**UNIT –III**

**Time and Global States:**

Time is an important and interesting issue in distributed systems, for several reasons. First, time is a quantity we often want to measure accurately. Second, algorithms that depend upon clock synchronization have been developed for several problems in distribution These include maintaining the consistency of distributed data, checking the authenticity of a request sent to a server and eliminating the processing of duplicate updates.

The notion of physical time is also problematic in a distributed system. This is not due to the effects of special relativity, which are negligible or nonexistent for normal computers.

**Clocks, events and process states:**

**Clocks •** Computers each contain their own physical clocks. These clocks are electronic devices that count oscillations occurring in a crystal at a definite frequency, and typically divide this count and store the result in a counter register. Clock devices can be programmed to generate interrupts at regular intervals.

The operating system reads the node’s hardware clock value, *Hi*(*t*) , scales it and adds an offset so as to produce a software clock *Ci*(*t*) = α*Hi*(*t*) + β that approximately measures real, physical time *t* for process *pi* .Note that successive events will correspond to different timestamps only if the *clock resolution* – the period between updates of the clock value – is smaller than the time interval between successive events. The rate at which events occur depends on such factors as the length of the processor instruction cycle.

**Clock skew and clock drift •** Computer clocks, like any others, tend not to be in perfect Agreement. The instantaneous difference between the readings of any two clocks is called their *skew*. Also, the crystal-based clocks used in computers are, like any other clocks, subject to *clock drift*, which means that they count time at different rates, and so diverge. The underlying oscillators are subject to physical variations, with the consequence that their frequencies of oscillation differ. A clock’s *drift rate* is the change in the offset between the clock and a nominal perfect reference clock per unit of time measured by the reference clock.

**Coordinated Universal Time •** Computer clocks can be synchronized to external sources of highly accurate time. The most accurate physical clocks use atomic oscillators, whose drift rate is about one part in 1013. The output of these atomic clocks is used as the standard for elapsed real time, known as *International Atomic Time*. Since 1967, the standard second has been defined as 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of Caesium-133 (Cs133).

***Coordinated Universal Time* –** abbreviated as UTC– is an international standard for timekeeping. It is based on atomic time, but a so-called ‘leap second’ is inserted – or, more rarely, deleted – occasionally to keep it in step with astronomical time. Satellite sources include the *Global Positioning System* (GPS).

**Synchronizing physical clocks:**

It is necessary to synchronize the processes’ clocks, *Ci* , with an authoritative, external source of time. This is *external synchronization*. And if the clocks *Ci* are synchronized with one another to a known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks, even though they are not necessarily synchronized to an external source of time. This is *internal synchronization*. We define these two modes of synchronization more closely as follows, over an interval of real time *I*:

***External synchronization*:** For a synchronization bound *D >*0, and for a source *S* of UTC time, *S*(*t*) – *Ci(t*) < *D*, for *i* = 1,2,..*N* and for all real times *t* in *I*. Another way of saying this is that the clocks *Ci* are *accurate* to within the bound *D*.

***Internal synchronization*:** For a synchronization bound *D* > 0 , [*Ci(t*) – *Cj*(*t*)] < *D*for *I*, *j* = 1,2,…*N* , and for all real times *t* in *I*. Another way of saying this is that the clocks *Ci agree* within the bound *D*.

Various notions of *correctness* for clocks have been suggested. It is common to define a hardware clock *H* to be correct if its drift rate falls within a known bound ϸ>0 (a value derived from one supplied by the manufacturer, such as 10–6 seconds/second).

This means that the error in measuring the interval between real times *t* and *t*1 (*t*1 > *t* ) is bounded:



This condition forbids jumps in the value of hardware clocks (during normal operation). Sometimes we also require our software clocks to obey the condition but a weaker condition of *monotonicity* may suffice. Monotonicity is the condition that a clock *C* only ever advances:



We need only change the rate at which updates are made to the time as given to applications. This can be achieved in software without changing the rate at which the underlying hardware clock ticks – recall that *Ci*(*t*) = α*Hi*(*t*) + β , where we are free to choose the values of α and β .

A clock that does not keep to whatever correctness conditions apply is defined to be *faulty*. A clock’s *crash failure* is said to occur when the clock stops ticking altogether; any other clock failure is an *arbitrary failure*.

14.3.1 Synchronization in a synchronous system

In a synchronous system, bounds are known for the drift rate of clocks, the maximum message transmission delay, and the time required executing each step of a process.

