DISTRIBUTED ALGORITHMS

AN INTUITIVE APPROACH | SECOND EDITION

WAN FOKKINK

Contents

A
Appendix: Pseudocode Descriptions

Pseudocode descriptions are presented for a considerable number of distributed algorithms discussed in the main body of this book. Several algorithms are excluded here either because their pseudocode descriptions are trivial or very similar to another algorithm that is included or because the main body contains a description that resembles the pseudocode.

Each piece of pseudocode is presented for a process p or p_i; its local variables are subscripted with p or i, respectively. We use $Neighbors_p$ to denote the set of neighbors of process p in the network and use *Processes* for the set of processes in the network.

Each pseudocode description starts with a variable declaration section. Let **bool**, **nat**, **int**, and **real** denote the data type of Booleans, natural numbers, integers, and reals, with default initial values *false* for the first data type and 0 for the latter three. The operations $\wedge$, $\vee$, and $\neg$ on Booleans denote conjunction, disjunction, and negation, respectively. The data type **dist**, representing distance, consists of the natural numbers extended with infinity ∞, where $\infty + d = d + \infty = \infty$ for all distance values d, and $d < \infty$ for all $d \neq \infty$; its default initial value is ∞.

The data type of processes in the network, **proc**, has default initial value $\perp$ (i.e., undefined). The data types **mess-queue** and **proc-queue** represent FIFO queues of basic messages and processes, respectively. Likewise,

mess-set, **proc-set**, **proc-nat-set**, **proc-real-nat-set**, and **proc-dist-set** represent sets of basic messages; processes; pairs of a process and a natural number; triples of a process, a natural number, and a real value; and pairs of a process and a distance value, respectively. Variables containing queues or sets have as default initial value $\emptyset$; that is, empty. There are three operations on queues: *head* produces the head and *tail* the tail of the queue (on the empty queue, these operations are undefined), while *append*(*Q, e*) appends element *e* at the end of queue *Q*.

We recall that assignment of a new value to a variable is written as $\leftarrow$. Equality between two data elements, $d_1 = d_2$ (or between two sets, $S_1 = S_2$), represents a Boolean value, which is *true* if and only if the two elements (or sets) are equal. We also recall that the network topology is supposed to be strongly connected. In the pseudocode, it is assumed that the network size N is greater than one.

A process is supposed to interrupt the execution of a procedure call (under a boxed text, such as "If *p* receives […]") only if it needs to wait for an incoming message, or in the case of a **while** *b* **do** *statement* **end while** construct, or when it enters its critical section.

In general, pseudocode tends to be error-prone because on the one hand it is condensed and intricate, while on the other hand it has never been executed. I welcome any comments on the pseudocode descriptions, as well as on the main body of the book.

A.1 Chandy-Lamport Snapshot Algorithm

The Boolean variable $recorded_p$ in the following pseudocode is set (to *true*) when *p* takes a local snapshot of its state. For each incoming channel *c* of *p*, the Boolean variable $marker_p[c]$ is set when a **marker** message arrives at *p* through *c*, and the queue $state_p[c]$ keeps track of the basic messages that arrive through channel *c* after *p* has taken its local snapshot and before a **marker** message arrives through *c*.

bool $recorded_p$, $marker_p[c]$ for all incoming channels *c* of *p*;
mess-queue $state_p[c]$ for all incoming channels *c* of *p*;

If *p* wants to initiate a snapshot

perform procedure $TakeSnapshot_p$;

If p receives a basic message m through an incoming channel c_0

if $recorded_p = true$ and $marker_p[c_0] = false$ **then**
    $state_p[c_0] \leftarrow append(state_p[c_0], m)$;
end if

If p receives ⟨ **marker** ⟩ through an incoming channel c_0

perform procedure $TakeSnapshot_p$;
$marker_p[c_0] \leftarrow true$;
if $marker_p[c] = true$ for all incoming channels c of p **then**
    terminate;
end if

Procedure $TakeSnapshot_p$

if $recorded_p = false$ **then**
    $recorded_p \leftarrow true$;
    send ⟨ **marker** ⟩ into each outgoing channel of p;
    take a local snapshot of the state of p;
end if

A.2 Lai-Yang Snapshot Algorithm

The variable $recorded_p$ is set when p takes a local snapshot of its state. The set $State_p[qp]$ keeps track of the basic messages that arrive at p through its incoming channel qp after p has taken its local snapshot and that were sent by q before it took its local snapshot. The variable $counter_q[qp]$ counts how many basic messages process q sent into its outgoing channel qp before taking its local snapshot. Right before taking its local snapshot, q sends the control message ⟨ **presnap**, $counter_q[qp] + 1$ ⟩ to p (the + 1 is present because the control message itself is also counted), and p stores the value within this message in the variable $counter_p[qp]$. Finally, p terminates when it has received a control message ⟨ **presnap**, k ⟩ and $k - 1$ basic messages with the tag *false* through each incoming channel qp.

bool $recorded_p$;

nat $counter_p[c]$ for all channels c of p;
mess-set $State_p[c]$ for all incoming channels c of p;

If p wants to initiate a snapshot

perform procedure $TakeSnapshot_p$;

If p sends a basic message m into an outgoing channel c_0

send $\langle m, recorded_p \rangle$ into c_0;
if $recorded_p = false$ **then**
    $counter_p[c_0] \leftarrow counter_p[c_0] + 1$;
end if

If p receives $\langle m, b \rangle$ through an incoming channel c_0

if $b = true$ **then**
    perform procedure $TakeSnapshot_p$;
else
    $counter_p[c_0] \leftarrow counter_p[c_0] - 1$;
    if $recorded_p = true$ **then**
        $State_p[c_0] \leftarrow State_p[c_0] \cup \{m\}$;
        if $|State_p[c]| + 1 = counter_p[c]$ for all incoming channels c of p **then**
            terminate;
        end if
    end if
end if

If p receives $\langle$ **presnap**, $\ell \rangle$ through an incoming channel c_0

$counter_p[c_0] \leftarrow counter_p[c_0] + \ell$;
perform procedure $TakeSnapshot_p$;
if $|State_p[c]| + 1 = counter_p[c]$ for all incoming channels c of p **then**
    terminate;
end if

Procedure $TakeSnapshot_p$

if $recorded_p = false$ **then**
    $recorded_p \leftarrow true$;
    send $\langle$ **presnap**, $counter_p[c] + 1 \rangle$ into each outgoing channel c;

take a local snapshot of the state of p;
end if

A.3 Cidon's Depth-First Search Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator (or $\perp$ if p has no parent); $info_p$ is set when p sends the token for the first time, and $token_p[q]$ is set when p is certain that neighbor q will receive or has received the token. In the variable $forward_p$, the neighbor is stored to which p forwarded the token last.

bool $info_p$, $token_p[r]$ for all $r \in Neighbors_p$;
proc $parent_p$, $forward_p$;

If p is the initiator

perform procedure $ForwardToken_p$;

If p receives ⟨**info**⟩ from a neighbor q

if $forward_p \neq q$ **then**
  $token_p[q] \leftarrow true$;
else
  perform procedure $ForwardToken_p$;
end if

