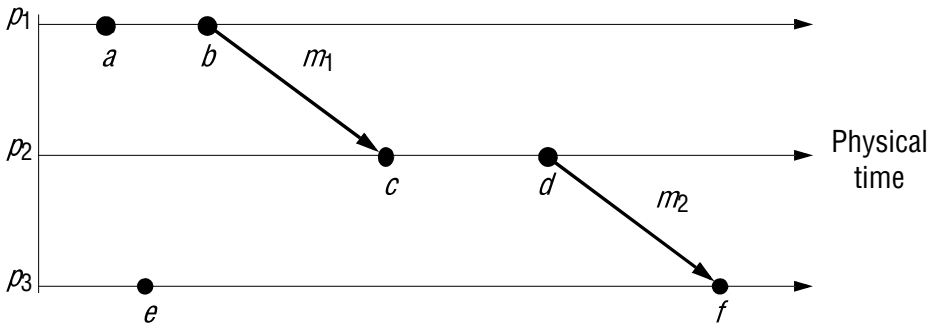
Figure 14.5 Events occurring at three processes



14.4 Logical time and logical clocks

From the point of view of any single process, events are ordered uniquely by times shown on the local clock. However, as Lamport [1978] pointed out, since we cannot synchronize clocks perfectly across a distributed system, we cannot in general use physical time to find out the order of any arbitrary pair of events occurring within it.

In general, we can use a scheme that is similar to physical causality but that applies in distributed systems to order some of the events that occur at different processes. This ordering is based on two simple and intuitively obvious points:

- If two events occurred at the same process p_i $(i = 1, 2, \ldots, N)$, then they occurred in the order in which p_i observes them – this is the order $\rightarrow_i$ that we defined above.
- Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

Lamport called the partial ordering obtained by generalizing these two relationships the *happened-before* relation. It is also sometimes known as the relation of *causal ordering* or *potential causal ordering*.

We can define the happened-before relation, denoted by $\rightarrow$, as follows:

HB1: If $\exists$ process $p_i : e \rightarrow_i e'$, then $e \rightarrow e'$.

HB2: For any message m, $send(m) \rightarrow receive(m)$
– where $send(m)$ is the event of sending the message, and $receive(m)$ is the event of receiving it.

HB3: If e, e' and e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

Thus, if e and e' are events, and if $e \rightarrow e'$, then we can find a series of events $e_1, e_2, \ldots, e_n$ occurring at one or more processes such that $e = e_1$ and $e' = e_n$, and for $i = 1, 2, \ldots, N-1$ either HB1 or HB2 applies between e_i and e_{i+1}. That is, either they occur in succession at the same process, or there is a message m such that $e_i = send(m)$ and $e_{i+1} = receive(m)$. The sequence of events $e_1, e_2, \ldots, e_n$ need not be unique.

The relation $\rightarrow$ is illustrated for the case of three processes, p_1, p_2 and p_3, in Figure 14.5. It can be seen that $a \rightarrow b$, since the events occur in this order at process p_1 $(a \rightarrow_i b)$, and similarly $c \rightarrow d$. Furthermore, $b \rightarrow c$, since these events are the sending and

reception of message m_1, and similarly $d \rightarrow f$. Combining these relations, we may also say that, for example, $a \rightarrow f$.

It can also be seen from Figure 14.5 that not all events are related by the relation $\rightarrow$. For example, $a \not\rightarrow e$ and $e \not\rightarrow a$, since they occur at different processes, and there is no chain of messages intervening between them. We say that events such as a and e that are not ordered by $\rightarrow$ are *concurrent* and write this $a \parallel e$.

The relation $\rightarrow$ captures a flow of data intervening between two events. Note, however, that in principle data can flow in ways other than by message passing. For example, if Smith enters a command to his process to send a message, then telephones Jones, who commands her process to issue another message, the issuing of the first message clearly happened-before that of the second. Unfortunately, since no network messages were sent between the issuing processes, we cannot model this type of relationship in our system.

Another point to note is that if the happened-before relation holds between two events, then the first might or might not actually have caused the second. For example, if a server receives a request message and subsequently sends a reply, then clearly the reply transmission is caused by the request transmission. However, the relation $\rightarrow$ captures only potential causality, and two events can be related by $\rightarrow$ even though there is no real connection between them. A process might, for example, receive a message and subsequently issue another message, but one that it issues every five minutes anyway and that bears no specific relation to the first message. No actual causality has been involved, but the relation $\rightarrow$ would order these events.

Logical clocks • Lamport [1978] invented a simple mechanism by which the happened-before ordering can be captured numerically, called a *logical clock*. A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process p_i keeps its own logical clock, L_i, which it uses to apply so-called *Lamport timestamps* to events. We denote the timestamp of event e at p_i by $L_i(e)$, and by $L(e)$ we denote the timestamp of event e at whatever process it occurred at.

To capture the happened-before relation $\rightarrow$, processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

LC1: L_i is incremented before each event is issued at process p_i: $L_i := L_i + 1$.

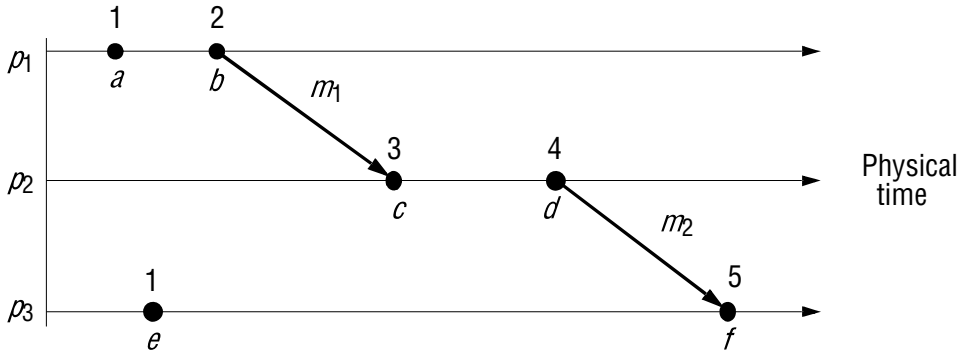
LC2: (a) When a process p_i sends a message m, it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive*(m).

Although we increment clocks by 1, we could have chosen any positive value. It can easily be shown, by induction on the length of any sequence of events relating two events e and e', that $e \rightarrow e' \Rightarrow L(e) < L(e')$.

Note that the converse is not true. If $L(e) < L(e')$, then we cannot infer that $e \rightarrow e'$. In Figure 14.6 we illustrate the use of logical clocks for the example given in Figure 14.5. Each of the processes p_1, p_2 and p_3 has its logical clock initialized to 0. The clock values given are those immediately after the event to which they are adjacent. Note that, for example, $L(b) > L(e)$ but $b \parallel e$.

Figure 14.6 Lamport timestamps for the events shown in Figure 14.5



Totally ordered logical clocks • Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps. However, we can create a total order on the set of events – that is, one for which all pairs of distinct events are ordered – by taking into account the identifiers of the processes at which events occur. If e is an event occurring at p_i with local timestamp T_i, and e' is an event occurring at p_j with local timestamp T_j, we define the global logical timestamps for these events to be (T_i, i) and (T_j, j), respectively. And we define $(T_i, i) < (T_j, j)$ if and only if either $T_i < T_j$, or $T_i = T_j$ and $i < j$. This ordering has no general physical significance (because process identifiers are arbitrary), but it is sometimes useful. Lamport used it, for example, to order the entry of processes to a critical section.

Vector clocks • Mattern [1989] and Fidge [1991] developed vector clocks to overcome the shortcoming of Lamport's clocks: the fact that from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$. A vector clock for a system of N processes is an array of N integers. Each process keeps its own vector clock, V_i, which it uses to timestamp local events. Like Lamport timestamps, processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:

VC1: Initially, $V_i[j] = 0$, for $i, j = 1, 2 \ldots, N$.

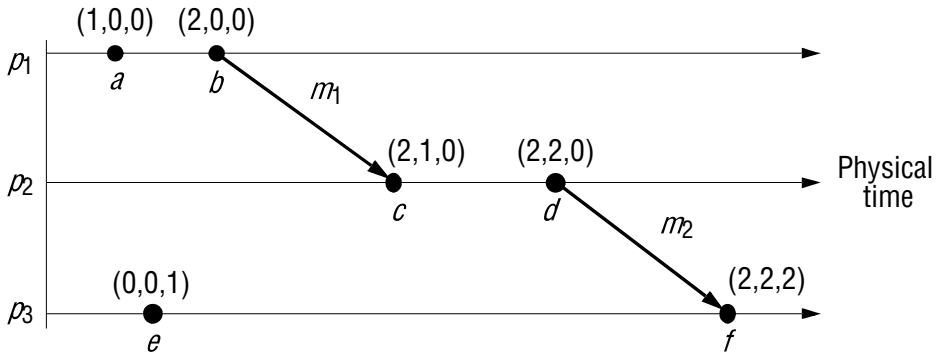
VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.

VC3: p_i includes the value $t = V_i$ in every message it sends.

VC4: When p_i receives a timestamp t in a message, it sets $V_i[j] := max(V_i[j], t[j])$, for $j = 1, 2 \ldots, N$. Taking the component-wise maximum of two vector timestamps in this way is known as a *merge* operation.

For a vector clock V_i, $V_i[i]$ is the number of events that p_i has timestamped, and $V_i[j]$ $(j \neq i)$ is the number of events that have occurred at p_j that have potentially affected p_i. (Process p_j may have timestamped more events by this point, but no information has flowed to p_i about them in messages as yet.)

Figure 14.7 Vector timestamps for the events shown in Figure 14.5



We may compare vector timestamps as follows:

$$V = V' \;\; iff \;\; V[j] = V'[j] \;\; \text{for } j = 1, 2 \ldots, N$$

$$V \leq V' \;\; iff \;\; V[j] \leq V'[j] \;\; \text{for } j = 1, 2 \ldots, N$$

$$V < V' \;\; iff \;\; V \leq V' \wedge V \neq V'$$

Let $V(e)$ be the vector timestamp applied by the process at which e occurs. It is straightforward to show, by induction on the length of any sequence of events relating two events e and e', that $e \rightarrow e' \Rightarrow V(e) < V(e')$. Exercise 10.13 leads the reader to show the converse: if $V(e) < V(e')$, then $e \rightarrow e'$.

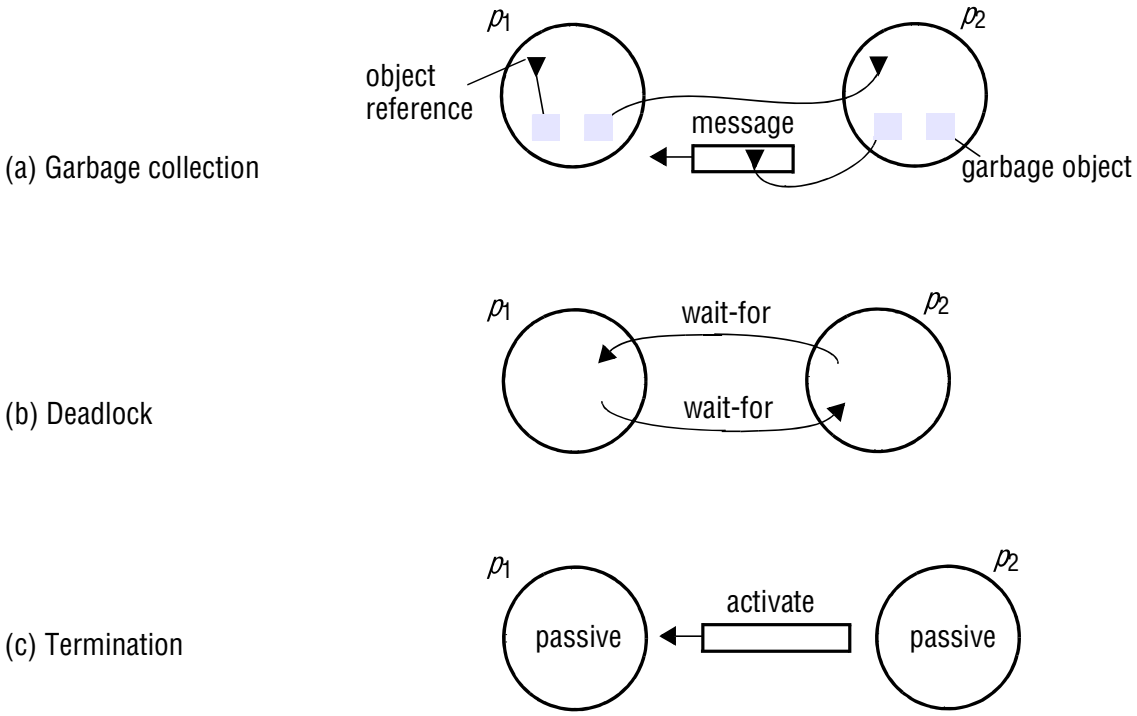
Figure 14.7 shows the vector timestamps of the events of Figure 14.5. It can be seen, for example, that $V(a) < V(f)$, which reflects the fact that $a \rightarrow f$. Similarly, we can tell when two events are concurrent by comparing their timestamps. For example, that $c \parallel e$ can be seen from the facts that neither $V(c) \leq V(e)$ nor $V(e) \leq V(c)$.

Vector timestamps have the disadvantage, compared with Lamport timestamps, of taking up an amount of storage and message payload that is proportional to N, the number of processes. Charron-Bost [1991] showed that, if we are to be able to tell whether or not two events are concurrent by inspecting their timestamps, then the dimension N is unavoidable. However, techniques exist for storing and transmitting smaller amounts of data, at the expense of the processing required to reconstruct complete vectors. Raynal and Singhal [1996] give an account of some of these techniques. They also describe the notion of *matrix clocks*, whereby processes keep estimates of other processes' vector times as well as their own.

14.5 Global states

In this and the next section we examine the problem of finding out whether a particular property is true of a distributed system as it executes. We begin by giving the examples of distributed garbage collection, deadlock detection, termination detection and debugging:

Figure 14.8 Detecting global properties



Distributed garbage collection: An object is considered to be garbage if there are no longer any references to it anywhere in the distributed system. The memory taken up by that object can be reclaimed once it is known to be garbage. To check that an object is garbage, we must verify that there are no references to it anywhere in the system. In Figure 14.8(a), process p_1 has two objects that both have references – one has a reference within p_1 itself, and p_2 has a reference to the other. Process p_2 has one garbage object, with no references to it anywhere in the system. It also has an object for which neither p_1 nor p_2 has a reference, but there is a reference to it in a message that is in transit between the processes. This shows that when we consider properties of a system, we must include the state of communication channels as well as the state of the processes.

Distributed deadlock detection: A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this 'waits-for' relationship. Figure 14.8(b) shows that processes p_1 and p_2 are each waiting for a message from the other, so this system will never make progress.

Distributed termination detection: The problem here is how to detect that a distributed algorithm has terminated. Detecting termination is a problem that sounds deceptively easy to solve: it seems at first only necessary to test whether each process has halted. To see that this is not so, consider a distributed algorithm executed by two processes p_1 and p_2, each of which may request values from the other. Instantaneously, we may find that a process is either active or passive – a passive process is not engaged in any activity of its own but is prepared to respond with a value requested by the other. Suppose we discover that p_1 is passive and that p_2 is

passive (Figure 14.8c). To see that we may not conclude that the algorithm has terminated, consider the following scenario: when we tested p_1 for passivity, a message was on its way from p_2, which became passive immediately after sending it. On receipt of the message, p_1 became active again – after we had found it to be passive. The algorithm had not terminated.

The phenomena of termination and deadlock are similar in some ways, but they are different problems. First, a deadlock may affect only a subset of the processes in a system, whereas all processes must have terminated. Second, process passivity is not the same as waiting in a deadlock cycle: a deadlocked process is attempting to perform a further action, for which another process waits; a passive process is not engaged in any activity.

Distributed debugging: Distributed systems are complex to debug [Bonnaire *et al.* 1995], and care needs to be taken in establishing what occurred during the execution. For example, suppose Smith has written an application in which each process p_i contains a variable x_i $(i = 1, 2 \ldots, N)$. The variables change as the program executes, but they are required always to be within a value δ of one another. Unfortunately, there is a bug in the program, and Smith suspects that under certain circumstances $|x_i - x_j| > \delta$ for some *i* and *j*, breaking her consistency constraints. Her problem is that this relationship must be evaluated for values of the variables that occur at the same time.

Each of the problems above has specific solutions tailored to it; but they all illustrate the need to observe a global state, and so motivate a general approach.

14.5.1 Global states and consistent cuts

It is possible in principle to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes – is much harder to address.

The essential problem is the absence of global time. If all processes had perfectly synchronized clocks, then we could agree on a time at which each process would record its state – the result would be an actual global state of the system. From the collection of process states we could tell, for example, whether the processes were deadlocked. But we cannot achieve perfect clock synchronization, so this method is not available to us.

So we might ask whether we can assemble a meaningful global state from local states recorded at different real times. The answer is a qualified 'yes', but in order to see this we must first introduce some definitions.

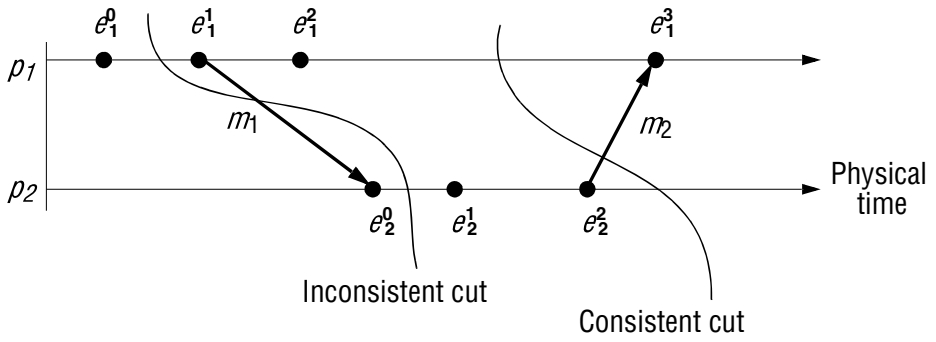
Let us return to our general system $\wp$ of N processes p_i $(i = 1, 2, \ldots, N)$, whose execution we wish to study. We said above that a series of events occurs at each process, and that we may characterize the execution of each process by its history:

$$history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$$

Similarly, we may consider any finite prefix of the process's history:

$$h_i^k = \langle e_i^0, e_i^1, \ldots e_i^k \rangle$$

Figure 14.9 Cuts



Each event either is an internal action of the process (for example, the updating of one of its variables), or is the sending or receipt of a message over the communication channels that connect the processes.

In principle, we can record what occurred in $\wp$'s execution. Each process can record the events that take place there, and the succession of states it passes through. We denote by s_i^k the state of process p_i immediately before the kth event occurs, so that s_i^0 is the initial state of p_i. We noted in the examples above that the state of the communication channels is sometimes relevant. Rather than introducing a new type of state, we make the processes record the sending or receipt of all messages as part of their state. If we find that process p_i has recorded that it sent a message m to process $p_j (i \neq j)$, then by examining whether p_j has received that message we can infer whether or not m is part of the state of the channel between p_i and p_j.

We can also form the *global history* of $\wp$ as the union of the individual process histories:

$$H = h_0 \cup h_1 \cup \ldots \cup h_{N-1}$$

Mathematically, we can take any set of states of the individual processes to form a global state $S = (s_1, s_2, \ldots s_N)$. But which global states are meaningful – that is, which process states could have occurred at the same time? A global state corresponds to initial prefixes of the individual process histories. A *cut* of the system's execution is a subset of its global history that is a union of prefixes of process histories:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_N^{c_N}$$

The state s_i in the global state S corresponding to the cut C is that of p_i immediately after the last event processed by p_i in the cut – $e_i^{c_i}$ $(i = 1, 2, \ldots, N)$. The set of events $\{e_i^{c_i} : i = 1, 2, \ldots, N\}$ is called the *frontier* of the cut.

Consider the events occurring at processes p_1 and p_2 shown in Figure 14.9. The figure shows two cuts, one with frontier $<e_1^0, e_2^0>$ and another with frontier $<e_1^2, e_2^2>$. The leftmost cut is *inconsistent*. This is because at p_2 it includes the receipt of the message m_1, but at p_1 it does not include the sending of that message. This is showing an 'effect' without a 'cause'. The actual execution never was in a global state corresponding to the process states at that frontier, and we can in principle tell this by examining the $\rightarrow$ relation between events. By contrast, the rightmost cut is *consistent*.

It includes both the sending and the receipt of message m_1 and the sending but not the receipt of message m_2. That is consistent with the actual execution – after all, the message took some time to arrive.

A cut C is consistent if, for each event it contains, it also contains all the events that happened-before that event:

For all events $e \in C$, $f \rightarrow e \Rightarrow f \in C$

A *consistent global state* is one that corresponds to a consistent cut. We may characterize the execution of a distributed system as a series of transitions between global states of the system:

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$$

In each transition, precisely one event occurs at some single process in the system. This event is either the sending of a message, the receipt of a message or an internal event. If two events happened simultaneously, we may nonetheless deem them to have occurred in a definite order – say, ordered according to process identifiers. (Events that occur simultaneously must be concurrent: neither happened-before the other.) A system evolves in this way through consistent global states.

A *run* is a total ordering of all the events in a global history that is consistent with each local history's ordering, $\varnothing_i$ $(i = 1, 2, \ldots, N)$. A *linearization* or *consistent run* is an ordering of the events in a global history that is consistent with this happened-before relation $\rightarrow$ on H. Note that a linearization is also a run.

Not all runs pass through consistent global states, but all linearizations pass only through consistent global states. We say that a state S' is *reachable* from a state S if there is a linearization that passes through S and then S'.

Sometimes we may alter the ordering of concurrent events within a linearization, and derive a run that still passes through only consistent global states. For example, if two successive events in a linearization are the receipt of messages by two processes, then we may swap the order of these two events.

14.5.2 Global state predicates, stability, safety and liveness

Detecting a condition such as deadlock or termination amounts to evaluating a *global state predicate*. A global state predicate is a function that maps from the set of global states of processes in the system $\wp$ to {*True*, *False*}. One of the useful characteristics of the predicates associated with the state of an object being garbage, of the system being deadlocked or the system being terminated is that they are all *stable*: once the system enters a state in which the predicate is *True*, it remains *True* in all future states reachable from that state. By contrast, when we monitor or debug an application we are often interested in non-stable predicates, such as that in our example of variables whose difference is supposed to be bounded. Even if the application reaches a state in which the bound obtains, it need not stay in that state.

We also note here two further notions relevant to global state predicates: safety and liveness. Suppose there is an undesirable property α that is a predicate of the system's global state – for example, α could be the property of being deadlocked. Let

S_0 be the original state of the system. *Safety* with respect to α is the assertion that α evaluates to *False* for all states S reachable from S_0. Conversely, let β be a desirable property of a system's global state – for example, the property of reaching termination. *Liveness* with respect to β is the property that, for any linearization L starting in the state S_0, β evaluates to *True* for some state S_L reachable from S_0.

14.5.3 The 'snapshot' algorithm of Chandy and Lamport

Chandy and Lamport [1985] describe a 'snapshot' algorithm for determining global states of distributed systems, which we now present. The goal of the algorithm is to record a set of process and channel states (a 'snapshot') for a set of processes p_i ($i = 1, 2, \ldots, N$) such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.

We shall see that the state that the snapshot algorithm records has convenient properties for evaluating stable global predicates.

The algorithm records state locally at processes; it does not give a method for gathering the global state at one site. An obvious method for gathering the state is for all processes to send the state they recorded to a designated collector process, but we shall not address this issue further here.

The algorithm assumes that:

- Neither channels nor processes fail – communication is reliable so that every message sent is eventually received intact, exactly once.
- Channels are unidirectional and provide FIFO-ordered message delivery.
- The graph of processes and channels is strongly connected (there is a path between any two processes).
- Any process may initiate a global snapshot at any time.
- The processes may continue their execution and send and receive normal messages while the snapshot takes place.

For each process p_i, let the *incoming channels* be those at p_i over which other processes send it messages; similarly, the *outgoing channels* of p_i are those on which it sends messages to other processes. The essential idea of the algorithm is as follows. Each process records its state and also, for each incoming channel, a set of messages sent to it. The process records, for each channel, any messages that arrived after it recorded its state and before the sender recorded its own state. This arrangement allows us to record the states of processes at different times but to account for the differentials between process states in terms of messages transmitted but not yet received. If process p_i has sent a message m to process p_j, but p_j has not received it, then we account for m as belonging to the state of the channel between them.

