PREV
9. Spaces

NEXT
11. Global Properties

Chapter 10. Distributed Algorithms

In this chapter and the next two, we present algorithms designed for loosely-connected distributed systems that communicate by sending and receiving messages over a communications network.

The algorithms in this chapter are for the critical section problem. (Traditionally, the problem is called distributed mutual exclusion, although mutual exclusion is just one of the correctness requirements.) The first algorithm is the Ricart–Agrawala algorithm, which is based upon the concept of *permissions*: a node wishing to enter its critical section must obtain permission from each of the other nodes. Algorithms based upon *token-passing* can be more efficient, because permission to enter the critical section resides in a token can that easily be passed from one node to another. We present a second algorithm by Ricart and Agrawala that uses token-passing, followed by a more efficient algorithm by Neilsen and Mizuno.

The variety of models studied in distributed systems is large, because you can independently vary assumptions concerning the topology of the network, and the reliability of the channels and the nodes. Therefore, we start with a section defining our model for distributed systems.

The Distributed Systems Model

We distinguish between *nodes* and *processes*: A node is intended to represent a physically identifiable object like a computer, while the individual computers may be running multiple processes, either by sharing a single processor or on multiple processors. We assume that the internal synchronization among processes in a node is accomplished using shared memory primitives, while processes in

In Chapters 10 and 11, we assume that the nodes do not fail. Actually, a somewhat weaker assumption will suffice: A node may fail in the sense of not performing its “own” computation, as long as it continues to execute the processes that send and receive messages as required by the algorithms in these chapters. In Chapter 12 we will present algorithms that are robust under failure of individual nodes.

Communications Channels

There are additional assumptions on the communications channels between the nodes:

- Each node has a two-way channel connecting it with each other node.
- The channels deliver the messages without error, although (except for the CL algorithm in Section 11.4) not necessarily in the order they were sent.
- The transit times of messages are finite but arbitrary.

The design of an algorithm must take into account the topology of the network. The algorithms presented here assume a fully-connected topology. In one sense this is arbitrary because algorithms for other topologies could also be presented, but in another sense it is a reasonable choice, because one node can usually send a message to any other node, although it may have to be relayed through intermediate nodes before reaching its destination.

The assumption that communications are error-free is an abstraction, but again, a reasonable abstraction. There is a large body of knowledge on techniques for ensuring reliable communications in the presence of errors, and we assume that the underlying network implements these techniques. The algorithms studied here are intended for higher-level software built upon the network. Finally, the assumption of finite but arbitrary transit times for messages is consistent with the type of models that we have used before: we do not want our algorithms to be sensitive to changes in the relative speeds of the channels and the processors at the nodes, so correctness will never depend on absolute times.

The algorithm for each node will be parameterized by a unique identification number for the node:

```
constant integer myID ←. . .
```

To save space in the algorithms, this declaration will not be explicitly written.

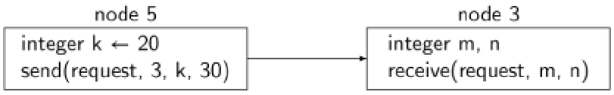
Sending and Receiving Messages

There are two statements for communications:

send(MessageType, Destination[, Parameters]). `MessageType` identifies the type of the message which is sent from this node to a `Destination` node with optional arbitrary `Parameters`.

receive(MessageType[, Parameters]). A message of type `MessageType` with optional arbitrary `Parameters` is received by this node.

Here is an example, where node 5 sends a message of type `request` to node 3, together with two integer parameters whose values are 20 and 30:



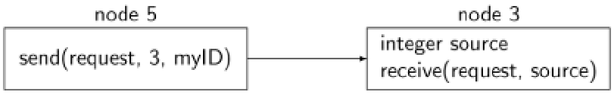
```
integer k ← 20
send(request, 3, k, 30)
```

```
integer m, n
receive(request, m, n)
```

When the message is received at node 3, these values are transferred to the variables `m` and `n` declared at that node.

There is an asymmetry between the two statements. In our algorithms, the data (message type and parameters) sent to each destination node will usually depend on the destination node. Therefore, the algorithm of the sending node must pass the destination ID to the underlying communications protocol so that the protocol can send the message to the correct node. Neither the ID of the source node nor the ID of the destination node need be sent within the message; the latter because the destination (receiving) node knows its own ID and does not have to be told what it is by the protocol, and the former because this information is not needed in most algorithms.

If the source node does wish to make its ID known to the destination node, for example to receive a reply, it must include `myID` as an explicit parameter:



```
send(request, 3, myID)
```

```
integer source
receive(request, source)
```

The mechanics of formatting parameters into a form suitable for a communications protocol are abstracted away. A process that executes a receive statement:

```
receive (message, parameters)
```

blocks until a `message` of the proper type is received; values are copied into the variable `parameters` and the process is awakened. We have thus abstracted away the processes executing the communications protocol, in particular, the process responsible for identifying the message type and unblocking processes waiting for such messages.

Concurrency within the Nodes

The model of distributed computation is intended to represent a network of nodes that are independent computers, synchronizing and communicating by sending and receiving messages. Clearly, the programs running at each node can be composed of concurrent processes that are synchronized using constructs such as semaphores and monitors. In the following chapters, the algorithms given for a node may be divided up into several concurrent processes.

NOTE

We assume atomicity of the algorithms at each process within a node. This does not apply to the critical and non-critical sections in a solution of the critical section problem, only to the pre- and postprotocols.

In particular, when a message is received, the handling of the message is considered part of the same atomic statement, and interleaving with other processes of the same node is prevented. The reason for this assumption is that we are interested in analyzing synchronization by message passing *between* nodes, and therefore ignore the synchronization within a node. The latter can be easily solved using constructs such as semaphores that will be familiar by now.

