PREV
10. Distributed Algorithms

NEXT
12. Consensus

Chapter 11. Global Properties

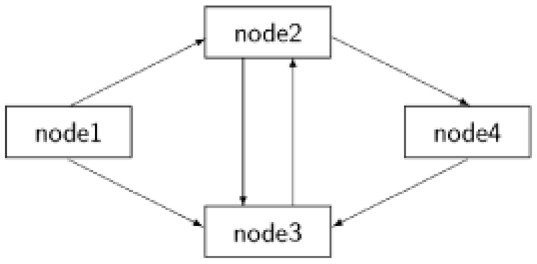
Almost by definition, there is no meaning to a global property of a distributed system. There is a parallel with Albert Einstein's theory of relativity: since information takes a finite amount of time to travel from one node to another, by the time you collect information on the global state of a system in a single node, it is "out of date." As with relativity, even the concept of time is problematic, and a central problem in the field of distributed systems is that of defining time and synchronizing clocks ([37], [48, Chapter 18]).

In this chapter, we present algorithms for two problems that can be characterized as detecting and recording global properties of distributed systems. The central concept is not *simultaneity*, but *consistency*: an unambiguous accounting of the state of the system. The first problem is to determine if the computations at each node in the system have terminated. This problem has no counterpart in concurrent systems that have shared resources, because each process can simply set a flag in shared memory to indicate that it has terminated. The second problem is to construct a snapshot of a distributed system. We would like to know where every message actually is at a "certain time," but since each node has its own clock and there are transmission delays, it is not feasible to talk about a "certain time." It does make sense, however, to compute a consistent snapshot, in the sense that every message is unambiguously attributed to a specific node or to a specific channel.

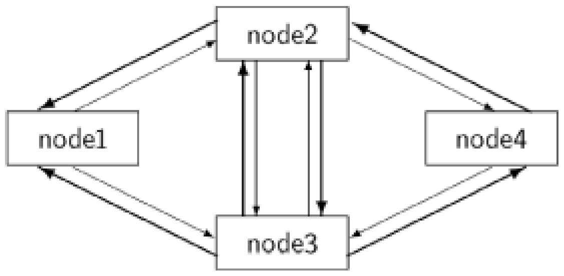
Distributed Termination

A concurrent program terminates when all of its processes have terminated, and similarly a distributed system terminates when the processes at all of its nodes have terminated. The problem posed in algorithm is developed that is live but not safe, and is then extended to the full Dijkstra–Scholten (DS) algorithm which is also safe.

The algorithm collects information from a set of nodes over a period of time in order to decide whether termination has occurred or not. We make one change from the model of distributed computation described in Section 10.1: we do not assume that each node is connected to each other node, only that the set of nodes form a connected directed graph. We do assume the existence of a unique *environment node* which has no incoming edges and from which every node is accessible by a path through the directed graph. The environment node is responsible for reporting termination. The following diagram shows such a directed graph with four nodes, where `node1` is the environment node with no incoming edges:



The DS algorithm to detect termination is to be run concurrently with the computation being carried out at each node. The computation sends messages over the edges from one node to another; the DS algorithm specifies additional statements that must be executed as part of the processing of messages by the sender and the receiver. The algorithm assumes that for each edge in the graph from node `i` to node `j`, there is a *back edge* that carries a special type of message called a *signal* from `j` to `i`. The back edges are shown by thick arrows in the following diagram:



A further assumption is that all the nodes except for the environment node are initially inactive, meaning that they are not performing any computation but are merely waiting to receive messages. The computation of the distributed system is initiated when the environment node sends messages on its outgoing edges. When a node that is not an environmental node receives its first message on any incoming edge, it can begin its computation. Eventually the computation in each node terminates and it will no longer send messages, although if it receives more messages it may be restarted. At all times, a node is able to receive, process and send signals, as required by the termination algorithm.

Under this set of assumptions, we want to develop an algorithm in which the environment node announces termination of the system if and only if the computation has terminated in all the nodes.

Preliminary Algorithm

For each message received by a destination node, it is required to eventually send a signal on the corresponding back edge to the source node. The difference between the number of messages received on an incoming edge E of node i and the number of signals sent on the corresponding back edge is denoted $inDeficit_i[E]$. The difference between the number of messages sent on outgoing edges of node i and the number of signals received on back edges is denoted $outDeficit_i$. $inDeficit_i[E]$ must be known for each incoming edge separately, while for outgoing edges it is sufficient to maintain the sum of the deficits over all the edges. When a node terminates it will no longer send messages; the sending of signals will continue as long as $inDeficit_i[E]$ is nonzero for any incoming edge. When $outDeficit_{env} = 0$ for the environment node, the algorithm announces termination.