The algorithm proceeds through use of special *marker* messages, which are distinct from any other messages the processes send and which the processes may send and receive while they proceed with their normal execution. The marker has a dual role: as a prompt for the receiver to save its own state, if it has not already done so; and as a means of determining which messages to include in the channel state.

Figure 14.10 Chandy and Lamport's 'snapshot' algorithm

Marker receiving rule for process p_i

On receipt of a *marker* message at p_i over channel c:

if (p_i has not yet recorded its state) it

records its process state now;

records the state of c as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of c as the set of messages it has received over c since it saved its state.

end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel c:

p_i sends one marker message over c

(before it sends any other message over c).

The algorithm is defined through two rules, the *marker receiving rule* and the *marker sending rule* (Figure 14.10). The marker sending rule obligates processes to send a marker after they have recorded their state, but before they send any other messages.

The marker receiving rule obligates a process that has not recorded its state to do so. In that case, this is the first marker that it has received. It notes which messages subsequently arrive on the other incoming channels. When a process that has already saved its state receives a marker (on another channel), it records the state of that channel as the set of messages it has received on it since it saved its state.

Any process may begin the algorithm at any time. It acts as though it has received a marker (over a nonexistent channel) and follows the marker receiving rule. Thus it records its state and begins to record messages arriving over all its incoming channels. Several processes may initiate recording concurrently in this way (as long as the markers they use can be distinguished).

We illustrate the algorithm for a system of two processes, p_1 and p_2, connected by two unidirectional channels, c_1 and c_2. The two processes trade in 'widgets'. Process p_1 sends orders for widgets over c_2 to p_2, enclosing payment at the rate of $10 per widget. Some time later, process p_2 sends widgets along channel c_1 to p_1. The

Figure 14.11 Two processes and their initial states

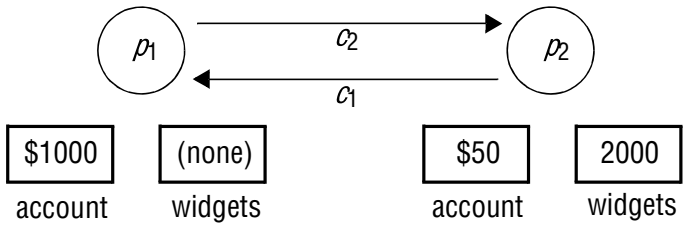
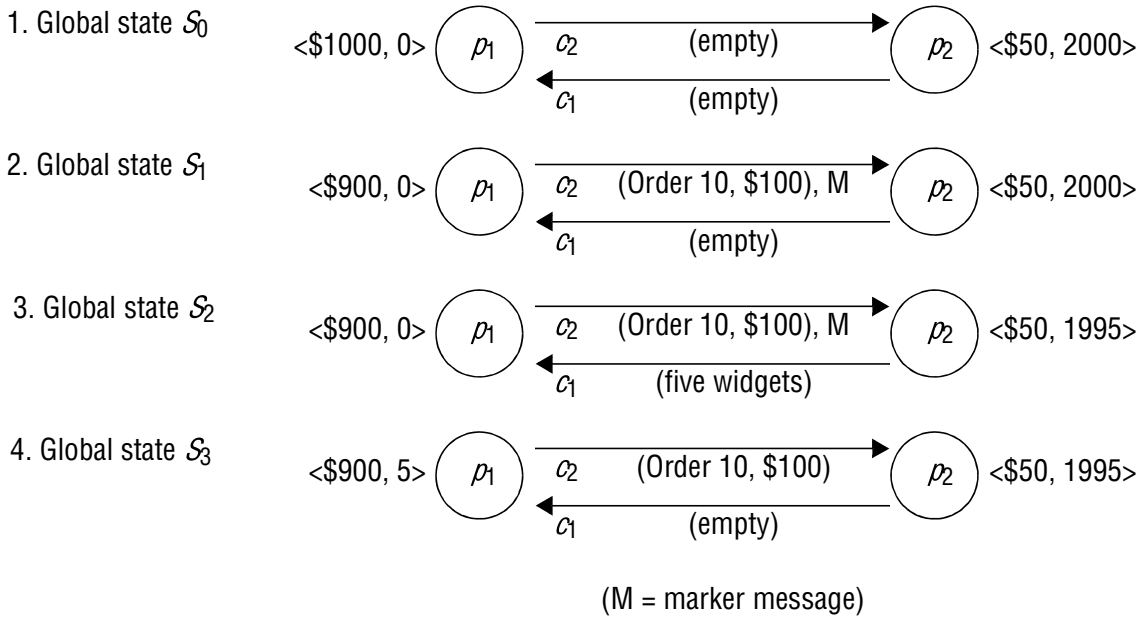


Figure 14.12 The execution of the processes in Figure 14.11



processes have the initial states shown in Figure 14.11. Process p_2 has already received an order for five widgets, which it will shortly dispatch to p_1.

Figure 14.12 shows an execution of the system while the state is recorded. Process p_1 records its state in the actual global state S_0, when the state of p_1 is <$1000, 0>. Following the marker sending rule, process p_1 then emits a marker message over its outgoing channel c_2 before it sends the next application-level message: (Order 10, $100), over channel c_2. The system enters actual global state S_1.

Before p_2 receives the marker, it emits an application message (five widgets) over c_1 in response to p_1's previous order, yielding a new actual global state S_2.

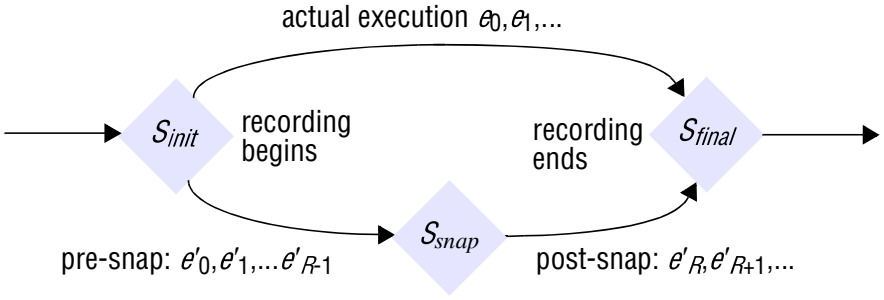
Now process p_1 receives p_2's message (five widgets), and p_2 receives the marker. Following the marker receiving rule, p_2 records its state as <$50, 1995> and that of channel c_2 as the empty sequence. Following the marker sending rule, it sends a marker message over c_1.

When process p_1 receives p_2's marker message, it records the state of channel c_1 as the single message (five widgets) that it received after it first recorded its state. The final actual global state is S_3.

The final recorded state is p_1: <$1000, 0>; p_2: <$50, 1995>; c_1: <(five widgets)>; c_2: < >. Note that this state differs from all the global states through which the system actually passed.

Termination of the snapshot algorithm • We assume that a process that has received a marker message records its state within a finite time and sends marker messages over each outgoing channel within a finite time (even when it no longer needs to send application messages over these channels). If there is a path of communication channels and processes from a process p_i to a process p_j $(j \neq i)$, then it is clear on these assumptions that p_j will record its state a finite time after p_i recorded its state. Since we are assuming the graph of processes and channels to be strongly connected, it follows

Figure 14.13 Reachability between states in the snapshot algorithm



that all processes will have recorded their states and the states of incoming channels a finite time after some process initially records its state.

Characterizing the observed state • The snapshot algorithm selects a cut from the history of the execution. The cut, and therefore the state recorded by this algorithm, is consistent. To see this, let e_i and e_j be events occurring at p_i and p_j, respectively, such that $e_i \rightarrow e_j$. We assert that if e_j is in the cut, then e_i is in the cut. That is, if e_j occurred before p_j recorded its state, then e_i must have occurred before p_i recorded its state. This is obvious if the two processes are the same, so we shall assume that $j \neq i$. Assume, for the moment, the opposite of what we wish to prove: that p_i recorded its state before e_i occurred. Consider the sequence of H messages $m_1, m_2 \ldots, m_H$ $(H \geq 1)$, giving rise to the relation $e_i \rightarrow e_j$. By FIFO ordering over the channels that these messages traverse, and by the marker sending and receiving rules, a marker message would have reached p_j ahead of each of $m_1, m_2 \ldots, m_H$. By the marker receiving rule, p_j would therefore have recorded its state before the event e_j. This contradicts our assumption that e_j is in the cut, and we are done.

We may further establish a reachability relation between the observed global state and the initial and final global states when the algorithm runs. Let $Sys = e_0, e_1, \ldots$ be the linearization of the system as it executed (where two events occurred at exactly the same time, we order them according to process identifiers). Let S_{init} be the global state immediately before the first process recorded its state; let S_{final} be the global state when the snapshot algorithm terminates, immediately after the last state-recording action; and let S_{snap} be the recorded global state.

We shall find a permutation of Sys, $Sys' = e'_0, e'_1, e'_2, \ldots$ such that all three states S_{init}, S_{snap} and S_{final} occur in Sys', S_{snap} is reachable from S_{init} in Sys', and S_{final} is reachable from S_{snap} in Sys'. Figure 14.13 shows this situation, in which the upper linearization is Sys and the lower linearization is Sys'.

We derive Sys' from Sys by first categorizing all events in Sys as *pre-snap* events or *post-snap* events. A pre-snap event at process p_i is one that occurred at p_i before it recorded its state; all other events are post-snap events. It is important to understand that a post-snap event may occur before a pre-snap event in Sys, if the events occur at different processes. (Of course, no post-snap event may occur before a pre-snap event at the same process.)

We shall show how we may order all pre-snap events before post-snap events to obtain Sys'. Suppose that e_j is a post-snap event at one process, and e_{j+1} is a pre-snap

event at a different process. It cannot be that $e_j \rightarrow e_{j+1}$ for then these two events would be the sending and receiving of a message, respectively. A marker message would have to have preceded the message, making the reception of the message a post-snap event, but by assumption e_{j+1} is a pre-snap event. We may therefore swap the two events without violating the happened-before relation (that is, the resultant sequence of events remains a linearization). The swap does not introduce new process states, since we do not alter the order in which events occur at any individual process.

We continue swapping pairs of adjacent events in this way as necessary until we have ordered all pre-snap events $e'_0, e'_1, e'_2, \ldots, e'_{R-1}$ prior to all post-snap events $e'_R, e'_{R+1}, e'_{R+2}, \ldots$ with Sys' the resulting execution. For each process, the set of events in $e'_0, e'_1, e'_2, \ldots, e'_{R-1}$ that occurred at it is exactly the set of events that it experienced before it recorded its state. Therefore the state of each process at that point, and the state of the communication channels, is that of the global state S_{snap} recorded by the algorithm. We have disturbed neither of the states S_{init} or S_{final} with which the linearization begins and ends. So we have established the reachability relationship.

Stability and the reachability of the observed state • The reachability property of the snapshot algorithm is useful for detecting stable predicates. In general, any non-stable predicate we establish as being *True* in the state S_{snap} may or may not have been *True* in the actual execution whose global state we recorded. However, if a stable predicate is *True* in the state S_{snap} then we may conclude that the predicate is *True* in the state S_{final}, since by definition a stable predicate that is *True* of a state S is also *True* of any state reachable from S. Similarly, if the predicate evaluates to *False* for S_{snap}, then it must also be *False* for S_{init}.

14.6 Distributed debugging

We now examine the problem of recording a system's global state so that we may make useful statements about whether a transitory state – as opposed to a stable state – occurred in an actual execution. This is what we require, in general, when debugging a distributed system. We gave an example above in which each of a set of processes p_i has a variable x_i. The safety condition required in this example is $|x_i - x_j| \leq \delta$ $(i, j = 1, 2, \ldots, N)$; this constraint is to be met even though a process may change the value of its variable at any time. Another example is a distributed system controlling a system of pipes in a factory where we are interested in whether all the valves (controlled by different processes) were open at some time. In these examples, we cannot in general observe the values of the variables or the states of the valves simultaneously. The challenge is to monitor the system's execution over time – to capture 'trace' information rather than a single snapshot – so that we can establish *post hoc* whether the required safety condition was or may have been violated.

Chandy and Lamport's [1985] snapshot algorithm collects state in a distributed fashion, and we pointed out how the processes in the system could send the state they gather to a monitor process for collection. The algorithm we describe next (due to Marzullo and Neiger [1991]) is centralized. The observed processes send their states to a process called a *monitor*, which assembles globally consistent states from what it receives. We consider the monitor to lie outside the system, observing its execution.

Our aim is to determine cases where a given global state predicate ϕ was definitely *True* at some point in the execution we observed, and cases where it was possibly *True*. The notion 'possibly' arises as a natural concept because we may extract a consistent global state S from an executing system and find that $\phi(S)$ is *True*. No single observation of a consistent global state allows us to conclude whether a non-stable predicate ever evaluated to *True* in the actual execution. Nevertheless, we may be interested to know whether they *might* have occurred, as far as we can tell by observing the execution.

The notion 'definitely' does apply to the actual execution and not to a run that we have extrapolated from it. It may sound paradoxical for us to consider what happened in an actual execution. However, it is possible to evaluate whether ϕ was definitely *True* by considering all linearizations of the observed events.

We now define the notions of *possibly* ϕ and *definitely* ϕ for a predicate ϕ in terms of linearizations of H, the history of the system's execution:

> *possibly* ϕ: The statement *possibly* ϕ means that there is a consistent global state S through which a linearization of H passes such that $\phi(S)$ is *True*.
>
> *definitely* ϕ:The statement *definitely* ϕ means that for all linearizations L of H, there is a consistent global state S through which L passes such that $\phi(S)$ is *True*.

When we use Chandy and Lamport's snapshot algorithm and obtain the global state S_{snap}, we may assert *possibly* ϕ if $\phi(S_{snap})$ happens to be *True*. But in general evaluating *possibly* ϕ entails a search through all consistent global states derived from the observed execution. Only if $\phi(S)$ evaluates to *False* for all consistent global states S is it not the case that *possibly* ϕ. Note also that while we may conclude *definitely* $(\neg\phi)$ from $\neg$*possibly* ϕ, we may not conclude $\neg$*possibly* ϕ from *definitely* $(\neg\phi)$. The latter is the assertion that $\neg\phi$ holds at some state on every linearization: ϕ may hold for other states.

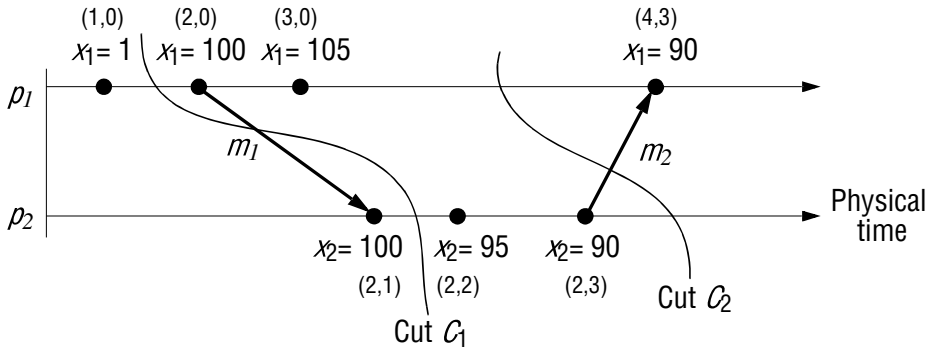
We now describe: how the process states are collected; how the monitor extracts consistent global states; and how the monitor evaluates *possibly* ϕ and *definitely* ϕ in both asynchronous and synchronous systems.

14.6.1 Collecting the state

The observed processes p_i $(i = 1, 2, \ldots, N)$ send their initial state to the monitor initially, and thereafter from time to time, in *state messages*. The monitor records the state messages from each process p_i in a separate queue Q_i, for each $i = 1, 2, \ldots, N$.

The activity of preparing and sending state messages may delay the normal execution of the observed processes, but it does not otherwise interfere with it. There is no need to send the state except initially and when it changes. There are two optimizations to reduce the state-message traffic to the monitor. First, the global state predicate may depend only on certain parts of the processes' states – for example, only on the states of particular variables – so the observed processes need only send the relevant state to the monitor. Second, they need only send their state at times when the predicate ϕ may become *True* or cease to be *True*. There is no point in sending changes to the state that do not affect the predicate's value.

Figure 14.14 Vector timestamps and variable values for the execution of Figure 14.9



For example, in the example system of processes p_i that are supposed to obey the constraint $|x_i - x_j| \leq \delta$. $(i, j = 1, 2, \ldots, N)$, each process need only notify the monitor when the values of its own variable x_i changes. When they send their state, they supply the value of x_i but do not need to send any other variables.

14.6.2 Observing consistent global states

The monitor must assemble consistent global states against which it evaluates ϕ. Recall that a cut C is consistent if and only if for all events e in the cut C, $f \rightarrow e \Rightarrow f \in C$.

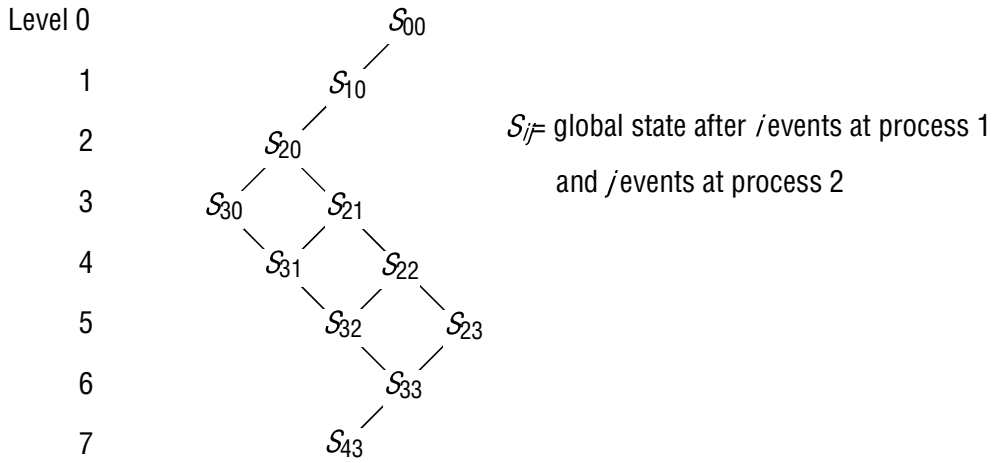
For example, Figure 14.14 shows two processes p_1 and p_2 with variables x_1 and x_2, respectively. The events shown on the timelines (with vector timestamps) are adjustments to the values of the two variables. Initially, $x_1 = x_2 = 0$. The requirement is $|x_1 - x_2| \leq 50$. The processes make adjustments to their variables, but 'large' adjustments cause a message containing the new value to be sent to the other process. When either of the processes receives an adjustment message from the other, it sets its variable equal to the value contained in the message.

Whenever one of the processes p_1 or p_2 adjusts the value of its variable (whether it is a 'small' adjustment or a 'large' one), it sends the value in a state message to the monitor. The latter keeps the state messages in the per-process queues for analysis. If the monitor were to use values from the inconsistent cut C_1 in Figure 14.14, then it would find that $x_1 = 1, x_2 = 100$, breaking the constraint $|x_1 - x_2| \leq 50$. But this state of affairs never occurred. On the other hand, values from the consistent cut C_2 show $x_1 = 105, x_2 = 90$.

In order that the monitor can distinguish consistent global states from inconsistent global states, the observed processes enclose their vector clock values with their state messages. Each queue Q_i is kept in sending order, which can immediately be established by examining the ith component of the vector timestamps. Of course, the monitor may deduce nothing about the ordering of states sent by different processes from their arrival order, because of variable message latencies. It must instead examine the vector timestamps of the state messages.

Let $S = (s_1, s_2, \ldots, s_N)$ be a global state drawn from the state messages that the monitor has received. Let $V(s_i)$ be the vector timestamp of the state s_i received from p_i. Then it can be shown that S is a consistent global state if and only if:

Figure 14.15 The lattice of global states for the execution of Figure 14.14



$$V(s_i)[i] \geq V(s_j)[i] \text{ for } i, j = 1, 2, \ldots, N \text{ – (Condition } CGS)$$

This says that the number of p_i's events known at p_j when it sent s_j is no more than the number of events that had occurred at p_i when it sent s_i. In other words, if one process's state depends upon another (according to happened-before ordering), then the global state also encompasses the state upon which it depends.

In summary, we now possess a method whereby the monitor may establish whether a given global state is consistent, using the vector timestamps kept by the observed processes and piggybacked on the state messages that they send to it.

Figure 14.15 shows the lattice of consistent global states corresponding to the execution of the two processes in Figure 14.14. This structure captures the relation of reachability between consistent global states. The nodes denote global states, and the edges denote possible transitions between these states. The global state S_{00} has both processes in their initial state; S_{10} has p_2 still in its initial state and p_1 in the next state in its local history. The state S_{01} is not consistent, because of the message m_1 sent from p_1 to p_2, so it does not appear in the lattice.

The lattice is arranged in levels with, for example, S_{00} in level 0 and S_{10} in level 1. In general, S_{ij} is in level $(i + j)$. A linearization traverses the lattice from any global state to any global state reachable from it on the next level – that is, in each step some process experiences one event. For example, S_{22} is reachable from S_{20}, but S_{22} is not reachable from S_{30}.

The lattice shows us all the linearizations corresponding to a history. It is now clear in principle how a monitor should evaluate *possibly* ϕ and *definitely* ϕ. To evaluate *possibly* ϕ, the monitor starts at the initial state and steps through all consistent states reachable from that point, evaluating ϕ at each stage. It stops when ϕ evaluates to *True*. To evaluate *definitely* ϕ, the monitor must attempt to find a set of states through which all linearizations must pass, and at each of which ϕ evaluates to *True*. For example, if $\phi(S_{30})$ and $\phi(S_{21})$ in Figure 14.15 are both *True* then, since all linearizations pass through these states, *definitely* ϕ holds.

Figure 14.16 Algorithms to evaluate *possibly* ϕ and *definitely* ϕ

1. *Evaluating possibly* ϕ *for global history* H *of* N *processes*

```
L := 0;
States := { (s_1^0, s_2^0, ..., s_N^0) };
while (φ(S) = False for all S ∈ States)
    L := L + 1;
    Reachable := {S' : S' reachable in H from some S ∈ States ∧ level(S') = L};
    States := Reachable
end while
output "possibly φ";
```

2. *Evaluating definitely* ϕ *for global history* H *of* N *processes*

```
L := 0;
if (φ(s_1^0, s_2^0, ..., s_N^0)) then States := {} else States := { (s_1^0, s_2^0, ..., s_N^0) };
while (States ≠ {})
    L := L + 1;
    Reachable := {S' : S' reachable in H from some S ∈ States ∧ level(S') = L};
    States := {S ∈ Reachable : φ(S) = False}
end while
output "definitely φ";
```

14.6.3 Evaluating possibly ϕ

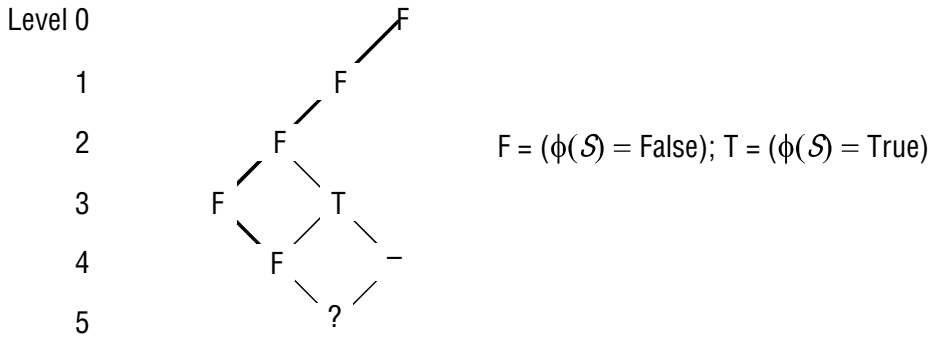
To evaluate *possibly* ϕ, the monitor must traverse the lattice of reachable states, starting from the initial state $(s_1^0, s_2^0 \ldots s_N^0)$. The algorithm is shown in Figure 14.16. The algorithm assumes that the execution is infinite. It may easily be adapted for a finite execution.

The monitor may discover the set of consistent states in level $L+1$ reachable from a given consistent state in level L by the following method. Let $S = (s_1, s_2, \ldots, s_N)$ be a consistent state. Then a consistent state in the next level reachable from S is of the form $S' = (s_1, s_2, \ldots s_i', \ldots, s_N)$, which differs from S only by containing the next state (after a single event) of some process p_i. The monitor can find all such states by traversing the queues of state messages Q_i $(i = 1, 2, \ldots, N)$. The state S' is reachable from S if and only if:

$$\text{for } j = 1, 2, \ldots, N, j \neq i: \; V(s_j)[j] \geq V(s_i')[j]$$

This condition comes from condition CGS above and from the fact that S was already a consistent global state. A given state may in general be reached from several states at the previous level, so the monitor should take care to evaluate the consistency of each state only once.