The algorithms in Chapters 11–12 are intended to perform a function for an *underlying computation* that is the real work being done at the node. The algorithms are grafted onto the underlying computations as explained in each case.

Studying Distributed Algorithms

There are several possibilities available for studying the execution of distributed algorithms. Many modern languages like Ada and Java include libraries for implementing distributed systems; in fact, you can even write distributed programs on a single personal computer by using a technique called *loopback*, where messages sent by the computer are received by the same computer without actually traversing a network. However, these libraries are complex and it takes some effort to learn how to use them, but more importantly, mastering the mechanics of distributed *programming* will tend to obscure the concepts of the distributed *algorithms*. You are likely to encounter many languages and systems over time, but the algorithmic principles will not change.

The real difficulty in understanding distributing algorithms comes from the “book-keeping”: at each step, you must remember the state of each node in the network, its local data and which messages have been sent to and received from the other nodes. To facilitate learning distributed algorithms, I have developed a software tool called DAJ for constructing scenarios step-by-step (see Appendix D.3 and [8]). At each step the tool displays the data structures at the nodes and optionally prompts for algorithm-specific messages that remain to be sent. Visualizations of virtual global data structures are also provided for some algorithms: the global queue of the Ricart–Agrawala algorithm, the spanning tree of the Dijkstra–Scholten algorithm and the knowledge trees of the Byzantine Generals algorithm.

Implementations

The model that we have described is appropriate for a network of computers connected by a protocol that enables reliable point-to-point communications. The *Transmission Control Protocol (TCP)* is the most commonly used protocol for these purposes, especially in local area networks and on the Internet. TCP is built upon a lower-level protocol called the *Internet Protocol (IP)* which is used to send packets of data. The assembling of packets into messages, together with recovery from lost or damaged packets, is the responsibility of TCP.

Implementing a distributed system, however, involves more than just sending and receiving messages. The nodes have to learn the network topology which may change dynamically as new nodes are added and as others are removed, perhaps as a result of faults.

Processes have to be assigned to nodes, and the location of data must be specified. If the nodes are *heterogeneous*, that is, if they use different computers, operating systems and languages, there will be problems adapting the data in the messages to the nodes. On a heterogeneous network, some nodes may be more appropriate for executing a certain process than others, and the assignment of processes to nodes must take this into account.

The subject of the implementation of distributed systems is beyond the scope of this book. We have already discussed two types of systems. Remote procedure calls, as implemented by the Java Remote Method Invocation and Ada, integrate extremely well with the object-oriented constructs of those languages. Implementations of Linda and JavaSpaces distribute the data—the space—over the nodes of the network, so that programming an application is relatively simple because transferring data from one node to another is automatic.

There are two important and very popular implementations of distributed systems that present the programmer with an abstract view of the underlying network: the *Parallel Virtual Machine (PVM)* [27] and the *Message Passing Interface (MPI)* [29].

PVM is a software system that presents the programmer with a virtual distributed machine. This means that, regardless of the actual network configuration, the programmer sees a set of nodes and can freely assign processes to nodes. The architecture of the virtual machine can be dynamically changed by any of the nodes in the system, so that PVM can be used to implement fault-tolerant systems. PVM is portable over a wide range of architectures, and furthermore, it is designed for *interoperability*, that is, the same program can run on a node of any type and it can send messages to and receive message from a node of any type.

While PVM is a specific software system that you can download and use, MPI is a *specification* that has been implemented for many computers and systems. It was originally designed to provide a common interface to multiprocessors so that applications programs written to execute on one computer could be easily ported to execute on another.

MPI was designed to emphasize performance by letting each implementation use the most efficient constructs on the target computer, while the designers of PVM emphasized interoperability at the expense of performance (an approach more suited to a network of computers than to a multiprocessor). While there has been some convergence between the two systems, there are significant differences that have to be taken into account when choosing a platform for implementing a distributed system. Detailed comparisons of PVM and MPI can be found in articles posted on the websites of the two systems listed in Appendix E.

Distributed Mutual Exclusion

Recall that a process in the critical section problem is structured as an infinite loop consisting of the non-critical section, the preprotocol, the critical section and the postprotocol. As before, we are not concerned with the computation within the non-critical or critical sections; the critical section might consist of the modification of a database accessible from any node within the network.

The algorithm we describe was proposed by Glenn Ricart and Ashok K. Agrawala, and it is based upon ticket numbers as in the bakery algorithm (Section 5.2). Nodes choose ticket numbers and compare them; the lowest number is allowed to enter the critical section, while other numbers must wait until that node has left the critical section. In a distributed system, the numbers cannot be directly compared, so they must be sent in messages.

Initial Development of the Algorithm

We will develop the algorithm in stages. Algorithm 10.1 is an initial outline. The algorithm is structured as two concurrent processes: a `Main` process that contains the familiar elements of the critical section problem, while the second process `Receive` executes a portion of the algorithm upon receipt of a `request` message.

Table 10.1. Ricart–Agrawala algorithm (outline)

```
integer myNum ← 0
set of node IDs deferred ← empty set
```

- **Main**

```
loop forever
   non-critical section
   myNum ← chooseNumber
   for all other nodes N
     send(request, N, myID, myNum)
   await reply's from all other nodes
   critical section
   for all nodes N in deferred
     remove N from deferred
     send(reply, N, myID)
```

- **Receive**

```
integer source, requestedNum
loop forever
   receive(request, source, requestedNum)
   if requestedNum < myNum
      send(reply, source, myID)
   else add source to deferred
```

The preprotocol begins with the selection of an arbitrary ticket number, which is then sent in a `request` message to all other processes. The process then waits until it has received `reply` messages from all these processes, at which point it may enter its critical section.