If p receives ⟨**token**⟩ from a neighbor q

if $forward_p = \perp$ **then**
  $parent_p \leftarrow q$; $token_p[q] \leftarrow true$;
  perform procedure $ForwardToken_p$;
else if $forward_p = q$ **then**
  perform procedure $ForwardToken_p$;
else
  $token_p[q] \leftarrow true$;
end if

Procedure $ForwardToken_p$

if $\{r \in Neighbors_p \mid token_p[r] = false\} \neq \emptyset$ **then**

```
    choose a q from this set, and send ⟨ token ⟩ to q;
    forward_p ← q;    token_p[q] ← true;
    if info_p = false then
        send ⟨ info ⟩ to each r ∈ Neighbors_p \ {q, parent_p};
        info_p ← true;
    end if
else if parent_p ≠ ⊥ then
    send ⟨ token ⟩ to parent_p;
else
    decide;
end if
```

A.4 Tree Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree. The variable $received_p[q]$ is set when p receives a wave message from neighbor q. Messages are included to inform all processes of the decision.

```
bool received_p[r] for all r ∈ Neighbors_p;
proc parent_p;

Initialization of p

perform procedure SendWave_p;

If p receives ⟨ wave ⟩ from a neighbor q

received_p[q] ← true;
perform procedure SendWave_p;

Procedure SendWave_p

if |{r ∈ Neighbors_p | received_p[r] = false}| = 1 then
    send ⟨ wave ⟩ to the only q ∈ Neighbors_p with received_p[q] = false;
    parent_p ← q;
else if |{r ∈ Neighbors_p | received_p[r] = false}| = 0 then
    decide;
    send ⟨ info ⟩ to each r ∈ Neighbors_p \ {parent_p};
end if
```

If p receives ⟨ **info** ⟩ from $parent_p$

send ⟨ **info** ⟩ to each $r \in Neighbors_p \setminus \{parent_p\}$;

A.5 Echo Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator. The variable $received_p$ counts how many wave messages have arrived at p.

nat $received_p$;
proc $parent_p$;

If p is the initiator

send ⟨ **wave** ⟩ to each $r \in Neighbors_p$;

If p receives ⟨ **wave** ⟩ from a neighbor q

$received_p \leftarrow received_p + 1$;
if $parent_p = \perp$ and p is a noninitiator **then**
    $parent_p \leftarrow q$;
    if $|Neighbors_p| > 1$ **then**
        send ⟨ **wave** ⟩ to each $r \in Neighbors_p \setminus \{q\}$;
    else
        send ⟨ **wave** ⟩ to q;
    end if
else if $received_p = |Neighbors_p|$ **then**
    if $parent_p \neq \perp$ **then**
        send ⟨ **wave** ⟩ to $parent_p$;
    else
        decide;
    end if
end if

A.6 Shavit-Francez Termination Detection Algorithm

The variable $parent_p$ contains the parent of p in a tree in the forest, and cc_p keeps track of (or better, estimates from above) the number of children of p

in its tree. The variable $active_p$ is set when p becomes active, and it is reset when p becomes passive.

bool $active_p$;
nat cc_p;
proc $parent_p$;

If p is an initiator

$active_p \leftarrow true$;

If p sends a basic message

$cc_p \leftarrow cc_p + 1$;

If p receives a basic message from a neighbor q

if $active_p = false$ **then**
    $active_p \leftarrow true$; $parent_p \leftarrow q$;
else
    send ⟨ **ack** ⟩ to q;
end if

If p receives ⟨ **ack** ⟩

$cc_p \leftarrow cc_p - 1$;
perform procedure $LeaveTree_p$;

If p becomes passive

$active_p \leftarrow false$;
perform procedure $LeaveTree_p$;

Procedure $LeaveTree_p$

if $active_p = false$ and $cc_p = 0$ **then**
    if $parent_p \neq \perp$ **then**
        send ⟨ **ack** ⟩ to $parent_p$;
        $parent_p \leftarrow \perp$;
    else
        start a wave, tagged with p;
    end if

end if

If p receives a wave message

if $active_p$ = *false* and $cc_p = 0$ **then**
    act according to the wave algorithm;
    in the case of a *decide* event, call *Announce*;
end if

A.7 Rana's Termination Detection Algorithm

The variable $active_p$ is set when p becomes active, and it is reset when p becomes passive. The variable $clock_p$ represents the clock value at p, and $unack_p$ keeps track of the number of unacknowledged basic messages sent by p.

bool $active_p$;
nat $clock_p$, $unack_p$;

If p is an initiator

$active_p \leftarrow true$;

If p sends a basic message

$unack_p \leftarrow unack_p + 1$;

If p receives a basic message from a neighbor q

$active_p \leftarrow true$;
send $\langle$ **ack**, $clock_p \rangle$ to q;

If p receives $\langle$ **ack**, $t \rangle$

$clock_p \leftarrow \max\{clock_p, t + 1\}$; $unack_p \leftarrow unack_p - 1$;
if $active_p$ = *false* and $unack_p = 0$ **then**
    start a wave, tagged with p and $clock_p$;
end if

If p becomes passive

$active_p \leftarrow false$;

if $unack_p = 0$ **then**
    start a wave, tagged with p and $clock_p$;
end if

If p receives a wave message tagged with q and t

if $active_p = false$ and $unack_p = 0$ and $clock_p \leq t$ **then**
    act according to the wave algorithm, for the wave tagged with q and t;
    in the case of a *decide* event, call *Announce*;
end if
$clock_p \leftarrow \max\{clock_p, t\}$;

A.8 Safra's Termination Detection Algorithm

The variable $active_p$ is set when p becomes active, and it is reset when p becomes passive. The variable $black_p$ is set when p receives a basic message, and it is reset when p forwards the token. Moreover, the initiator of the control algorithm at the start sets this variable to make sure it sends out the token when it becomes passive for the first time. As long as p is holding the token, $token_p$ is set. When p sends or receives a basic message, $mess\text{-}counter_p$ is increased or decreased by 1, respectively. The variable $token\text{-}counter_p$ is used to store the counter value of the token. For simplicity, we assume that the initiator of the control algorithm is also an initiator of the basic algorithm.

bool $active_p$, $token_p$, $black_p$;
int $mess\text{-}counter_p$, $token\text{-}counter_p$;

If p is the initiator of the control algorithm

$token_p \leftarrow true$; $black_p \leftarrow true$;

If p is an initiator of the basic algorithm

$active_p \leftarrow true$;

If p sends a basic message

$mess\text{-}counter_p \leftarrow mess\text{-}counter_p + 1$;

If p receives a basic message

$active_p \leftarrow true$; $black_p \leftarrow true$; $mess\text{-}counter_p \leftarrow mess\text{-}counter_p - 1$;

If p becomes passive

$active_p \leftarrow false$;
perform procedure $TreatToken_p$;

If p receives ⟨ **token**, b, k⟩

$token_p \leftarrow true$; $black_p \leftarrow black_p \vee b$; $token\text{-}counter_p \leftarrow k$;
perform procedure $TreatToken_p$;