Figure 14.17 Evaluating *definitely* ϕ



14.6.4 Evaluating definitely ϕ

To evaluate *definitely* ϕ, the monitor again traverses the lattice of reachable states a level at a time, starting from the initial state $(s_1^0, s_2^0, \ldots, s_N^0)$. The algorithm (shown in Figure 14.16) again assumes that the execution is infinite but may easily be adapted for a finite execution. It maintains the set *States*, which contains those states at the current level that may be reached on a linearization from the initial state by traversing only states for which ϕ evaluates to *False*. As long as such a linearization exists, we may not assert *definitely* ϕ: the execution could have taken this linearization, and ϕ would be *False* at every stage along it. If we reach a level for which no such linearization exists, we may conclude *definitely* ϕ.

In Figure 14.17, at level 3 the set *States* consists of only one state, which is reachable by a linearization on which all states are *False* (marked in bold lines). The only state considered at level 4 is the one marked 'F'. (The state to its right is not considered, since it can only be reached via a state for which ϕ evaluates to *True*.) If ϕ evaluates to *True* in the state at level 5, then we may conclude *definitely* ϕ. Otherwise, the algorithm must continue beyond this level.

Cost • The algorithms we have just described are combinatorially explosive. Suppose that k is the maximum number of events at a single process. Then the algorithms we have described entail $O(k^N)$ comparisons (the monitor compares the states of each of the N observed processes with one another).

There is also a space cost to these algorithms of $O(kN)$. However, we observe that the monitor may delete a message containing state s_i from queue Q_i when no other item of state arriving from another process could possibly be involved in a consistent global state containing s_i. That is, when:

$$V(s_j^{last})[i] > V(s_i)[i] \text{ for } j = 1, 2, \ldots, N, j \neq i$$

where s_j^{last} is the last state that the monitor has received from process p_j.

14.6.5 Evaluating possibly ϕ and definitely ϕ in synchronous systems

The algorithms we have given so far work in an asynchronous system: we have made no timing assumptions. But the price paid for this is that the monitor may examine a consistent global state $S = (s_1, s_2, \ldots, s_N)$ for which any two local states s_i and s_j occurred an arbitrarily long time apart in the actual execution of the system. Our requirement, by contrast, is to consider only those global states that the actual execution could in principle have traversed.

In a synchronous system, suppose that the processes keep their physical clocks internally synchronized within a known bound, and that the observed processes provide physical timestamps as well as vector timestamps in their state messages. Then the monitor need consider only those consistent global states whose local states could possibly have existed simultaneously, given the approximate synchronization of the clocks. With good enough clock synchronization, these will number many less than all globally consistent states.

We now give an algorithm to exploit synchronized clocks in this way. We assume that each observed process p_i $(i = 1, 2, \ldots, N)$ and the monitor, which we shall call p_0, keep a physical clock C_i $(i = 0, 1, \ldots, N)$. These are synchronized to within a known bound $D > 0$; that is, at the same real time:

$$|C_i(t) - C_j(t)| < D \text{ for } i, j = 0, 1, \ldots, N$$

The observed processes send both their vector time and physical time with their state messages to the monitor. The monitor now applies a condition that not only tests for consistency of a global state $S = (s_1, s_2, \ldots, s_N)$, but also tests whether each pair of states could have happened at the same real time, given the physical clock values. In other words, for $i, j = 1, 2, \ldots, N$:

$V(s_i)[i] \geq V(s_j)[i]$ and s_i and s_j could have occurred at the same real time.

The first clause is the condition that we used earlier. For the second clause, note that p_i is in the state s_i from the time it first notifies the monitor, $C_i(s_i)$, to some later local time $L_i(s_i)$ – say, when the next state transition occurs at p_i. For s_i and s_j to have obtained at the same real time we thus have, allowing for the bound on clock synchronization:

$C_i(s_i) - D \leq C_j(s_j) \leq L_i(s_i) + D$ – or vice versa (swapping *i and j*).

The monitor must calculate a value for $L_i(s_i)$, which is measured against p_i's clock. If the monitor has received a state message for p_i's next state s_i', then $L_i(s_i)$ is $C_i(s_i')$. Otherwise, the monitor estimates $L_i(s_i)$ as $C_0 - max + D$, where C_0 is the monitor's current local clock value and *max* is the maximum transmission time for a state message.

14.7 Summary

This chapter began by describing the importance of accurate timekeeping for distributed systems. It then described algorithms for synchronizing clocks despite the drift between them and the variability of message delays between computers.

The degree of synchronization accuracy that is practically obtainable fulfils many requirements but is nonetheless not sufficient to determine the ordering of an arbitrary pair of events occurring at different computers. The happened-before relation is a partial order on events that reflects a flow of information between them – within a process, or via messages between processes. Some algorithms require events to be ordered in happened-before order, for example, successive updates made to separate copies of data. Lamport clocks are counters that are updated in accordance with the happened-before relationship between events. Vector clocks are an improvement on Lamport clocks, in that it is possible to determine by examining their vector timestamps whether two events are ordered by happened-before or are concurrent.

We introduced the concepts of events, local and global histories, cuts, local and global states, runs, consistent states, linearizations (consistent runs) and reachability. A consistent state or run is one that is in accord with the happened-before relation.

We went on to consider the problem of recording a consistent global state by observing a system's execution. Our objective was to evaluate a predicate on this state. An important class of predicates are the stable predicates. We described the snapshot algorithm of Chandy and Lamport, which captures a consistent global state and allows us to make assertions about whether a stable predicate holds in the actual execution. We went on to give Marzullo and Neiger's algorithm for deriving assertions about whether a predicate held or may have held in the actual run. This algorithm employs a monitor process to collect states. The monitor examines vector timestamps to extract consistent global states, and it constructs and examines the lattice of all consistent global states. This algorithm involves great computational complexity but is valuable for understanding and can be of some practical benefit in real systems where relatively few events change the global predicate's value. The algorithm has a more efficient variant in synchronous systems, where clocks may be synchronized.

EXERCISES

14.1 Why is computer clock synchronization necessary? Describe the design requirements for a system to synchronize the clocks in a distributed system. *page 596*

14.2 A clock is reading 10:27:54.0 (hr:min:sec) when it is discovered to be 4 seconds fast. Explain why it is undesirable to set it back to the right time at that point and show (numerically) how it should be adjusted so as to be correct after 8 seconds have elapsed. *page 600*

14.3 A scheme for implementing *at-most-once* reliable message delivery uses synchronized clocks to reject duplicate messages. Processes place their local clock value (a 'timestamp') in the messages they send. Each receiver keeps a table giving, for each sending process, the largest message timestamp it has seen. Assume that clocks are synchronized to within 100 ms, and that messages can arrive at most 50 ms after transmission.

i) When may a process ignore a message bearing a timestamp T, if it has recorded the last message received from that process as having timestamp T' ?

ii) When may a receiver remove a timestamp 175,000 (ms) from its table? (Hint: use the receiver's local clock value.)

iii) Should the clocks be internally synchronized or externally synchronized? *page 601*

14.4 A client attempts to synchronize with a time server. It records the round-trip times and timestamps returned by the server in the table below.

Which of these times should it use to set its clock? To what time should it set it? Estimate the accuracy of the setting with respect to the server's clock. If it is known that the time between sending and receiving a message in the system concerned is at least 8 ms, do your answers change?

Round-trip (ms)	*Time (hr:min:sec)*
22	*10:54:23.674*
25	*10:54:25.450*
20	*10:54:28.342*

page 601

14.5 In the system of Exercise 14.4 it is required to synchronize a file server's clock to within ±1 millisecond. Discuss this in relation to Cristian's algorithm. *page 601*

14.6 What reconfigurations would you expect to occur in the NTP synchronization subnet? *page 604*

14.7 An NTP server B receives server A's message at 16:34:23.480, bearing a timestamp of 16:34:13.430, and replies to it. A receives the message at 16:34:15.725, bearing B's timestamp, 16:34:25.7. Estimate the offset between B and A and the accuracy of the estimate. *page 605*

14.8 Discuss the factors to be taken into account when deciding to which NTP server a client should synchronize its clock. *page 606*

14.9 Discuss how it is possible to compensate for clock drift between synchronization points by observing the drift rate over time. Discuss any limitations to your method. *page 607*

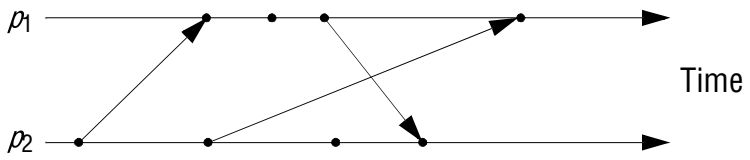
14.10 By considering a chain of zero or more messages connecting events e and e' and using induction, show that $e \rightarrow e' \Rightarrow L(e) < L(e')$. *page 608*

14.11 Show that $V_j[i] \leq V_i[i]$. *page 609*

14.12 In a similar fashion to Exercise 14.10, show that $e \rightarrow e' \Rightarrow V(e) < V(e')$. *page 610*

14.13 Using the result of Exercise 14.11, show that if events e and e' are concurrent then neither $V(e) \leq V(e')$ nor $V(e') \leq V(e)$. Hence show that if $V(e) < V(e')$ then $e \rightarrow e'$. *page 610*

14.14 Two processes P and Q are connected in a ring using two channels, and they constantly rotate a message m. At any one time, there is only one copy of m in the system. Each process's state consists of the number of times it has received m, and P sends m first. At a certain point, P has the message and its state is 101. Immediately after sending m, P initiates the snapshot algorithm. Explain the operation of the algorithm in this case, giving the possible global state(s) reported by it. *page 615*



14.15 The figure above shows events occurring for each of two processes, p_1 and p_2. Arrows between processes denote message transmission.

Draw and label the lattice of consistent states (p_1 state, p_2 state), beginning with the initial state (0,0).

page 622

14.16 Jones is running a collection of processes $p_1, p_2, \ldots, p_N$. Each process p_i contains a variable v_i. She wishes to determine whether all the variables $v_1, v_2, \ldots, v_N$ were ever equal in the course of the execution.

i) Jones' processes run in a synchronous system. She uses a monitor process to determine whether the variables were ever equal. When should the application processes communicate with the monitor process, and what should their messages contain?

ii) Explain the statement *possibly* ($v_1 = v_2 = \ldots = v_N$). How can Jones determine whether this statement is true of her execution?

page 623

15

COORDINATION AND AGREEMENT

In this chapter, we introduce some topics and algorithms related to the issue of how processes coordinate their actions and agree on shared values in distributed systems, despite failures. The chapter begins with algorithms to achieve mutual exclusion among a collection of processes, so as to coordinate their accesses to shared resources. It goes on to examine how an election can be implemented in a distributed system – that is, how a group of processes can agree on a new coordinator of their activities after the previous coordinator has failed.

The second half of the chapter examines the related problems of group communication, consensus, Byzantine agreement and interactive consistency. In the context of group communication, the issue is how to agree on such matters as the order in which messages are to be delivered. Consensus and the other problems generalize from this: how can any collection of processes agree on some value, no matter what the domain of the values in question? We encounter a fundamental result in the theory of distributed systems: that under certain conditions – including surprisingly benign failure conditions – it is impossible to guarantee that processes will reach consensus.

15.1 Introduction

This chapter introduces a collection of algorithms whose goals vary but that share an aim that is fundamental in distributed systems: for a set of processes to coordinate their actions or to agree on one or more values. For example, in the case of a complex piece of machinery such as a spaceship, it is essential that the computers controlling it agree on such conditions as whether the spaceship's mission is proceeding or has been aborted. Furthermore, the computers must coordinate their actions correctly with respect to shared resources (the spaceship's sensors and actuators). The computers must be able to do so even where there is no fixed master-slave relationship between the components (which would make coordination particularly simple). The reason for avoiding fixed master-slave relationships is that we often require our systems to keep working correctly even if failures occur, so we need to avoid single points of failure, such as fixed masters.

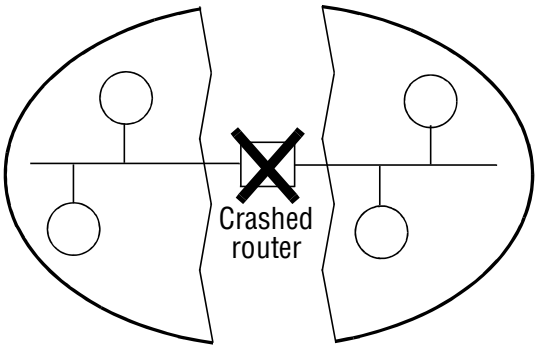
An important distinction for us, as in Chapter 14, will be whether the distributed system under study is asynchronous or synchronous. In an asynchronous system we can make no timing assumptions. In a synchronous system, we shall assume that there are bounds on the maximum message transmission delay, on the time taken to execute each step of a process, and on clock drift rates. The synchronous assumptions allow us to use timeouts to detect process crashes.

Another important aim of the chapter is to consider failures, and how to deal with them when designing algorithms. Section 2.4.2 introduced a failure model, which we shall use in this chapter. Coping with failures is a subtle business, so we begin by considering some algorithms that tolerate no failures and progress through benign failures before exploring how to tolerate arbitrary failures. Along the way, we encounter a fundamental result in the theory of distributed systems: even under surprisingly benign failure conditions, it is impossible to guarantee in an asynchronous system that a collection of processes can agree on a shared value – for example, for all of a spaceship's controlling processes to agree 'mission proceed' or 'mission abort'.

Section 15.2 examines the problem of distributed mutual exclusion. This is the extension to distributed systems of the familiar problem of avoiding race conditions in kernels and multi-threaded applications. Since much of what occurs in distributed systems is resource sharing, this is an important problem to solve. Next, Section 15.3 introduces the related but more general issue of how to 'elect' one of a collection of processes to perform a special role. For example, in Chapter 14 we saw how processes synchronize their clocks to a designated time server. If this server fails and several surviving servers can fulfil that role, then for the sake of consistency it is necessary to choose just one server to take over.

Coordination and agreement related to group communication is the subject of Section 15.4. As Section 4.4.1 explained, the ability to multicast a message to a group is a very useful communication paradigm, with applications from locating resources to coordinating the updates to replicated data. Section 15.4 examines multicast reliability and ordering semantics, and gives algorithms to achieve the variations. Multicast delivery is essentially a problem of agreement between processes: the recipients agree on which messages they will receive, and in which order they will receive them. Section 15.5 discusses the problem of agreement more generally, primarily in the forms known as consensus and Byzantine agreement.

Figure 15.1 A network partition



The treatment followed in this chapter involves stating the assumptions and the goals to be met, and giving an informal account of why the algorithms presented are correct. There is insufficient space to provide a more rigorous approach. For that, we refer the reader to a text that gives a thorough account of distributed algorithms, such as Attiya and Welch [1998] and Lynch [1996].

Before presenting the problems and algorithms, we discuss failure assumptions and the practical matter of detecting failures in distributed systems.

15.1.1 Failure assumptions and failure detectors

For the sake of simplicity, this chapter assumes that each pair of processes is connected by reliable channels. That is, although the underlying network components may suffer failures, the processes use a reliable communication protocol that masks these failures – for example, by retransmitting missing or corrupted messages. Also for the sake of simplicity, we assume that no process failure implies a threat to the other processes' ability to communicate. This means that none of the processes depends upon another to forward messages.

Note that a reliable channel *eventually* delivers a message to the recipient's input buffer. In a synchronous system, we suppose that there is hardware redundancy where necessary, so that a reliable channel not only eventually delivers each message despite underlying failures, but does so within a specified time bound.

In any particular interval of time, communication between some processes may succeed while communication between others is delayed. For example, the failure of a router between two networks may mean that a collection of four processes is split into two pairs, such that intra-pair communication is possible over their respective networks; but inter-pair communication is not possible while the router has failed. This is known as a *network partition* (Figure 15.1). Over a point-to-point network such as the Internet, complex topologies and independent routing choices mean that connectivity may be *asymmetric*: communication is possible from process p to process q, but not vice versa. Connectivity may also be *intransitive*: communication is possible from p to q and from q to r, but p cannot communicate directly with r. Thus our reliability assumption entails that eventually any failed link or router will be repaired or circumvented. Nevertheless, the processes may not all be able to communicate at the same time.

The chapter assumes, unless we state otherwise, that processes fail only by crashing – an assumption that is good enough for many systems. In Section 15.5, we shall consider how to treat the cases where processes have arbitrary (Byzantine) failures. Whatever the type of failure, a *correct* process is one that exhibits no failures at any point in the execution under consideration. Note that correctness applies to the whole execution, not just to a part of it. So a process that suffers a crash failure is 'non-failed' before that point, not 'correct' before that point.

One of the problems in the design of algorithms that can overcome process crashes is that of deciding when a process has crashed. A *failure detector* [Chandra and Toueg 1996, Stelling *et al.* 1998] is a service that processes queries about whether a particular process has failed. It is often implemented by an object local to each process (on the same computer) that runs a failure-detection algorithm in conjunction with its counterparts at other processes. The object local to each process is called a *local failure detector*. We outline how to implement failure detectors shortly, but first we concentrate on some of the properties of failure detectors.

A failure 'detector' is not necessarily accurate. Most fall into the category of *unreliable failure detectors*. An unreliable failure detector may produce one of two values when given the identity of a process: *Unsuspected* or *Suspected*. Both of these results are hints, which may or may not accurately reflect whether the process has actually failed. A result of *Unsuspected* signifies that the detector has recently received evidence suggesting that the process has not failed; for example, a message was recently received from it. But of course, the process may have failed since then. A result of *Suspected* signifies that the failure detector has some indication that the process may have failed. For example, it may be that no message from the process has been received for more than a nominal maximum length of silence (even in an asynchronous system, practical upper bounds can be used as hints). The suspicion may be misplaced: for example, the process could be functioning correctly but be on the other side of a network partition, or it could be running more slowly than expected.

A *reliable failure detector* is one that is always accurate in detecting a process's failure. It answers processes' queries with either a response of *Unsuspected* – which, as before, can only be a hint – or *Failed*. A result of *Failed* means that the detector has determined that the process has crashed. Recall that a process that has crashed stays that way, since by definition a process never takes another step once it has crashed.

It is important to realize that, although we speak of one failure detector acting for a collection of processes, the response that the failure detector gives to a process is only as good as the information available at that process. A failure detector may sometimes give different responses to different processes, since communication conditions vary from process to process.

We can implement an unreliable failure detector using the following algorithm. Each process p sends a 'p is here' message to every other process, and it does this every T seconds. The failure detector uses an estimate of the maximum message transmission time of D seconds. If the local failure detector at process q does not receive a 'p is here' message within $T + D$ seconds of the last one, then it reports to q that p is *Suspected*. However, if it subsequently receives a 'p is here' message, then it reports to q that p is *OK*.

In a real distributed system, there are practical limits on message transmission times. Even email systems give up after a few days, since it is likely that communication

links and routers will have been repaired in that time. If we choose small values for T and D (so that they total 0.1 second, say), then the failure detector is likely to suspect non-crashed processes many times, and much bandwidth will be taken up with 'p is here' messages. If we choose a large total timeout value (a week, say), then crashed processes will often be reported as *Unsuspected*.

A practical solution to this problem is to use timeout values that reflect the observed network delay conditions. If a local failure detector receives a 'p is here' in 20 seconds instead of the expected maximum of 10 seconds, it can reset its timeout value for p accordingly. The failure detector remains unreliable, and its answers to queries are still only hints, but the probability of its accuracy increases.

In a synchronous system, our failure detector can be made into a reliable one. We can choose D so that it is not an estimate but an absolute bound on message transmission times; the absence of a 'p is here' message within $T + D$ seconds entitles the local failure detector to conclude that p has crashed.

The reader may wonder whether failure detectors are of any practical use. Unreliable failure detectors may suspect a process that has not failed (they may be *inaccurate*), and they may not suspect a process that has in fact failed (they may be *incomplete*). Reliable failure detectors, on the other hand, require that the system is synchronous (and few practical systems are).

We have introduced failure detectors because they help us to think about the nature of failures in a distributed system. And any practical system that is designed to cope with failures must detect them – however imperfectly. But it turns out that even unreliable failure detectors with certain well-defined properties can help us to provide practical solutions to the problem of coordinating processes in the presence of failures. We return to this point in Section 15.5.

15.2 Distributed mutual exclusion

Distributed processes often need to coordinate their activities. If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources. This is the *critical section* problem, familiar in the domain of operating systems. In a distributed system, however, neither shared variables nor facilities supplied by a single local kernel can be used to solve it, in general. We require a solution to *distributed mutual exclusion*: one that is based solely on message passing.

In some cases shared resources are managed by servers that also provide mechanisms for mutual exclusion – Chapter 16 describes how some servers synchronize client accesses to resources. But in some practical cases, a separate mechanism for mutual exclusion is required.

Consider users who update a text file. A simple means of ensuring that their updates are consistent is to allow them to access it only one at a time, by requiring the editor to lock the file before updates can be made. NFS file servers, described in Chapter 12, are designed to be stateless and therefore do not support file locking. For this reason, UNIX systems provide a separate file-locking service, implemented by the daemon *lockd*, to handle locking requests from clients.

A particularly interesting example is where there is no server, and a collection of peer processes must coordinate their accesses to shared resources amongst themselves. This occurs routinely on networks such as Ethernets and IEEE 802.11 wireless networks in 'ad hoc' mode, where network interfaces cooperate as peers so that only one node transmits at a time on the shared medium. Consider, also, a system monitoring the number of vacancies in a car park with a process at each entrance and exit that tracks the number of vehicles entering and leaving. Each process keeps a count of the total number of vehicles within the car park and displays whether or not it is full. The processes must update the shared count of the number of vehicles consistently. There are several ways of achieving that, but it would be convenient for these processes to be able to obtain mutual exclusion solely by communicating among themselves, eliminating the need for a separate server.

It is useful to have a generic mechanism for distributed mutual exclusion at our disposal – one that is independent of the particular resource management scheme in question. We now examine some algorithms for achieving that.

15.2.1 Algorithms for mutual exclusion

We consider a system of N processes p_i, $i = 1, 2, \ldots, N$, that do not share variables. The processes access common resources, but they do so in a critical section. For the sake of simplicity, we assume that there is only one critical section. It is straightforward to extend the algorithms we present to more than one critical section.

We assume that the system is asynchronous, that processes do not fail and that message delivery is reliable, so that any message sent is eventually delivered intact, exactly once.

The application-level protocol for executing a critical section is as follows:

```
enter()              // enter critical section – block if necessary
resourceAccesses()   // access shared resources in critical section
exit()               // leave critical section – other processes may now enter
```

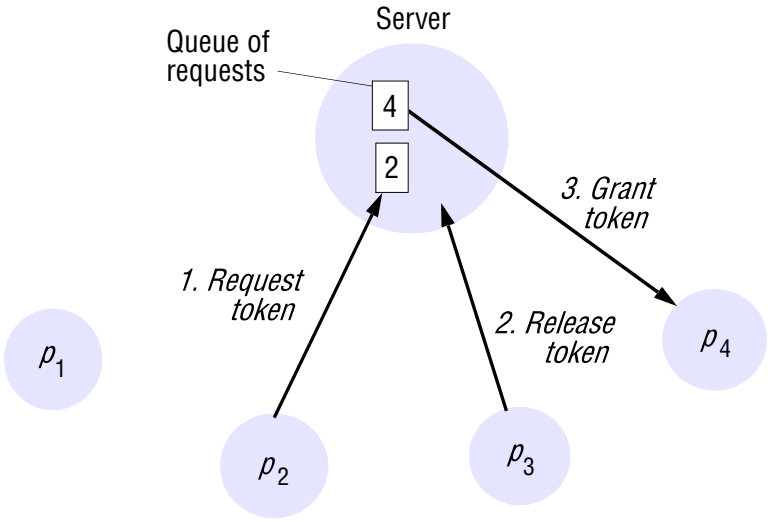
Our essential requirements for mutual exclusion are as follows:

ME1: (safety)	At most one process may execute in the critical section (CS) at a time.
ME2: (liveness)	Requests to enter and exit the critical section eventually succeed.

Condition ME2 implies freedom from both deadlock and starvation. A deadlock would involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the critical section, by virtue of their mutual interdependence. But even without a deadlock, a poor algorithm might lead to *starvation*: the indefinite postponement of entry for a process that has requested it.