Before discussing the postprotocol, let us consider what happens in the `Receive` process. That process receives the `request` message and compares the `requested-Num` with the number `myNum` chosen by the node. (For now we assume that the numbers chosen are distinct.) If `requestedNum` is less than `myNum`, this means that the sending node has taken a *lower* ticket number, so the receiving node agrees to let it enter the critical section by sending a `reply` message to the node that was the `source` of the `request` message. The node that has the lowest ticket number will receive replies from all other nodes and enter its critical section.

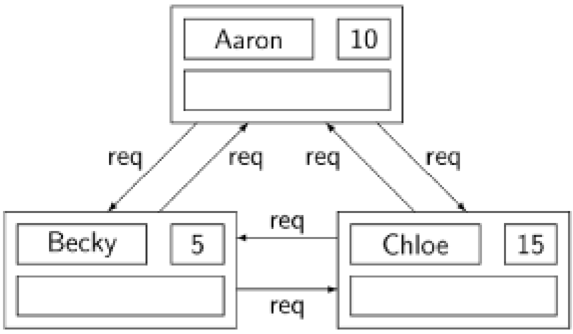
If, however, `requestedNum` is greater than `myNum`, the receiving node has the lower ticket number, so it notifies the sending node *not* to enter the critical section. Cleverly, it does this by simply not sending a `reply` message! Since a node must receive replies from *all* other nodes in order to enter its critical section, the absence of a reply is sufficient to prevent it from prematurely entering its critical section.

A set of processes called `deferred` is maintained. This set contains the IDs of nodes that sent `request` messages with higher ticket numbers than the number chosen by the node. In the postprotocol that is executed upon completion of the critical section, these nodes are finally sent their `reply` messages.

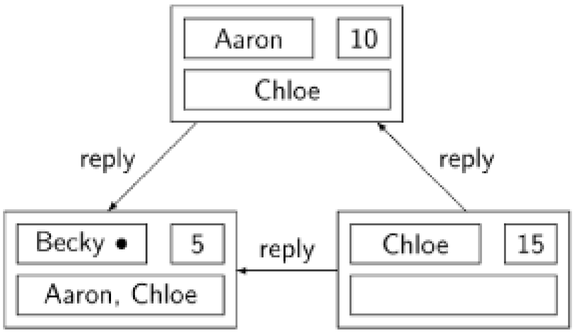
The Scenario of an Example

We now demonstrate a scenario using diagrams to show the states that occur in the scenario. Rather than use numbers, we personify the nodes by giving them names for IDs. Nodes are shown as rectangles containing the data structures: `myID`, the identity of the node, in the upper left corner; `myNum`, the number chosen by the node, in the upper right corner; and `deferred`, the set of deferred nodes, in the lower part of the rectangle. Arrows show the messages sent from one node to another.

Here is the state of the system after all nodes have chosen ticket numbers and sent `request` messages (abbreviated `req`) to all other nodes:

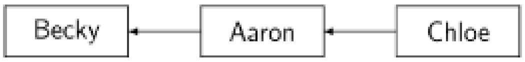


The following diagram shows the result of the execution of the loop body for `Receive` at each node:



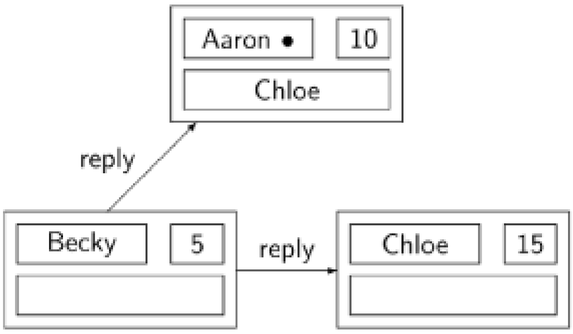
Chloe sends reply messages to both Aaron and Becky, because both have lower numbers than she does. Becky does not send any reply messages, instead adding Aaron and Chloe to her set of deferred nodes, because both processes have higher numbers than she does. Aaron's number is in between the other two, so he sends a reply to Becky while appending Chloe to his deferred set.

At this point, the three nodes have constructed a *virtual queue*:

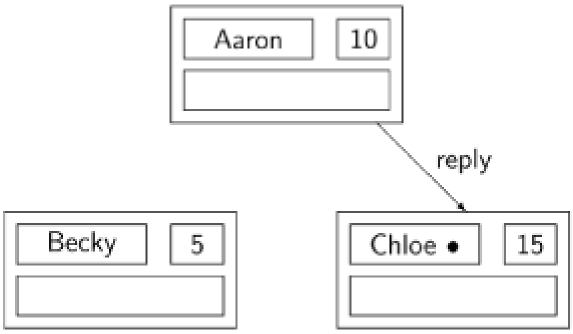


The queue is virtual because it does not actually exist as a data structure in any node, but the effect of the messages is to order the nodes as if in a queue.

Becky can now execute her critical section, denoted by the symbol • next to her name. When she completes her critical section, Becky sends the two deferred reply messages to Aaron and Chloe:



Aaron has now received both replies and can enter his critical section. Upon completion, he sends a reply message to Chloe who can now enter her critical section:



There are many more possible scenarios even with this choice of ticket numbers. In our scenario, all request messages were sent before any `reply` messages, but it is also possible that a node immediately replies to a received `request` message.