Procedure $TreatToken_p$

if $active_p = false$ and $token_p = true$ **then**
  if $black_p = false$ **then**
    $mess\text{-}counter_p \leftarrow mess\text{-}counter_p + token\text{-}counter_p$
  end if
  if p is a noninitiator **then**
    forward ⟨ **token**, $black_p$, $mess\text{-}counter_p$⟩ ;
    $token_p \leftarrow false$; $black_p \leftarrow false$;
  else if $black_p = true$ or $mess\text{-}counter_p > 0$ **then**
    send ⟨ **token**, *false*, 0⟩ on a round trip through the network;
    $token_p \leftarrow false$; $black_p \leftarrow false$;
  else
    call *Announce*;
  end if
end if

A.9 Weight-Throwing Termination Detection Algorithm

The variable $active_p$ is set when p becomes active, and it is reset when p becomes passive. The variable $weight_p$ contains the weight at p, and *total* contains the total amount of weight in the network. The constant *minimum*, a real value between 0 and $\frac{1}{2}$, represents the minimum allowed weight at a process. In the case of underflow, a noninitiator informs the initiator that it has added one extra unit of weight to the system, and it waits for an acknowledgment from the initiator. For simplicity, we assume that there is

an undirected channel between the initiator and every other process in the network.

bool $active_p$;
real $weight_p$, *total* only at the initiator;

If p is the initiator

$active_p \leftarrow true$; $weight_p \leftarrow 1$; $total \leftarrow 1$;

If p sends a basic message m to a neighbor q

if $\frac{1}{2} \cdot weight_p < minimum$ **then**
  if p is a noninitiator **then**
    send ⟨ **more-weight** ⟩ to the initiator to ask for extra weight;
    wait for an acknowledgment from the initiator to arrive;
  else
    $total \leftarrow total + 1$;
  end if
  $weight_p \leftarrow weight_p + 1$;
end if
send ⟨ m, $\frac{1}{2} \cdot weight_p$ ⟩ to q;
$weight_p \leftarrow \frac{1}{2} \cdot weight_p$;

If p receives a basic message ⟨ m, w ⟩

$active_p \leftarrow true$; $weight_p \leftarrow weight_p + w$;

If p becomes passive

$active_p \leftarrow false$;
if p is a noninitiator **then**
  send ⟨ **return-weight**, $weight_p$ ⟩ to the initiator;
  $weight_p \leftarrow 0$;
else if $total = weight_p$ **then**
  call *Announce*;
end if

If initiator p receives ⟨ **more-weight** ⟩ from a process q

$total \leftarrow total + 1$;

send an acknowledgment to q;

If initiator p receives ⟨ **return-weight**, w⟩

$weight_p \leftarrow weight_p + w$;
if $active_p = false$ and $total = weight_p$ **then**
    call *Announce*;
end if

A.10 Chandy-Misra Routing Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator, and $dist_p$ is the distance value of p toward the initiator.

dist $dist_p$;
proc $parent_p$;

If p is the initiator

$dist_p \leftarrow 0$;
send ⟨ **dist**, 0⟩ to each $r \in Neighbors_p$;

If p receives ⟨ **dist**, d⟩ from a neighbor q

if $d + weight(pq) < dist_p$ **then**
    $dist_p \leftarrow d + weight(pq)$; $parent_p \leftarrow q$;
    send ⟨ **dist**, $dist_p$⟩ to each $r \in Neighbors_p \setminus \{q\}$;
end if

A.11 Merlin-Segall Routing Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator, and $dist_p$ is the distance value of p toward the initiator. In $new\text{-}parent_p$, the process is stored that sent the message to p on which the current value of $dist_p$ is based; at the end of a round, the value of $new\text{-}parent_p$ is passed on to $parent_p$. In $counter_p$, p keeps track of how many messages it has received in the current round.

nat $counter_p$;

dist $dist_p$;
proc $parent_p$, $new\text{-}parent_p$;

If p is the initiator

$dist_p \leftarrow 0$;
initiate a wave that determines a spanning tree of the network, captured by values of $parent_r$ for all $r \in Processes$, with p as root;
wait until this wave has terminated;
for $k = 1$ **to** $N - 1$ **do**
    send ⟨ **dist**, 0⟩ to each $r \in Neighbors_p$;
    while $counter_p < |Neighbors_p|$ **do**
        wait for a message ⟨ **dist**, d⟩ to arrive;
        $counter_p \leftarrow counter_p + 1$;
    end while
    $counter_p \leftarrow 0$;
end for

If p is a noninitiator

take part in the wave, and provide $parent_p$ with the resulting parent value;
for $k = 1$ **to** $N - 1$ **do**
    while $counter_p < |Neighbors_p|$ **do**
        wait for a message ⟨ **dist**, d⟩ from a $q \in Neighbors_p$;
        $counter_p \leftarrow counter_p + 1$;
        if $d + weight(pq) < dist_p$ **then**
            $dist_p \leftarrow d + weight(pq)$; $new\text{-}parent_p \leftarrow q$;
        end if
        if $q = parent_p$ **then**
            send ⟨ **dist**, $dist_p$⟩ to each $r \in Neighbors_p \setminus \{parent_p\}$;
        end if
    end while
    send ⟨ **dist**, $dist_p$⟩ to $parent_p$;
    $parent_p \leftarrow new\text{-}parent_p$; $counter_p \leftarrow 0$;
end for

A.12 Toueg's Routing Algorithm

The variable $parent_p[q]$ contains the parent of p in the spanning tree rooted at process q, and $dist_p[q]$ is the distance value of p toward destination q. In $round_p$, p keeps track of its round number. The distance values of the pivot in round k are stored in $Distances_p[k]$. Each process that sends a request to p for the distance values of the pivot in the current or a future round k is stored in $Forward_p[k]$. We assume that p only treats incoming requests when it is idle, to avoid having a request stored in $Forward_p[k]$ after p forwarded the distance values of the pivot in round k. The pivot in round k is denoted by $pivot(k)$. We include the optimization that a process, upon receiving distance values from the pivot, first checks which of its distance values are improved and then forwards only those elements of the set that gave rise to an improved distance value.

nat $round_p$;
dist $dist_p[r]$ for all $r \in Processes$;
proc $parent_p[r]$ for all $r \in Processes$;
proc-set $Forward_p[k]$ for all $k \in \{0,...,N-1\}$;
proc-dist-set $Distances_p[k]$ for all $k \in \{0,...,N-1\}$;

Initialization of p

$dist_p[p] \leftarrow 0$; $parent_p[r] \leftarrow r$ and $dist_p[r] \leftarrow weight(pr)$ for all $r \in Neighbors_p$;
perform procedure $Request_p$;

Procedure $Request_p$

if $p = pivot(round_p)$ **then**
  send ⟨ **dist-set**, $\{(r, dist_p[r]) \mid r \in Processes$ and $dist_p[r] < \infty\}$ ⟩
  to each $q \in Forward_p[round_p]$;
  perform procedure $NextRound_p$;
else if $parent_p[pivot(round_p)] \neq \perp$ **then**
  send ⟨ **request**, $round_p$ ⟩ to $parent_p[pivot(round_p)]$;
else
  perform procedure $NextRound_p$;
end if

If p receives ⟨ **request**, k⟩ from a neighbor q

if $k < round_p$ **then**
  send ⟨ **dist-set**, $Distances_p[k]$⟩ to q;
else
  $Forward_p[k] \leftarrow Forward_p[k] \cup \{q\}$;
end if