The absence of starvation is a *fairness* condition. Another fairness issue is the order in which processes enter the critical section. It is not possible to order entry to the critical section by the times that the processes requested it, because of the absence of global clocks. But a useful fairness requirement that is sometimes made makes use of the happened-before ordering (Section 14.4) between messages that request entry to the critical section:

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



ME3: ($\rightarrow$ ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

If a solution grants entry to the critical section in happened-before order, and if all requests are related by happened-before, then it is not possible for a process to enter the critical section more than once while another waits to enter. This ordering also allows processes to coordinate their accesses to the critical section. A multi-threaded process may continue with other processing while a thread waits to be granted entry to a critical section. During this time, it might send a message to another process, which consequently also tries to enter the critical section. ME3 specifies that the first process be granted access before the second.

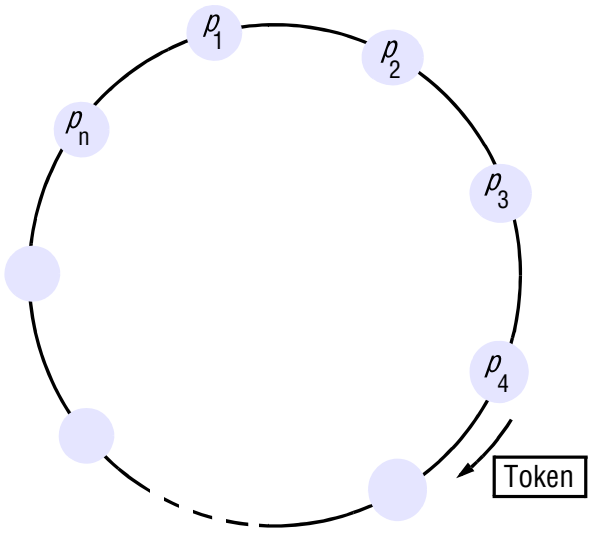
We evaluate the performance of algorithms for mutual exclusion according to the following criteria:

- the *bandwidth* consumed, which is proportional to the number of messages sent in each *entry* and *exit* operation;
- the *client delay* incurred by a process at each *entry* and *exit* operation;
- the algorithm's effect upon the *throughput* of the system. This is the rate at which the collection of processes as a whole can access the critical section, given that some communication is necessary between successive processes. We measure the effect using the *synchronization delay* between one process exiting the critical section and the next process entering it; the throughput is greater when the synchronization delay is shorter.

We do not take the implementation of resource accesses into account in our descriptions. We do, however, assume that the client processes are well behaved and spend a finite time accessing resources within their critical sections.

The central server algorithm • The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the critical section. Figure 15.2 shows the use of this server. To enter a critical section, a process sends a request message to

Figure 15.3 A ring of processes transferring a mutual exclusion token



the server and awaits a reply from it. Conceptually, the reply constitutes a token signifying permission to enter the critical section. If no other process has the token at the time of the request, then the server replies immediately, granting the token. If the token is currently held by another process, then the server does not reply, but queues the request. When a process exits the critical section, it sends a message to the server, giving it back the token.

If the queue of waiting processes is not empty, then the server chooses the oldest entry in the queue, removes it and replies to the corresponding process. The chosen process then holds the token. In the figure, we show a situation in which p_2's request has been appended to the queue, which already contained p_4's request. p_3 exits the critical section, and the server removes p_4's entry and grants permission to enter to p_4 by replying to it. Process p_1 does not currently require entry to the critical section.

Given our assumption that no failures occur, it is easy to see that the safety and liveness conditions are met by this algorithm. The reader should verify, however, that the algorithm does not satisfy property ME3.

We now evaluate the performance of this algorithm. Entering the critical section – even when no process currently occupies it – takes two messages (a *request* followed by a *grant*) and delays the requesting process by the time required for this round-trip. Exiting the critical section takes one *release* message. Assuming asynchronous message passing, this does not delay the exiting process.

The server may become a performance bottleneck for the system as a whole. The synchronization delay is the time taken for a round-trip: a *release* message to the server, followed by a *grant* message to the next process to enter the critical section.

A ring-based algorithm • One of the simplest ways to arrange mutual exclusion between the N processes without requiring an additional process is to arrange them in a logical ring. This requires only that each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$. The idea is that exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction –

Figure 15.4 Ricart and Agrawala's algorithm

```
On initialization
    state := RELEASED;

To enter the section
    state := WANTED;
    Multicast request to all processes;          } Request processing deferred here
    T := request's timestamp;
    Wait until (number of replies received = (N – 1));
    state := HELD;

On receipt of a request <Ti, pi> at pj (i≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
                queue request from pi without replying;
    else
                reply immediately to pi;
    end if

To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

clockwise, say – around the ring. The ring topology may be unrelated to the physical interconnections between the underlying computers.

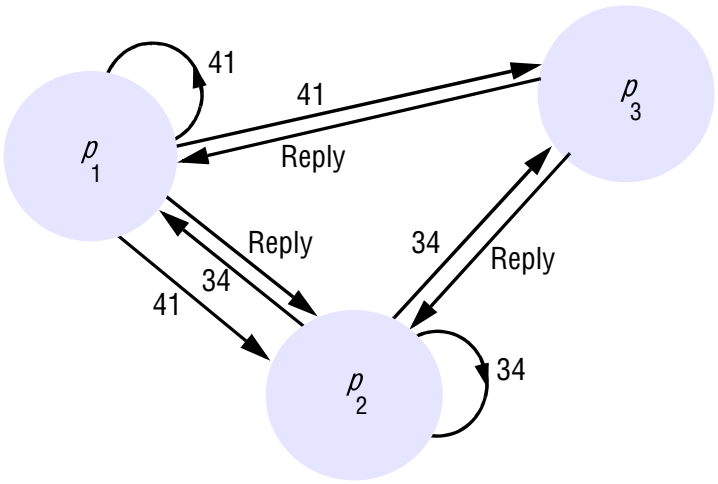
If a process does not require to enter the critical section when it receives the token, then it immediately forwards the token to its neighbour. A process that requires the token waits until it receives it, but retains it. To exit the critical section, the process sends the token on to its neighbour.

The arrangement of processes is shown in Figure 15.3. It is straightforward to verify that the conditions ME1 and ME2 are met by this algorithm, but that the token is not necessarily obtained in happened-before order. (Recall that the processes may exchange messages independently of the rotation of the token.)

This algorithm continuously consumes network bandwidth (except when a process is inside the critical section): the processes send messages around the ring even when no process requires entry to the critical section. The delay experienced by a process requesting entry to the critical section is between 0 messages (when it has just received the token) and N messages (when it has just passed on the token). To exit the critical section requires only one message. The synchronization delay between one process's exit from the critical section and the next process's entry is anywhere from 1 to N message transmissions.

An algorithm using multicast and logical clocks • Ricart and Agrawala [1981] developed an algorithm to implement mutual exclusion between N peer processes that is based upon multicast. The basic idea is that processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have

Figure 15.5 Multicast synchronization



replied to this message. The conditions under which a process replies to a request are designed to ensure that conditions ME1–ME3 are met.

The processes $p_1, p_2, \ldots, p_N$ bear distinct numeric identifiers. They are assumed to possess communication channels to one another, and each process p_i keeps a Lamport clock, updated according to the rules LC1 and LC2 of Section 14.4. Messages requesting entry are of the form $<T, p_i>$, where T is the sender's timestamp and p_i is the sender's identifier.

Each process records its state of being outside the critical section (*RELEASED*), wanting entry (*WANTED*) or being in the critical section (*HELD*) in a variable *state*. The protocol is given in Figure 15.4.

If a process requests entry and the state of all other processes is *RELEASED*, then all processes will reply immediately to the request and the requester will obtain entry. If some process is in the state *HELD*, then that process will not reply to requests until it has finished with the critical section, and so the requester cannot gain entry in the meantime. If two or more processes request entry at the same time, then whichever process's request bears the lowest timestamp will be the first to collect $N-1$ replies, granting it entry next. If the requests bear equal Lamport timestamps, the requests are ordered according to the processes' corresponding identifiers. Note that, when a process requests entry, it defers processing requests from other processes until its own request has been sent and it has recorded the timestamp T of the request. This is so that processes make consistent decisions when processing requests.

This algorithm achieves the safety property ME1. If it were possible for two processes p_i and p_j $(i \neq j)$ to enter the critical section at the same time, then both of those processes would have to have replied to the other. But since the pairs $<T_i, p_i>$ are totally ordered, this is impossible. We leave the reader to verify that the algorithm also meets requirements ME2 and ME3.

To illustrate the algorithm, consider a situation involving three processes, p_1, p_2 and p_3, shown in Figure 15.5. Let us assume that p_3 is not interested in entering the critical section, and that p_1 and p_2 request entry concurrently. The timestamp of p_1's request is 41, and that of p_2 is 34. When p_3 receives their requests, it replies

immediately. When p_2 receives p_1's request, it finds that its own request has the lower timestamp and so does not reply, holding p_1 off. However, p_1 finds that p_2's request has a lower timestamp than that of its own request and so replies immediately. On receiving this second reply, p_2 can enter the critical section. When p_2 exits the critical section, it will reply to p_1's request and so grant it entry.

Gaining entry takes $2(N-1)$ messages in this algorithm: $N-1$ to multicast the request, followed by $N-1$ replies. Or, if there is hardware support for multicast, only one message is required for the request; the total is then N messages. It is thus a more expensive algorithm, in terms of bandwidth consumption, than the algorithms just described. However, the client delay in requesting entry is again a round-trip time (ignoring any delay incurred in multicasting the request message).

The advantage of this algorithm is that its synchronization delay is only one message transmission time. Both the previous algorithms incurred a round-trip synchronization delay.

The performance of the algorithm can be improved. First, note that the process that last entered the critical section and that has received no other requests for it still goes through the protocol as described, even though it could simply decide locally to reenter the critical section. Second, Ricart and Agrawala refined this protocol so that it requires N messages to obtain entry in the worst (and common) case, without hardware support for multicast. This is described in Raynal [1988].

Maekawa's voting algorithm • Maekawa [1985] observed that in order for a process to enter a critical section, it is not necessary for all of its peers to grant it access. Processes need only obtain permission to enter from *subsets* of their peers, as long as the subsets used by any two processes overlap. We can think of processes as voting for one another to enter the critical section. A 'candidate' process must collect sufficient votes to enter. Processes in the intersection of two sets of voters ensure the safety property ME1, that at most one process can enter the critical section, by casting their votes for only one candidate.

Maekawa associated a *voting set* V_i with each process p_i ($i = 1, 2, \ldots, N$), where $V_i \subseteq \{p_1, p_1, \ldots, p_N\}$. The sets V_i are chosen so that, for all $i, j = 1, 2, \ldots, N$:

- $p_i \in V_i$
- $V_i \cap V_j \neq \varnothing$ – there is at least one common member of any two voting sets
- $|V_i| = K$ – to be fair, each process has a voting set of the same size
- Each process p_j is contained in M of the voting sets V_i.

Maekawa showed that the optimal solution, which minimizes K and allows the processes to achieve mutual exclusion, has $K \sim \sqrt{N}$ and $M = K$ (so that each process is in as many of the voting sets as there are elements in each one of those sets). It is non-trivial to calculate the optimal sets R_i. As an approximation, a simple way of deriving sets R_i such that $|R_i| \sim 2\sqrt{N}$ is to place the processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix and let V_i be the union of the row and column containing p_i.

Maekawa's algorithm is shown in Figure 15.6. To obtain entry to the critical section, a process p_i sends *request* messages to all K members of V_i (including itself). p_i cannot enter the critical section until it has received all K *reply* messages. When a process p_j in V_i receives p_i's *request* message, it sends a *reply* message immediately,

Figure 15.6 Maekawa's algorithm

```
On initialization
    state := RELEASED;
    voted := FALSE;

For p_i to enter the critical section
    state := WANTED;
    Multicast request to all processes in V_i;
    Wait until (number of replies received = K);
    state := HELD;

On receipt of a request from p_i at p_j
    if (state = HELD or voted = TRUE)
    then
                queue request from p_i without replying;
    else
                send reply to p_i;
                voted := TRUE;
    end if

For p_i to exit the critical section
    state := RELEASED;
    Multicast release to all processes in V_i;

On receipt of a release from p_i at p_j
    if (queue of requests is non-empty)
    then
                remove head of queue – from p_k, say;
                send reply to p_k;
                voted := TRUE;
    else
                voted := FALSE;
    end if
```

unless either its state is *HELD* or it has already replied ('voted') since it last received a *release* message. Otherwise, it queues the request message (in the order of its arrival) but does not yet reply. When a process receives a *release* message, it removes the head of its queue of outstanding requests (if the queue is nonempty) and sends a *reply* message (a 'vote') in response to it. To leave the critical section, p_i sends *release* messages to all K members of V_i (including itself).

This algorithm achieves the safety property, ME1. If it were possible for two processes p_i and p_j to enter the critical section at the same time, then the processes in $V_i \cap V_j \neq \varnothing$ would have to have voted for both. But the algorithm allows a process to make at most one vote between successive receipts of a *release* message, so this situation is impossible.

Unfortunately, the algorithm is deadlock-prone. Consider three processes, p_1, p_2 and p_3, with $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ and $V_3 = \{p_3, p_1\}$. If the three

processes concurrently request entry to the critical section, then it is it is possible for p_1 to reply to itself and hold off p_2, for p_2 to reply to itself and hold off p_3, and for p_3 to reply to itself and hold off p_1. Each process has received one out of two replies, and none can proceed.

The algorithm can be adapted [Sanders 1987] so that it becomes deadlock-free. In the adapted protocol, processes queue outstanding requests in happened-before order, so that requirement ME3 is also satisfied.

The algorithm's bandwidth utilization is $2\sqrt{N}$ messages per entry to the critical section and $\sqrt{N}$ messages per exit (assuming no hardware multicast facilities). The total of $3\sqrt{N}$ is superior to the $2(N-1)$ messages required by Ricart and Agrawala's algorithm, if $N > 4$. The client delay is the same as that of Ricart and Agrawala's algorithm, but the synchronization delay is worse: a round-trip time instead of a single message transmission time.

Fault tolerance • The main points to consider when evaluating the above algorithms with respect to fault tolerance are:

- What happens when messages are lost?
- What happens when a process crashes?

None of the algorithms that we have described would tolerate the loss of messages, if the channels were unreliable. The ring-based algorithm cannot tolerate a crash failure of any single process. As it stands, Maekawa's algorithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required, then its failure will not affect the other processes. The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token. The Ricart and Agrawala algorithm as we have described it can be adapted to tolerate the crash failure of such a process, by taking it to grant all requests implicitly.

We invite the reader to consider how to adapt the algorithms to tolerate failures, on the assumption that a reliable failure detector is available. Even with a reliable failure detector, care is required to allow for failures at any point (including during a recovery procedure), and to reconstruct the state of the processes after a failure has been detected. For example, in the central-server algorithm, if the server fails it must be established whether it or one of the client processes held the token.

We examine the general problem of how processes should coordinate their actions in the presence of faults in Section 15.5.

15.3 Elections

An algorithm for choosing a unique process to play a particular role is called an *election algorithm*. For example, in a variant of our central-server algorithm for mutual exclusion, the 'server' is chosen from among the processes p_i, $(i = 1, 2, \ldots, N)$ that need to use the critical section. An election algorithm is needed for this choice. It is essential that all the processes agree on the choice. Afterwards, if the process that plays the role of server wishes to retire then another election is required to choose a replacement.

We say that a process *calls the election* if it takes an action that initiates a particular run of the election algorithm. An individual process does not call more than one election at a time, but in principle the N processes could call N concurrent elections. At any point in time, a process p_i is either a *participant* – meaning that it is engaged in some run of the election algorithm – or a *non-participant* – meaning that it is not currently engaged in any election.

An important requirement is for the choice of elected process to be unique, even if several processes call elections concurrently. For example, two processes could decide independently that a coordinator process has failed, and both call elections.

Without loss of generality, we require that the elected process be chosen as the one with the largest identifier. The 'identifier' may be any useful value, as long as the identifiers are unique and totally ordered. For example, we could elect the process with the lowest computational load by having each process use $<1/load, i>$ as its identifier, where $load > 0$ and the process index i is used to order identifiers with the same load.

Each process p_i $(i = 1, 2, \ldots, N)$ has a variable $elected_i$, which will contain the identifier of the elected process. When the process first becomes a participant in an election it sets this variable to the special value '$\perp$' to denote that it is not yet defined.

Our requirements are that, during any particular run of the algorithm:

E1: (safety) A participant process p_i has $elected_i = \perp$ or $elected_i = P$, where P is chosen as the non-crashed process at the end of the run with the largest identifier.

E2: (liveness) All processes p_i participate and eventually either set $elected_i \neq \perp$ – or crash.

Note that there may be processes p_j that are not yet participants, which record in $elected_j$ the identifier of the previous elected process.

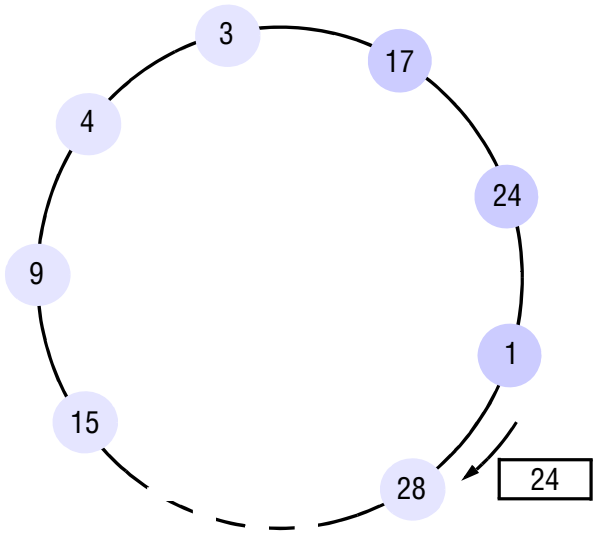
We measure the performance of an election algorithm by its total network bandwidth utilization (which is proportional to the total number of messages sent), and by the *turnaround time* for the algorithm: the number of serialized message transmission times between the initiation and termination of a single run.

A ring-based election algorithm • The algorithm of Chang and Roberts [1979] is suitable for a collection of processes arranged in a logical ring. Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) mod N}$, and all messages are sent clockwise around the ring. We assume that no failures occur, and that the system is asynchronous. The goal of this algorithm is to elect a single process called the *coordinator*, which is the process with the largest identifier.

Initially, every process is marked as a *non-participant* in an election. Any process can begin an election. It proceeds by marking itself as a *participant*, placing its identifier in an *election* message and sending it to its clockwise neighbour.

When a process receives an *election* message, it compares the identifier in the message with its own. If the arrived identifier is greater, then it forwards the message to its neighbour. If the arrived identifier is smaller and the receiver is not a *participant*, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a *participant*. On forwarding an *election* message in any case, the process marks itself as a *participant*.

Figure 15.7 A ring-based election in progress



Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown in a darker tint.

If, however, the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the coordinator. The coordinator marks itself as a *non-participant* once more and sends an *elected* message to its neighbour, announcing its election and enclosing its identity.

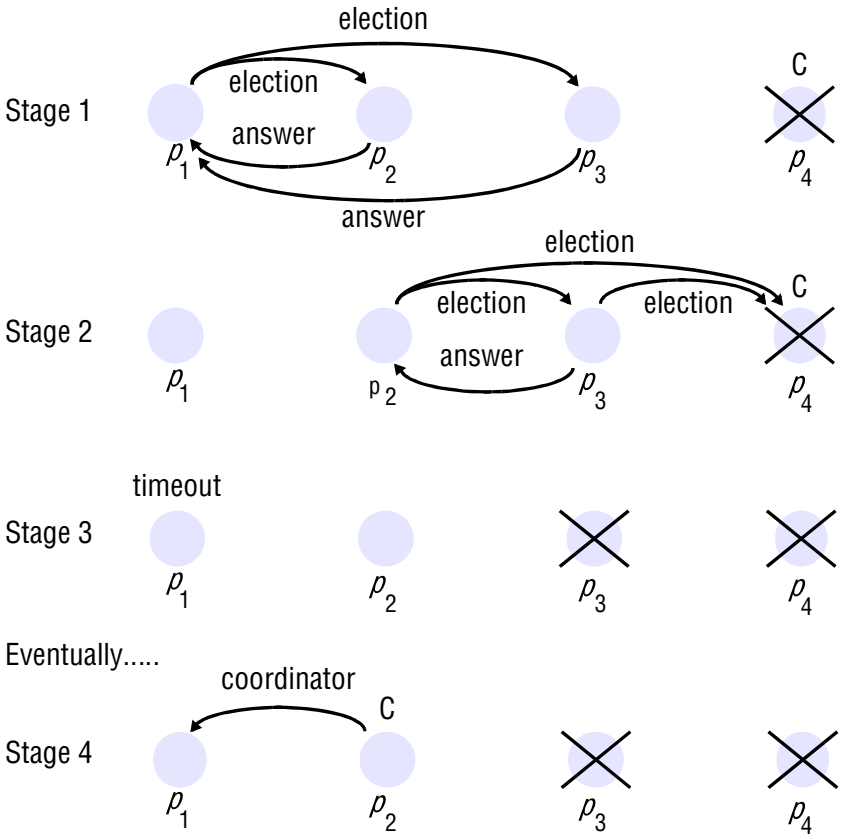
When a process p_i receives an *elected* message, it marks itself as a *non-participant*, sets its variable $elected_i$ to the identifier in the message and, unless it is the new coordinator, forwards the message to its neighbour.

It is easy to see that condition E1 is met. All identifiers are compared, since a process must receive its own identifier back before sending an *elected* message. For any two processes, the one with the larger identifier will not pass on the other's identifier. It is therefore impossible that both should receive their own identifier back.

Condition E2 follows immediately from the guaranteed traversals of the ring (there are no failures). Note how the *non-participant* and *participant* states are used so that duplicate messages arising when two processes start an election at the same time are extinguished as soon as possible, and always before the 'winning' election result has been announced.

If only a single process starts an election, then the worst-performing case is when its anti-clockwise neighbour has the highest identifier. A total of $N-1$ messages are then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further N messages. The *elected* message is then sent N times, making $3N-1$ messages in all. The turnaround time is also $3N-1$, since these messages are sent sequentially.

Figure 15.8 The bully algorithm



The election of coordinator p_2, after the failure of p_4 and then p_3

An example of a ring-based election in progress is shown in Figure 15.7. The *election* message currently contains 24, but process 28 will replace this with its identifier when the message reaches it.

While the ring-based algorithm is useful for understanding the properties of election algorithms in general, the fact that it tolerates no failures makes it of limited practical value. However, with a reliable failure detector it is in principle possible to re-constitute the ring when a process crashes.

The bully algorithm • The bully algorithm [Garcia-Molina 1982] allows processes to crash during an election, although it assumes that message delivery between processes is reliable. Unlike the ring-based algorithm, this algorithm assumes that the system is synchronous: it uses timeouts to detect a process failure. Another difference is that the ring-based algorithm assumed that processes have minimal *a priori* knowledge of one another: each knows only how to communicate with its neighbour, and none knows the identifiers of the other processes. The bully algorithm, on the other hand, assumes that each process knows which processes have higher identifiers, and that it can communicate with all such processes.

There are three types of message in this algorithm: an *election* message is sent to announce an election; an *answer* message is sent in response to an election message and a *coordinator* message is sent to announce the identity of the elected process – the new

'coordinator'. A process begins an election when it notices, through timeouts, that the coordinator has failed. Several processes may discover this concurrently.

Since the system is synchronous, we can construct a reliable failure detector. There is a maximum message transmission delay, T_{trans}, and a maximum delay for processing a message $T_{process}$. Therefore, we can calculate a time $T = 2T_{trans} + T_{process}$ that is an upper bound on the time that can elapse between sending a message to another process and receiving a response. If no response arrives within time *T*, then the local failure detector can report that the intended recipient of the request has failed.

The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a *coordinator* message to all processes with lower identifiers. On the other hand, a process with a lower identifier can begin an election by sending an *election* message to those processes that have a higher identifier and awaiting *answer* messages in response. If none arrives within time *T*, the process considers itself the coordinator and sends a *coordinator* message to all processes with lower identifiers announcing this. Otherwise, the process waits a further period T' for a *coordinator* message to arrive from the new coordinator. If none arrives, it begins another election.

If a process p_i receives a *coordinator* message, it sets its variable $elected_i$ to the identifier of the coordinator contained within it and treats that process as the coordinator.

If a process receives an *election* message, it sends back an *answer* message and begins another election – unless it has begun one already.

When a process is started to replace a crashed process, it begins an election. If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes. Thus it will become the coordinator, even though the current coordinator is functioning. It is for this reason that the algorithm is called the 'bully' algorithm.

The operation of the algorithm is shown in Figure 15.8. There are four processes, $p_1 - p_4$. Process p_1 detects the failure of the coordinator p_4 and announces an election (stage 1 in the figure). On receiving an *election* message from p_1, processes p_2 and p_3 send *answer* messages to p_1 and begin their own elections; p_3 sends an *answer* message to p_2, but p_3 receives no *answer* message from the failed process p_4 (stage 2). It therefore decides that it is the coordinator. But before it can send out the *coordinator* message, it too fails (stage 3). When p_1's timeout period T' expires (which we assume occurs before p_2's timeout expires), it deduces the absence of a *coordinator* message and begins another election. Eventually, p_2 is elected coordinator (stage 4).

This algorithm clearly meets the liveness condition E2, by the assumption of reliable message delivery. And if no process is replaced, then the algorithm meets condition E1. It is impossible for two processes to decide that they are the coordinator, since the process with the lower identifier will discover that the other exists and defer to it.

But the algorithm is *not* guaranteed to meet the safety condition E1 if processes that have crashed are replaced by processes with the same identifiers. A process that replaces a crashed process *p* may decide that it has the highest identifier just as another process (which has detected *p*'s crash) decides that *it* has the highest identifier. Two processes will therefore announce themselves as the coordinator concurrently. Unfortunately, there are no guarantees on message delivery order, and the recipients of these messages may reach different conclusions on which is the coordinator process.

Furthermore, condition E1 may be broken if the assumed timeout values turn out to be inaccurate – that is, if the processes' failure detector is unreliable.

Taking the example just given, suppose that either p_3 had not failed but was just running unusually slowly (that is, that the assumption that the system is synchronous is incorrect), or that p_3 had failed but was then replaced. Just as p_2 sends its *coordinator* message, p_3 (or its replacement) does the same. p_2 receives p_3 's *coordinator* message after it has sent its own and so sets $elected_2 = p_3$. Due to variable message transmission delays, p_1 receives p_2's *coordinator* message after p_3's and so eventually sets $elected_1 = p_2$. Condition E1 has been broken.

With regard to the performance of the algorithm, in the best case the process with the second-highest identifier notices the coordinator's failure. Then it can immediately elect itself and send $N-2$ coordinator messages. The turnaround time is one message. The bully algorithm requires $O(N^2)$ messages in the worst case – that is, when the process with the lowest identifier first detects the coordinator's failure. For then $N-1$ processes altogether begin elections, each sending messages to processes with higher identifiers.

15.4 Coordination and agreement in group communication

This chapter examines the key coordination and agreement problems related to group communication – that is, how to achieve the desired reliability and ordering properties across all members of a group. Chapter 6 introduced group communication as an example of an indirect communication technique whereby processes can send messages to a group. This message is propagated to all members of the group with certain guarantees in terms of reliability and ordering. We are particularly seeking reliability in terms of the properties of validity, integrity and agreement, and ordering in terms of FIFO ordering, causal ordering and total ordering.

In this chapter, we study multicast communication to groups of processes whose membership is known. Chapter 18 will expand our study to fully fledged group communication, including the management of dynamically varying groups.

System model • The system under consideration contains a collection of processes, which can communicate reliably over one-to-one channels. As before, processes may fail only by crashing.

The processes are members of groups, which are the destinations of messages sent with the *multicast* operation. It is generally useful to allow processes to be members of several groups simultaneously – for example, to enable processes to receive information from several sources by joining several groups. But to simplify our discussion of ordering properties, we shall sometimes restrict processes to being members of at most one group at a time.

The operation *multicast*(g, m) sends the message m to all members of the group g of processes. Correspondingly, there is an operation *deliver*(m) that delivers a message sent by multicast to the calling process. We use the term *deliver* rather than *receive* to make clear that a multicast message is not always handed to the application layer inside

the process as soon as it is received at the process's node. This is explained when we discuss multicast delivery semantics shortly.

Every message m carries the unique identifier of the process *sender*(m) that sent it, and the unique destination group identifier *group*(m). We assume that processes do not lie about the origin or destinations of messages.

Some algorithms assume that groups are closed (as defined in Chapter 6).

15.4.1 Basic multicast

It is useful to have at our disposal a basic multicast primitive that guarantees, unlike IP multicast, that a correct process will eventually deliver the message, as long as the multicaster does not crash. We call the primitive *B-multicast* and its corresponding basic delivery primitive *B-deliver*. We allow processes to belong to several groups, and each message is destined for some particular group.

A straightforward way to implement *B-multicast* is to use a reliable one-to-one *send* operation, as follows:

To *B-multicast*(g, m): for each process $p \in g$, *send*(p, m);

On *receive*(m) at p: *B-deliver*(m) at p.

The implementation may use threads to perform the *send* operations concurrently, in an attempt to reduce the total time taken to deliver the message. Unfortunately, such an implementation is liable to suffer from a so-called *ack-implosion* if the number of processes is large. The acknowledgements sent as part of the reliable *send* operation are liable to arrive from many processes at about the same time. The multicasting process's buffers will rapidly fill, and it is liable to drop acknowledgements. It will therefore retransmit the message, leading to yet more acknowledgements and further waste of network bandwidth. A more practical basic multicast service can be built using IP multicast, and we invite the reader to show this in Exercise 15.10.

15.4.2 Reliable multicast

Chapter 6 discussed reliable multicast in terms of validity, integrity and agreement. This section builds on this informal discussion, presenting a more complete definition.

Following Hadzilacos and Toueg [1994] and Chandra and Toueg [1996], we define a *reliable multicast* with corresponding operations *R-multicast* and *R-deliver*. Properties analogous to integrity and validity are clearly highly desirable in reliable multicast delivery, but we add another: a requirement that *all* correct processes in the group must receive a message if *any* of them does. It is important to realize that this is not a property of the *B-multicast* algorithm that is based on a reliable one-to-one *send* operation. The sender may fail at any point while *B-multicast* proceeds, so some processes may deliver a message while others do not.

A reliable multicast is one that satisfies the following properties:

Integrity: A correct process p delivers a message m at most once. Furthermore, $p \in group(m)$ and m was supplied to a *multicast* operation by *sender*(m). (As with one-to-one communication, messages can always be distinguished by a sequence number relative to their sender.)

Figure 15.9 Reliable multicast algorithm