Let us start with a preliminary algorithm (Algorithm 11.1). There are three variables for keeping track of deficits. `outDeficit` and the array `inDeficit[E]` were described above. In addition to the array of incoming deficits, the algorithm needs the sum of all these deficits:

$$\sum_{E \in incoming} inDeficit_i[E].$$

No confusion will result if we use `inDeficit` without an index to represent this sum. (Alternatively, the sum could be generated as needed in `send signal`.)

There are four parts to this algorithm that must be integrated into the underlying computation of each node:

send message. When the underlying computation sends a `message` (whatever it is), an additional statement is executed to increment the outgoing deficit.

receive message. When the underlying computation receives a `message` (whatever it is), additional statements are executed to increment the incoming deficit on that edge, as well as the total incoming deficit.

Table 11.1. Dijkstra–Scholten algorithm (preliminary)

```
integer array[incoming] inDeficit ← [0,. . . ,0]
integer inDeficit ← 0
integer outDeficit ← 0
```

- **`send message`**

```
p1: send(message, destination, myID)
p2: increment outDeficit
```

- **`receive message`**

```
p3: receive(message,source)
p4: increment inDeficit[source] and inDeficit
```

- **send signal**

```
p5: when inDeficit > 1 or
          (inDeficit = 1 and is Terminated and outDeficit = 0)
p6:   E ← some edge E with inDeficit[E] 6= 0
p7:   send(signal, E, myID)
p8:   decrement inDeficit[E] and inDeficit
```

- **receive signal**

```
p9:  receive(signal, _)
p10: decrement outDeficit
```

send signal. This is an additional process added to the program at each node. The statements of this process may be executed if the condition in the `when` clause is true; otherwise, the process is blocked. Whenever the incoming deficit is nonzero, the node may send a signal on the back edge, but it does not send the final signal until the underlying computation has terminated and the outgoing deficit is zero. It is assumed that a node can decide if its local computation has terminated; this is denoted by calling the boolean-valued function `isTerminated`.

receive signal. This is also an additional process that is executed whenever a signal is received; it decrements the outgoing deficit.

For the environment node, the only field needed is the counter for the outgoing deficit (Algorithm 11.2).

Correctness of the Preliminary Algorithm

We now prove that if the computation terminates at all nodes, eventually the environment node announces termination. For the purpose of simplifying the invariants and their proofs, it will be convenient to assume that communications are synchronous, so that every message and signal sent is immediately received; by doing this, we do not have to account for messages that are in transit. Since we are proving a liveness formula with "eventually," this does not affect correctness, because we have assumed that each individual message and signal is eventually received.

Table 11.2. Dijkstra–Scholten algorithm (env., preliminary)

```
integer outDeficit ← 0
```

- **computation**

```
p1: for all outgoing edges E
p2:    send(message, E, myID)
p3:    increment outDeficit
p4: await outDeficit = 0
p5: announce system termination
```

- **receive signal**

```
p6: receive(signal, source)
p7: decrement outDeficit
```

Notation: $inDeficit_i$ is the value of `inDeficit` at node i and similarly for $outDeficit_i$.

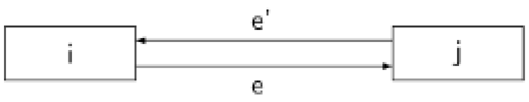
Example 11.1. Lemma

At each node `i`, $inDeficit_i \geq 0$ and $outDeficit_i \geq 0$ are invariant, as is the equality of the sums of the two deficits over *all* nodes:

Equation 11.1.

$$\sum_{i \in nodes} inDeficit_i = \sum_{i \in nodes} outDeficit_i .$$

Proof: Let `i` and `j` be arbitrary nodes such that `e` is the edge from node `i` to `j`, and `e'` is the matching back edge for sending signals from `j` to `i`:



Let n be the number of messages received by `j` on `e` and n' the number of signals sent by `j` on `e'`; then $inDeficit_j[e] = n - n'$. But n is also the number of messages sent by node `i` on `e` and n' is also the number of signals received by `i` on `e'` so $outDeficit_i[e] = n - n'$, where the notation $outDeficit_i[e]$ means that we are counting the contribution of edge `e` to the total $outDeficit_i$. We have shown that for an arbitrary edge `e`, $inDeficit_j[e] = outDeficit_i[e]$. Since each edge is an outgoing edge for exactly one node and an incoming edge for exactly one node, summing over all the edges in the graph gives

Equation 11.2.

$$\sum_{j\in nodes}\sum_{e\in incoming_j} inDeficit_j[e] = \sum_{i\in nodes}\sum_{e\in outgoing_i} outDeficit_i[e].$$

By definition,

Equation 11.3.

$$inDeficit_i = \sum_{e\in incoming_i} inDeficit_i[e]$$

Equation 11.4.

$$outDeficit_i = \sum_{e\in outgoing_i} outDeficit_i[e],$$

Substituting (11.3) and (11.4) into (11.2) proves the invariance of (11.1).