Equal Ticket Numbers

Since the system is distributed it is impossible to coordinate the choice of ticket numbers, so several processes can choose the same number. In a truly symmetric algorithm where all nodes execute exactly the same program, there is no way to break ties, but here we can use the symmetry-violating assumption that each process has a distinct ID. The comparison of numbers will use the IDs to break ties, by replacing line `p11` in the `Receive` process by:

```
if   (requestedNum < myNum) or
    ((requestedNum = myNum) and (source < myID))
```

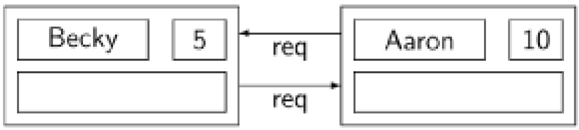
As with the bakery algorithm (Section 5.2), it is convenient to introduce a new notation for this comparison:

```
if requestedNum ≪ myNum
```

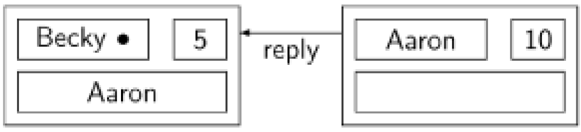
Choosing Ticket Numbers

We have not specified the manner in which ticket numbers are chosen, but have left it as an arbitrary function `chooseNumber`. If the choice is in fact arbitrary, it is not hard to find a scenario even with just two nodes that leads to a violation of mutual exclusion.

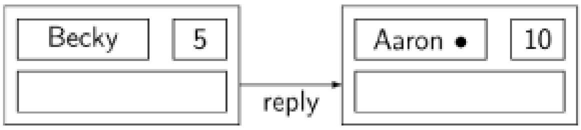
Let `Aaron` and `Becky` choose the same ticket numbers as before and send requests to each other:



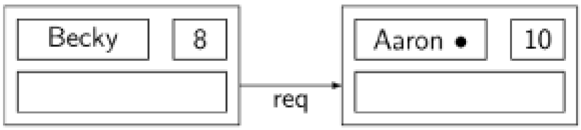
`Aaron` sends a reply enabling `Becky` to enter her critical section:



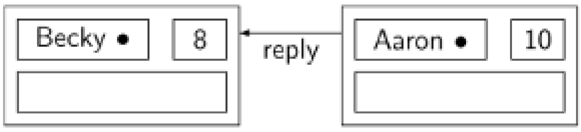
Eventually, `Becky` completes her critical section and sends a reply to the deferred node `Aaron`:



Suppose now that Becky quickly re-executes her loop body, choosing a new ticket number `8` and sending the request to `Aaron`:



`Aaron` will compare ticket numbers and send a reply to `Becky`, enabling her to enter her critical section before `Aaron` leaves his:



To prevent this violation of mutual exclusion, we must ensure (as in a real bakery with physical tickets) that the numbers chosen are monotonic, in the sense that a node will choose a number that is higher than all other ticket numbers *it knows about*. To each node we add a variable `highestNum`, which stores the highest number received in any `request` message so far. `p2: myNum ← chooseNumber` is implemented to assign a new ticket number that is larger than `highestNum`:

```
myNum ← highestNum + 1
```

In process `Receive` the following statement is inserted after `p10: receive(. . .)`:

```
highestNum ← max(highestNum, requestedNum)
```

Quiescent Nodes

There is a problem associated with the possibility that a node is not required by the specification of the critical section problem to actually attempt to enter its critical section. In our model of a distributed system, all that we require is that the `Receive` process continues to receive requests and send replies even if the `Main` process has terminated in its non-critical section.

Table 10.2. Ricart–Agrawala algorithm

```
integer myNum ← 0
set of node IDs deferred ← empty set
integer highestNum ← 0
boolean requestCS ← false
```

- **`Main`**

```
    loop forever
p1:    non-critical section
p2:    requestCS ← true
p3:    myNum ← highestNum + 1
p4:    for all other nodes N
p5:      send(request, N, myID, myNum)
p6:    await reply's from all other nodes
p7:    critical section
p8:    requestCS ← false
p9:    for all nodes N in deferred
p10:     remove N from deferred
p11:     send(reply, N, myID)
```

- **`Receive`**

```
     integer source, requestedNum
     loop forever
p12:   receive(request, source, requestedNum)
p13:   highestNum ← max(highestNum, requestedNum)
p14:   if not requestCS or requestedNum ≪ myNum
p15:      send(reply, source, myID)
p16:   else add source to deferred
```

However, `Receive` will only send a reply if `myNum` is greater than `requestedNum`. Initially, `myNum` is zero so the node will *not* send a reply. Similarly, if the node remains inactive for a long period while other nodes attempt to enter their critical sections, the selection of ever-increasing ticket numbers will cause them to become larger than the current value of `myNum` at the quiescent node.

To solve this problem, we add an additional flag `requestCS` which the `Main` process sets before choosing a ticket number and resets upon exit from its critical section. If this flag is not set, the `Receive` process will immediately send a reply; otherwise, it will compare ticket numbers to decide if a reply should be sent or deferred.

The complete RA algorithm is shown as Algorithm 10.2.

Correctness of the Ricart–Agrawala Algorithm

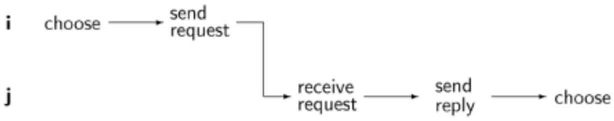
The RA algorithm satisfies the mutual exclusion property, does not deadlock and does not starve individual nodes.

Example 10.1. Theorem

Mutual exclusion holds in the Ricart–Agrawala algorithm.