One process sends the time *t* on its local clock to the other in a message *m.* In principle, the receiving process could set its clock to the time *t* + *Ttrans* , where *Ttrans* is the time taken to transmit *m* between them. The two clocks would then agree.

In a synchronous system, by definition, there is also an upper bound *max* on the time taken to transmit any message. Let the uncertainty in the message transmission time be *u*, so that *u* = (*max* – *min*). If the receiver sets its clock to be *t* + *min,* then the clock skew may be as much as *u*, since the message may in fact have taken time *max* to arrive. Similarly, if it sets its clock to *t* + *max* , the skew may again be as large as *u*. If, however, it sets its clock to the halfway point, *t* + (*max* + *min*)/2, then the skew is at most *u* /2. In general, for a synchronous system, the optimum bound that can be achieved on clock skew when synchronizing *N* clocks is *u*(1 – 1/*N*).

14.3.2 Cristian’s method for synchronizing clocks

Cristian observed that while there is no upper bound on message transmission delays in an asynchronous system, the round-trip times for messages exchanged between pairs of processes are often reasonably short – a small fraction of a second. He describes the algorithm as *probabilistic*: the method achieves synchronization only if the observed round-trip times between client and server are sufficiently short compared with the required accuracy.

A process *p* requests the time in a message *mr* , and receives the time value *t* in a message *mt* (*t* is inserted in *mt* at the last possible point before transmission from *S*’s computer). Process *p* records the total round-trip time *Tround* taken to send the request *mr* and receive the reply *mt* . It can measure this time with reasonable accuracy if its rate of clock drift is small. For example, the round-trip time should be on the order of 1–10 milliseconds on a LAN, over which time a clock with a drift rate of 10–6 seconds/second varies by at most 10–5 milliseconds.

A simple estimate of the time to which *p* should set its clock is *t* + *Tround/* 2 , which assumes that the elapsed time is split equally before and after *S* placed *t* in *mt* . This is normally a reasonably accurate assumption, unless the two messages are transmitted over different networks. If the value of the minimum transmission time *min* is known or can be conservatively estimated, then we can determine the accuracy of this result as follows.

The earliest point at which *S* could have placed the time in *mt* was *min* after *p* dispatched *mr* . The latest point at which it could have done this was *min* before *mt* arrived at *p*. The time by *S*’s clock when the reply message arrives is therefore in the range [*t* + *min*, *t* + *Tround* – *min*] . The width of this range is *Tround* – 2*min* , so the accuracy is *±*(*Tround* / 2 – *min*).

14.3.3 The Berkeley algorithm

Gusella and Zatti describe an algorithm for internal synchronization that they developed for collections of computers running Berkeley UNIX. In it, a coordinator computer is chosen to act as the *master*. Unlike in Cristian’s protocol, this computer periodically polls the other computers whose clocks are to be synchronized, called *slaves*. The slaves send back their clock values to it. The master estimates their local clock times by observing the round-trip times, and it averages the values obtained.

The Berkeley algorithm eliminates readings from faulty clocks. Such clocks could have a significant adverse effect if an ordinary average was taken so instead the master takes a *fault-tolerant average*.

14.3.4 The Network Time Protocol

The Network Time Protocol (NTP)] defines architecture for a time service and a protocol to distribute time information over the Internet.

NTP’s chief design aims and features are as follows:

* To provide a service enabling clients across the Internet to be synchronized accurately to UTC
* To provide a reliable service that can survive lengthy losses of connectivity
* To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers
* To provide protection against interference with the time service, whether malicious or accidental

The NTP service is provided by a network of servers located across the Internet. *Primary servers* are connected directly to a time source such as a radio clock receiving UTC; *secondary servers* are synchronized, ultimately, with primary servers. The servers are connected in a logical hierarchy called a *synchronization subnet* whose levels are called *strata*. Primary servers occupy stratum 1: they are at the root. Stratum 2 servers are secondary servers that are synchronized directly with the primary servers; stratum 3 servers are synchronized with stratum 2 servers, and so on. The lowest-level (leaf) servers execute in users’ workstations.

The clocks belonging to servers with high stratum numbers are liable to be less accurate than those with low stratum numbers, because errors are introduced at each level of synchronization. NTP also takes into account the total message round-trip delays to the root in assessing the quality of timekeeping data held by a particular server.

NTP servers synchronize with one another in one of three modes: multicast, procedure-call and symmetric mode. *Multicast mode* is intended for use on a high-speed LAN. One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay.

