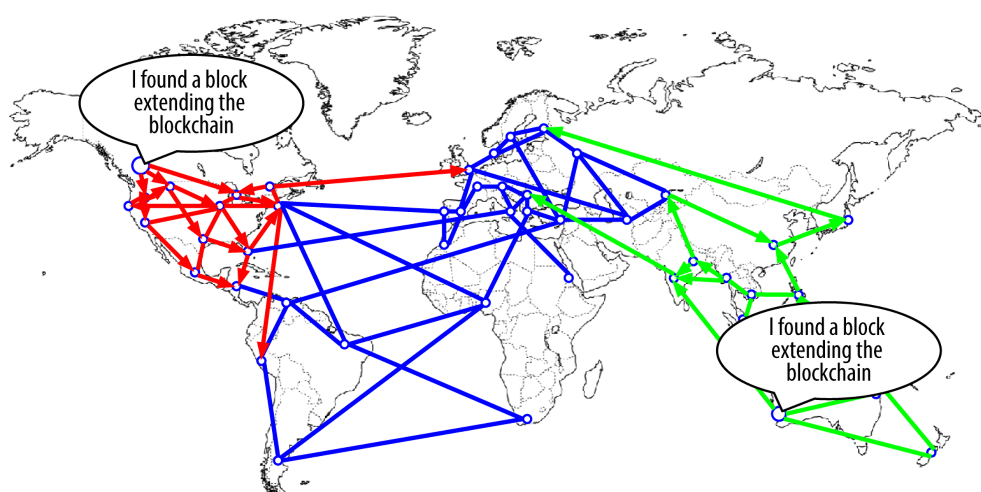
- Solving the PoW => 'found' a block; can add block to blockchain
  - Miner who found block adds "**coinbase transaction**"
    - contains mining reward (currently 12.5 BTC)
  - Miner broadcasts block
  - Other nodes verify, then add to their own copy of the blockchain
    - Verification is easy: just compute the SHA256 digest with the given nonce, and check that it gives a legit value
- This happens roughly every 10 minutes
  - Difficulty of the problem adjusted to keep block generation rate constant

A blockchain fork event: before the fork

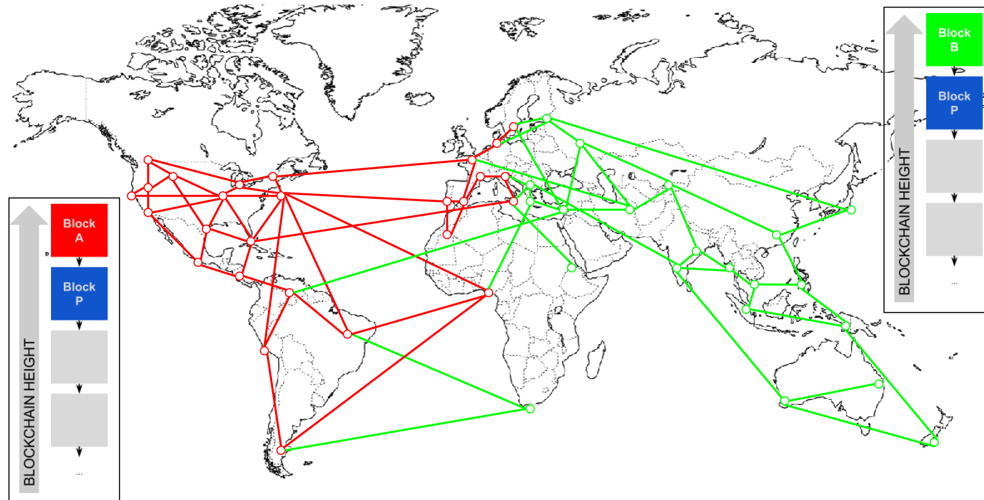
- A fork can occur when two miners publish blocks simultaneously. Such blocks are almost always in conflict.