Proof: To prove mutual exclusion, suppose to the contrary that two nodes `i` and `j` are in the critical section. By the structure of the algorithm, there must have been a last time when each node chose a ticket number, $myNum_i$ and $myNum_j$, respectively. There are three cases to consider.

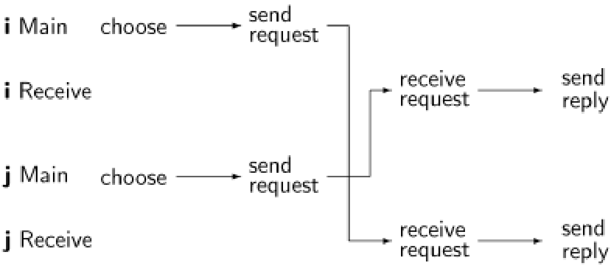
Case 1: Node `j` chose $myNum_j$ after it sent its reply to node `i`. The arrows in the following diagram show the precedence relationships between the statements of the two processes.



The horizontal arrows follow either from the sequential nature of the processes at the nodes or from the assumption of this case. The precedence represented by the vertical arrow from `send request` in node `i` to `receive request` in node `j` follows because a message cannot be received until after it is sent. Node `j` sends the `reply` only after updating (in `Receive`) `highestNum` to a value greater than or equal to $myNum_i$, and therefore it will choose a number $myNum_j$ greater than $myNum_i$. When node `i` receives this `request` it will not send a `reply` as long as it is in its critical section. We conclude that in this case both nodes cannot be in their critical sections at the same time.

Case 2: Node `i` chose *myNum* $_i$ *after* it sent its reply to node `j`. This case is symmetric with Case1.

Case 3: Nodes `i` and `j` chose their ticket numbers *myNum* $_i$ and *myNum* $_j$, respectively, *before* sending `reply` messages to each other. The diagram for this case is somewhat more complicated, because we have to take into account the two concurrent processes at each node, shown one under another. No relative timing information should be read from the diagram, only the precedence information derived from the structure of the algorithm and the fact that a message is received after it is sent.



Consider now what happens when the nodes `i` and `j` decide to reply. For the entire period between choosing ticket numbers and making this decision, they both want to enter the critical section so that *requestCS* = *true*, and the ticket numbers that are being compared are the same ticket numbers that they have chosen. So `i` will reply if and only if *myNum* $_j$ ≪ *myNum* $_i$ and `j` will reply if and only if *myNum* $_i$ ≪ *myNum* $_j$. By the definition of ≪ both expressions cannot be true, so one of these expressions will evaluate to false, causing that node to defer its reply. We conclude that in this case too both nodes cannot be in their critical sections at the same time.

Example 10.2. Theorem

The Ricart–Agrawala algorithm is free from starvation and therefore free from deadlock.

Proof: Suppose that node `i` requests entry to the critical section by setting `requestCS`, choosing a ticket number *myNum* $_i$ and sending `request` messages to all other nodes. Can node `i` be blocked forever waiting for `reply` messages at `p6`? At some point in time *t*, these `request` messages will have arrived at all the other nodes, and their variables `highestNum` will be set to values greater than or equal to *myNum* $_i$. So from *t* onwards, *any* process attempting to enter its critical section will choose a ticket number higher than *myNum* $_i$.

Let *aheadOf* (*i*) be the set of nodes that at time *t* are requesting to enter the critical section and have chosen a ticket numbers lower than *myNum* $_i$. We have shown that processes can only leave *aheadOf* (*i*), never join it. If we can show that eventually some node will leave *aheadOf* (*i*), it follows by (numerical) induction that the set is eventually empty, enabling node `i` to enter its critical section.

Since the ticket numbers form a monotonic sequence under ≪, there must be a node `k` in *aheadOf* (*i*) with a minimal number. By the assumptions on sending and receiving messages, the `request` messages from node `k` must eventually be sent and received; since `k` has the minimal ticket number, all other nodes will send `reply` messages that are eventually received. Therefore, node `k` will eventually enter and exit its critical section, and leave the set *aheadOf* (*i*).

The RA Algorithm in Promela^L

Promela is well-suited for modeling distributed systems. Although we have emphasized the distinction between the distributed message passing among the nodes and the shared-memory synchronization of concurrent processes at individual nodes, for modeling purposes a single program can do both. In the model of the RA algorithm, for each of the *NPROC* nodes there will be two processes, a `Main` process and a `Receive` process; each pair is parameterized by a node number that we have been denoting as `myID`. The global variables *within the nodes* are declared as arrays, one element for each node:

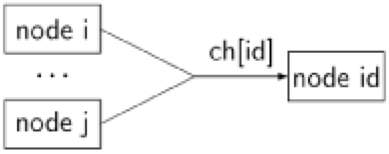
```
byte myNum[NPROCS];
byte highestNum[NPROCS];
bool requestCS[NPROCS];
chan deferred[ NPROCS] = [NPROCS] of { byte };
```

The declaration of the sets `deferred` as channels is explained below.

The channels between the nodes are modeled, of course, by channels:

```
mtype = { request, reply };
chan ch[NPROCS] = [NPROCS] of { mtype, byte, byte };
```

The intended interpretation is that `ch[id]` is the channel *to* the receiving node `id`:



We are using the fact that a Promela channel can be used by more than one process. The channel capacity is defined to be `NPROCS` in order to model out-of-order delivery of messages.

The `Main` process is shown in Listing 10.1. **atomic** is used to prevent interleaving between setting `requestCS` and choosing a number. The symbol `??` in

```
ch[myID] ?? reply, _, _;
```

means to remove any arbitrary `reply` message from the channel, without regard to the FIFO order normally inherent in the definition of a channel.