For any edge `e`, $inDeficit_j[e] \geq 0$ is invariant, because $inDeficit_j[e]$ is decremented only in `send signal` and only after explicitly checking that its value is positive. Since $outDeficit_i[e] = inDeficit_j[e]$ for the node `j` at the other end of `e`, $outDeficit_i[e] \geq 0$ is also invariant. The invariance of $inDeficit_i \geq 0$ and $outDeficit_i \geq 0$ follows by summing the individual deficits of the edges.

Example 11.2. Theorem

If the system terminates, the source node eventually announces termination.

Proof: The system terminates when the underlying computation terminates at each node; therefore, no more messages will be sent so neither $inDeficit_i$ nor $outDeficit_i$ increase after termination. By the condition in `send signal`, each non-environment node will continue sending signals until

$inDeficit_i > 1 \vee (inDeficit_i = 1 \wedge outDeficit_i = 0)$

becomes false. Using the invariants $inDeficit_i \geq 0$ and $outDeficit_i \geq 0$ of Lemma 11.1, it can be shown (exercise) that the negation of this formula is equivalent to:

Equation 11.5.

$$inDeficit_i = 0 \vee (inDeficit_i \leq 1 \wedge outDeficit_i > 0).$$

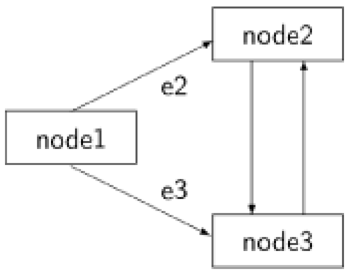
If $inDeficit_i = 0$, the formula is true regardless of the truth of $outDeficit_i > 0$), so this formula is in turn equivalent to:

$inDeficit_i = 0 \vee (inDeficit_i = 1 \wedge outDeficit_i > 0)$.

From this we deduce $inDeficit_i \leq outDeficit_i$ as follows: if $inDeficit_i = 0$ the formula follows from $outDeficit_i \geq 0$, while if $inDeficit_i = 1$, from the second disjunct we have $outDeficit_i > 0$ which is the same as $outDeficit_i \geq 1$.

We have shown that at termination, $inDeficit_i \leq outDeficit_i$ holds for all non-environment nodes. For the environment node, $inDeficit_i = 0$ (since there are no incoming edges) and the invariant $outDeficit_i \geq 0$ also imply that $inDeficit_i \leq outDeficit_i$. Since this holds for all i, it follows from (11.2) that $inDeficit_i$ must be equal to $outDeficit_i$ at all nodes, in particular for the environment node where *inDeficit*$_i$ = 0. Therefore, *outDeficit*$_i$ = 0 and the node announces termination.

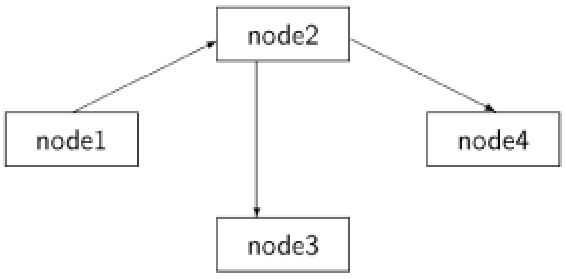
The algorithm is not safe. Consider the following set of nodes and edges:



Let `node1` send messages to both `node2` and `node3`, which in turn send messages to each other. At both nodes, *inDeficit*$_i$ = 2 and furthermore, *inDeficit*$_2$[*e2*] = 1 at `node2` and *inDeficit* $_3$[*e3*] = 1 at `node3`. By the statements at `p5` and `p6` in Algorithm 11.1, both can send signals to `node1`, which will now have *outDeficit*$_i$ = 0 and can announce termination although the other two nodes have not terminated.

The Dijkstra–Scholten Algorithm

Let us first consider the case where the directed graph is a tree. In a tree, each node except for the root has exactly one parent node, and furthermore, a tree necessarily has leaves, which are nodes with no outgoing edges. The following diagram shows a tree constructed from our directed graph:



The root is the environment node `node1` and the edges are a subset of the edges of the graph. If the graph is a tree, it is trivial to detect termination. When a leaf terminates, it sends a signal to its parent. A non-leaf node waits for signals from each of its children and then sends a signal to its parent. The root has no parent, so when it receives signals from all its children, it can conclude that the distributed system has terminated.

The tree shown in the diagram is called a *spanning tree* because every node in the graph is included in the tree. The Dijkstra–Scholten algorithm implicitly constructs a spanning tree from the directed graph. By “implicitly,” we mean that the tree is not held in any actual data structure, but it can be deduced from the internal states of the nodes. The trivial algorithm for termination in a tree can then be executed on the spanning tree.