```
On initialization
    Received := {};

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

On B-deliver(m) at process q with g = group(m)
    if ( m ∉ Received )
    then
                Received := Received ∪ { m } ;
                if ( q ≠ p ) then B-multicast(g, m); end if
                R-deliver m;
    end if
```

Validity: If a correct process multicasts message *m*, then it will eventually deliver *m*.

Agreement: If a correct process delivers message *m*, then all other correct processes in *group*(*m*) will eventually deliver *m*.

The integrity property is analogous to that for reliable one-to-one communication. The validity property guarantees liveness for the sender. This may seem an unusual property, because it is asymmetric (it mentions only one particular process). But notice that validity and agreement together amount to an overall liveness requirement: if one process (the sender) eventually delivers a message *m*, since the correct processes agree on the set of messages they deliver, it follows that *m* will eventually be delivered to all the group's correct members.

The advantage of expressing the validity condition in terms of self-delivery is simplicity. What we require is that the message be delivered eventually by *some* correct member of the group.

The agreement condition is related to atomicity, the property of 'all or nothing', applied to delivery of messages to a group. If a process that multicasts a message crashes before it has delivered it, then it is possible that the message will not be delivered to any process in the group; but if it is delivered to some correct process, then all other correct processes will deliver it. Many papers in the literature use the term 'atomic' to include a total ordering condition; we define this shortly.

Implementing reliable multicast over B-multicast • Figure 15.9 gives a reliable multicast algorithm, with primitives *R-multicast* and *R-deliver*, that allows processes to belong to several closed groups simultaneously. To *R-multicast* a message, a process *B-multicast*s the message to the processes in the destination group (including itself). When the message is *B-deliver*ed, the recipient in turn *B-multicast*s the message to the group (if it is not the original sender), and then *R-deliver*s the message. Since a message may arrive more than once, duplicates of the message are detected and not delivered.

This algorithm clearly satisfies the validity property, since a correct process will eventually *B-deliver* the message to itself. By the integrity property of the underlying communication channels used in *B-multicast*, the algorithm also satisfies the integrity property.

Agreement follows from the fact that every correct process *B-multicast*s the message to the other processes after it has *B-deliver*ed it. If a correct process does not *R-deliver* the message, then this can only be because it never *B-deliver*ed it. That in turn can only be because no other correct process *B-deliver*ed it either; therefore none will *R-deliver* it.

The reliable multicast algorithm that we have described is correct in an asynchronous system, since we made no timing assumptions. But the algorithm is inefficient for practical purposes. Each message is sent $|g|$ times to each process.

Reliable multicast over IP multicast • An alternative realization of *R-multicast* is to use a combination of IP multicast, piggybacked acknowledgements (that is, acknowledgements attached to other messages) and negative acknowledgements. This *R-multicast* protocol is based on the observation that IP multicast communication is often successful. In the protocol, processes do not send separate acknowledgement messages; instead, they piggyback acknowledgements on the messages that they send to the group. Processes send a separate response message only when they detect that they have missed a message. A response indicating the absence of an expected message is known as a *negative acknowledgement.*

The description assumes that groups are closed. Each process p maintains a sequence number S_g^p for each group g to which it belongs. The sequence number is initially zero. Each process also records R_g^q, the sequence number of the latest message it has delivered from process q that was sent to group g.

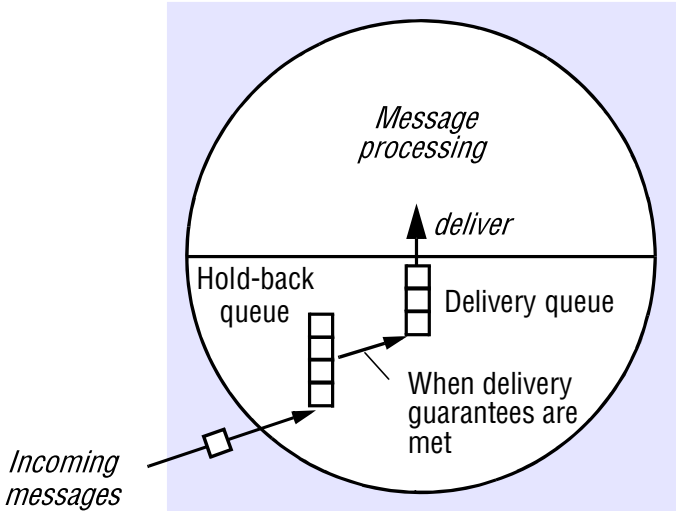
For p to *R-multicast* a message to group g, it piggybacks onto the message the value S_g^p and acknowledgements, of the form $<q, R_g^q>$. An acknowledgement states, for some sender q, the sequence number of the latest message from q destined for g that p has delivered since it last multicast a message. The multicaster p then IP-multicasts the message with its piggybacked values to g, and increments S_g^p by one.

The piggybacked values in a multicast message enable the recipients to learn about messages that they have not received. A process *R-deliver*s a message destined for g bearing the sequence number S from p if and only if $S = R_g^p + 1$, and it increments R_g^p by one immediately after delivery. If an arriving message has $S \leq R_g^p$, then r has delivered the message before and it discards it. If $S > R_g^p + 1$, or if $R > R_g^q$ for an enclosed acknowledgement $<q, R>$, then there are one or more messages that it has not yet received (and which are likely to have been dropped, in the first case). It keeps any message for which $S > R_g^p + 1$ in a *hold-back queue* (Figure 15.10) – such queues are often used to meet message delivery guarantees. It requests missing messages by sending negative acknowledgements, either to the original sender or to a process q from which it has received an acknowledgement $<q, R_g^q>$ with R_g^q no less than the required sequence number.

The hold-back queue is not strictly necessary for reliability, but it simplifies the protocol by enabling us to use sequence numbers to represent sets of delivered messages. It also provides us with a guarantee of delivery order (see Section 15.4.3).

The integrity property follows from the detection of duplicates and the underlying properties of IP multicast (which uses checksums to expunge corrupted messages). The validity property holds because IP multicast has that property. For agreement we require, first, that a process can always detect missing messages. That in turn means that it will always receive a further message that enables it to detect the omission. As this

Figure 15.10 The hold-back queue for arriving multicast messages



simplified protocol stands, we guarantee detection of missing messages only in the case where correct processes multicast messages indefinitely. Second, the agreement property requires that there is always an available copy of any message needed by a process that did not receive it. We therefore assume that processes retain copies of the messages they have delivered – indefinitely, in this simplified protocol.

Neither of the assumptions we made to ensure agreement is practical (see Exercise 15.15). However, agreement is practically addressed in the protocols from which ours is derived: the Psync protocol [Peterson *et al.* 1989], Trans protocol [Melliar-Smith *et al.* 1990] and scalable reliable multicast protocol [Floyd *et al.* 1997]. Psync and Trans also provide further delivery ordering guarantees.

Uniform properties • The definition of agreement given above refers only to the behaviour of *correct* processes – processes that never fail. Consider what would happen in the algorithm of Figure 15.9 if a process was not correct and crashed after it had *R-deliver*ed a message. Since any process that *R-deliver*s the message must first *B-multicast* it, it follows that all correct processes will still eventually deliver the message.

Any property that holds whether or not processes are correct is called a *uniform* property. We define uniform agreement as follows:

> *Uniform agreement*: If a process, whether it is correct or fails, delivers message *m*, then all correct processes in *group*(*m*) will eventually deliver *m*.

Uniform agreement allows a process to crash after it has delivered a message, while still ensuring that all correct processes will deliver the message. We have argued that the algorithm of Figure 15.9 satisfies this property, which is stronger than the non-uniform agreement property defined above.

Uniform agreement is useful in applications where a process may take an action that produces an observable inconsistency before it crashes. For example, suppose that the processes are servers that manage copies of a bank account, and that updates to the account are sent using reliable multicast to the group of servers. If the multicast does not satisfy uniform agreement, then a client that accesses a server just before it crashes may observe an update that no other server will process.

It is interesting to note that if we reverse the lines '*R-deliver m*' and '*if* ($q \neq p$) *then B-multicast*(g, m); *end if*' in Figure 15.9, then the resultant algorithm does not satisfy uniform agreement.

Just as there is a uniform version of agreement, there are also uniform versions of any multicast property, including validity and integrity and the ordering properties that we are about to define.

15.4.3 Ordered multicast

The basic multicast algorithm of Section 15.4.1 delivers messages to processes in an arbitrary order, due to arbitrary delays in the underlying one-to-one *send* operations. This lack of an ordering guarantee is not satisfactory for many applications. For example, in a nuclear power plant it may be important that events signifying threats to safety conditions and events signifying actions by control units are observed in the same order by all processes in the system.

As discussed in Chapter 6, the common ordering requirements are total ordering, causal ordering and FIFO ordering, together with hybrid solutions (in particular, total-causal and total-FIFO). To simplify our discussion, we define these orderings under the assumption that any process belongs to at most one group (later we discuss the implications of allowing groups to overlap):

FIFO ordering: If a correct process issues *multicast*(g, m) and then *multicast*(g, m'), then every correct process that delivers m' will deliver m before m'.

Causal ordering: If *multicast*(g, m) $\rightarrow$ *multicast*(g, m'), where $\rightarrow$ is the happened-before relation induced only by messages sent between the members of g, then any correct process that delivers m' will deliver m before m'.

Total ordering: If a correct process delivers message m before it delivers m', then any other correct process that delivers m' will deliver m before m'.

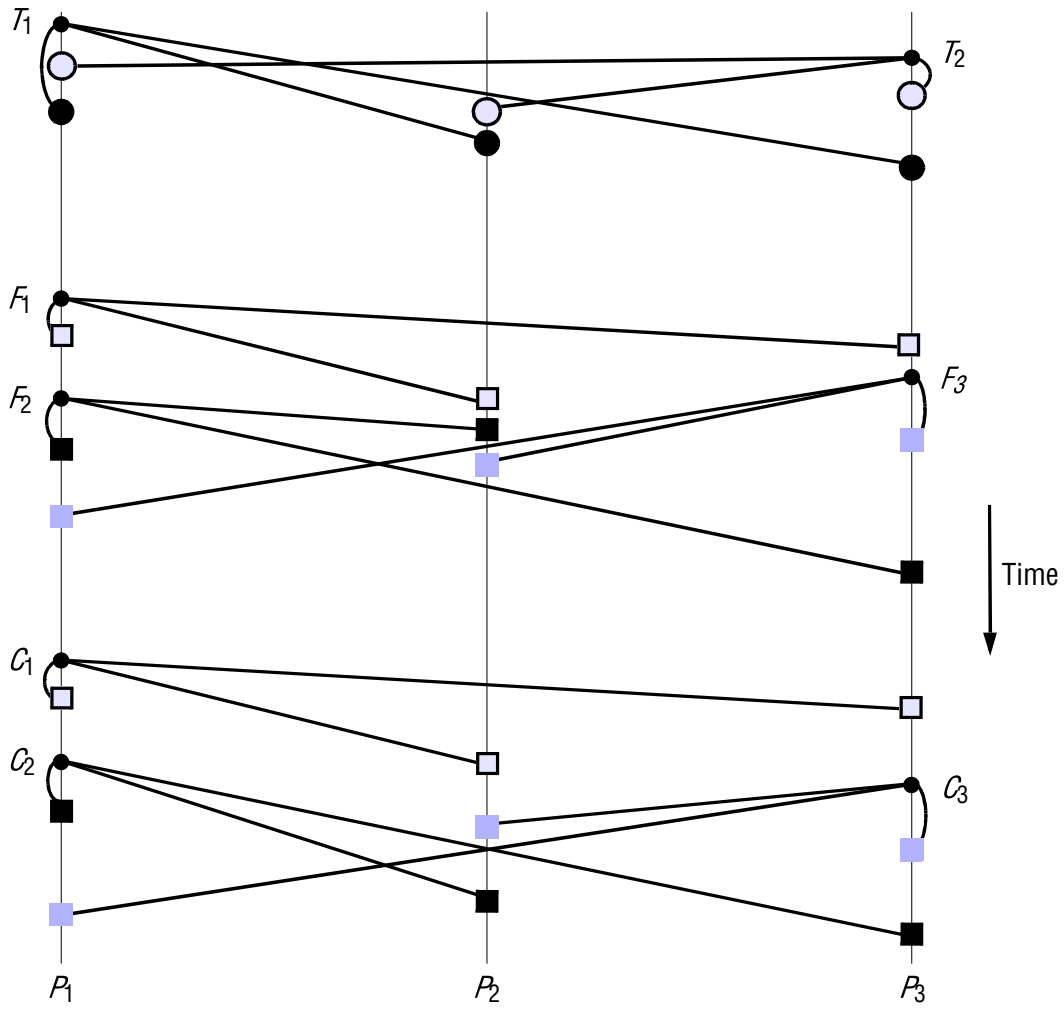
Causal ordering implies FIFO ordering, since any two multicasts by the same process are related by happened-before. Note that FIFO ordering and causal ordering are only partial orderings: not all messages are sent by the same process, in general; similarly, some multicasts are concurrent (not ordered by happened-before).

Figure 15.11 illustrates the orderings for the case of three processes. Close inspection of the figure shows that the totally ordered messages are delivered in the opposite order to the physical time at which they were sent. In fact, the definition of total ordering allows message delivery to be ordered arbitrarily, as long as the order is the same at different processes. Since total ordering is not necessarily also a FIFO or causal ordering, we define the hybrid of *FIFO-total* ordering as one for which message delivery obeys both FIFO and total ordering; similarly, under *causal-total* ordering message delivery obeys both causal and total ordering.

The definitions of ordered multicast do not assume or imply reliability. For example, the reader should check that, under total ordering, if correct process p delivers message m and then delivers m', then a correct process q can deliver m without also delivering m' or any other message ordered after m.

We can also form hybrids of ordered and reliable protocols. A reliable totally ordered multicast is often referred to in the literature as an *atomic multicast*. Similarly,

Figure 15.11 Total, FIFO and causal ordering of multicast messages



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

we may form reliable FIFO multicast, reliable causal multicast and reliable versions of the hybrid ordered multicasts.

Ordering the delivery of multicast messages, as we shall see, can be expensive in terms of delivery latency and bandwidth consumption. The ordering semantics that we have described may delay the delivery of messages unnecessarily. That is, at the application level, a message may be delayed for another message that it does not in fact depend upon. For this reason, some have proposed multicast systems that use the application-specific message semantics alone to determine the order of message delivery [Cheriton and Skeen 1993, Pedone and Schiper 1999].

The example of the bulletin board • To make multicast delivery semantics more concrete, consider an application in which users post messages to bulletin boards. Each user runs a bulletin-board application process. Every topic of discussion has its own process group. When a user posts a message to a bulletin board, the application

Figure 15.12 Display from bulletin board program

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

multicasts the user's posting to the corresponding group. Each user's process is a member of the group for the topic in which that user is interested, so they will receive just the postings concerning that topic.

Reliable multicast is required if every user is to receive every posting eventually. The users also have ordering requirements. Figure 15.12 shows the postings as they appear to a particular user. At a minimum, FIFO ordering is desirable, since then every posting from a given user – 'A.Hanlon', say – will be received in the same order, and users can talk consistently about A.Hanlon's second posting.

Note that the messages whose subjects are 'Re: Microkernels' (25) and 'Re: Mach' (27) appear after the messages to which they refer. A causally ordered multicast is needed to guarantee this relationship. Otherwise, arbitrary message delays could mean that, say, the message 'Re: Mach' could appear before the original message about Mach.

If the multicast delivery was totally ordered, then the numbering in the lefthand column would be consistent between users. Users could refer unambiguously, for example, to 'message 24'.

In practice, the USENET bulletin board system implements neither causal nor total ordering. The communication costs of achieving these orderings on a large scale outweigh their advantages.

Implementing FIFO ordering • FIFO-ordered multicast (with operations *FO-multicast* and *FO-deliver*) is achieved with sequence numbers, much as we would achieve it for one-to-one communication. We shall consider only non-overlapping groups. The reader should verify that the reliable multicast protocol that we defined on top of IP multicast in Section 15.4.2 also guarantees FIFO ordering, but we shall show how to construct a FIFO-ordered multicast on top of any given basic multicast. We use the variables S_g^p and R_g^q held at process p from the reliable multicast protocol of Section 15.4.2: S_g^p is a count of how many messages p has sent to g and, for each q, R_g^q is the sequence number of the latest message p has delivered from process q that was sent to group g.

For p to *FO-multicast* a message to group g, it piggybacks the value S_g^p onto the message, *B-multicast*s the message to g and then increments S_g^p by 1. Upon receipt of a message from q bearing the sequence number S, p checks whether $S = R_g^q + 1$. If so, this message is the next one expected from the sender q and p *FO-deliver*s it, setting

R_g^q :=S. If $S > R_g^q + 1$, it places the message in the hold-back queue until the intervening messages have been delivered and $S = R_g^q + 1$.

Since all messages from a given sender are delivered in the same sequence, and since a message's delivery is delayed until its sequence number has been reached, the condition for FIFO ordering is clearly satisfied. But this is so only under the assumption that groups are non-overlapping.

Note that we can use any implementation of *B-multicast* in this protocol. Moreover, if we use a reliable *R-multicast* primitive instead of *B-multicast*, then we obtain a reliable FIFO multicast.

Implementing total ordering • The basic approach to implementing total ordering is to assign totally ordered identifiers to multicast messages so that each process makes the same ordering decision based upon these identifiers. The delivery algorithm is very similar to the one we described for FIFO ordering; the difference is that processes keep group-specific sequence numbers rather than process-specific sequence numbers. We only consider how to totally order messages sent to non-overlapping groups. We call the multicast operations *TO-multicast* and *TO-deliver*.

We discuss two main methods for assigning identifiers to messages. The first of these is for a process called a *sequencer* to assign them (Figure 15.13). A process wishing to *TO-multicast* a message m to group g attaches a unique identifier $id(m)$ to it. The messages for g are sent to the sequencer for g, *sequencer*(g), as well as to the members of g. (The sequencer may be chosen to be a member of g.) The process *sequencer*(g) maintains a group-specific sequence number s_g, which it uses to assign increasing and consecutive sequence numbers to the messages that it *B-deliver*s. It announces the sequence numbers by *B-multicast*ing *order* messages to g (see Figure 15.13 for the details).

A message will remain in the hold-back queue indefinitely until it can be *TO-deliver*ed according to the corresponding sequence number. Since the sequence numbers are well defined (by the sequencer), the criterion for total ordering is met. Furthermore, if the processes use a FIFO-ordered variant of *B-multicast*, then the totally ordered multicast is also causally ordered. We leave the reader to show this.

The obvious problem with a sequencer-based scheme is that the sequencer may become a bottleneck and is a critical point of failure. Practical algorithms exist that address the problem of failure. Chang and Maxemchuk [1984] first suggested a multicast protocol employing a sequencer (which they called a *token site*). Kaashoek *et al.* [1989] developed a sequencer-based protocol for the Amoeba system. These protocols ensure that a message is in the hold-back queue at $f + 1$ nodes before it is delivered; up to f failures can thus be tolerated. Like Chang and Maxemchuk, Birman *et al.* [1991] also employ a token-holding site that acts as a sequencer. The token can be passed from process to process so that, for example, if only one process sends totally ordered multicasts that process can act as the sequencer, saving communication.

The protocol of Kaashoek *et al.* uses hardware-based multicast – available on an Ethernet, for example – rather than reliable point-to-point communication. In the simplest variant of their protocol, processes send the message to be multicast to the sequencer, one-to-one. The sequencer multicasts the message itself, as well as the identifier and sequence number. This has the advantage that the other members of the

Figure 15.13 Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

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

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

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

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

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

group receive only one message per multicast; its disadvantage is increased bandwidth utilization. The protocol is described in full at www.cdk5.net/coordination.

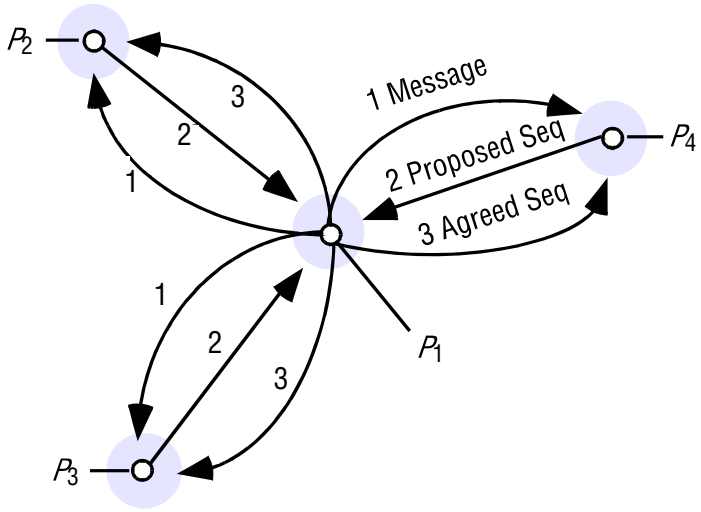
The second method that we examine for achieving totally ordered multicast is one in which the processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion. A simple algorithm – similar to one that was originally developed to implement totally ordered multicast delivery for the ISIS toolkit [Birman and Joseph 1987a] – is shown in Figure 15.14. Once more, a process *B-multicast*s its message to the members of the group. The group may be open or closed. The receiving processes propose sequence numbers for messages as they arrive and return these to the sender, which uses them to generate *agreed* sequence numbers.

Each process q in group g keeps A_g^q, the largest agreed sequence number it has observed so far for group g, and P_g^q, its own largest proposed sequence number. The algorithm for process p to multicast a message m to group g is as follows:

1. p *B-multicast*s <m, i> to g, where i is a unique identifier for m.

2. Each process q replies to the sender p with a proposal for the message's agreed sequence number of $P_g^q := Max(A_g^q, P_g^q) + 1$. In reality, we must include process identifiers in the proposed values P_g^q to ensure a total order, since otherwise different processes could propose the same integer value; but for the sake of simplicity we shall not make that explicit here. Each process provisionally assigns the proposed sequence number to the message and places it in its hold-back queue, which is ordered with the *smallest* sequence number at the front.

Figure 15.14 The ISIS algorithm for total ordering



3. *p* collects all the proposed sequence numbers and selects the largest one, *a,* as the next agreed sequence number. It then *B-multicast*s <*i, a*> to *g*. Each process *q* in *g* sets $A_g^q := Max(A_g^q, a)$ and attaches *a* to the message (which is identified by *i*). It reorders the message in the hold-back queue if the agreed sequence number differs from the proposed one. When the message at the front of the hold-back queue has been assigned its agreed sequence number, it is transferred to the tail of the delivery queue. Messages that have been assigned their agreed sequence number but are not at the head of the hold-back queue are not yet transferred, however.

If every process agrees the same set of sequence numbers and delivers them in the corresponding order, then total ordering is satisfied. It is clear that correct processes ultimately agree on the same set of sequence numbers, but we must show that they are monotonically increasing and that no correct process can deliver a message prematurely.

Assume that a message m_1 has been assigned an agreed sequence number and has reached the front of the hold-back queue. By construction, a message that is received after this stage will and should be delivered after m_1: it will have a larger proposed sequence number and thus a larger agreed sequence number than m_1. So let m_2 be any other message that has not yet been assigned its agreed sequence number but that is on the same queue. We have that:

$$agreedSequence(m_2) \geq proposedSequence(m_2)$$

by the algorithm just given. Since m_1 is at the front of the queue:

$$proposedSequence(m_2) > agreedSequence(m_1)$$

Therefore:

$$agreedSequence(m_2) > agreedSequence(m_1)$$

Figure 15.15 Causal ordering using vector timestamps

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

On initialization

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

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j $(j \neq i)$, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

and total ordering is assured.

This algorithm has higher latency than the sequencer-based multicast algorithm: three messages are sent serially between the sender and the group before a message can be delivered.

Note that the total ordering chosen by this algorithm is not also guaranteed to be causally or FIFO-ordered: any two messages are delivered in an essentially arbitrary total order, influenced by communication delays.

For other approaches to implementing total ordering, see Melliar-Smith *et al.* [1990], Garcia-Molina and Spauster [1991] and Hadzilacos and Toueg [1994].

Implementing causal ordering • Next we give an algorithm for non-overlapping closed groups based on that developed by Birman *et al.* [1991], shown in Figure 15.15, in which the causally ordered multicast operations are *CO-multicast* and *CO-deliver*. The algorithm takes account of the happened-before relationship only as it is established by *multicast* messages. If the processes send one-to-one messages to one another, then these will not be accounted for.

Each process p_i $(i = 1, 2, \ldots, N)$ maintains its own vector timestamp (see Section 14.4). The entries in the timestamp count the number of multicast messages from each process that happened-before the next message to be multicast.

To *CO-multicast* a message to group g, the process adds 1 to its entry in the timestamp and *B-multicast*s the message along with its timestamp to g.

When a process p_i *B-deliver*s a message from p_j, it must place it in the hold-back queue before it can *CO-deliver* it – that is, until it is assured that it has delivered any messages that causally preceded it. To establish this, p_i waits until (a) it has delivered any earlier message sent by p_j, and (b) it has delivered any message that p_j had delivered at the time it multicast the message. Both of those conditions can be detected by examining vector timestamps, as shown in Figure 15.15. Note that a process can immediately *CO-deliver* to itself any message that it *CO-multicast*s, although this is not described in Figure 15.15.

Each process updates its vector timestamp upon delivering any message, to maintain the count of causally precedent messages. It does this by incrementing the *j*th entry in its timestamp by one. This is an optimization of the *merge* operation that appears in the rules for updating vector clocks in Section 14.4. We can make the optimization in view of the delivery condition in the algorithm of Figure 15.15, which guarantees that only the *j*th entry will increase.

We outline the proof of the correctness of this algorithm as follows. Suppose that *multicast*(g, m) $\rightarrow$ *multicast*(g, m'). Let V and V' be the vector timestamps of m and m', respectively. It is straightforward to prove inductively from the algorithm that $V < V'$. In particular, if process p_k multicast m, then $V[k] \leq V'[k]$.

Consider what happens when some correct process p_i *B-deliver*s m' (as opposed to *CO-deliver*ing it) without first *CO-deliver*ing m. By the algorithm, $V_i[k]$ can increase only when p_i delivers a message from p_k, when it increases by 1. But p_i has not received m, and therefore $V_i[k]$ cannot increase beyond $V[k] - 1$. It is therefore not possible for p_i to *CO-deliver* m', since this would require that $V_i[k] \geq V'[k]$, and therefore that $V_i[k] \geq V[k]$.

The reader should check that if we substitute the reliable *R-multicast* primitive in place of *B-multicast*, then we obtain a multicast that is both reliable and causally ordered.

Furthermore, if we combine the protocol for causal multicast with the sequencer-based protocol for totally ordered delivery, then we obtain message delivery that is both total and causal. The sequencer delivers messages according to the causal order and multicasts the sequence numbers for the messages in the order in which it receives them. The processes in the destination group do not deliver a message until they have received an *order* message from the sequencer and the message is next in the delivery sequence.

Since the sequencer delivers messages in causal order, and since all other processes deliver messages in the same order as the sequencer, the ordering is indeed both total and causal.

Overlapping groups • We have considered only non-overlapping groups in the preceding definitions and algorithms for FIFO, total and causal ordering semantics. This simplifies the problem, but it is not satisfactory, since in general processes need to be members of multiple overlapping groups. For example, a process may be interested in events from multiple sources and thus join a corresponding set of event-distribution groups.

We can extend the ordering definitions to global orders [Hadzilacos and Toueg 1994], in which we have to consider that if message m is multicast to g, and if message m' is multicast to g', then both messages are addressed to the members of $g \cap g'$:

> *Global FIFO ordering*: If a correct process issues *multicast*(g, m) and then *multicast*(g', m'), then every correct process in $g \cap g'$ that delivers m' will deliver m before m'.
>
> *Global causal ordering*: If *multicast*(g, m) $\rightarrow$ *multicast*(g', m'), where $\rightarrow$ is the happened-before relation induced by any chain of multicast messages, then any correct process in $g \cap g'$ that delivers m' will deliver m before m'.

Pairwise total ordering: If a correct process delivers message m sent to g before it delivers m' sent to g', then any other correct process in $g \cap g'$ that delivers m' will deliver m before m'.

Global total ordering: Let '<' be the relation of ordering between delivery events. We require that '<' obeys pairwise total ordering and that it is acyclic – under pairwise total ordering, '<' is not acyclic by default.

One way of implementing these orders would be to multicast each message m to the group of *all* processes in the system. Each process either discards or delivers the message according to whether it belongs to *group*(m). This would be an inefficient and unsatisfactory implementation: a multicast should involve as few processes as possible beyond the members of the destination group. Alternatives are explored in Birman *et al.* [1991], Garcia-Molina and Spauster [1991], Hadzilacos and Toueg [1994], Kindberg [1995] and Rodrigues *et al.* [1998].

Multicast in synchronous and asynchronous systems • In this section, we have described algorithms for reliable unordered multicast, (reliable) FIFO-ordered multicast, (reliable) causally ordered multicast and totally ordered multicast. We also indicated how to achieve a multicast that is both totally and causally ordered. We leave the reader to devise an algorithm for a multicast primitive that guarantees both FIFO and total ordering. All the algorithms that we have described work correctly in asynchronous systems.

We did not, however, give an algorithm that guarantees both reliable and totally ordered delivery. Surprising though it may seem, while possible in a *synchronous* system, a protocol with these guarantees is impossible in an *asynchronous* distributed system – even one that has at worst suffered a single process crash failure. We return to this point in the next section.

15.5 Consensus and related problems

This section introduces the problem of consensus [Pease *et al.* 1980, Lamport *et al.* 1982] and the related problems of Byzantine generals and interactive consistency. We refer to these collectively as problems of *agreement*. Roughly speaking, the problem is for processes to agree on a value after one or more of the processes has proposed what that value should be.

For example, in Chapter 2 we described a situation in which two armies should decide consistently to attack the common enemy or retreat. Similarly, we may require that all the correct processes controlling a spaceship's engines should decide to either 'proceed' or 'abort' after each has proposed one action or the other, and in a transaction to transfer funds from one account to another the processes involved must consistently agree to perform the respective debit and credit. In mutual exclusion, the processes agree on which process can enter the critical section. In an election, the processes agree on which is the elected process. In totally ordered multicast, the processes agree on the order of message delivery.

Protocols exist that are tailored to these individual types of agreement. We described some of them above, and Chapters 16 and 17 examine transactions. But it is

useful for us to consider more general forms of agreement, in a search for common characteristics and solutions.

This section defines consensus more precisely and relates it to three related agreement problems: Byzantine generals, interactive consistency and totally ordered multicast. We go on to examine under what circumstances the problems can be solved, and to sketch some solutions. In particular, we discuss the well-known impossibility result of Fischer *et al.* [1985], which states that in an asynchronous system a collection of processes containing only one faulty process cannot be guaranteed to reach consensus. Finally, we consider how it is that practical algorithms exist despite the impossibility result.

15.5.1 System model and problem definitions

Our system model includes a collection of processes p_i ($i = 1, 2, \ldots, N$) communicating by message passing. An important requirement that applies in many practical situations is for consensus to be reached even in the presence of faults. We assume, as before, that communication is reliable but that processes may fail. In this section we consider Byzantine (arbitrary) process failures, as well as crash failures. We sometimes specify an assumption that up to some number f of the N processes are faulty – that is, they exhibit some specified types of fault; the remainder of the processes are correct.

If arbitrary failures can occur, then another factor in specifying our system is whether the processes digitally sign the messages that they send (see Section 11.4). If processes sign their messages, then a faulty process is limited in the harm it can do. Specifically, during an agreement algorithm it cannot make a false claim about the values that a correct process has sent to it. The relevance of message signing will become clearer when we discuss solutions to the Byzantine generals problem. By default, we assume that signing does not take place.

Definition of the consensus problem • To reach consensus, every process p_i begins in the *undecided* state and *proposes* a single value v_i, drawn from a set D ($i = 1, 2, \ldots, N$). The processes communicate with one another, exchanging values. Each process then sets the value of a *decision variable*, d_i. In doing so it enters the *decided* state, in which it may no longer change d_i ($i = 1, 2, \ldots, N$). Figure 15.16 shows three processes engaged in a consensus algorithm. Two processes propose 'proceed' and a third proposes 'abort' but then crashes. The two processes that remain correct each decide 'proceed'.

The requirements of a consensus algorithm are that the following conditions should hold for every execution of it:

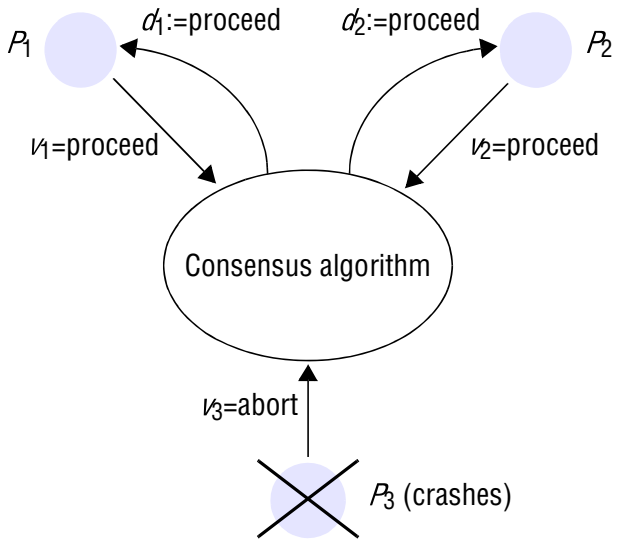
Termination: Eventually each correct process sets its decision variable.

Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the *decided* state, then $d_i = d_j$ ($i, j = 1, 2, \ldots, N$).

Integrity: If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value.

Variations on the definition of integrity may be appropriate, according to the application. For example, a weaker type of integrity would be for the decision value to

Figure 15.16 Consensus for three processes



equal a value that some correct process proposed – not necessarily all of them. We use the definition above except where stated otherwise. Integrity is also known as *validity* in the literature.

To help in understanding how the formulation of the problem translates into an algorithm, consider a system in which processes cannot fail. It is then straightforward to solve consensus. For example, we can collect the processes into a group and have each process reliably multicast its proposed value to the members of the group. Each process waits until it has collected all N values (including its own). It then evaluates the function $majority(v_1, v_2, \ldots, v_N)$, which returns the value that occurs most often among its arguments, or the special value $\perp \notin D$ if no majority exists. Termination is guaranteed by the reliability of the multicast operation. Agreement and integrity are guaranteed by the definition of *majority* and the integrity property of a reliable multicast. Every process receives the same set of proposed values, and every process evaluates the same function of those values. So they must all agree, and if every process proposed the same value, then they all decide on this value.

Note that *majority* is only one possible function that the processes could use to agree upon a value from the candidate values. For example, if the values are ordered, then the functions *minimum* and *maximum* may be appropriate.

If processes can crash this introduces the complication of detecting failures, and it is not immediately clear that a run of the consensus algorithm can terminate. In fact, if the system is asynchronous, then it may not; we shall return to this point shortly.

If processes can fail in *arbitrary* (Byzantine) ways, then faulty processes can in principle communicate random values to the others. This may seem unlikely in practice, but it is not beyond the bounds of possibility for a process with a bug to fail in this way. Moreover, the fault may not be accidental, but the result of mischievous or malevolent operation. Someone could deliberately make a process send different values to different peers in an attempt to thwart the others, which are trying to reach consensus. In case of inconsistency, correct processes must compare what they have received with what other processes claim to have received.

The Byzantine generals problem • In the informal statement of the *Byzantine generals problem* [Lamport *et al.* 1982], three or more generals are to agree to attack or to retreat. One, the commander, issues the order. The others, lieutenants to the commander, must decide whether to attack or retreat. But one or more of the generals may be 'treacherous' – that is, faulty. If the commander is treacherous, he proposes attacking to one general and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.

The Byzantine generals problem differs from consensus in that a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value. The requirements are:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the *decided* state, then $d_i = d_j$ $(i, j = 1, 2, \ldots, N)$.

Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.

Note that, for the Byzantine generals problem, integrity implies agreement when the commander is correct; but the commander need not be correct.