Visualization of a blockchain fork event: two blocks found simultaneously

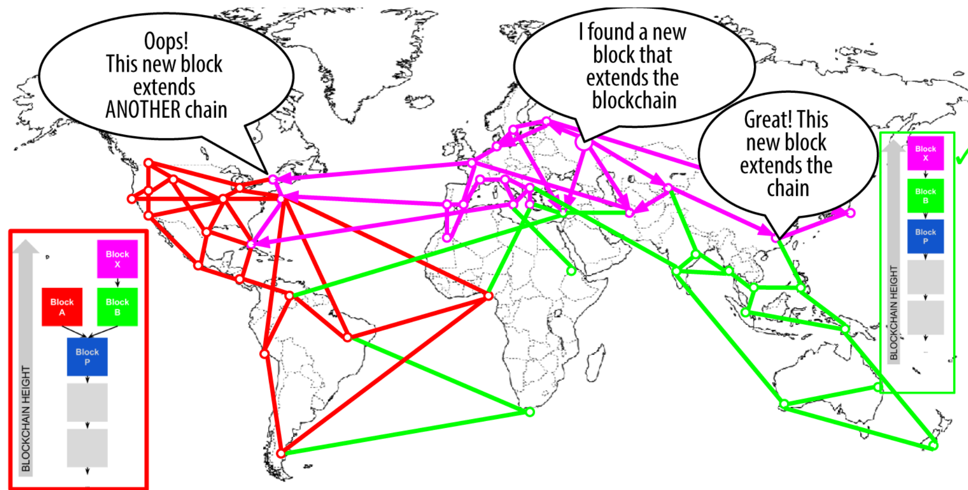


A blockchain fork event: two blocks propagate, splitting the network



Visualization of a blockchain fork event: a new block extends one fork

Visualization of a blockchain fork event: the network converges on a new longest chain



Sketch of Bitcoin Mining - 51% Attacks

Major assumption of Bitcoin:

Strictly less than 50% percent of the whole CPU power in the network is controlled by dishonest nodes

Under this assumption, the honest (CPU) majority will always form, eventually, the longest proof-of-work chain

51% Attack: Attempt to overwhelm the mining power of the network: if attacker nodes are able to assemble so much CPU power, they can «change» the history as they desires.

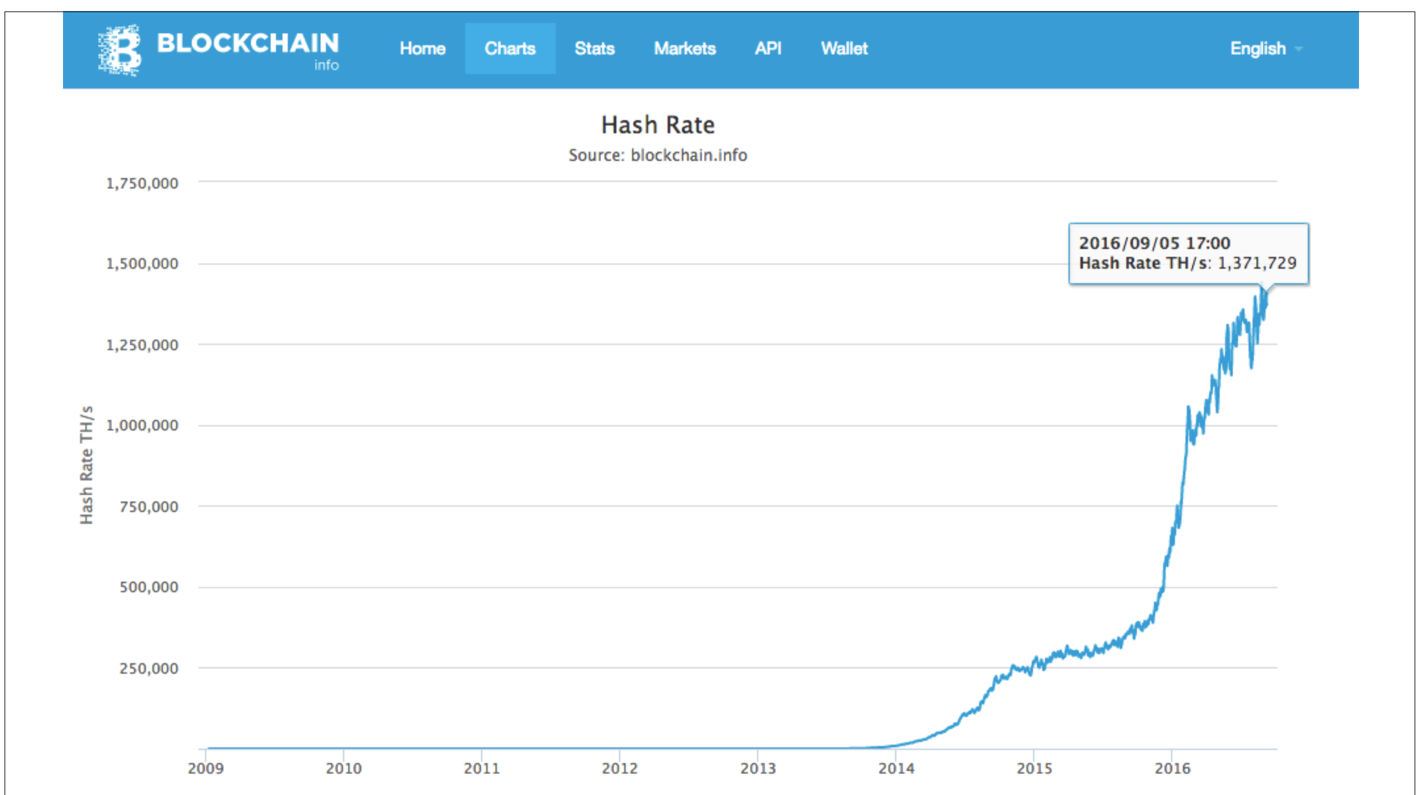
Sketch of Bitcoin Mining - 51% Attacks

However, if a group of nodes assembles so much CPU power, they have this dilemma:

1. Either use it to attack the system – but then nobody would use it anymore, and the value of bitcoins would become zero.
2. Or use it to mine bitcoins, gaining more bitcoins honestly.

If the nodes have interest IN the Bitcoin system, they are incentivized to act honestly – otherwise they would break the system.

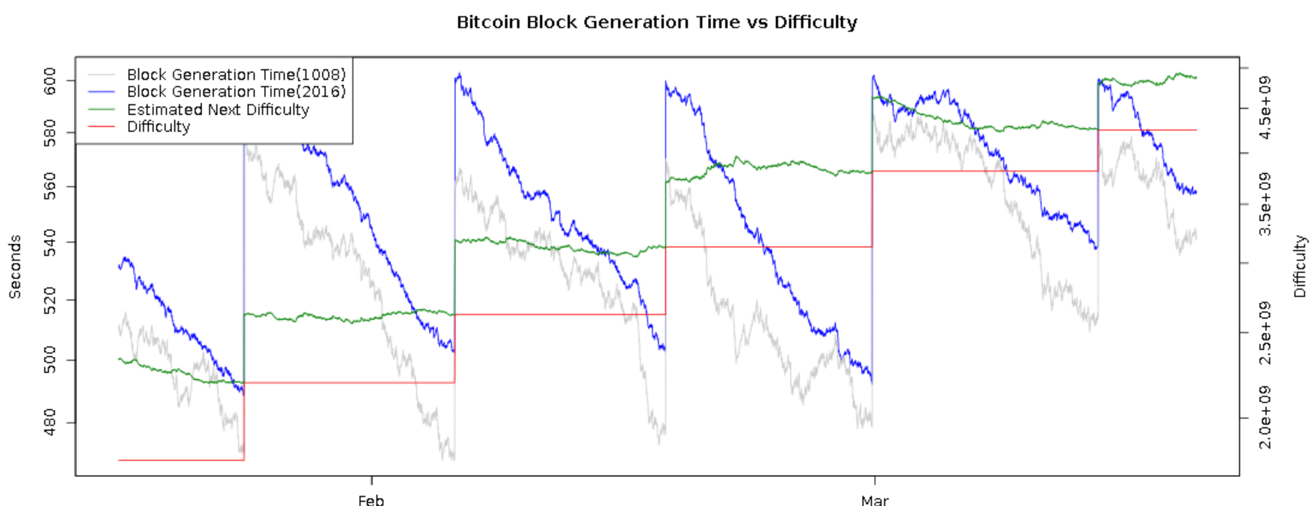
This is however a problem for altcoins – called *altcoin infanticide*
(e.g., a single Bitcoin miner, Luke-Jr., killed CoiledCoin by filling its blockchain with useless blocks much faster than honest Coiledcoin miners)



Sketch of Bitcoin Mining - Finding blocks

- Timeline + stats
  - This happens roughly every 10 minutes
    - Difficulty of the problem adjusted every 2 weeks
  - Block reward halving every 4 years
  - Bitcoin is in limited supply: 21 million bitcoins by 2141
    - Deflationary!
  - 80% have been already mined
  - After that, no further bitcoins will be created
  - And bitcoins may go (and have got) lost - forever!
    - Owned by addresses whose private key has got lost