If p receives ⟨ **dist-set**, $Distances$⟩ from $parent_p[pivot(round_p)]$

for each $s \in Processes$ **do**
  if there is a pair (s, d) in $Distances$ **then**
    if $d + dist_p[pivot(round_p)] < dist_p[s]$ **then**
      $parent_p[s] \leftarrow parent_p[pivot(round_p)]$;
      $dist_p[s] \leftarrow d + dist_p[pivot(round_p)]$;
    else
      remove entry (s, d) from $Distances$;
    end if
  end if
end for
send ⟨ **dist-set**, $Distances$⟩ to each $r \in Forward_p[round_p]$;
$Distances_p[round_p] \leftarrow Distances$;
perform procedure $NextRound_p$;

Procedure $NextRound_p$

if $round_p < N - 1$ **then**
  $round_p \leftarrow round_p + 1$;
  perform procedure $Request_p$;
else
  terminate;
end if

A.13 Frederickson's Breadth-First Search Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree rooted at the initiator, and $dist_p$ is the distance value of p toward the initiator. In $dist_p[r]$, p stores the best-known distance value of neighbor r. After p has

sent **forward** or **explore** messages, it uses Ack_p to keep track of the neighbors that should still send a (positive or negative) reply. In $Reported_p$, p keeps track of the neighbors to which it must send a **forward** message in the next round. The initiator maintains the round number in *counter*. During each round, ℓ levels are explored. For uniformity, messages $\langle$ **reverse**, $b\rangle$ are always supplied with the distance value of the sender.

nat *counter* only at the initiator;
dist $dist_p$, $dist_p[r]$ for all $r \in Neighbors_p$;
proc $parent_p$;
proc-set Ack_p, $Reported_p$;

If p is the initiator

send $\langle$ **explore**, $1\rangle$ to each $r \in Neighbors_p$;
$dist_p \leftarrow 0$; $Ack_p \leftarrow Neighbors_p$; $counter \leftarrow 1$;

If p receives $\langle$ **explore**, $k\rangle$ from a neighbor q

$dist_p[q] \leftarrow \min\{dist_p[q], k-1\}$;
if $k < dist_p$ **then**
  $parent_p \leftarrow q$; $dist_p \leftarrow k$; $Reported_p \leftarrow \emptyset$;
  if $k \bmod \ell \neq 0$ **then**
    send $\langle$ **explore**, $k+1\rangle$ to each $r \in Neighbors_p \setminus \{q\}$;
    $Ack_p \leftarrow \{r \in Neighbors_p \mid dist_p[r] > k+1\}$;
    if $Ack_p = \emptyset$ **then**
      send $\langle$ **reverse**, *k, false*$\rangle$ to q;
    end if
  else
    send $\langle$ **reverse**, *k, true*$\rangle$ to q;
  end if
else if $k > dist_p$ or $k \bmod \ell \neq 0$ **then**
  if $k \leq dist_p + 2$ and $q \in Ack_p$ **then**
    $Ack_p \leftarrow Ack_p \setminus \{q\}$;
    perform procedure $ReceivedAck_p$;
  else if $k = dist_p$ **then**
    $Reported_p \leftarrow Reported_p \setminus \{q\}$;
  end if

else
send ⟨ **reverse**, *k*, *false*⟩ to *q*;
end if

If *p* receives ⟨ **reverse**, *k*, *b*⟩ from a neighbor *q*

$dist_p[q] \leftarrow \min\{dist_p[q], k\}$;
if $k = dist_p + 1$ **then**
if $b = true$ and $dist_p[q] = k$ **then**
$Reported_p \leftarrow Reported_p \cup \{q\}$;
end if
if $q \in Ack_p$ **then**
$Ack_p \leftarrow Ack_p \setminus \{q\}$;
perform procedure $ReceivedAck_p$;
end if
end if

Procedure $ReceivedAck_p$

if $Ack_p = \emptyset$ **then**
if $parent_p \neq \perp$ **then**
send ⟨ **reverse**, $dist_p$, $Reported_p \neq \emptyset$ ⟩ to $parent_p$;
else if $Reported_p \neq \emptyset$ **then**
send ⟨ **forward**, $\ell \cdot counter$⟩ to each $r \in Reported_p$;
$Ack_p \leftarrow Reported_p$; $Reported_p \leftarrow \emptyset$; $counter \leftarrow counter + 1$;
else
terminate;
end if
end if

If *p* receives ⟨ **forward**, *k*⟩ from a neighbor *q*

if $q = parent_p$ **then**
if $k < depth_p$ **then**
send ⟨ **forward**, *k*⟩ to each $r \in Reported_p$;
$Ack_p \leftarrow Reported_p$; $Reported_p \leftarrow \emptyset$;
else
$Ack_p \leftarrow \{r \in Neighbors_p \mid dist_p[r] = \infty\}$;
if $Ack_p \neq \emptyset$ **then**

```
            send ⟨ explore, k + 1⟩ to each r ∈ Ack_p;
        else
            send ⟨ reverse, k, false⟩ to q;
        end if
    end if
end if
```

A.14 Dolev-Klawe-Rodeh Election Algorithm

Initially, $active_p$ is set if p is an initiator, and it is reset when p becomes passive. If p terminates as the leader, it sets $leader_p$. Since messages of two consecutive rounds can overtake each other, p keeps track of the parity of its round number in $parity_p$ and attaches this Boolean value to its message. In $election\text{-}id_p$, p stores the ID it assumes for the current election round. In $neighb\text{-}id_p[0, b]$ and $neighb\text{-}id_p[1, b]$, p stores the process IDs of its two nearest active predecessors in the directed ring, with b the parity of the corresponding election round. We assume a total order $<$ on process IDs.

```
bool active_p, leader_p, parity_p;
proc election-id_p, neighb-id_p[n, b] for n = 0, 1 and Booleans b;

If p is an initiator

active_p ← true;    election-id_p ← p;
send ⟨ id, p, 0, b⟩ ;

If p receives ⟨ id, q, n, b⟩

if active_p = true then
    if n = 0 then
        send ⟨ id, q, 1, b⟩ ;
    end if
    neighb-id_p[n, b] ← q;
    if neighb-id_p[n, parity_p] ≠ ⊥ for n = 0 and n = 1 then
        perform procedure CompareIds_p;
    end if
else
    send ⟨ id, q, n, b⟩ ;
```

end if

Procedure $CompareIds_p$

if $\max\{election\text{-}id_p, neighb\text{-}id_p[1, parity_p]\} < neighb\text{-}id_p[0, parity_p]$ **then**
  $election\text{-}id_p \leftarrow neighb\text{-}id_p[0, parity_p]$;
  $neighb\text{-}id_p[n, parity_p] \leftarrow \perp$ for $n = 0$ and $n = 1$; $parity_p \leftarrow \neg parity_p$;
  send $\langle$ **id**, $election\text{-}id_p$, 0, $parity_p\rangle$;
  if $neighb\text{-}id_p[n, parity_p] \neq \perp$ for $n = 0$ and $n = 1$ **then**
    perform procedure $CompareIds_p$;
  end if
else if $neighb\text{-}id_p[0, parity_p] < election\text{-}id_p$ **then**
  $active_p \leftarrow false$;
else
  $leader_p \leftarrow true$;
end if