*In Procedure-call mode* one server accepts requests from other computers, which it processes by replying with its timestamp. This mode is suitable where higher accuracies are required than can be achieved with multicast, or where multicast is not supported in hardware. Finally, *symmetric mode* is intended for use by the servers that supply time information in LANs and by the higher levels (lower strata) of the synchronization subnet, where the highest accuracies are to be achieved.

For each pair of messages sent between two servers the NTP calculates an *offset oi* , which is an estimate of the actual offset between the two clocks, and a *delay di* , which is the total transmission time for the two messages. If the true offset of the clock at *B* relative to that at *A* is *o*, and if the actual transmission times for *m* and *m'* are *t* and *t'*, respectively, then we have:



COORDINATION AND AGREEMENT

It 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.

15.1.1 Failure assumptions and failure detectors

For the sake of simplicity, we assume 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.

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*.

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* is a service that processes queries about whether a particular process has failed. It is often implemented by an object local to each process 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*.

An unreliable failure detector may produce one of two values when given the identity of a process: *Unsuspected* or *Suspected*. A result of *Unsuspected* signifies that the detector has recently received evidence suggesting that the process has not failed; A result of *Suspected* signifies that the failure detector has some indication that the process may have failed.

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.

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 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.

**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 some cases shared resources are managed by servers that also provide mechanisms for mutual exclusion –But in some practical cases, a separate mechanism for mutual exclusion is required.

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 *pi* *i* = 1, 2…*N* , that do not share variables. The processes access common resources, but they do so in a 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. The absence of starvation is a *fairness* condition.

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

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. 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.

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

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 *pi* has a communication channel to the next process in the ring, *pi* + 1*mod 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 – clockwise, say – around the ring. The ring topology may be unrelated to the physical interconnections between the underlying computers.

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.

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 •** 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 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…*pN* bear distinct numeric identifiers. They are assumed to possess communication channels to one another, and each process *pi* keeps a Lamport clock, updated according to the rules LC1 and LC2. Messages requesting entry are of the form <*Ti, pi* >, where *T* is the sender’s timestamp and *pi* 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*.

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.

This algorithm achieves the safety property ME1. If it were possible for two processes *pi* and *pj* ( *i≠* *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 *<Ti, pi>* are totally ordered, this is impossible. We leave the reader to verify that the algorithm also meets requirements ME2 and ME3*.*

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.

**Maekawa’s voting algorithm •** 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 Vi* with each process *pi* ( *i* = 1,2… *N* ), where *Vi,< p*1,*p*2.. *pN*  . The sets *Vi* are chosen so that, for all *I*, *j* = 1,2,..*N* :

**•** *pi €Vi*

**•** *Vi Vj≠φ*– there is at least one common member of any two voting sets

**• |***Vi|* = *K* – to be fair, each process has a voting set of the same size

**•** Each process *pj* is contained in *M* of the voting sets *Vi* .

Maekawa showed that the optimal solution, which minimizes *K* and allows the processes to achieve mutual exclusion, has *K* 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 nontrivial to calculate the optimal sets *Ri* . As an approximation, a simple way of deriving sets *Ri* such that |*Ri|* = 2 is to place the processes in a *N* by *N* matrix and let *Vi* be the union of the row and column containing *pi.*



To obtain entry to the critical section, a process *pi* sends *request* messages to all *K* members of *Vi* (including itself). *pi* cannot enter the critical section until it has received all *K reply* messages. When a process *pj* in *Vi* receives *pi’s* *request* message, it sends a *reply* message immediately, 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, *pi* sends *release* messages to all *K* members of *Vi* (including itself).

This algorithm achieves the safety property, ME1. If it were possible for two processes *pi* and *pj* to enter the critical section at the same time, then the processes in *Vi*  *Vj ≠* 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 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 *N* messages per entry to the critical section and messages per exit (assuming no hardware multicast facilities). The total of 3 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?

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 *pi,(* 1,2,.. *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 *pi* 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 *pi,(* 1,2,.. *N* ) has a variable *electedi* , 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 ‘ ’ to denote that it is not yet defined.

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

E1: (safety) A participant process *pi* has *electedi* = † or *electedi* = *P*, where *P* is chosen as the non-crashed process at the end of the run with the largest identifier.

E2: (liveness) All processes *pi* participate and eventually either set *electedi* ≠†– or crash.

Note that there may be processes *pj* that are not yet participants, which record in *electedj* 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 *pi* 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 neighbor.