To model the set of deferred nodes, we could have defined an array of ID numbers, but it is simpler to use a channel as a queue. The `Receive` process (below) sends deferred processes to the channel and the `Main` process receives node IDs until the channel is empty. These IDs are then used as indices to send `reply` messages.

Example 10.1. Main process for Ricart–Agrawala algorithm

The `Re-`

```
proctype Main(byte myID) {
   do ::
      atomic {
         requestCS[myID] = true;
         myNum[myID] = highestNum[myID] + 1;
      }

      for (J, 0, NPROCS-1)
         if
         :: J != myID ->
              ch[J] ! request, myID, myNum[myID];
         :: else
         fi
      rof (J);

      for (K, 0, NPROCS-2)
         ch[myID] ?? reply, _, _;
      rof (K);

      critical_section ();
      requestCS[myID] = false;

      byte N;
      do
        :: empty(deferred[myID]) -> break;
        :: deferred [myID] ? N -> ch[N] ! reply, 0, 0
      od
   od
}
```

ceive process is shown in Listing 10.2. **atomic** is used to prevent interleaving with the `Main` process, and `??` is used to receive an arbitrary `request` message.

Example 10.2. Receive process for Ricart–Agrawala algorithm

```
proctype Receive(byte myID) {
   byte reqNum, source;
   do ::
      ch[myID] ?? request, source, reqNum;

      highestNum[myID] =
          ((reqNum > highestNum[myID]) ->
              reqNum : highestNum[myID]);

      atomic {
         if
         :: requestCS[myID] &&
              ((myNum[myID] < reqNum) ||
              ((myNum[myID] == reqNum) &&
                    (myID < source)
             )) ->
                 deferred [myID] ! source
         :: else ->
              ch[source] ! reply, 0, 0
         fi
      }
   od
}
```

Token-Passing Algorithms

The problem with a permission-based algorithm is that it can be inefficient if there are a large number of nodes. Furthermore, the algorithm does not show improved performance in the absence of contention; a node wishing to enter its critical section must always send and receive messages from all other nodes.

In token-based algorithms, permission to enter the critical section is associated with the possession of an object called a *token*. Mutual exclusion is trivially satisfied by token-based algorithms, because their structure clearly shows this association. The algorithms are also efficient: only one message is needed to transfer the token and its associated permission from one node to another, and a node possessing the token can enter its critical section any number of times without sending or receiving any messages. The challenge is to construct an algorithm for passing the token that ensures freedom from deadlock and starvation.

Table 10.3. Ricart–Agrawala token-passing algorithm

```
boolean haveToken ← true in node 0, false in others
integer array[NODES] requested ← [0,. . . ,0]
integer array[NODES] granted ← [0,. . . ,0]
integer myNum ← 0
boolean inCS ← false
```

```
    sendToken
    if exists N such that requested[N] > granted[N]
       for some such N
          send(token, N, granted)
          haveToken ← false
```

- **Main**

```
    loop forever
p1:    non-critical section
p2:    if not haveToken
p3:       myNum ← myNum + 1
p4:       for all other nodes N
p5:          send(request, N, myID, myNum)
p6:       receive(token, granted)
p7:       haveToken ← true
p8:    inCS ← true
p9:    critical section
p10:   granted[myID] ← myNum
p11:   inCS ← false
p12:   sendToken
```

- **Receive**

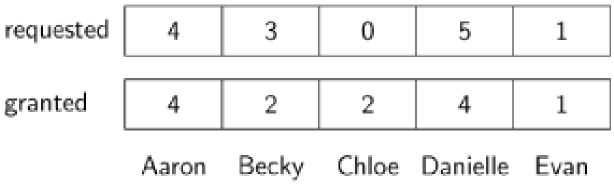
```
    integer source, reqNum
    loop forever
p13:   receive(request, source, reqNum)
p14:   requested[source] ← max(requested[source], reqNum)
p15:   if haveToken and not inCS
p16:      sendToken
```

Algorithm 10.3 was developed by Ricart and Agrawala as a token-passing version of their original algorithm.[1]

Look first at the algorithm without considering the content of the `token` message sent in `sendToken` (called from `p12` and `p16`), and received in `p6`. Unlike the permissions in the RA algorithm, the passing of the permission by a token is contingent; a token will not be passed unless it is needed. As long as it is not needed, a node may hold the token as indicated by the boolean variable `haveToken`; the `if` statement at `p2` enables the node to repeatedly enter its critical section.

Two data structures are used by the algorithm to decide if there are outstanding requests that require a node to give up the token it holds. The `token` message includes an array `granted` whose elements are the ticket numbers held by each node the *last* time it was granted permission to enter its critical section. In addition, each node stores in the array `requested` the ticket numbers accompanying the *last* `request` messages from the other nodes. While each node may have different data in `requested` depending on when `request` messages are received, only the copy of `granted` maintained by the node with the token is meaningful, and it is passed from one node to another as part of the token. It enables the algorithm to decide unambiguously what outstanding `request` messages have not been satisfied.

Here is an example of these two data structures maintained by the node `Chloe` in a five-node system:



	Aaron	Becky	Chloe	Danielle	Evan
requested	4	3	0	5	1
granted	4	2	2	4	1

`request` messages have been received at node `Chloe` from `Becky` and `Danielle` that were sent *after* the last time they were `granted` permission to enter their critical sections. Therefore, if Chloe holds the token and is not in her critical section, she must send it to one of them. If she is in the critical section, Chloe sends the token upon leaving, thereby preventing starvation that would occur if she immediately reentered her critical section. However, if Chloe has not received a `request` message from another process, she can retain the token and re-enter her critical section.