A.15 Gallager-Humblet-Spira Minimum Spanning Tree Algorithm

The variable $parent_p$ contains p's parent toward the core edge of p's fragment. The name and level of p's fragment are stored in $name_p$ and $level_p$. Initially, $state_p$ has the value *find*; for simplicity, the state *sleep* and the corresponding wake-up phase are omitted. The channel states, $state_p[q]$ for each $q \in Neighbors_p$, initially are *basic*. While looking for a lowest-weight outgoing edge, p stores the optimal intermediate result in $best\text{-}weight_p$. If the optimal result was reported through the basic or branch edge pq, then $best\text{-}edge_p$ has the value q. While p is testing whether basic edge pq is outgoing, $test\text{-}edge_p$ has the value q. In $counter_p$, p keeps track of how many branch edges have reported their minimum values; it starts at 1 to account for the fact that p's parent in general does not report a value (except for the core nodes). In $parent\text{-}report_p$, a core node p can keep the value reported by its parent; if there is no report yet, its value is 0, while the value ∞ means that p's parent has reported no outgoing edges at its side. In $Connects_p$ and $Tests_p$, p stores incoming **connect** and **test** messages to

which a reply is being delayed until the level of p's fragment is high enough.

```
{find, found} state_p;
{basic, branch, rejected} state_p[r] for all r ∈ Neighbors_p;
real name_p;
nat level_p, counter_p;
dist best-weight_p, parent-report_p;
proc parent_p, test-edge_p, best-edge_p;
proc-nat-set Connects_p;
proc-real-nat-set Tests_p;
```

Initialization of p

```
determine the lowest-weight channel pq;
state_p ← found;  state_p[q] ← branch;  counter_p ← 1;  parent-report_p ← 0;
send ⟨ connect, 0 ⟩ to q;
```

If p receives ⟨ **connect**, ℓ ⟩ from a neighbor q

```
if ℓ < level_p then
    send ⟨ initiate, name_p, level_p, state_p ⟩ to q;
    state_p[q] ← branch;
else if state_p[q] = branch then
    send ⟨ initiate, weight(pq), level_p + 1, find ⟩ to q;
else
    Connects_p = Connects_p ∪ {(q, ℓ)};
end if
```

If p receives ⟨ **initiate**, fn, ℓ, st ⟩ from a neighbor q

```
name_p ← fn;  level_p ← ℓ;  state_p ← st;  parent_p ← q;
best-edge_p ← ⊥;  best-weight_p ← ∞;  counter_p ← 1;  parent-report_p ← 0;
for each (q_0, ℓ_0) ∈ Connects_p do
    if ℓ_0 < level_p then
        state_p[q_0] ← branch;  Connects_p ← Connects_p \ {(q_0, ℓ_0)};
    end if
```

end for
send ⟨ **initiate**, *fn, ℓ, st*⟩ to each $r \in Neighbors_p \setminus \{q\}$ with $state_p[r] = branch$;
for each $(q_1, fn_1, \ell_1) \in Tests_p$ **do**
  if $\ell_1 \leq level_p$ **then**
    perform procedure $ReplyTest_p(q_1)$;
    $Tests_p \leftarrow Tests_p \setminus \{(q_1, fn_1, \ell_1)\}$;
  end if
end for
if *st* = *find* **then**
  perform procedure $FindMinimalOutgoing_p$;
end if

Procedure $FindMinimalOutgoing_p$

if $\{pr \mid r \in Neighbors_p$ and $state_p(pr) = basic\} \neq \emptyset$ **then**
  send ⟨ **test**, $name_p$, $level_p$⟩ into the lowest-weight channel *pq* in this collection;
  $test\text{-}edge_p \leftarrow q$;
else
  $test\text{-}edge_p \leftarrow \perp$;
  if $counter_p = |\{r \in Neighbors_p \mid state_p[r] = branch\}|$ **then**
    perform procedure $SendReport_p$
  end if
end if

If *p* receives ⟨ **test**, *fn, ℓ*⟩ from a neighbor *q*

if $\ell \leq level_p$ **then**
  perform procedure $ReplyTest_p(q)$;
else
  $Tests_p = Tests_p \cup \{(q, fn, \ell)\}$;
end if

Procedure $ReplyTest_p(q)$

if $name_p \neq fn$ **then**
  send ⟨ **accept**⟩ to *q*;
else

$state_p[pq] \leftarrow rejected$;
if $test\text{-}edge_p \neq q$ **then**
send ⟨ **reject** ⟩ to q;
else
perform procedure $FindMinimalOutgoing_p$;
end if
end if

If p receives ⟨ **reject** ⟩ from a neighbor q

$state_p[q] \leftarrow rejected$;
perform procedure $FindMinimalOutgoing_p$;

If p receives ⟨ **accept** ⟩ from a neighbor q

$test\text{-}edge_p \leftarrow \perp$;
if $weight(pq) < best\text{-}weight_p$ **then**
$best\text{-}edge_p \leftarrow q$; $best\text{-}weight_p \leftarrow weight(pq)$;
end if
if $counter_p = |\{r \in Neighbors_p \mid state_p[r] = branch\}|$ **then**
perform procedure $SendReport_p$
end if

Procedure $SendReport_p$

$state_p \leftarrow found$;
send ⟨ **report**, $best\text{-}weight_p$ ⟩ to $parent_p$;
if $parent\text{-}report_p > 0$ and $best\text{-}weight_p < parent\text{-}report_p$ **then**
perform procedure $ChangeRoot_p$;
end if

If p receives ⟨ **report**, λ ⟩ from a neighbor q

if $q \neq parent_p$ **then**
$counter_p \leftarrow counter_p + 1$;
if $\lambda < best\text{-}weight_p$ **then**
$best\text{-}edge_p \leftarrow q$; $best\text{-}weight_p \leftarrow \lambda$;
end if
if $counter_p = |\{r \in Neighbors_p \mid state_p[r] = branch\}|$ and $test\text{-}edge_p = \perp$
then

```
        perform procedure SendReport_p
    end if
else if state_p = find then
    parent-report_p ← λ;
else
    if best-weight_p < λ then
        perform procedure ChangeRoot_p;
    else if λ = ∞ then
        terminate;
    end if
end if
```

Procedure $ChangeRoot_p$

```
if state_p[best-edge_p] = branch then
    send ⟨ changeroot ⟩ to best-edge_p;
else
    state_p[best-edge_p] ← branch;
    send ⟨ connect, level_p ⟩ to best-edge_p;
    if (best-edge_p, level_p) ∈ Connects_p then
        send ⟨ initiate, best-weight_p, level_p + 1, find ⟩ to best-edge_p;
        Connects_p ← Connects_p \ {(best-edge_p, level_p)};
    end if
end if
```

If p receives ⟨ **changeroot** ⟩

```
perform procedure ChangeRoot_p;
```

A.16 IEEE 1394 Election Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree. The variable $received_p[q]$ is set when p receives a parent request from a neighbor q to which p has not sent a parent request. If p gets into root contention and chooses to start a timer, it sets $waiting_p$. If p terminates as the leader, it sets $leader_p$.

```
bool leader_p, waiting_p, received_p[r] for all r ∈ Neighbors_p;
```

proc $parent_p$;

Initialization of p

perform procedure $SendRequest_p$;