Table 11.3. Dijkstra–Scholten algorithm

```
integer array[incoming] inDeficit ← [0,.. . ,0]
integer inDeficit ← 0
integer outDeficit ← 0
integer parent ← -1
```

- **`send message`**

```
p1: when parent ≠ -1        // Only active nodes send messages
p2:   send(message, destination, myID)
p3:   increment outDeficit
```

- **`receive message`**

```
p4: receive(message, source)
p5: if parent = -1
p6:     parent ← source
p7: increment inDeficit[source] and inDeficit
```

- **`send signal`**

```
p8:  when inDeficit > 1
p9:    E ← some edge E for which
         (inDeficit[E] > 1) or (inDeficit[E] = 1 and E ≠ parent)
p10:   send(signal, E, myID)
p11:   decrement inDeficit[E] and inDeficit
p12: or when inDeficit = 1 and is Terminated and outDeficit = 0
p13:    send(signal, parent, myID)
p14:    inDeficit[parent] ← 0
p15:    inDeficit ← 0
p16: parent ← -1
```

- **`receive signal`**

```
p17: receive(signal, _)
p18: decrement outDeficit
```

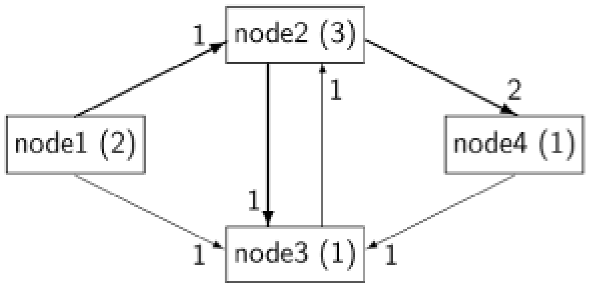
The `source` field of the *first* message to arrive at a node defines the parent of that node. We will call that incoming edge the *parent edge* of the node. The preliminary algorithm is modified so that the *last* signal from a node is sent on its parent edge; the parent then knows that no more signals will ever be received from the node. A node sends its last signal only when $outDeficit_i$ has become zero, so it becomes a leaf node in the spanning tree of non-terminated nodes.

The modifications are shown in Algorithm 11.3. The new variable `parent` stores the identity of the parent edge. The value −1 is used as a flag to indicate that the parent edge is not yet known. `send message` is modified to restrict the sending of messages to nodes that have already received a message and thus have an edge to their `parent` nodes. `received message` is modified to save the `parent` edge when the first message is received. `send signal` is modified to make sure that the last signal is sent on the `parent` edge; signals can be sent along any edge with an outstanding deficit (including the parent), as long as the final signal is saved for the parent node. When terminating, a node resets its `parent` variable because it may receive new messages causing it to be restarted.

Here a partial scenario:

Action	node1	node2	node3	node4
1 ⇒ 2	(-1,[],0)	(-1,[0,0],0)	(-1,[0,0,0],0)	(-1,[0],0)
2 ⇒ 4	(-1,[],1)	(1,[1,0],0)	(-1,[0,0,0],0)	(-1,[0],0)
2 ⇒ 3	(-1,[],1)	(1,[1,0],1)	(-1,[0,0,0],0)	(2,[1],0)
2 ⇒ 4	(-1,[],1)	(1,[1,0],2)	(2,[0,1,0],0)	(2,[1],0)
1 ⇒ 3	(-1,[],1)	(1,[1,0],3)	(2,[0,1,0],0)	(2,[2],0)
3 ⇒ 2	(-1,[],2)	(1,[1,0],3)	(2,[1,1,0],0)	(2,[2],0)
4 ⇒ 3	(-1,[],2)	(1,[1,1],3)	(2,[1,1,0],1)	(2,[2],0)
	(-1,[],2)	(1,[1,1],3)	(2,[1,1,1],1)	(2,[2],1)

The first column show the actions: `n ⇒ m` means that node `n` sends a message to node `m`. The other columns show the local data structures at each node: `(parent, inDeficit[E], outDeficit)`. The values of the array `inDeficit[E]` are shown in increasing numerical order of the nodes. (The variable containing the sum of the values in the array can be easily computed as needed.) The data structures upon completion of this part of the scenario are graphically shown in the following diagram:



The outgoing deficits are shown in parentheses next to the node labels, while the incoming deficits are shown next to the edges.

We leave it as an exercise to expand the scenario to include decisions to terminate, and sending and receiving signals.

Correctness of the Dijkstra–Scholten Algorithm

The proof of the liveness of the algorithm is almost the same as it is for the preliminary algorithm. We have delayed the sending of the last signal on the parent edge, but eventually it is sent, maintaining the liveness property.

Let us now prove the safety of the the algorithm. Define a non-environment node as *active* if and only if $parent \neq -1$.

Example 11.3. Lemma

$inDeficit_i = 0 \rightarrow outDeficit_i = 0$ is invariant at non-environment nodes.