The rest of the algorithm simply maintains these data structures. The current ticket number of a node is incremented in `p3` and then sent in `request` messages to enable the array `requested` to be updated in the other nodes. `granted` is updated when a node completes its critical section.

It is easy to show (exercise) that there is only a single token and that this ensures that mutual exclusion is satisfied.

Example 10.3. Theorem

The algorithm does not deadlock.

Proof: If some nodes wish to enter their critical sections and cannot, they must be blocked indefinitely waiting at `receive(token, granted)`. For all such nodes `i`, `requested[i]` is eventually greater than `granted[i]` in the node holding the token, so at `p16` a `token` message will be sent to some node when its `request` is received, unless the node with the token is in its critical section. By the assumed progress of critical sections, the token will eventually be sent in `p12`.

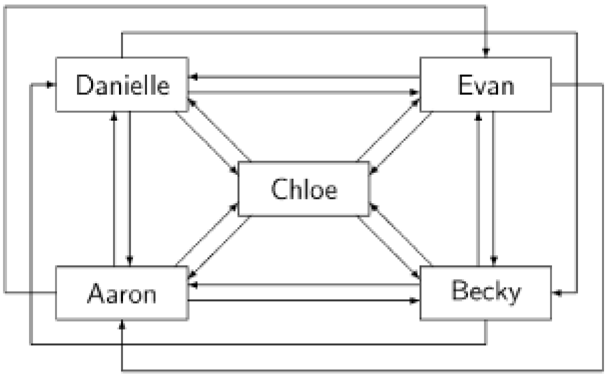
Starvation is possible because of the arbitrary selection of a requesting process `for some such N` in the algorithm for `send-Token`. In the exercises, you are asked to modify the algorithm to prevent starvation.

The original RA algorithm required that a node entering its critical section send $N - 1$ `request` messages and receive $N - 1$ `reply` messages. These messages are short, containing only a few values of fixed size. In the token-passing algorithm, a node need not send messages if it possesses the token; otherwise, it needs to send $N - 1$ `request` messages as before, but it only need receive one `token` message. However, the token message is long, containing N ticket numbers for the other processes. If N is large, this can be inefficient, but in practice, the token-passing algorithm should be more efficient than the RA algorithm because there is a large overhead to sending a message, regardless of its size.

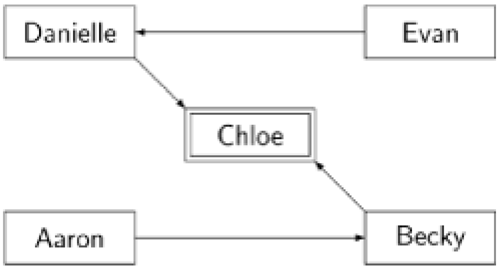
Tokens in Virtual Trees^A

The problem with the RA token-passing algorithm is that the queue of waiting processes is carried along with the token message. In this section we describe the Neilsen–Mizuno algorithm for distributed mutual exclusion [54] that is based upon passing a small token in a set of virtual trees that is implicitly constructed by the algorithm. It is recommended that you first study the concept of virtual data structures in the context of the Dijkstra–Scholten algorithm (Section 11.1) before studying this algorithm.

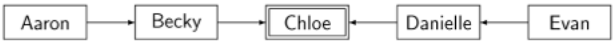
Before presenting the algorithm, let us work out an example on a five-node distributed system, where we assume that the nodes are fully connected, that is, that any node can send a message directly to any other node:



Let us now suppose that the system is initialized so that a set of edges is selected that define an arbitrary spanning tree with the directed edges pointing to the root, for example:

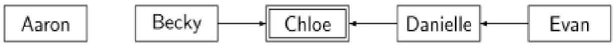


It will be convenient to draw the nodes in a horizontal line:



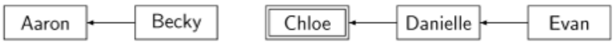
The node at the root of the tree possesses the token and is possibly in its critical section. The node possessing the token, in this case `Chloe`, is drawn with double lines. As in the RA token-passing algorithm, `Chloe` can enter her critical section repeatedly as long as she does not receive any `request` messages.

Suppose now that `Aaron` wishes to enter his critical section. He sends to his `parent` node, `Becky`, a message `(request, Aaron, Aaron)`; the first parameter is the ID of the *sender* of the message, while the second parameter is the ID of the *originator* of the request. In the first message sent by a node wishing to enter its critical section, the two parameters are, of course, the same. After sending the message, `Aaron` zeroes out his `parent` field, indicating that he is the root of a new tree:

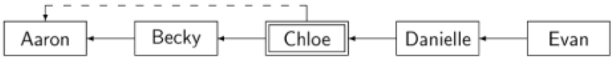


The intent is that `Aaron` will eventually become the root with the token, and enter his critical section; he receives requests from other nodes so that he can pass the token to another node when he leaves his critical section.

`Becky` will now relay the request to the root by sending the message `(request, Becky, Aaron)`. `Becky` is sending a message on behalf of `Aaron` who wishes to enter his critical section. `Becky` also changes her `parent` field, but this time to `Aaron`, the sender of the message she is relaying:

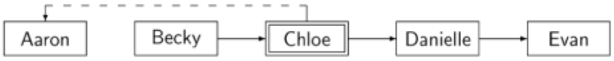


The node receiving this message, `Chloe`, possesses the token; however, by the assumption in this scenario, she is in her critical section and must therefore defer sending the token. `Chloe` sets a field `deferred` to the value of the `originator` parameter in the message. This is indicated by the dashed arrow in the following diagram:

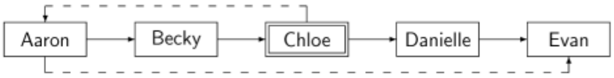


`Chloe` also sets her `parent` field to the sender of the message so that she can relay other messages.