Procedure $SendRequest_p$

if $|\{r \in Neighbors_p \mid received_p[r] = false\}| = 1$ **then**
    send ⟨ **parent-req** ⟩ to the only $q \in Neighbors_p$ with $received_p[q] = false$;
    $parent_p \leftarrow q$;
end if

If p receives ⟨ **parent-req** ⟩ from a neighbor q

if $q \neq parent_p$ **then**
    $received_p[q] \leftarrow true$;
    send ⟨ **ack** ⟩ to q;
    perform procedure $SendRequest_p$;
else if $waiting_p = false$ **then**
    perform procedure $RootContention_p$;
else
    $leader_p \leftarrow true$;
end if

If p receives ⟨ **ack** ⟩ from $parent_p$

terminate;

Procedure $RootContention_p$

either send ⟨ **parent-req** ⟩ to q and $waiting_p \leftarrow false$,
or start a timer and $waiting_p \leftarrow true$;

If a *timeout* occurs at p

perform procedure $RootContention_p$;

A.17 Awerbuch's Synchronizer

The variable $parent_p$ is the parent of p in the spanning tree within its cluster, $Children_p$ contains the children of p in this spanning tree, and $Designated_p$ contains the processes q for which there is a designated channel pq. Note that these three values are fixed after the initialization phase. In $1st\text{-}counter_p$ and $2nd\text{-}counter_p$, p keeps track of how many messages still need to be received in the first and second phases of this synchronizer, respectively.

nat $1st\text{-}counter_p$, $2nd\text{-}counter_p$;
proc $parent_p$;
proc-set $Children_p$, $Designated_p$;

Initialization

The network is divided into clusters, and within each cluster a spanning tree is built. Between each pair of distinct clusters that are connected by a channel, one of these connecting channels is labeled as *designated*. Furthermore, a wake-up phase makes sure that each process starts its first pulse, meaning that it performs the procedure *NewPulse*.

Procedure $NewPulse_p$

send $k \geq 0$ basic messages;
$1st\text{-}counter_p \leftarrow k + |Children_p|$; $2nd\text{-}counter_p \leftarrow |Children_p| + |Designated_p|$;
perform procedure $FirstReport_p$;

If p receives a basic message from a neighbor q

send ⟨ **ack** ⟩ to q;

If p receives ⟨ **ack** ⟩ or ⟨ **safe** ⟩

$1st\text{-}counter_p \leftarrow 1st\text{-}counter_p - 1$;
perform procedure $FirstReport_p$;

Procedure $FirstReport_p$

if $1st\text{-}counter_p = 0$ **then**
  if $parent_p \neq \perp$ **then**
    send ⟨ **safe** ⟩ to $parent_p$;
  else

perform procedure $SendNext_p$;
end if
end if

Procedure $SendNext_p$

send ⟨ **next** ⟩ to each $q \in Children_p$;
send ⟨ **cluster-safe** ⟩ to each $r \in Designated_p$;
perform procedure $SecondReport_p$;

If p receives ⟨ **next** ⟩

perform procedure $SendNext_p$;

If p receives ⟨ **cluster-safe** ⟩

$2nd\text{-}counter_p \leftarrow 2nd\text{-}counter_p - 1$;
perform procedure $SecondReport_p$;

Procedure $SecondReport_p$

if $2nd\text{-}counter_p = 0$ **then**
if $parent_p \neq \perp$ **then**
send ⟨ **cluster-safe** ⟩ to $parent_p$;
else
perform procedure $SendClusterNext_p$;
end if
end if

Procedure $SendClusterNext_p$

send ⟨ **cluster-next** ⟩ to each $q \in Children_p$;
perform procedure $NewPulse_p$;

If p receives ⟨ **cluster-next** ⟩

perform procedure $SendClusterNext_p$;

A.18 Ricart-Agrawala Mutual Exclusion Algorithm

We use the lexicographical order on pairs (t, i) with t a time stamp and i a process index. The variable $clock_i$ represents the clock value at p_i; it starts at

1. In $req\text{-}stamp_i$, p_i stores the time stamp of its current request; if there is none, $req\text{-}stamp_i = 0$. The number of permissions that p_i has received for its current request is maintained in $counter_i$. In $Pending_i$, p_i keeps track of the processes it has received a request from but to which it has not yet sent permission. The Carvalho-Roucairol optimization is taken into account. In $Requests_i$, p_i keeps track of the processes to which it must send (or has sent) its next (or current) request.

nat $clock_i$, $req\text{-}stamp_i$, $counter_i$;
proc-set $Pending_i$, $Requests_i$;

Initialization of p_i

$Requests_i \leftarrow Neighbors_i$; $clock_i \leftarrow 1$;

If p_i wants to enter its critical section

if $Requests_i \neq \emptyset$ **then**
  send ⟨ **request**, $clock_i$, i ⟩ to each $q \in Requests_i$;
  $req\text{-}stamp_i \leftarrow clock_i$; $counter_i \leftarrow 0$;
else
  perform procedure $CriticalSection_p$;
end if

If p_i receives ⟨ **permission** ⟩

$counter_i \leftarrow counter_i + 1$;
if $counter_i = |Requests_i|$ **then**
  perform procedure $CriticalSection_p$;
end if

Procedure $CriticalSection_p$

enter critical section;
exit critical section;
send ⟨ **permission** ⟩ to each $q \in Pending_i$;
$req\text{-}stamp_i \leftarrow 0$; $Requests_i \leftarrow Pending_i$; $Pending_i \leftarrow \emptyset$;

If p_i receives ⟨ **request**, t, j ⟩ from a p_j

$clock_i \leftarrow \max\{clock_i, t + 1\}$;

if $req\text{-}stamp_i = 0$ or $(t, j) < (req\text{-}stamp_i, i)$ **then**
    send ⟨ **permission** ⟩ to p_j;
    if $p_j \notin Requests_i$ **then**
        $Requests_i \leftarrow Requests_i \cup \{p_j\}$;
        if $req\text{-}stamp_i > 0$ **then**
            send ⟨ **request**, $req\text{-}stamp_i$, i ⟩ to p_j;
        end if
    end if
else
    $Pending_i \leftarrow Pending_i \cup \{p_j\}$;
end if

A.19 Raymond's Mutual Exclusion Algorithm

The variable $parent_p$ contains the parent of p in the spanning tree. The queue $pending_p$ contains the children of p in the tree that have asked for the token and possibly p itself.

proc $parent_p$;
proc-queue $pending_p$;

If p is the initiator

Initiate a wave that determines a spanning tree of the network, captured by values of $parent_r$ for all $r \in Processes$, with p as root;

If p wants to enter its critical section

if $parent_p \neq \perp$ **then**
    $pending_p \leftarrow append(pending_p, p)$;
    if $head(pending_p) = p$ **then**
        send ⟨ **request** ⟩ to $parent_p$;
    end if
else
    perform procedure $CriticalSection_p$;
end if

If p receives ⟨ **request** ⟩ from a neighbor q

$pending_p \leftarrow append(pending_p, q)$;
if $head(pending_p) = q$ **then**
  if $parent_p \neq \perp$ **then**
    send ⟨ **request** ⟩ to $parent_p$;
  else if p is not in its critical section **then**
    perform procedure $SendToken_p$;
  end if
end if