Proof: `send message` can make $outDeficit_i = 0$ false by sending a message, but it never does so unless the node is active. A node is not active initially and becomes active in `receive message` when `inDeficit[source]` is incremented; therefore, the antecedent must be false. $inDeficit_i = 0$ becomes true only after waiting for $outDeficit_i$ to become zero, so the truth of the formula is again maintained.

Example 11.4. Lemma

The edges defined by the `parent` variables in each node form a spanning tree of the active nodes with the environment node as its root. Furthermore, for each active node, $inDeficit_i \neq 0$.

Proof: A non-environment node can become active only by receiving a message, and this causes `parent` to be set to the parent edge. Therefore the the parent edges span the active nodes. Do these edges form a tree? Yes, because `parent` is set to a non-negative value when the first message is received and is never changed again as long as as the node is active.

A node can become inactive only by resetting `inDeficit` to zero. By Lemma 11.3, this implies that $outDeficit_i = 0$, so the children of this node must not be active. Therefore, this node must have been a leaf of the spanning tree of active nodes, and its removal maintains the property that all active nodes are in the tree.

Example 11.5. Theorem

If the environment node announces termination, then the system has terminated.

Proof: If the environment node announces termination, $outDeficit_{env}$ = 0. By Lemma 11.4, if there were active nodes, they would form a spanning tree and at least one child of the environment node would have $inDeficit_i \neq 0$, which contradicts $outDeficit_{env} = 0$.

Performance

A problem with the DS algorithm is that the number of signals equals the number of messages that are sent during the entire execution of the system. But consider a distributed system that is shutting down after it has been executing for several days sending billions of messages; a similarly large number of signals will have to be sent. In such a computation, it should be possible to reduce the number of signals.

A first idea is to reduce the deficit as much as possible in a signal. The algorithm for `send signal` becomes:

```
when inDeficit > 1
  (E, N) ← selectEdgeAndNum
  send(signal, E, myID, N)
  inDeficit [ E] ← inDeficit [ E] - N
  inDeficit ← inDeficit - N
```

where `selectEdgeAndNum` is:

```
if  E ≠ parent and inDeficit [ E] ≥ 1
   return (E, inDeficit [ E])
else if inDeficit [ parent] > 1
   return (parent, inDeficit [ parent]-1)
```

The algorithm for `receive signal` is changed to:

```
receive (signal, source, N)
outDeficit ← outDeficit - N
```

A further improvement in the algorithm can be obtained by initializing all the `parent` edges to point to the environment node with ID zero, forming a “spanning bush” rather than creating a spanning tree on-the-fly. The initialization for non-environment nodes now becomes:

```
integer parent ← 0
integer array [ incoming] inDeficit ← [1, 0,. . ., 0]
integer inDeficit ← 1
integer outDeficit ← 0
```

In the exercises, you are asked to show the correctness of these modifications and to discuss the conditions under which performance is improved.

Credit-Recovery Algorithms

The DS algorithm can be difficult to implement if the deficit counts become too large to store in simple integer variables. *Credit-recovery algorithms* [53, 34] attempt to avoid this problem by using clever representations of the deficits.

In a credit-recovery algorithm, the environment node starts out with a unit “sum of money” or “weight,” $W = 1.0$. When it sends a message to a node, it “lends” the destination node part of the weight. Non-environment nodes are active if they possess a nonzero amount of weight; in turn, they share this weight when sending messages to other nodes. When a non-environment node becomes terminated, it returns all its weight to the environment node. Once all the weights have been recovered, the environment node can declare global termination.

Algorithm 11.4 is for the environment node. Every outgoing message takes with it half the weight. Once the system has been initialized, the environment node simply waits for the weight that it has lent out to be recovered. This is done by signals sent to the environment node.

Algorithm 11.5 for the non-environment nodes is very similar. A non-environment node starts with zero weight and becomes active when it receives its first message together with an initial weight, which can then be lent to other nodes. When the node terminates, it returns its current weight directly to the environment node.

Table 11.4. Credit-recovery algorithm (environment node)

```
float weight ← 1.0
```

- **computation**

```
p1: for all outgoing edges E
p2:    weight ← weight / 2.0
p3:    send(message, E, weight)
p4: await weight = 1.0
p5: announce system termination
```

- **receive signal**

```
p6: receive(signal, w)
p7: weight ← weight + w
```

Table 11.5. Credit-recovery algorithm (non-environment node)

```
constant integer parent ← 0 // Environment node
boolean active ← false
float weight ← 0.0
```

- **send message**

```
p1: if active                   // Only active nodes send messages
p2:    weight ← weight / 2.0
p3:    send(message, destination, myID, weight)
```

- **receive message**

```
p4: receive(message, source, w)
p5: active ← true
p6: weight ← weight + w
```

- **send signal**

```
p7: when terminated
p8:   send(signal, parent, weight)
p9:   weight ← 0.0
p10:  active ← false
```