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 neighbor. 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*.

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 neighbor, announcing its election and enclosing its identity.

When a process *pi* receives an *elected* message, it marks itself as a *nonparticipant*, sets its variable *electedi* to the identifier in the message and, unless it is the new coordinator, forwards the message to its neighbor.

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 3*N* – 1 messages in all. The turnaround time is also 3*N* – 1 , since these messages are sent sequentially.

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 reconstitute 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, *Ttrans* , and a maximum delay for processing a message *Tprocess* . Therefore, we can calculate a time *T* = 2*Ttrans* + *Tprocess* 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 *pi* receives a *coordinator* message, it sets its variable *electedi* 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 process altogether begin elections, each sending messages to processes with higher identifiers.

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 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 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.)
* *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 *Bmulticast*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 correct processes 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 *Rdeliver* 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 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 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 *Rdeliver*ed a message. Since any process that *R-deliver*s the message must first *Bmulticast* 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.

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* ≠ *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*1), then every correct process that delivers *m*􀁣 will deliver *m* before *m*1.
* *Causal ordering*: If *multicast*(*g*, *m*)🡪 *multicast*(*g*, *m*1), where 🡪is the happened-before relation induced only by messages sent between the members of *g*, then any correct process that delivers *m*1 will deliver *m* before *m*1 .
* *Total ordering*: If a correct process delivers message *m* before it delivers *m*1 , then any other correct process that delivers *m*1 will deliver *m* before *m*1 .

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, 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 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. 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 left hand 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 *Sg p* and *Rg q* held at process *p* from the reliable multicast protocol of Section 15.4.2: *Sg p* is a count of how many messages *p* has sent to *g* and, for each *q*, *Rg 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 *Sg p* onto the message, *B-multicast*s the message to *g* and then increments *Sg p* by 1. Upon receipt of a message from *q* bearing the sequence number *S*, *p* checks whether *S Rg* = *q* + 1. If so, this message is the next one expected from the sender *q* and *p FO-deliver*s it, setting *Rg q*:=*S*. If *S Rg* 􀀡 *q* + 1, it places the message in the hold-back queue until the intervening messages have been delivered and *S Rg 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 *sg* , 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 *TOdeliver*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 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 *Ag q* , the largest agreed sequence number it has observed so far for group *g*, and *Pg 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 *Pg q* := *Max*(*Ag q* , *Pg q* ) + 1. In reality, we must include process identifiers in the proposed values *Pg 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.

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 *Ag q* := *Max*(*Ag 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 ) ≥ *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 )

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] 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 *pi* ( *i* = 1,2,…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 *pi B-deliver*s a message from *pj* , 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, *pi* waits until (a) it has delivered any earlier message sent by *pj* , and (b) it has delivered any message that *pj* 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*)🡪 *multicast* (*g*, *m1*). Let *V* and *V*1 be the vector timestamps of *m* and *m1* , respectively. It is straightforward to prove inductively from the algorithm that *V* < *V*1. In particular, if process *pk* multicast *m*, then *V*[*k*] ≤ *V*1 *[k*] .

Consider what happens when some correct process *pi B-deliver*s *m1* (as opposed to *CO-deliver*ing it) without first *CO-deliver*ing *m*. By the algorithm, *Vi*[*k*] can increase only when *pi* delivers a message from *pk* , when it increases by 1. But *pi* has not received *m*, and therefore *Vi*[*k*] cannot increase beyond *V*[*k*] – 1 . It is therefore not possible for *pi* to *CO-deliver m1* , since this would require that *Vi*[*k*] ≥ *V*1[*k*] , and therefore that *Vi*[*k*] ≥ *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*1, then both messages are addressed to the members of *g* *g*1 :

*Global FIFO ordering*: If a correct process issues *multicast*(*g*, *m*) and then *multicast*(*g*1 , *m1* ), then every correct process in *g* *g*1 that delivers *m1* will deliver *m* before *m1* .

*Global causal ordering*: If *multicast*(*g*, *m*) 🡪 *multicast*(*g*1 , *m1*), where 🡪is the happened-before relation induced by any chain of multicast messages, then any correct process in *g* *g*1 that delivers *m1* will deliver *m* before *m1* .

*Pairwise total ordering*: If a correct process delivers message *m* sent to *g* before it delivers *m1* sent to *g*1, then any other correct process in *g* *g*1 that delivers *m1* will deliver *m* before *m1* .

*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.