If p receives ⟨ **token** ⟩

if $head(pending_p) \neq p$ **then**
  perform procedure $SendToken_p$;
else
  $parent_p \leftarrow \perp$; $pending_p \leftarrow tail(pending_p)$;
  perform procedure $CriticalSection_p$;
end if

Procedure $SendToken_p$

$parent_p \leftarrow head(pending_p)$; $pending_p \leftarrow tail(pending_p)$;
send ⟨ **token** ⟩ to $parent_p$;
if $pending_p \neq \emptyset$ **then**
  send ⟨ **request** ⟩ to $parent_p$;
end if

Procedure $CriticalSection_p$

enter critical section;
exit critical section;
if $pending_p \neq \emptyset$ **then**
  perform procedure $SendToken_p$;
end if

A.20 Agrawal–El Abbadi Mutual Exclusion Algorithm

The variable $requests_p$ contains a queue of processes from which p must still obtain permission to enter its critical section. The set $Permissions_p$ contains the processes from which p has received permission during its current

attempt to become privileged. The queue $pending_p$ contains the processes from which p has received a request; it has only sent permission to the head of this queue. (We recall that p may have to ask permission from itself.) We assume that $N = 2^k - 1$ for some $k > 1$, so that the binary tree has depth $k - 1$. The constant *root* denotes the root node of the binary tree, and for any nonleaf q in the tree, *left-child*(q) and *right-child*(q) denote its children at the left and right, respectively. Processes may crash, and they are provided with a complete and strongly accurate failure detector. When p detects that another process has crashed, it puts the corresponding process ID in the set $Crashed_p$.

proc-queue $requests_p$, $pending_p$;
proc-set $Permissions_p$, $Crashed_p$;

If p wants to enter its critical section

$requests_p \leftarrow append(\emptyset, root)$;
perform procedure $SendRequest_p$;

Procedure $SendRequest_p$

if $head(requests_p) \notin Crashed_p$ **then**
    send ⟨ **request** ⟩ to $head(requests_p)$;
else
    perform procedure $HeadRequestsCrashed_p$;
end if

Procedure $HeadRequestsCrashed_p$

if $head(requests_p)$ is not a leaf of the binary tree **then**
    $requests_p \leftarrow append(append(tail(requests_p), left\text{-}child(head(requests_p))),$
                $right\text{-}child(head(requests_p)))$;
    perform procedure $SendRequest_p$;
else
    $Permissions_p \leftarrow \emptyset$;
    start a new attempt to enter the critical section;
end if

If p receives ⟨ **request** ⟩ from a process q

$pending_p \leftarrow append(pending_p, q)$;
if $head(pending_p) = q$ **then**
    perform procedure $SendPermission_p$;
end if

Procedure $SendPermission_p$

if $pending_p \neq \emptyset$ **then**
    if $head(pending_p) \notin Crashed_p$ **then**
        send ⟨ **permission** ⟩ to $head(pending_p)$;
    else
        $pending_p \leftarrow tail(pending_p)$;
        perform procedure $SendPermission_p$;
    end if
end if

If p receives ⟨ **permission** ⟩ from process q

$Permissions_p \leftarrow Permissions_p \cup \{q\}$; $requests_p \leftarrow tail(requests_p)$;
if q is not a leaf of the binary tree **then**
    either $requests_p \leftarrow append(requests_p, left\text{-}child(q))$
    or $requests_p \leftarrow append(requests_p, right\text{-}child(q))$;
end if
if $requests_p \neq \emptyset$ **then**
    perform procedure $SendRequest_p$;
else
    enter critical section;
    exit critical section;
    send ⟨ **released** ⟩ to each $r \in Permissions_p$;
    $Permissions_p \leftarrow \emptyset$;
end if

If p receives ⟨ **released** ⟩

$pending_p \leftarrow tail(pending_p)$;
perform procedure $SendPermission_p$;

If p detects that a process q has crashed

$Crashed_p \leftarrow Crashed_p \cup \{q\}$;
if $head(requests_p) = q$ **then**
    perform procedure $HeadRequestsCrashed_p$;
end if
if $head(pending_p) = q$ **then**
    $pending_p \leftarrow tail(pending_p)$;
    perform procedure $SendPermission_p$;
end if

A.21 MCS Queue Lock

Since processes can use the same element for each lock access, we take the liberty of representing the queue of waiting processes by using process IDs instead of elements. The multi-writer register $wait_p$ is *true* as long as p must wait to get the lock. The multi-writer register $succ_p$ points to the successor of p in the queue of waiting processes. The single-writer register $pred_p$ points to the predecessor of p in the queue of waiting processes. When a process arrives at this queue, it assigns its process ID to the multi-writer register *last*. The operation *last.get-and-set*(p) assigns the value p to *last* and returns the previous value of *last*, all in one atomic step. And *last.compare-and-set*(p, $\perp$) in one atomic operation reads the value of *last* and either assigns the value $\perp$ to *last*, if its current value is p, or leaves the value of *last* unchanged otherwise. In the first case, this operation returns *true*, while in the second case it returns *false*.

bool $wait_p$;
proc *last*, $succ_p$, $pred_p$;

If p wants to enter its critical section

$pred_p \leftarrow last.get\text{-}and\text{-}set(p)$;
if $pred_p \neq \perp$ **then**
    $wait_p \leftarrow true$; $succ_{pred_p} \leftarrow p$;
    while $wait_p = true$ **do**
        {};
    end while
end if

enter critical section;
exit critical section;
if $succ_p \neq \perp$ **then**
    $wait_{succ_p} \leftarrow false$;
else if *last.compare-and-set*(p, $\perp$) returns *false* **then**
    while $succ_p = \perp$ **do**
        {};
    end while
    $wait_{succ_p} \leftarrow false$;
end if

A.22 CLH Queue Lock with Timeouts

The data type **pointer** consists of pointers to an element, with `null` as default initial value. If p wants to enter its critical section, it creates an element ε containing a pointer $pred_\varepsilon$. Let element ε' denote the nearest nonabandoned predecessor of ε in the queue. The pointer $pred_p$ points to ε'. In *pred-pred*$_p$, p repeatedly stores the value of $pred_{\varepsilon'}$. If p decides to abandon its attempt to get the lock and has a successor in the queue, then it lets $pred_\varepsilon$ point to ε'. The multi-writer register *last* points to the last element in the queue.

pointer *last*, $pred_p$, *pred-pred*$_p$, $pred_\varepsilon$ for all elements ε;

If p wants to enter its critical section

create an element ε;
$pred_p \leftarrow$ *last.get-and-set*(ε);
if $pred_p =$ `null` **then**
    perform procedure *CriticalSection*$_p$(ε);
else
    while no *timeout* occurs **do**
        pred-pred$_p \leftarrow pred_{pred_p}$;
        if *pred-pred*$_p =$ `released` **then**
            perform procedure *CriticalSection*$_p$(ε);
        else if *pred-pred*$_p \neq$ `null` **then**

```
            pred_p ← pred-pred_p;
        end if
    end while
    if last.compare-and-set(ε, pred_p) returns false then
        pred_ε ← pred_p;
    end if
    abandon the attempt to take the lock;
end if
```

Procedure $CriticalSection_p(\varepsilon)$

```
enter critical section;
exit critical section;
if last.compare-and-set(ε, null) returns false then
    pred_ε ← released;
end if
terminate;
```

A.23 Afek-Kutten-Yung Spanning Tree Algorithm

Self-stabilizing algorithms are always defined in a shared-memory framework. Therefore, the message-passing description of the Afek-Kutten-Yung spanning tree algorithm in section 17.3 is here cast in shared variables. We recall that a variable can be initialized with any value in its domain.