While we have shown non-environment nodes waiting until they terminate to return their weight, the algorithm could have enabled or required them to return excess weight earlier. For example, in Mattern's algorithm, weight received while a node is active is immediately returned to the environment node:

```
receive (message, source, w)
if active then
   send(signal, parent, w)
else
   active ← true
   weight ← w
```

Just as the deficit counters in the DS algorithm can grow to be very large, the values of `weight` can grow to be very small. The first node to receive a message from the environment node receives a value of 2^{-1}. If it now sends out one million messages, the values of `weight` will become $2^{-2}, 2^{-3}, \ldots, 2^{-1000001}$. By storing just the negative exponent, this is no more difficult than storing the value 1000000 in `outDeficit`. The problem arises when arbitrary weights are added in the environment node, leading to values such as: $2^{-1} + 2^{-15} + 2^{-272} + \ldots + 2^{-204592} + \ldots + 2^{-822850}$. In the exercises you are asked to explore various data structures for improving the space efficiency of the algorithm.

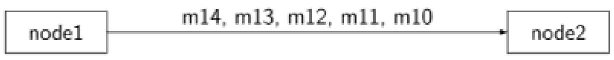
Snapshots

There is no real meaning to the global state of distributed systems, because global properties cannot be measured. The internal clocks of the processors at each node run independently and cannot be fully synchronized, and precise coordination is impossible because communications channels transfer data at uncertain rates. Nevertheless, there is meaning to the concept of a *global snapshot*: a consistent recording of the states of all the nodes and channels in a distributed system.

The state of a node is defined to be the values of its internal variables, together with the sequences of messages that have been sent and received along all edges incident with the node. (It is convenient to identify channels with the corresponding edge in the directed graph.) The state of an edge is defined as the sequence of messages sent on the edge but not yet delivered to the receiving node. For a snapshot to be consistent, each message must be in exactly one of these states: sent and in transit in an edge, or already received. It is not required that all the information of a snapshot be gathered at one particular instant, only that the assembly of the information over time be such that consistency is achieved. Algorithms for snapshots are useful because they generalize algorithms for several specific problems, such as detection of termination and deadlock.

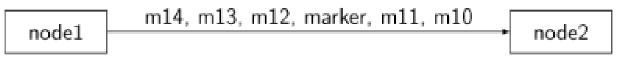
The algorithm we present was developed by K. Mani Chandy and Leslie Lamport [22]. Unlike other distributed algorithms in this book, the CL algorithm works only if the channels are FIFO, that is, if the messages are delivered in the order they were sent.

Consider two nodes and the stream of messages sent from `node1` to `node2`:



Suppose that both of these nodes are instructed to take a snapshot. (We leave for now the question of how this occurs.) Presumably we want the snapshot to indicate that `node1` has sent fourteen messages (`m1`, . . . , `m14`), that `node2` has received nine messages (`m1`,. . . , `m9`), and that messages `m10` through `m14` are still on the edge. But `node1` has no idea which messages have been received and which are still on the edge, and similarly, `node2` can only know which messages it has received, not which messages are on the edge.

To ensure consistency when a snapshot is taken, an additional message called a *marker* is sent on each edge, in order to place a boundary between messages sent before the snapshot is taken and messages sent after it is taken. Suppose again that both nodes have been instructed to take a snapshot. Let us suppose further that `node1` received this command after sending message `m11`, but before sending message `m12`. It will send the marker to `node2` immediately upon reception of the snapshot command, so the messages on the edge will be as follows:



node1 records its state immediately when the snapshot command is given; its state, therefore, includes messages m1 through m11. node2 also records its state, so the messages received by then—messages m1 through m9—are part of its state. This leaves messages m10 and m11, which have been sent but not received, to be assigned to the state of the edge. Receiving nodes are responsible for recording the state of an edge, which is defined as the set of messages received after it has recorded its state but before it has received the marker.

There is an additional possibility: a node may receive a marker before it is instructed to take a snapshot. In that case, the edge will be considered empty, as all messages received before the marker are considered to be part of the state of the receiving node.

As with the DS algorithm, it is convenient to assume that an environment node is responsible for initiating the algorithm, although the algorithm is very flexible and only requires that every node be reachable from some node that spontaneously decides to record the state. In fact, several nodes could concurrently decide to record the state and the algorithm would still succeed. The environment node will initiate the snapshot by sending a marker on each of its outgoing edges.

```
for all outgoing edges E
   send(marker, E, myID)
```

Algorithm 11.6 is the Chandy–Lamport (CL) algorithm for global snapshots. It consists of modifications to the sending and receiving of messages by the underlying algorithm, as well as a process that receives the markers. There is also a process that waits until all markers have been received and then records the state as described below.