Interactive consistency • The interactive consistency problem is another variant of consensus, in which every process proposes a single value. The goal of the algorithm is for the correct processes to agree on a *vector* of values, one for each process. We call this the 'decision vector'. For example, the goal could be for each of a set of processes to obtain the same information about their respective states.

The requirements for interactive consistency are:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision vector of all correct processes is the same.

Integrity: If p_i is correct, then all correct processes decide on v_i as the ith component of their vector.

Relating consensus to other problems • Although it is common to consider the Byzantine generals problem with arbitrary process failures, in fact each of the three problems – consensus, Byzantine generals and interactive consistency – is meaningful in the context of either arbitrary or crash failures. Similarly, each can be framed assuming either a synchronous or an asynchronous system.

It is sometimes possible to derive a solution to one problem using a solution to another. This is a very useful property, both because it increases our understanding of the problems and because by reusing solutions we can potentially save on implementation effort and complexity.

Suppose that there exist solutions to consensus (C), Byzantine generals (BG) and interactive consistency (IC) as follows:

$C_i(v_1, v_2, \ldots, v_N)$ returns the decision value of p_i in a run of the solution to the consensus problem, where $v_1, v_2, \ldots, v_N$ are the values that the processes proposed.

$BG_i(j, v)$ returns the decision value of p_i in a run of the solution to the Byzantine generals problem, where p_j, the commander, proposes the value v.

$IC_i(v_1, v_2, \ldots, v_N)[j]$ returns the jth value in the decision vector of p_i in a run of the solution to the interactive consistency problem, where $v_1, v_2, \ldots, v_N$ are the values that the processes proposed.

The definitions of C_i, BG_i and IC_i assume that a faulty process proposes a single notional value, even though it may have given different proposed values to each of the other processes. This is only a convenience: the solutions will not rely on any such notional value.

It is possible to construct solutions out of the solutions to other problems. We give three examples:

IC from BG: We construct a solution to IC from BG by running BG N times, once with each process p_i $(i, j = 1, 2, \ldots, N)$ acting as the commander:

$$IC_i(v_1, v_2, \ldots, v_N)[j] = BG_i(j, v_j) \quad (i, j = 1, 2, \ldots, N)$$

C from IC: For the case where a majority of processes are correct, we construct a solution to C from IC by running IC to produce a vector of values at each process, then applying an appropriate function on the vector's values to derive a single value:

$$C_i(v_1, \ldots, v_N) = majority(IC_i(v_1, \ldots, v_N)[1], \ldots, IC_i(v_1, \ldots, v_N)[N])$$

where $i = 1, 2 \ldots N$ and *majority* is as defined above.

BG from C: We construct a solution to BG from C as follows:

- The commander p_j sends its proposed value v to itself and each of the remaining processes.
- All processes run C with the values $v_1, v_2, \ldots, v_N$ that they receive (p_j may be faulty).
- They derive $BG_i(j, v) = C_i(v_1, v_2, \ldots, v_N)$ $(i = 1, 2, \ldots, N)$.

The reader should check that the termination, agreement and integrity conditions are preserved in each case. Fischer [1983] relates the three problems in more detail.

In systems with crash failures, consensus is equivalent to solving reliable and totally ordered multicast: given a solution to one, we can solve the other. Implementing consensus with a reliable and totally ordered multicast operation *RTO-multicast* is straightforward. We collect all the processes into a group, g. To achieve consensus, each process p_i performs *RTO-multicast*(g, v_i). Then each process p_i chooses $d_i = m_i$, where m_i is the *first* value that p_i *RTO-deliver*s. The termination property follows from the reliability of the multicast. The agreement and integrity properties follow from the reliability and total ordering of multicast delivery. Chandra and Toueg [1996] demonstrate how reliable and totally ordered multicast can be derived from consensus.

15.5.2 Consensus in a synchronous system

This section describes an algorithm to solve consensus in a synchronous system, although it is based on a modified form of the integrity requirement. The algorithm uses only a basic multicast protocol. It assumes that up to f of the N processes exhibit crash failures.

Figure 15.17 Consensus in a synchronous system

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

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

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

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

To reach consensus, each correct process collects proposed values from the other processes. The algorithm proceeds in $f+1$ rounds, in each of which the correct processes *B-multicast* the values between themselves. At most f processes may crash, by assumption. At worst, all f crashes will occur during the rounds, but the algorithm guarantees that at the end of the rounds all the correct processes that have survived will be in a position to agree.

The algorithm, shown in Figure 15.17, is based on that by Dolev and Strong [1983] and its presentation by Attiya and Welch [1998]. Their modified form of the integrity requirement applies to the proposed values of all processes, not just the correct ones: if all processes, whether correct or not, proposed the same value, then any correct process in the *decided* state would chose that value. Given that the algorithm assumes crash failures at worst, the proposed values of correct and non-correct processes would not be expected to differ, at least not on the basis of failures. The revised form of integrity enables the convenient use of the *minimum* function to choose a decision value from those proposed.

The variable $Values_i^r$ holds the set of proposed values known to process p_i at the beginning of round r. Each process multicasts the set of values that it has not sent in previous rounds. It then takes delivery of similar multicast messages from other processes and records any new values. Although this is not shown in Figure 15.17, the duration of a round is limited by setting a timeout based on the maximum time for a correct process to multicast a message. After $f+1$ rounds, each process chooses the minimum value it has received as its decision value.

Termination is obvious from the fact that the system is synchronous. To check the correctness of the algorithm, we must show that each process arrives at the same set of values at the end of the final round. Agreement and integrity (in its modified form) will then follow, because the processes apply the *minimum* function to this set.

Assume, to the contrary, that two processes differ in their final set of values. Without loss of generality, some correct process p_i possesses a value v that another correct process p_j $(i \neq j)$ does not possess. The only explanation for p_i possessing a proposed value v at the end that p_j does not possess is that any third process, p_k, say, that managed to send v to p_i crashed before v could be delivered to p_j. In turn, any process sending v in the previous round must have crashed, to explain why p_k possesses v in that round but p_j did not receive it. Proceeding in this way, we have to posit at least one crash in each of the preceding rounds. But we have assumed that at most f crashes can occur, and there are $f+1$ rounds. We have arrived at a contradiction.

It turns out that *any* algorithm to reach consensus despite up to f crash failures requires at least $f+1$ rounds of message exchanges, no matter how it is constructed [Dolev and Strong 1983]. This lower bound also applies in the case of Byzantine failures [Fischer and Lynch 1982].

15.5.3 The Byzantine generals problem in a synchronous system

Now we discuss the Byzantine generals problem in a synchronous system. Unlike the algorithm for consensus described in the previous section, here we assume that processes can exhibit arbitrary failures. That is, a faulty process may send any message with any value at any time; and it may omit to send any message. Up to f of the N processes may be faulty. Correct processes can detect the absence of a message through a timeout; but they cannot conclude that the sender has crashed, since it may be silent for some time and then send messages again.

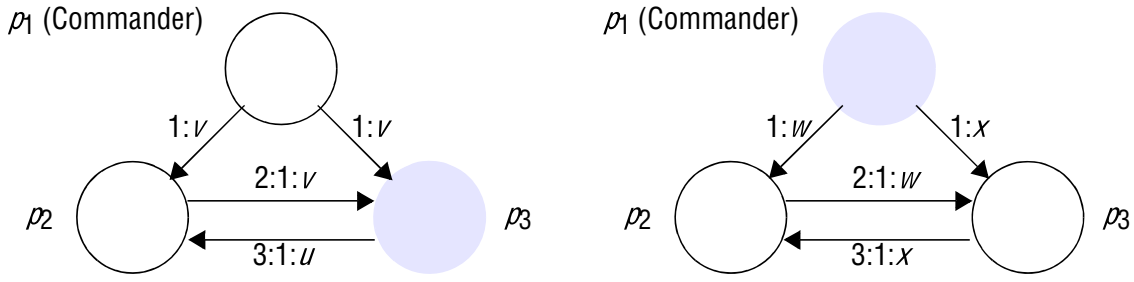
We assume that the communication channels between pairs of processes are private. If a process could examine all the messages that other processes sent, then it could detect the inconsistencies in what a faulty process sends to different processes. Our default assumption of channel reliability means that no faulty process can inject messages into the communication channel between correct processes.

Lamport *et al.* [1982] considered the case of three processes that send unsigned messages to one another. They showed that there is no solution that guarantees to meet the conditions of the Byzantine generals problem if one process is allowed to fail. They generalized this result to show that no solution exists if $N \leq 3f$. We shall demonstrate these results shortly. They went on to give an algorithm that solves the Byzantine generals problem in a synchronous system if $N \geq 3f+1$, for unsigned (they call them 'oral') messages.

Impossibility with three processes • Figure 15.18 shows two scenarios in which just one of three processes is faulty. In the lefthand configuration one of the lieutenants, p_3, is faulty; on the right the commander, p_1, is faulty. Each scenario in Figure 15.18 shows two rounds of messages: the values the commander sends, and the values that the lieutenants subsequently send to each other. The numeric prefixes serve to specify the sources of messages and to show the different rounds. Read the ':' symbol in messages as 'says'; for example, '3:1:u' is the message '3 says 1 says u'.

In the lefthand scenario, the commander correctly sends the same value v to each of the other two processes, and p_2 correctly echoes this to p_3. However, p_3 sends a value $u \neq v$ to p_2. All p_2 knows at this stage is that it has received differing values; it cannot tell which were sent out by the commander.

Figure 15.18 Three Byzantine generals



Faulty processes are shown in grey

In the righthand scenario, the commander is faulty and sends differing values to the lieutenants. After p_3 has correctly echoed the value x that it received, p_2 is in the same situation as it was in when p_3 was faulty: it has received two differing values.

If a solution exists, then process p_2 is bound to decide on value v when the commander is correct, by the integrity condition. If we accept that no algorithm can possibly distinguish between the two scenarios, p_2 must also choose the value sent by the commander in the righthand scenario.

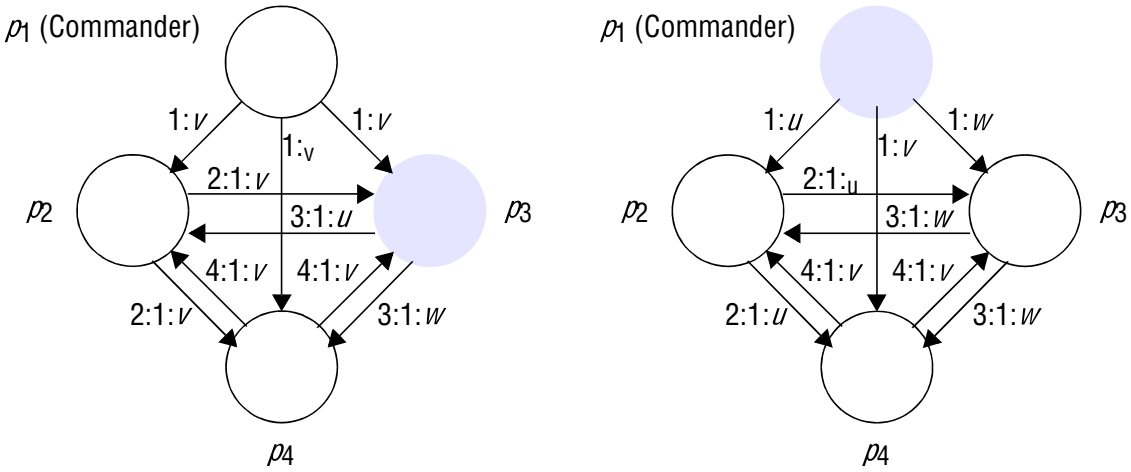
Following exactly the same reasoning for p_3, assuming that it is correct, we are forced to conclude (by symmetry) that p_3 also chooses the value sent by the commander as its decision value. But this contradicts the agreement condition (the commander sends differing values if it is faulty). So no solution is possible.

Note that this argument rests on our intuition that nothing can be done to improve a correct general's knowledge beyond the first stage, where it cannot tell which process is faulty. It is possible to prove the correctness of this intuition [Pease *et al.* 1980]. Byzantine agreement *can* be reached for three generals, with one of them faulty, if the generals digitally sign their messages.

Impossibility with N ≤ 3f • Pease *et al.* generalized the basic impossibility result for three processes, to prove that no solution is possible if $N \leq 3f$. In outline, the argument is as follows. Assume that a solution exists with $N \leq 3f$. Let each of three processes p_1, p_2 and p_3 use the solution to simulate the behaviour of n_1, n_2 and n_3 generals, respectively, where $n_1 + n_2 + n_3 = N$ and $n_1, n_2, n_3 \leq N/3$. Assume, furthermore, that one of the three processes is faulty. Those of p_1, p_2 and p_3 that are correct simulate correct generals: they simulate the interactions of their own generals internally and send messages from their generals to those simulated by other processes. The faulty process's simulated generals are faulty: the messages that it sends as part of the simulation to the other two processes may be spurious. Since $N \leq 3f$ and $n_1, n_2, n_3 \leq N/3$, at most f simulated generals are faulty.

Because the algorithm that the processes run is assumed to be correct, the simulation terminates. The correct simulated generals (in the two correct processes) agree and satisfy the integrity property. But now we have a means for the two correct processes out of the three to reach consensus: each decides on the value chosen by all of their simulated generals. This contradicts our impossibility result for three processes, with one faulty.

Figure 15.19 Four Byzantine generals



Faulty processes are shown in grey

Solution with one faulty process • There is not sufficient space to describe fully the algorithm of Pease *et al.* that solves the Byzantine generals problem in a synchronous system with $N \geq 3f + 1$. Instead, we give the operation of the algorithm for the case $N \geq 4$, $f = 1$ and illustrate it for $N = 4$, $f = 1$.

The correct generals reach agreement in two rounds of messages:

- In the first round, the commander sends a value to each of the lieutenants.
- In the second round, each of the lieutenants sends the value it received to its peers.

A lieutenant receives a value from the commander, plus $N-2$ values from its peers. If the commander is faulty, then all the lieutenants are correct and each will have gathered exactly the set of values that the commander sent out. Otherwise, one of the lieutenants is faulty; each of its correct peers receives $N-2$ copies of the value that the commander sent, plus a value that the faulty lieutenant sent to it.

In either case, the correct lieutenants need only apply a simple majority function to the set of values they receive. Since $N \geq 4$, $(N-2) \geq 2$. Therefore, the *majority* function will ignore any value that a faulty lieutenant sent, and it will produce the value that the commander sent if the commander is correct.

We now illustrate the algorithm that we have just outlined for the case of four generals. Figure 15.19 shows two scenarios similar to those in Figure 15.18, but in this case there are four processes, one of which is faulty. As in Figure 15.18, in the lefthand configuration one of the lieutenants, p_3, is faulty; on the right, the commander, p_1, is faulty.

In the lefthand case, the two correct lieutenant processes agree, deciding on the commander's value:

p_2 decides on $majority(v, u, v) = v$

p_4 decides on $majority(v, v, w) = v$

In the righthand case the commander is faulty, but the three correct processes agree:

p_2, p_3 and p_4 decide on $majority(u, v, w) = \perp$ (the special value $\perp$ applies where no majority of values exists).

The algorithm takes account of the fact that a faulty process may omit to send a message. If a correct process does not receive a message within a suitable time limit (the system is synchronous), it proceeds as though the faulty process had sent it the value $\perp$.

Discussion • We can measure the efficiency of a solution to the Byzantine generals problem – or any other agreement problem – by asking:

- How many message rounds does it take? (This is a factor in how long it takes for the algorithm to terminate.)
- How many messages are sent, and of what size? (This measures the total bandwidth utilization and has an impact on the execution time.)

In the general case ($f \geq 1$) the Lamport *et al.* [1982] algorithm for unsigned messages operates over $f+1$ rounds. In each round, a process sends to a subset of the other processes the values that it received in the previous round. The algorithm is very costly: it involves sending $O(N^{f+1})$ messages.

Fischer and Lynch [1982] proved that any deterministic solution to consensus assuming Byzantine failures (and hence to the Byzantine generals problem, as Section 15.5.1 showed) will take at least $f+1$ message rounds. So no algorithm can operate faster in this respect than that of Lamport *et al*. But there have been improvements in the message complexity, for example Garay and Moses [1993].

Several algorithms, such as that of Dolev and Strong [1983], take advantage of signed messages. Dolev and Strong's algorithm again takes $f+1$ rounds, but the number of messages sent is only $O(N^2)$.

The complexity and cost of the solutions suggest that they are applicable only where the threat is great. Solutions that are based on more detailed knowledge of the fault model may be more efficient [Barborak *et al.* 1993]. If malicious users are the source of the threat, then a system to counter them is likely to use digital signatures; a solution without signatures is impractical.

15.5.4 Impossibility in asynchronous systems

We have provided solutions to consensus and the Byzantine generals problem (and hence, by derivation, to interactive consistency). However, all these solutions relied upon the system being synchronous. The algorithms assume that message exchanges take place in rounds, and that processes are entitled to time out and assume that a faulty process has not sent them a message within the round, because the maximum delay has been exceeded.

Fischer *et al.* [1985] proved that no algorithm can guarantee to reach consensus in an asynchronous system, even with one process crash failure. In an asynchronous system, processes can respond to messages at arbitrary times, so a crashed process is indistinguishable from a slow one. Their proof, which is beyond the scope of this book, involves showing that there is always some continuation of the processes' execution that avoids consensus being reached.

We immediately know from the result of Fischer *et al.* that there is no guaranteed solution in an asynchronous system to the Byzantine generals problem, to interactive consistency or to totally ordered and reliable multicast. If there were such a solution

then, by the results of Section 15.5.1, we would have a solution to consensus – contradicting the impossibility result.

Note the word 'guarantee' in the statement of the impossibility result. The result does not mean that processes can *never* reach distributed consensus in an asynchronous system if one is faulty. It allows that consensus can be reached with some probability greater than zero, confirming what we know in practice. For example, despite the fact that our systems are often effectively asynchronous, transaction systems have been reaching consensus regularly for many years.

One approach to working around the impossibility result is to consider *partially synchronous* systems, which are sufficiently weaker than synchronous systems to be useful as models of practical systems, and sufficiently stronger than asynchronous systems for consensus to be solvable in them [Dwork *et al.* 1988]. That approach is beyond the scope of this book. However, we shall now outline three other techniques for working around the impossibility result: fault masking, and reaching consensus by exploiting failure detectors and by randomizing aspects of the processes' behaviour.

Masking faults • The first technique is to avoid the impossibility result altogether by masking any process failures that occur (see Section 2.4.2 for an introduction to fault masking). For example, transaction systems employ persistent storage, which survives crash failures. If a process crashes, then it is restarted (automatically, or by an administrator). The process places sufficient information in persistent storage at critical points in its program so that if it should crash and be restarted, it will find sufficient data to be able to continue correctly with its interrupted task. In other words, it will behave like a process that is correct, but that sometimes takes a long time to perform a processing step.

Of course, fault masking is generally applicable in system design. Chapter 16 discusses how transactional systems take advantage of persistent storage. Chapter 18 describes how process failures can also be masked by replicating software components.

Consensus using failure detectors • Another method for circumventing the impossibility result is to employ failure detectors. Some practical systems employ 'perfect by design' failure detectors to reach consensus. No failure detector in an asynchronous system that works solely by message passing can really be perfect. However, processes can agree to *deem* a process that has not responded for more than a bounded time to have failed. An unresponsive process may not really have failed, but the remaining processes act as if it had done. They make the failure 'fail-silent' by discarding any subsequent messages that they do in fact receive from a 'failed' process. In other words, we have effectively turned an asynchronous system into a synchronous one. This technique is used in the ISIS system [Birman 1993].

This method relies upon the failure detector usually being accurate. When it is inaccurate, then the system has to proceed without a group member that otherwise could potentially have contributed to the system's effectiveness. Unfortunately, making the failure detector reasonably accurate involves using long timeout values, forcing processes to wait a relatively long time (and not perform useful work) before concluding that a process has failed. Another issue that arises for this approach is network partitioning, which we discuss in Chapter 18.

A quite different approach is to use imperfect failure detectors, and to reach consensus while allowing suspected processes to behave correctly instead of excluding

them. Chandra and Toueg [1996] analyzed the properties that a failure detector must have in order to solve the consensus problem in an asynchronous system. They showed that consensus can be solved in an asynchronous system, even with an unreliable failure detector, if fewer than $N/2$ processes crash and communication is reliable. The weakest type of failure detector for which this is so is called an *eventually weak failure detector*. This is one that is both:

Eventually weakly complete: Each faulty process is eventually suspected permanently by some correct process.

Eventually weakly accurate: After some point in time, at least one correct process is never suspected by any correct process.

Chandra and Toueg show that we cannot implement an eventually weak failure detector in an asynchronous system by message passing alone. However, we described a message-based failure detector in Section 15.1 that adapts its timeout values according to observed response times. If a process or the connection to it is very slow, then the timeout value will grow so that cases of falsely suspecting a process become rare. In the case of many real systems, this algorithm behaves sufficiently closely to an eventually weak failure detector for practical purposes.

Chandra and Toueg's consensus algorithm allows falsely suspected processes to continue their normal operations and allows processes that have suspected them to receive messages from them and process those messages normally. This makes the application programmer's life complicated, but it has the advantage that correct processes are not wasted by being falsely excluded. Moreover, timeouts for detecting failures can be set less conservatively than with the ISIS approach.

Consensus using randomization • The result of Fischer *et al.* [1985] depends on what we can consider to be an 'adversary'. This is a 'character' (actually, just a collection of random events) who can exploit the phenomena of asynchronous systems so as to foil the processes' attempts to reach consensus. The adversary manipulates the network to delay messages so that they arrive at just the wrong time, and similarly it slows down or speeds up the processes just enough so that they are in the 'wrong' state when they receive a message.

The third technique that addresses the impossibility result is to introduce an element of chance in the processes' behaviour, so that the adversary cannot exercise its thwarting strategy effectively. Consensus might still not be reached in some cases, but this method enables processes to reach consensus in a finite *expected* time. A probabilistic algorithm that solves consensus even with Byzantine failures can be found in Canetti and Rabin [1993].

15.6 Summary

We began this chapter by discussing the need for processes to access shared resources under conditions of mutual exclusion. Locks are not always implemented by the servers that manage the shared resources, and a separate distributed mutual exclusion service is then required. Three algorithms were considered that achieve mutual exclusion: one employing a central server, a ring-based algorithm and a multicast-based algorithm using logical clocks. None of these mechanisms can withstand failure as we described them, although they can be modified to tolerate some faults.

Next we explored elections, considering a ring-based algorithm and the bully algorithm, whose common aim is to elect a process uniquely from a given set – even if several elections take place concurrently. The bully algorithm could be used, for example, to elect a new master time server, or a new lock server, when the previous one fails.

The following section described coordination and agreement in group communication. It discussed reliable multicast, in which the correct processes agree on the set of messages to be delivered, and multicast with FIFO, causal and total delivery ordering. We gave algorithms for reliable multicast and for all three types of delivery ordering.

Finally, we described the three problems of consensus, Byzantine generals and interactive consistency. We defined the conditions for their solution and we showed relationships between these problems – including the relationship between consensus and reliable, totally ordered multicast.

Solutions exist in a synchronous system, and we described some of them. In fact, solutions exist even when arbitrary failures are possible.We outlined part of the solution to the Byzantine generals problem of Lamport *et al.* [1982]. More recent algorithms have lower complexity, but in principle none can better the $f + 1$ rounds taken by this algorithm, unless messages are digitally signed.

The chapter ended by describing the fundamental result of Fischer *et al.* [1982] concerning the impossibility of guaranteeing consensus in an asynchronous system. We discussed how it is that, nonetheless, systems regularly do reach agreement in asynchronous systems.

EXERCISES

15.1 Is it possible to implement either a reliable or an unreliable (process) failure detector using an unreliable communication channel? *page 632*

15.2 If all client processes are single-threaded, is mutual exclusion condition ME3, which specifies entry in happened-before order, relevant? *page 635*

15.3 Give a formula for the maximum throughput of a mutual exclusion system in terms of the synchronization delay. *page 635*

15.4 In the central server algorithm for mutual exclusion, describe a situation in which two requests are not processed in happened-before order. *page 636*

15.5 Adapt the central server algorithm for mutual exclusion to handle the crash failure of any client (in any state), assuming that the server is correct and given a reliable failure detector. Comment on whether the resultant system is fault-tolerant. What would happen if a client that possesses the token is wrongly suspected to have failed? *page 636*

15.6 Give an example execution of the ring-based algorithm to show that processes are not necessarily granted entry to the critical section in happened-before order. *page 637*

15.7 In a certain system, each process typically uses a critical section many times before another process requires it. Explain why Ricart and Agrawala's multicast-based mutual exclusion algorithm is inefficient for this case, and describe how to improve its performance. Does your adaptation satisfy liveness condition ME2? *page 639*

15.8 In the bully algorithm, a recovering process starts an election and will become the new coordinator if it has a higher identifier than the current incumbent. Is this a necessary feature of the algorithm? *page 644*

15.9 Suggest how to adapt the bully algorithm to deal with temporary network partitions (slow communication) and slow processes. *page 646*

15.10 Devise a protocol for basic multicast over IP multicast. *page 647*

15.11 How, if at all, should the definitions of integrity, agreement and validity for reliable multicast change for the case of open groups? *page 647*

15.12 Explain why reversing the order of the lines '*R-deliver m*' and '*if* ($q \neq p$) *then B-multicast*(*g, m*); *end if*' in Figure 15.9 makes the algorithm no longer satisfy uniform agreement. Does the reliable multicast algorithm based on IP multicast satisfy uniform agreement? *page 648*

15.13 Explain whether the algorithm for reliable multicast over IP multicast works for open as well as closed groups. Given any algorithm for closed groups, how, simply, can we derive an algorithm for open groups? *page 649*

15.14 Explain how to adapt the algorithm for reliable multicast over IP multicast to eliminate the hold-back queue – so that a received message that is not a duplicate can be delivered immediately, but without any ordering guarantees. Hint: use sets of sequence numbers to represent the messages that have been delivered so far. *page 649*

15.15 Consider how to address the impractical assumptions we made in order to meet the validity and agreement properties for the reliable multicast protocol based on IP multicast. Hint: add a rule for deleting retained messages when they have been delivered everywhere, and consider adding a dummy 'heartbeat' message, which is never delivered to the application, but which the protocol sends if the application has no message to send. *page 649*

15.16 Show that the FIFO-ordered multicast algorithm does not work for overlapping groups, by considering two messages sent from the same source to two overlapping groups, and considering a process in the intersection of those groups. Adapt the protocol to work for this case. Hint: processes should include with their messages the latest sequence numbers of messages sent to *all* groups. *page 654*

15.17 Show that, if the basic multicast that we use in the algorithm of Figure 15.13 is also FIFO-ordered, then the resultant totally-ordered multicast is also causally ordered. Is it the case that any multicast that is both FIFO-ordered and totally ordered is thereby causally ordered? *page 655*

15.18 Suggest how to adapt the causally ordered multicast protocol to handle overlapping groups. *page 657*

15.19 In discussing Maekawa's mutual exclusion algorithm, we gave an example of three subsets of a set of three processes that could lead to a deadlock. Use these subsets as multicast groups to show how a pairwise total ordering is not necessarily acyclic. *page 658*

15.20 Construct a solution to reliable, totally ordered multicast in a synchronous system, using a reliable multicast and a solution to the consensus problem. *page 659*

15.21 We gave a solution to consensus from a solution to reliable and totally ordered multicast, which involved selecting the first value to be delivered. Explain from first principles why, in an asynchronous system, we could not instead derive a solution by using a reliable but not totally ordered multicast service and the 'majority' function. (Note that, if we could, this would contradict the impossibility result of Fischer *et al*. [1985]!) Hint: consider slow/failed processes. *page 663*

15.22 Consider the algorithm given in Figure 15.17 for consensus in a synchronous system, which uses the following integrity definition: if all processes, whether correct or not, proposed the same value, then any correct process in the decided state would chose that value. Now consider an application in which correct processes may propose different results, e.g., by running different algorithms to decide which action to take in a control system's operation. Suggest an appropriate modification to the integrity definition and thus to the algorithm. *page 664*

15.23 Show that Byzantine agreement can be reached for three generals, with one of them faulty, if the generals digitally sign their messages. *page 665*

This page intentionally left blank

16

TRANSACTIONS AND CONCURRENCY CONTROL

This chapter discusses the application of transactions and concurrency control to shared objects managed by servers.

A transaction defines a sequence of server operations that is guaranteed by the server to be atomic in the presence of multiple clients and server crashes. Nested transactions are structured from sets of other transactions. They are particularly useful in distributed systems because they allow additional concurrency.

All of the concurrency control protocols are based on the criterion of serial equivalence and are derived from rules for conflicts between operations. Three methods are described:

- Locks are used to order transactions that access the same objects according to the order of arrival of their operations at the objects.
- Optimistic concurrency control allows transactions to proceed until they are ready to commit, whereupon a check is made to see whether they have performed conflicting operations on objects.
- Timestamp ordering uses timestamps to order transactions that access the same objects according to their starting times.

16.1 Introduction

The goal of transactions is to ensure that all of the objects managed by a server remain in a consistent state when they are accessed by multiple transactions and in the presence of server crashes. Chapter 2 introduced a failure model for distributed systems. Transactions deal with crash failures of processes and omission failures in communication, but not any type of arbitrary (or Byzantine) behaviour. The failure model for transactions is presented in Section 16.1.2.

Objects that can be recovered after their server crashes are called *recoverable* objects. In general, the objects managed by a server may be stored in volatile memory (for example, RAM) or persistent memory (for example, a hard disk). Even if objects are stored in volatile memory, the server may use persistent memory to store sufficient information for the state of the objects to be recovered if the server process crashes. This enables servers to make objects recoverable. A transaction is specified by a client as a set of operations on objects to be performed as an indivisible unit by the servers managing those objects. The servers must guarantee that either the entire transaction is carried out and the results recorded in permanent storage or, in the case that one or more of them crashes, its effects are completely erased. The next chapter discusses issues related to transactions that involve several servers, in particular how they decide on the outcome of a distributed transaction. This chapter concentrates on the issues for a transaction at a single server. A client's transaction is also regarded as indivisible from the point of view of other clients' transactions in the sense that the operations of one transaction cannot observe the partial effects of the operations of another. Section 16.1.1 discusses simple synchronization of access to objects, and Section 16.2 introduces transactions, which require more advanced techniques to prevent interference between clients. Section 16.3 discusses nested transactions. Sections 16.4 to 16.6 discuss three methods of concurrency control for transactions whose operations are all addressed to a single server (locks, optimistic concurrency control and timestamp ordering). Chapter 17 discusses how these methods are extended for use with transactions whose operations are addressed to several servers.

To explain some of the points made in this chapter, we use a banking example, shown in Figure 16.1. Each account is represented by a remote object whose interface, *Account*, provides operations for making deposits and withdrawals and for enquiring about and setting the balance. Each branch of the bank is represented by a remote object whose interface, *Branch*, provides operations for creating a new account, for looking up an account by name and for enquiring about the total funds at that branch.

16.1.1 Simple synchronization (without transactions)

One of the main issues of this chapter is that unless a server is carefully designed, its operations performed on behalf of different clients may sometimes interfere with one another. Such interference may result in incorrect values in the objects. In this section, we discuss how client operations may be synchronized without recourse to transactions.

Atomic operations at the server • We have seen in earlier chapters that the use of multiple threads is beneficial to performance in many servers. We have also noted that the use of threads allows operations from multiple clients to run concurrently and

Figure 16.1 Operations of the *Account* interface

deposit(amount)
deposit *amount* in the account

withdraw(amount)
withdraw *amount* from the account

getBalance()→ amount
return the balance of the account

setBalance(amount)
set the balance of the account to *amount*

Operations of the *Branch* interface

create(name)→ account
create a new account with a given name

lookUp(name)→ account
return a reference to the account with the given name

branchTotal()→ amount
return the total of all the balances at the branch

possibly access the same objects. Therefore, the methods of objects should be designed for use in a multi-threaded context. For example, if the methods *deposit* and *withdraw* are not designed for use in a multi-threaded program, then it is possible that the actions of two or more concurrent executions of the method could be interleaved arbitrarily and have strange effects on the instance variables of the account objects.

Chapter 7 explains the use of the *synchronized* keyword, which can be applied to methods in Java to ensure that only one thread at a time can access an object. In our example, the class that implements the *Account* interface will be able to declare the methods as synchronized. For example:

```
public synchronized void deposit(int amount) throws RemoteException{
    // adds amount to the balance of the account
}
```

If one thread invokes a synchronized method on an object, then that object is effectively locked, and another thread that invokes one of its synchronized methods will be blocked until the lock is released. This form of synchronization forces the execution of threads to be separated in time and ensures that the instance variables of a single object are accessed in a consistent manner. Without synchronization, two separate *deposit* invocations might read the balance before either has incremented it – resulting in an incorrect value. Any method that accesses an instance variable that can vary should be synchronized.

Operations that are free from interference from concurrent operations being performed in other threads are called *atomic operations*. The use of synchronized

methods in Java is one way of achieving atomic operations. But in other programming environments for multi-threaded servers the operations on objects still need to have atomic operations in order to keep their objects consistent. This may be achieved by the use of any available mutual exclusion mechanism, such as a mutex.

Enhancing client cooperation by synchronization of server operations • Clients may use a server as a means of sharing some resources. This is achieved by some clients using operations to update the server's objects and other clients using operations to access them. The above scheme for synchronized access to objects provides all that is required in many applications – it prevents threads interfering with one another. However, some applications require a way for threads to communicate with each other.

For example, a situation may arise in which the operation requested by one client cannot be completed until an operation requested by another client has been performed. This can happen when some clients are producers and others are consumers – the consumers may have to wait until a producer has supplied some more of the commodity in question. It can also occur when clients are sharing a resource – clients needing the resource may have to wait for other clients to release it. We shall see later in this chapter that a similar situation arises when locks or timestamps are used for concurrency control in transactions.

The Java *wait* and *notify* methods introduced in Chapter 7 allow threads to communicate with one another in a manner that solves the above problems. They must be used within synchronized methods of an object. A thread calls *wait* on an object so as to suspend itself and to allow another thread to execute a method of that object. A thread calls *notify* to inform any thread waiting on that object that it has changed some of its data. Access to an object is still atomic when threads wait for one another: a thread that calls *wait* gives up its lock and suspends itself as a single atomic action; when a thread is restarted after being notified it acquires a new lock on the object and resumes execution from after its *wait*. A thread that calls *notify* (from within a synchronized method) completes the execution of that method before releasing the lock on the object.

Consider the implementation of a shared *Queue* object with two methods: *first* removes and returns the first object in the queue, and *append* adds a given object to the end of the queue. The method *first* will test whether the queue is empty, in which case it will call *wait* on the queue. If a client invokes *first* when the queue is empty, it will not get a reply until another client has added something to the queue – the *append* operation will call *notify* when it has added an object to the queue. This allows one of the threads waiting on the queue object to resume and to return the first object in the queue to its client. When threads can synchronize their actions on an object by means of *wait* and *notify*, the server holds onto requests that cannot immediately be satisfied and the client waits for a reply until another client has produced whatever it needs.

In Section 16.4, we discuss the implementation of a lock as an object with synchronized operations. When clients attempt to acquire a lock, they can be made to wait until the lock is released by other clients.

Without the ability to synchronize threads in this way, a client that cannot be satisfied immediately – for example, a client that invokes the *first* method on an empty queue – is told to try again later. This is unsatisfactory, because it will involve the client in polling the server and the server in carrying out extra requests. It is also potentially unfair because other clients may make their requests before the waiting client tries again.

16.1.2 Failure model for transactions

Lampson [1981] proposed a fault model for distributed transactions that accounts for failures of disks, servers and communication. In this model, the claim is that the algorithms work correctly in the presence of predictable faults, but no claims are made about their behaviour when a disaster occurs. Although errors may occur, they can be detected and dealt with before any incorrect behaviour results. The model states the following:

- Writes to permanent storage may fail, either by writing nothing or by writing a wrong value – for example, writing to the wrong block is a disaster. File storage may also decay. Reads from permanent storage can detect (by a checksum) when a block of data is bad.
- Servers may crash occasionally. When a crashed server is replaced by a new process, its volatile memory is first set to a state in which it knows none of the values (for example, of objects) from before the crash. After that it carries out a recovery procedure using information in permanent storage and obtained from other processes to set the values of objects including those related to the two-phase commit protocol (see Section 17.6). When a processor is faulty, it is made to crash so that it is prevented from sending erroneous messages and from writing wrong values to permanent storage – that is, so it cannot produce arbitrary failures. Crashes can occur at any time; in particular, they may occur during recovery.
- There may be an arbitrary delay before a message arrives. A message may be lost, duplicated or corrupted. The recipient can detect corrupted messages using a checksum. Both forged messages and undetected corrupt messages are regarded as disasters.

The fault model for permanent storage, processors and communications was used to design a stable system whose components can survive any single fault and present a simple failure model. In particular, *stable storage* provided an atomic *write* operation in the presence of a single fault of the *write* operation or a crash failure of the process. This was achieved by replicating each block on two disk blocks. A *write* operation was applied to the pair of disk blocks, and in the case of a single fault, one good block was always available. A *stable processor* used stable storage to enable it to recover its objects after a crash. Communication errors were masked by using a reliable remote procedure calling mechanism.

16.2 Transactions

In some situations, clients require a sequence of separate requests to a server to be atomic in the sense that:

1. They are free from interference by operations being performed on behalf of other concurrent clients.
2. Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

Figure 16.2 A client's banking transaction