The variable $root_p$ is the root of the spanning tree according to p, $parent_p$ represents the parent of p in the spanning tree, and $dist_p$ is the distance value of p toward the root. The variables req_p, $from_p$, to_p, and $direction_p$ deal with join requests and corresponding grant messages. The process ID of the process that originally issued the request is stored in req_p, the neighbor from which p received the request is stored in $from_p$, the neighbor to which p forwarded the request is stored in to_p, and whether a request is being forwarded to the root of the fragment or a grant message is being forwarded to the process that originally issued the request is remembered in $direction_p$. The variable $toggle_p$ makes sure that p performs an event only when all its neighbors have copied the current values of p's local variables; $toggle_q(p)$ represents the copy at neighbor q of the value of $toggle_p$. It is assumed that a

process copies the values of all local variables of a neighbor in one atomic step.

We use the following abbreviations. $AmRoot_p$ states that p considers itself the root:

$$parent_p = \perp \ \wedge \ root_p = p \ \wedge \ dist_p = 0.$$

$NotRoot_p$ states that p does not consider itself the root and that the values of p's local variables are in line with those of its parent:

$$\begin{array}{l} parent_p \in Neighbors_p \ \wedge \ root_p > p \\ \wedge \ root_p = root_{parent_p} \ \wedge \ dist_p = dist_{parent_p} + 1. \end{array}$$

$MaxRoot_p$ states that no neighbor of p has a root value greater than $root_p$:

$$root_p \geq root_r \text{ for all } r \in Neighbors_p.$$

The network is stable, with the process with the largest ID as root, if at each process p either $AmRoot_p$ or $NotRoot_p$ holds, as well as $MaxRoot_p$.

In the following pseudocode, p repeatedly copies the values of the local variables of its neighbors, checks whether all its neighbors have copied the current values of p's local variables, and, if so, tries to perform one of several possible events. First of all, if $NotRoot_p \wedge MaxRoot_p$ does not hold and p does not yet consider itself the root, then p makes itself the root. The second kind of event arises if $MaxRoot_p$ does not hold (and so, since p skipped the first case, $AmRoot_p$ does hold). Then p asks a neighbor with a maximum root value to become its parent, if p is not already making such a request to a neighbor q, expressed by (the negation of) the predicate $Asking_p(q)$:

$$\begin{array}{l} root_q \geq root_r \text{ for all } r \in Neighbors_p \\ \wedge \ req_p = from_p = p \ \wedge \ to_p = q \ \wedge \ direction_p = \texttt{ask}. \end{array}$$

Or such a request by p may be granted by a neighbor q, expressed by the predicate $Granted_p(q)$:

$$req_q = req_p \wedge from_q = from_p \wedge direction_q = \texttt{grant} \wedge direction_p = \texttt{ask},$$

where we take q to be to_p. It is, moreover, required that p issued a request to to_p, expressed by the predicate $Requestor_p$:

$$to_p \in Neighbors_p \wedge root_{to_p} > p \wedge req_p = from_p = p.$$

In this case, to_p becomes p's parent. The third kind of event arises when p is not yet handling a request from a neighbor q, expressed by (the negation of) the predicate $Handling_p(q)$:

$$req_q = req_p \wedge from_p = q \wedge to_q = p \wedge to_p = parent_p \wedge direction_q = \texttt{ask}.$$

Here q should be either a root that issued a join request or a child of p in the spanning tree, expressed by the predicate $Request_p(q)$:

$$(AmRoot_q \wedge req_q = from_q = q) \vee (parent_q = p \wedge req_q \notin \{q, \perp\}).$$

If the four variables that capture requests are not all undefined, expressed by (the negation of) the predicate $NotHandling_p$,

$$req_p = \perp \wedge from_p = \perp \wedge to_p = \perp \wedge direction_p = \perp,$$

then p sets the values of these four variables to $\perp$. Otherwise, p forwards a request, but only if $from_{parent_p} \neq p$ (allowing $parent_p$ to first reset its join request variables). The fourth kind of event is a root p that is handling a request of a neighbor setting $direction_p$ to *grant*. Finally, the fifth kind of event is a nonroot p that finds that its request has been granted by its parent setting $direction_p$ to *grant*. Note that for the third, fourth, and fifth kinds of event, $AmRoot_p \vee NotRoot_p$ (because p skipped the first case) and $MaxRoot_p$

(because p skipped the second case). And for the fourth and fifth kinds of event, $Request_p(q) \wedge Handling_p(q)$ for some $q \in Neighbors_p$ (because p skipped the third case).

bool $toggle_p$, $toggle_p(r)$ for all $r \in Neighbors_p$;
dist $dist_p$;
proc $parent_p$, $root_p$, req_p, $from_p$, to_p;
{`ask`, `grant`, $\perp$}$direction_p$;

while *true* **do**
  copy the values of variables of all neighbors into a local copy;
  if $toggle_r(p) = toggle_p$ for all $r \in Neighbors_p$ **then**
    if $\neg(NotRoot_p \wedge MaxRoot_p) \wedge \neg AmRoot_p$ **then**
      $parent_p \leftarrow \perp$; $root_p \leftarrow p$; $dist_p \leftarrow 0$;
    else if $\neg MaxRoot_p$ **then**
      if $\neg Asking_p(r)$ for all $r \in Neighbors_p$ **then**
        $req_p \leftarrow p$; $from_p \leftarrow p$; $direction_p \leftarrow$ `ask`;
        $to_p \leftarrow q$ for a $q \in Neighbors_p$ with $root_q$ as large as possible;
      else if $Requestor_p \wedge Granted_p(to_p)$ **then**
        $parent_p \leftarrow to_p$; $root_p \leftarrow root_{to_p}$ $dist_p \leftarrow dist_{to_p} + 1$;
        $req_p \leftarrow \perp$; $from_p \leftarrow \perp$; $to_p \leftarrow \perp$; $direction_p \leftarrow \perp$;
      end if
    else if $\neg(Request_p(r) \wedge Handling_p(r))$ for all $r \in Neighbors_p$ **then**
      if $\neg NotHandling_p$ **then**
        $req_p \leftarrow \perp$; $from_p \leftarrow \perp$; $to_p \leftarrow \perp$; $direction_p \leftarrow \perp$;
      else if $from_{parent_p} \neq p \wedge Request_p(q)$ for some $q \in Neighbors_p$ **then**
        $req_p \leftarrow req_q$; $from_p \leftarrow q$; $to_p \leftarrow parent_p$; $direction_p \leftarrow$ `ask`;
      end if
    else if $AmRoot_p \wedge direction_p =$ `ask` **then**
      $direction_p \leftarrow$ `grant`;
    else if $Granted_p(parent_p)$ **then**
      $direction_p \leftarrow$ `grant`;
    end if
    $toggle_p \leftarrow \neg toggle_p$;
  end if

end while