To simplify the presentation, we do not store the contents of the messages, only their number within the sequence. Since FIFO channels are assumed these numbers suffice to specify which messages are sent and received. The internal state consists of simply the (number of the) last message sent on each outgoing edge—stored in the variable lastSent during send message, and the (number of the) last message received on each incoming edge—stored in the variable lastReceived during receive message. When the first marker is received, the state of the outgoing edges is recorded in stateAtRecord. All elements of the array are initialized to −1 and this is also used as a flag to indicate that the marker has not been received. (The assignments at p8 and p9 are array assignments; similarly, the test at p7 compares two arrays, although an additional boolean flag would suffice.)

Table 11.6. Chandy–Lamport algorithm for global snapshots

```
integer array[outgoing] lastSent ← [0, . . . , 0]
integer array[incoming] lastReceived ← [0, . . . , 0]
integer array[outgoing] stateAtRecord ← [-1, . . . , -1]
integer array[incoming] messageAtRecord ← [-1, . . . , -1]
integer array[incoming] messageAtMarker ← [-1, . . . , -1]
```

- **send message**

```
p1: send(message, destination, myID)
p2: lastSent[destination] ← message
```

- **`receive message`**

```
p3: receive(message, source)
p4: lastReceived[source] ← message
```

- **`receive marker`**

```
p5: receive(marker, source)
p6: messageAtMarker[source] ← lastReceived[source]
p7: if stateAtRecord = [-1,. . . ,-1]  // Not yet recorded
p8:     stateAtRecord ← lastSent
p9:     messageAtRecord ← lastReceived
p10:    for all outgoing edges E
p11:       send(marker, E, myID)
```

- **`record state`**

```
p12: await markers received on all incoming edges
p13: recordState
```

For incoming edges, two array variables are needed: `messageAtRecord` stores the (number of the) last message received on each edge before the state was recorded, and `messageAtMarker` stores the (number of the) last message received on each edge before the first marker was received.

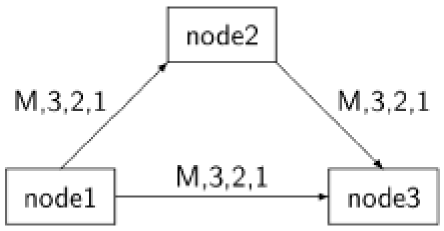
When the *first* marker is received, it initiates the recording of the state. For outgoing edges this is easy: the state is simply the last message sent on each edge. Similarly, for the incoming edge upon which the first marker was received, all messages received before the marker are part of the state of the node. For other incoming edges, it is possible that additional messages will be received after the state is recorded, but before the marker is received. The state of the edge is defined as the difference between the last message received before the node recorded its state (in `messageAtRecord`) and the last message received before the marker was received on this edge (in `messageAtMarker`).

When marker messages have been received from each incoming edge, the node can record its state, which consists of the following data:

- `stateAtRecord[E]`: the last message sent on each outgoing edge `E`.
- `messageAtRecord[E]`: the last message received on each incoming edge `E`.
- For each incoming edge `E`, if `messageAtRecord[E]` is not equal to `messgeAtMarker[E]`, then the messages from `messageAtRecord[E]+1` to `messgeAtMarker[E]` form the state of edge `E`.

The recorded global state is composed of the state recorded at the individual nodes. An algorithm similar to the DS algorithm can be used to send all the state information back to an environment node.

Let us construct a scenario for an example on the following graph:



Each of the nodes sends three messages `1`, `2` and `3` in that order and then a marker `M`. In the display of the scenario, we have abbreviated the variables: `ls` for `lastSent`; `lr` for `lastReceived`; `sr` for `stateAtRecord`; `mr` for `messageAtRecord`; `mm` for `messageAt-Marker`. Variables that have not changed from their initial values are indicated by a blank cell. The scenario starts by sending all three messages from `node1` to `node2`, where they are received. Then three messages are sent from `node1` to `node3` and from `node2` to `node3`, but they are not yet received. We display the scenario starting from the first state in this table:

Action	node1					node2	
	ls	lr	sr	mr	mm	ls	lr
	[3,3]					[3]	[3]
1M⇒2	[3,3]		[3,3]			[3]	[3]
1M⇒3	[3,3]		[3,3]			[3]	[3]
2⇐1M	[3,3]		[3,3]			[3]	[3]

2M⇒3	[3,3]	[3,3]	[3]	[3]

(To save space, the data structures of `node3`, which are empty, are omitted.) `node1`, the source node, decides to initiate a snapshot. It sends markers to `node2` and `node3` (denoted `1M⇒2` and `1M⇒3`), and records its own state as having sent three messages on each of its outgoing edges. `node2` receives the marker (`2⇐1M`) and records its state: it has sent three messages and received three messages. We can see that there are no messages on the edge from `node1` to `node2`. Finally, `node2` sends a marker to `node3` (`2M⇒3`).