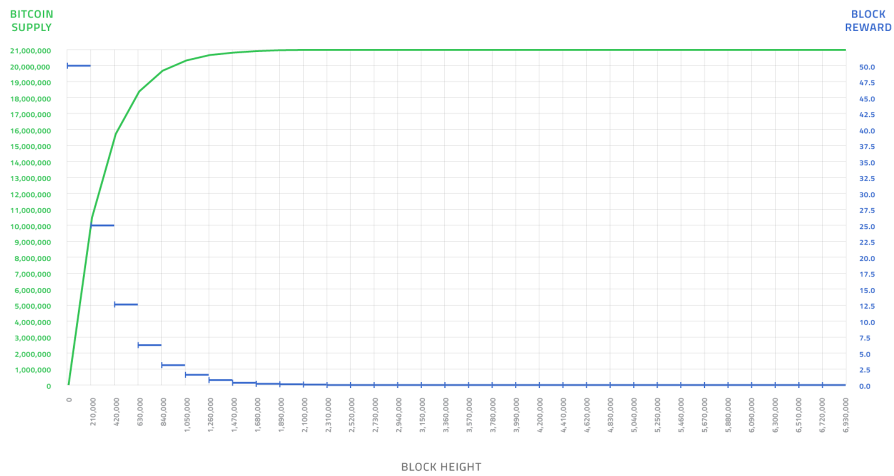
Block Reward :: Difficulty Adjustment



Block Reward :: Bitcoin Halving

Controlled Supply of Bitcoin

Number of bitcoins as a function of Block Height

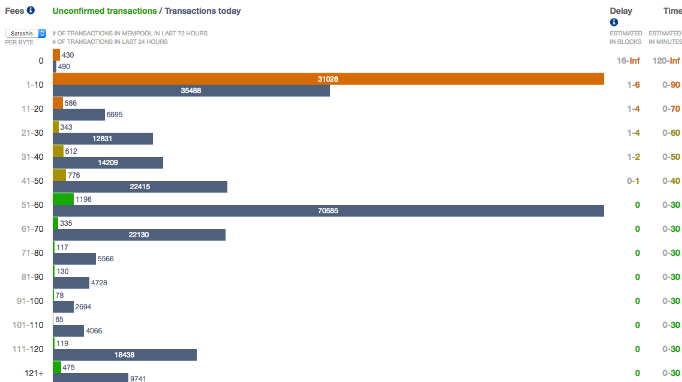


Transaction Fees

PREDICTING BITCOIN FEES FOR TRANSACTIONS.

WANT LOW FEES? TRY PAYMENT CHANNELS

LEARN MORE

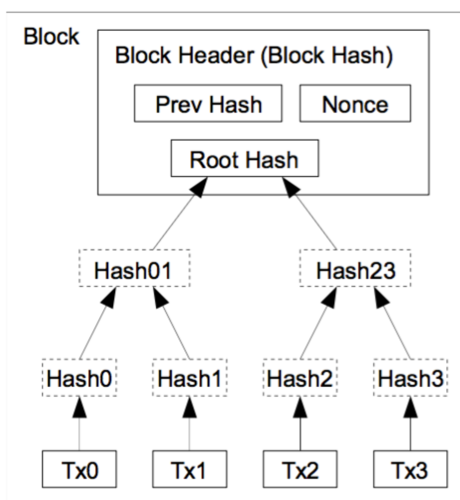


ı larger transaction fees
ıext block they're mining
after 2141

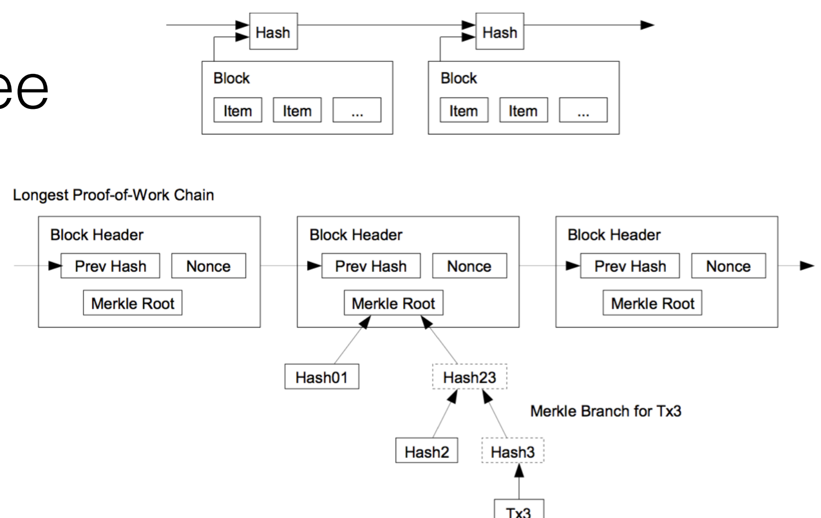
Bringing it all together - Back to a transaction

- I want to send money to Sunny
  - Sign transaction
  - Broadcast to network
- Miners receives transaction, adds to "**zero-conf pool**"
  - Verify transaction: i.e. signature matches, enough money, etc.
- Miner finds PoW, broadcasts block (with its reward inside)
  - Block propagates; others verify
- Miners work on the next problem

Bonus: Merkle Tree



Transactions Hashed in a Merkle Tree



- Makes transaction history immutable
- PoW to add chains

Source: Nakamoto 2009