Suppose now that `Evan` concurrently originates a request to enter his critical section, and suppose that his request is received by `Chloe` *after* the request from `Aaron`. `Chloe` is no longer a root node, so she will simply relay the message as an ordinary node, setting her `parent` to be `Danielle`. The chain of relays will continue until the message `(request, Becky, Evan)` arrives at the root, `Aaron`:

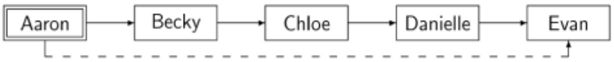


`Aaron` is a *root node without the token*, so he knows that he must appear in the `deferred` field of some other node. (In fact, he is in the `deferred` field of `Chloe` who holds the token.) Therefore, `Aaron` places the `originator` he has just received in his `deferred` field and as usual sets his `parent` field to the sender:

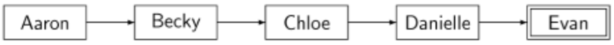


The deferred fields implicitly define a queue of processes waiting to enter their critical sections.

When Chloe finally leaves her critical section, she sends a token message to the node Aaron listed in her deferred field, enabling Aaron to enter his critical section:



When Aaron leaves his critical section, and then sends the token to Evan who enters his critical section:



The state of the computation returns to a quiescent state with one node possessing the token.

Algorithm 10.4, the Neilsen–Mizuno algorithm, is very memory-efficient. Only three variables are needed at each node: the boolean flag holding, which is used to indicate that a root node holds the token but is not in its critical section, and the node numbers for parent and deferred. The messages are also very small: the request message has two integer-valued parameters and the token message has no parameters. We leave it as an exercise to write the statements that initialize the parent fields.

The time that elapses between sending a request and receiving the token will vary, depending on the topology of the virtual trees. In Algorithm 10.3, a request message was sent to *every* node, so the token, if available, could be directly sent to the next requesting process; here the request message might have to be relayed through many nodes, although the token itself will be sent directly to the node whose ID is passed along in the originator field.

A proof of the correctness of the algorithm can be found in [54].

Transition

We have shown several algorithms for solving the critical section problem, which is the basic problem of concurrent programming. In a certain sense, the critical section problem is not typical of distributed algorithms, because it assumes that there is a centralized resource that needs to be protected from being accessed by several nodes. The next chapter poses problems that go to the heart of distributed programming: implementing cooperation among a set of independent nodes.

Table 10.4. Neilsen–Mizuno token-passing algorithm

```
integer parent ← (initialized to form a tree)
integer deferred ← 0
boolean holding ← true in the root, false in others
```

- **Main**

```
    loop forever
p1:    non-critical section
p2:    if not holding
p3:       send(request, parent, myID, myID)
p4:       parent ← 0
p5:       receive(token)
p6:    holding ← false
p7:    critical section
p8:    if deferred ≠ 0
p9:       send(token, deferred)
p10:      deferred ← 0
p11:   else holding ← true
```

- **Receive**

```
    integer source, originator
    loop forever
p12:   receive(request, source, originator)
p13:   if parent = 0
p14:      if holding
p15:         send(token, originator)
p16:         holding ← false
p17:      else deferred ← originator
p18:   else send(request, parent, myID, originator)
p19:   parent ← source
```

Exercises

RICART–AGRAWALA

1. What is the total number of messages sent in a scenario in which all nodes enter their critical sections once?

2.

1. Construct a scenario in which the ticket numbers are unbounded.
2. What is the maximum difference between two ticket numbers?

3. What would happen if several nodes had the same value for `myID`?

4.

1. Can the `deferred` lists of all the nodes be nonempty?
2. What is the maximum number of entries in a single `deferred` list?

3. What is the maximum number of entries in all the `deferred` lists together?

The following four questions refer to Algorithm 10.2 and do not assume that the processes in each node are executed atomically.

5. Suppose that we exchanged the lines `p8` and `p9`-`p11`. Would the algorithm still be correct?

6. Why don't we have to add `highestNum ← max(highestNum, myNum)` after statement `p3: myNum ← highestNum + 1`?

7. Can the statement `p13: highestNum ← max(highestNum, requestNum)` be replaced by `p13: highestNum ← requestNum`?

8. Construct a faulty scenario in which there is interleaving between the choosing statements `p2`-`p3` and the statements that make the decision to defer `p13-16`.

9. Modify the Promela program for the RA algorithm to implement the deferred list using an array instead of a channel.

RICART–AGRAWALA TOKEN-PASSING ALGORITHM

10. Prove that mutual exclusion holds.

11. In node `i`, can `requested[j]` be less than `granted[j]` for $j \neq i$?

12. Show that the algorithm is not correct if the pre- and postprotocols and the processing of a `request` message in `Receive` are not executed under mutual exclusion.

13. Construct a scenario leading to starvation. Modify the algorithm so that it is free from starvation. You can either modify the statement `for some such N` in `SendToken` (Ricart–Agrawala) or modify the data structure `granted` (Suzuki–Kasami).

NEILSEN–MIZUNO ALGORITHM

14. Develop a distributed algorithm to initialize the virtual tree in Algorithm 10.4.

[1] The algorithm appears in the *Authors' Response* to a *Technical Correspondence* by Carvalho and Roucairol [21]. A similar algorithm was discovered independently by Suzuki and Kasami [64].

PREV
9. Spaces

NEXT
11. Global Properties