The scenario continues as follows:

Action	node3				
	ls	lr	sr	mr	mm
3⇐2					
3⇐2		[0,1]			
3⇐2		[0,2]			
3⇐2M		[0,3]			
3⇐1		[0,3]		[0,3]	[0,3]
3⇐1		[1,3]		[0,3]	[0,3]
3⇐1		[2,3]		[0,3]	[0,3]
3⇐1M		[3,3]		[0,3]	[0,3]
		[3,3]		[0,3]	[3,3]

(The data structures for `node1` and `node2`, which do not change, are omitted.)

`node3` receives the three messages from `node2` and updates its `lastReceived` variable. Then it reads the marker sent by `node2` and records its state; again the state shows that the edge from `node2` is empty. Finally, `node3` receives the three messages from `node1` (updating `lastReceived` as usual), and then receives the marker, recording its state. Since a marker has already been received by this node, `messageAtRecord` is *not* updated (`p9`), but `messageAtMarker` is updated to reflect the messages received on this edge (`p6`). The difference between the first components of these two variables indicates that the three messages sent from `node1` to `node3` are considered to have been on that edge when the snapshot was taken.

Correctness of the Chandy–Lamport Algorithm

Example 11.6. Theorem

If the environment node initiates a snapshot, eventually the algorithm displays a consistent snapshot.

Proof: The termination of the algorithm follows by the connectedness of the graph. The environment node sends a marker to each child node; when the marker is received by any node, it immediately sends a marker to each of its children. By induction a marker is sent on each outgoing edge and hence received on each incoming edge, eventually causing `display state` to be executed at each node.

Since we assume that messages (and markers) are never lost, the only way that a snapshot could be inconsistent is if some message `m` appears in the state of `i`, but does not appear in exactly one of the states of the destination node or the edge. There are four cases, depending on whether `m` was sent before or after the marker from node `i` to node `j`, and whether it was received before or after node `j` recorded its state.

The first two cases are those where the message `m` was sent before `i` sent the marker.

Case 1: If `m` was received before `j` recorded its state, it will be stored in `lastReceived` by `receive message` and appear in the state of node `j` only.

Case 2: If `m` was received after `j` recorded its state, it will be stored in `lastReceived` by `receive message` but not in `messageAtRecorded`. Eventually the marker from node `i` will be received, so `m` will have a value greater than the value `messageAtRecorded` and less than or equal to the value of `messageAtMarker`; it will appear only in the state of the edge from `i` to `j`.

The other two cases are those where the message `m` was sent after `i` sent the marker to `j`. In these cases, the message is *not* part of the state of `i`, so we have to ensure that it does not appear in the state of `j` or in the state of the edge from `i` to `j`.

Case 3: `m` was received before `j` recorded its state. But this is impossible because `j` already recorded its state when it received the marker sent before `m`.

Case 4: `m` was received after `j` recorded its state. Clearly, `m` will not be recorded in the state of `j` or in the state of the edge.

In the exercises you are asked to show that the state displayed by the snapshot need not be an actual state that occurred during the computation.

Transition

This chapter and the preceding one have presented algorithms for the solution of three classical problems in distributed programming: mutual exclusion, termination detection and snapshots. The algorithms have been given under certain assumptions about the topology of the network; furthermore, we have assumed that nodes do not fail and that messages are reliably received. In the next chapter, we show how reliable algorithms can be built even if some nodes or channels are not reliable.

Exercises

DIJKSTRA–SCHOLTEN

1. How many spanning trees are there for the example with four nodes? Construct scenarios that create each of the trees. (Assume that `node1` remains the environment node.)

2. Given the details of the derivation of Equation 11.5 in Theorem 11.2.

3. Complete the scenario on page 245 assuming that no more messages are sent.

4. Prove the correctness of the modifications to the DS algorithm discussed on page 247.

5. Under what conditions does the modified DS algorithm lead to an improvement in performance?

6. Develop an algorithm that enables the environment node to collect a full description of the topology of the distributed system.

CREDIT-RECOVERY

7. When an active node receives a message with a weight, it can be added to the current node's `weight` or it can be returned immediately to the environment node. Discuss the advantages and disadvantages of each possibility.

8. Develop efficient data structures for storing arbitrary sums of weights.

1. (Huang) Store fixed ranges of nonzero values, thus $\{2^{-4}, 2^{-5}, 2^{-7}, 2^{-8}\}$ would be stored as (4, 1101).
2. (Mattern) Store the values as a set and perform implicit addition as new values are received; for example, if 2^{-8} is received and added to the above set, we get the smaller set $\{2^{-4}, 2^{-5}, 2^{-6}\}$.

CHANDY–LAMPORT

9. Construct a scenario such that the state displayed by the snapshot is not a state that occurred during the computation.

10. Where in the proof of correctness have we implicitly used the fact that the channels must be FIFO?

PREV
10. Distributed Algorithms

NEXT
12. Consensus