```
Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);
```

We return to our banking example to illustrate transactions. A client that performs a sequence of operations on a particular bank account on behalf of a user will first *lookUp* the account by name and then apply the *deposit*, *withdraw* and *getBalance* operations directly to the relevant account. In our examples, we use accounts with names *A*, *B* and *C*. The client looks them up and stores references to them in variables *a*, *b* and *c* of type *Account*. The details of looking up the accounts by name and the declarations of the variables are omitted from the examples.

Figure 16.2 shows an example of a simple client transaction specifying a series of related actions involving the bank accounts *A*, *B* and *C*. The first two actions transfer $100 from *A* to *B* and the second two transfer $200 from *C* to *B*. A client achieves a transfer operation by doing a withdrawal followed by a deposit.

Transactions originate from database management systems. In that context, a transaction is an execution of a program that accesses a database. Transactions were introduced to distributed systems in the form of transactional file servers such as XDFS [Mitchell and Dion 1982]. In the context of a transactional file server, a transaction is an execution of a sequence of client requests for file operations. Transactions on distributed objects were provided in several research systems, including Argus [Liskov 1988] and Arjuna [Shrivastava *et al*. 1991]. In this last context, a transaction consists of the execution of a sequence of client requests such as, for example, those in Figure 16.2. From the client's point of view, a transaction is a sequence of operations that forms a single step, transforming the server data from one consistent state to another.

Transactions can be provided as a part of middleware. For example, CORBA provides the specification for an Object Transaction Service [OMG 2003] with IDL interfaces allowing clients' transactions to include multiple objects at multiple servers. The client is provided with operations to specify the beginning and end of a transaction. The client maintains a context for each transaction, which it propagates with each operation in that transaction. In CORBA, transactional objects are invoked within the scope of a transaction and generally have some persistent store associated with them.

In all of these contexts, a transaction applies to recoverable objects and is intended to be atomic. It is often called an *atomic transaction*. There are two aspects to atomicity:

All or nothing: A transaction either completes successfully, in which case the effects of all of its operations are recorded in the objects, or (if it fails or is deliberately aborted) has no effect at all. This all-or-nothing effect has two further aspects of its own:

Failure atomicity: The effects are atomic even when the server crashes.

Durability: After a transaction has completed successfully, all its effects are saved in permanent storage. We use the term 'permanent storage' to refer to files held on disk or another permanent medium. Data saved in a file will survive if the server process crashes.

Isolation: Each transaction must be performed without interference from other transactions; in other words, the intermediate effects of a transaction must not be visible to other transactions. The box below introduces a mnemonic, ACID, for remembering the properties of atomic transactions.

To support the requirement for failure atomicity and durability, the objects must be *recoverable*; that is, when a server process crashes unexpectedly due to a hardware fault or a software error, the changes due to all completed transactions must be available in permanent storage so that when the server is replaced by a new process, it can recover the objects to reflect the all-or-nothing effect. By the time a server acknowledges the completion of a client's transaction, all of the transaction's changes to the objects must have been recorded in permanent storage.

A server that supports transactions must synchronize the operations sufficiently to ensure that the isolation requirement is met. One way of doing this is to perform the transactions serially – one at a time, in some arbitrary order. Unfortunately, this solution would generally be unacceptable for servers whose resources are shared by multiple interactive users. For instance, in our banking example it is desirable to allow several bank clerks to perform online banking transactions at the same time as one another.

The aim for any server that supports transactions is to maximize concurrency. Therefore transactions are allowed to execute concurrently if this would have the same effect as a serial execution – that is, if they are *serially equivalent* or *serializable*.

ACID properties

Härder and Reuter [1983] suggested the mnemonic 'ACID' to remember the properties of transactions, which are as follows:

Atomicity: a transaction must be all or nothing;

Consistency: a transaction takes the system from one consistent state to another consistent state;

Isolation;

Durability.

We have not included 'consistency' in our list of the properties of transactions because it is generally the responsibility of the programmers of servers and clients to ensure that transactions leave the database consistent.

As an example of consistency, suppose that in the banking example, an object holds the sum of all the account balances and its value is used as the result of *branchTotal*. Clients can get the sum of all the account balances either by using *branchTotal* or by calling *getBalance* on each of the accounts. For consistency, they should get the same result from both methods. To maintain this consistency, the *deposit* and *withdraw* operations must update the object holding the sum of all the account balances.

Figure 16.3 Operations in the *Coordinator* interface

openTransaction( ) → trans;
Starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans)→ (commit, abort);
Ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);
Aborts the transaction.

Transaction capabilities can be added to servers of recoverable objects. Each transaction is created and managed by a coordinator, which implements the *Coordinator* interface shown in Figure 16.3. The coordinator gives each transaction an identifier, or TID. The client invokes the *openTransaction* method of the coordinator to introduce a new transaction – a transaction identifier or TID is allocated and returned. At the end of a transaction, the client invokes the *closeTransaction* method to indicate its end – all of the recoverable objects accessed by the transaction should be saved. If, for some reason, the client wants to abort a transaction, it invokes the *abortTransaction* method – all of its effects should be removed from sight.

A transaction is achieved by cooperation between a client program, some recoverable objects and a coordinator. The client specifies the sequence of invocations on recoverable objects that are to comprise a transaction. To achieve this, the client sends with each invocation the transaction identifier returned by *openTransaction*. One way to make this possible is to include an extra argument in each operation of a recoverable object to carry the TID. For example, in the banking service the *deposit* operation might be defined:

deposit(trans, amount)
Deposits *amount* in the account for transaction with TID *trans*

When transactions are provided as middleware, the TID can be passed implicitly with all remote invocations between *openTransaction* and *closeTransaction* or *abortTransaction*. This is what the CORBA Transaction Service does. We shall not show TIDs in our examples.

Normally, a transaction completes when the client makes a *closeTransaction* request. If the transaction has progressed normally, the reply states that the transaction is *committed* – this constitutes a promise to the client that all of the changes requested in the transaction are permanently recorded and that any future transactions that access the same data will see the results of all of the changes made during the transaction.

Alternatively, the transaction may have to *abort* for one of several reasons related to the nature of the transaction itself, to conflicts with another transaction or to the crashing of a process or computer. When a transaction is aborted the parties involved (the recoverable objects and the coordinator) must ensure that none of its effects are visible to future transactions, either in the objects or in their copies in permanent storage.

A transaction is either successful or is aborted in one of two ways – the client aborts it (using an *abortTransaction* call to the server) or the server aborts it. Figure 16.4

Figure 16.4 Transaction life histories

Successful	*Aborted by client*	*Aborted by server*	
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•	server aborts	•
•	•	*transaction* →	•
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

shows these three alternative life histories for transactions. We refer to a transaction as *failing* in both of the latter cases.

Service actions related to process crashes • If a server process crashes unexpectedly, it is eventually replaced. The new server process aborts any uncommitted transactions and uses a recovery procedure to restore the values of the objects to the values produced by the most recently committed transaction. To deal with a client that crashes unexpectedly during a transaction, servers can give each transaction an expiry time and abort any transaction that has not completed before its expiry time.

Client actions related to server process crashes • If a server crashes while a transaction is in progress, the client will become aware of this when one of the operations returns an exception after a timeout. If a server crashes and is then replaced during the progress of a transaction, the transaction will no longer be valid and the client must be informed via an exception to the next operation. In either case, the client must then formulate a plan, possibly in consultation with the human user, for the completion or abandonment of the task of which the transaction was a part.

16.2.1 Concurrency control

This section illustrates two well-known problems of concurrent transactions in the context of the banking example – the 'lost update' problem and the 'inconsistent retrievals' problem. We then show how both of these problems can be avoided by using serially equivalent executions of transactions. We assume throughout that each of the operations *deposit*, *withdraw*, *getBalance* and *setBalance* is a synchronized operation – that is, that its effects on the instance variable that records the balance of an account are atomic.

The lost update problem • The lost update problem is illustrated by the following pair of transactions on bank accounts A, B and C, whose initial balances are \$100, \$200 and \$300, respectively. Transaction T transfers an amount from account A to account B. Transaction U transfers an amount from account C to account B. In both cases, the

Figure 16.5 The lost update problem

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance();		*balance = b.getBalance();*	
*b.setBalance(balance*1.1);*		*b.setBalance(balance*1.1);*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

amount transferred is calculated to increase the balance of *B* by 10%. The net effects on account *B* of executing the transactions *T* and *U* should be to increase the balance of account *B* by 10% twice, so its final value is $242.

Now consider the effects of allowing the transactions *T* and *U* to run concurrently, as in Figure 16.5. Both transactions get the balance of *B* as $200 and then deposit $20. The result is incorrect, increasing the balance of account *B* by $20 instead of $42. This is an illustration of the 'lost update' problem. *U*'s update is lost because *T* overwrites it without seeing it. Both transactions have read the old value before either writes the new value.

In Figure 16.5 onwards, we show the operations that affect the balance of an account on successive lines down the page, and the reader should assume that an operation on a particular line is executed at a later time than the one on the line above it.

Inconsistent retrievals • Figure 16.6 shows another example related to a bank account in which transaction *V* transfers a sum from account *A* to *B* and transaction *W* invokes the *branchTotal* method to obtain the sum of the balances of all the accounts in the bank.

Figure 16.6 The inconsistent retrievals problem

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100)		*aBranch.branchTotal()*	
b.deposit(100)			
a.withdraw(100);	$100		
		total = a.getBalance( )	$100
		total = total + b.getBalance()	$300
		total = total + c.getBalance()	
b.deposit(100)	$300	•	
		•	

Figure 16.7 A serially equivalent interleaving of *T* and *U*

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

The balances of the two bank accounts, *A* and *B*, are both initially $200. The result of *branchTotal* includes the sum of *A* and *B* as $300, which is wrong. This is an illustration of the 'inconsistent retrievals' problem. *W*'s retrievals are inconsistent because *V* has performed only the withdrawal part of a transfer at the time the sum is calculated.

Serial equivalence • If each of several transactions is known to have the correct effect when it is done on its own, then we can infer that if these transactions are done one at a time in some order the combined effect will also be correct. An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order is a *serially equivalent* interleaving. When we say that two different transactions have the *same effect* as one another, we mean that the *read* operations return the same values and that the instance variables of the objects have the same values at the end.

The use of serial equivalence as a criterion for correct concurrent execution prevents the occurrence of lost updates and inconsistent retrievals.

The lost update problem occurs when two transactions read the old value of a variable and then use it to calculate the new value. This cannot happen if one transaction is performed before the other, because the later transaction will read the value written by the earlier one. As a serially equivalent interleaving of two transactions produces the same effect as a serial one, we can solve the lost update problem by means of serial equivalence. Figure 16.7 shows one such interleaving in which the operations that affect the shared account, *B*, are actually serial, for transaction *T* does all its operations on *B* before transaction *U* does. Another interleaving of *T* and *U* that has this property is one in which transaction *U* completes its operations on account *B* before transaction *T* starts.

We now consider the effect of serial equivalence in relation to the inconsistent retrievals problem, in which transaction *V* is transferring a sum from account *A* to *B* and transaction *W* is obtaining the sum of all the balances (see Figure 16.6). The inconsistent retrievals problem can occur when a retrieval transaction runs concurrently with an update transaction. It cannot occur if the retrieval transaction is performed before or after the update transaction. A serially equivalent interleaving of a retrieval transaction and an update transaction, for example as in Figure 16.8, will prevent inconsistent retrievals occurring.

Figure 16.8 A serially equivalent interleaving of *V* and *W*

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal( )*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance( )	$100
		total = total + b.getBalance()	$400
		total = total + c.getBalance()	
		...	

Conflicting operations • When we say that a pair of operations *conflicts* we mean that their combined effect depends on the order in which they are executed. To simplify matters we consider a pair of operations, *read* and *write*. *read* accesses the value of an object and *write* changes its value. The *effect* of an operation refers to the value of an object set by a *write* operation and the result returned by a *read* operation. The conflict rules for *read* and *write* operations are given in Figure 16.9.

For any pair of transactions, it is possible to determine the order of pairs of conflicting operations on objects accessed by both of them. Serial equivalence can be defined in terms of operation conflicts as follows:

> For two transactions to be *serially equivalent*, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

Figure 16.9 *Read* and *write* operation conflict rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Figure 16.10 A non–serially-equivalent interleaving of operations of transactions *T* and *U*

Transaction *T*:	Transaction *U*:
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Consider as an example the transactions *T* and *U,* defined as follows:

T: x = *read(i)*; *write(i, 10)*; *write(j, 20)*;
U: y = *read(j)*; *write(j, 30)*; *z = read (i)*;

Then consider the interleaving of their executions, shown in Figure 16.10. Note that each transaction's access to objects *i* and *j* is serialized with respect to one another, because *T* makes all of its accesses to *i* before *U* does and *U* makes all of its accesses to *j* before *T* does. But the ordering is not serially equivalent, because the pairs of conflicting operations are not done in the same order at both objects. Serially equivalent orderings require one of the following two conditions:

1. *T* accesses *i* before *U* and *T* accesses *j* before *U*.
2. *U* accesses *i* before *T* and *U* accesses *j* before *T*.

Serial equivalence is used as a criterion for the derivation of concurrency control protocols. These protocols attempt to serialize transactions in their access to objects. Three alternative approaches to concurrency control are commonly used: locking, optimistic concurrency control and timestamp ordering. However, most practical systems use locking, which is discussed in Section 16.4. When locking is used, the server sets a lock, labelled with the transaction identifier, on each object just before it is accessed and removes these locks when the transaction has completed. While an object is locked, only the transaction that it is locked for can access that object; other transactions must either wait until the object is unlocked or, in some cases, share the lock. The use of locks can lead to deadlocks, with transactions waiting for each other to release locks – for example, when a pair of transactions each has an object locked that the other needs to access. We discuss the deadlock problem and some remedies for it in Section 16.4.1.

Optimistic concurrency control is described in Section 16.5. In optimistic schemes, a transaction proceeds until it asks to commit, and before it is allowed to commit the server performs a check to discover whether it has performed operations on any objects that conflict with the operations of other concurrent transactions, in which case the server aborts it and the client may restart it. The aim of the check is to ensure that all the objects are correct.

Timestamp ordering is described in Section 16.6. In timestamp ordering, a server records the most recent time of reading and writing of each object and for each

Figure 16.11 A dirty read when transaction *T* aborts

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

operation, the timestamp of the transaction is compared with that of the object to determine whether it can be done immediately or must be delayed or rejected. When an operation is delayed, the transaction waits; when it is rejected, the transaction is aborted.

Basically, concurrency control can be achieved either by clients' transactions waiting for one another or by restarting transactions after conflicts between operations have been detected, or by a combination of the two.

16.2.2 Recoverability from aborts

Servers must record all the effects of committed transactions and none of the effects of aborted transactions. They must therefore allow for the fact that a transaction may abort by preventing it affecting other concurrent transactions if it does so.

This section illustrates two problems associated with aborting transactions in the context of the banking example. These problems are called 'dirty reads' and 'premature writes', and both of them can occur in the presence of serially equivalent executions of transactions. These issues are concerned with the effects of operations on objects such as the balance of a bank account. To simplify things, operations are considered in two categories: *read* operations and *write* operations. In our illustrations, *getBalance* is a *read* operation and *setBalance* a *write* operation.

Dirty reads • The isolation property of transactions requires that transactions do not see the uncommitted state of other transactions. The 'dirty read' problem is caused by the interaction between a *read* operation in one transaction and an earlier *write* operation in another transaction on the same object. Consider the executions illustrated in Figure 16.11, in which *T* gets the balance of account *A* and sets it to $10 more, then *U* gets the balance of account *A* and sets it to $20 more, and the two executions are serially equivalent. Now suppose that the transaction *T* aborts after *U* has committed. Then the transaction *U* will have seen a value that never existed, since *A* will be restored to its original value. We say that the transaction *U* has performed a *dirty read*. As it has committed, it cannot be undone.

Figure 16.12 Overwriting uncommitted values

Transaction *T*:		Transaction *U*:	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

Recoverability of transactions • If a transaction (like *U*) has committed after it has seen the effects of a transaction that subsequently aborted, the situation is not recoverable. To ensure that such situations will not arise, any transaction (like *U*) that is in danger of having a dirty read delays its commit operation. The strategy for recoverability is to delay commits until after the commitment of any other transaction whose uncommitted state has been observed. In our example, *U* delays its commit until after *T* commits. In the case that *T* aborts, then *U* must abort as well.

Cascading aborts • In Figure 16.11, suppose that transaction *U* delays committing until after *T* aborts. As we have said, *U* must abort as well. Unfortunately, if any other transactions have seen the effects due to *U*, they too must be aborted. The aborting of these latter transactions may cause still further transactions to be aborted. Such situations are called *cascading aborts*. To avoid cascading aborts, transactions are only allowed to read objects that were written by committed transactions. To ensure that this is the case, any *read* operation must be delayed until other transactions that applied a *write* operation to the same object have committed or aborted. The avoidance of cascading aborts is a stronger condition than recoverability.

Premature writes • Consider another implication of the possibility that a transaction may abort. This one is related to the interaction between *write* operations on the same object belonging to different transactions. For an illustration, we consider two *setBalance* transactions, *T* and *U*, on account *A*, as shown in Figure 16.12. Before the transactions, the balance of account A was $100. The two executions are serially equivalent, with *T* setting the balance to $105 and *U* setting it to $110. If the transaction *U* aborts and *T* commits, the balance should be $105.

Some database systems implement the action of *abort* by restoring 'before images' of all the *writes* of a transaction. In our example, *A* is $100 initially, which is the 'before image' of *T*'s *write*; similarly, $105 is the 'before image' of *U*'s *write*. Thus if *U* aborts, we get the correct balance of $105.

Now consider the case when *U* commits and then *T* aborts. The balance should be $110, but as the 'before image' of *T*'s *write* is $100, we get the wrong balance of $100. Similarly, if *T* aborts and then *U* aborts, the 'before image' of *U*'s *write* is $105 and we get the wrong balance of $105 – the balance should revert to $100.

To ensure correct results in a recovery scheme that uses before images, *write* operations must be delayed until earlier transactions that updated the same objects have either committed or aborted.