14

TIME AND GLOBAL STATES

In this chapter, we introduce some topics related to the issue of time in distributed systems. Time is an important practical issue. For example, we require computers around the world to timestamp electronic commerce transactions consistently. Time is also an important theoretical construct in understanding how distributed executions unfold. But time is problematic in distributed systems. Each computer may have its own physical clock, but the clocks typically deviate, and we cannot synchronize them perfectly. We examine algorithms for synchronizing physical clocks approximately and then go on to explain logical clocks, including vector clocks, which are a tool for ordering events without knowing precisely when they occurred.

The absence of global physical time makes it difficult to find out the state of our distributed programs as they execute. We often need to know what state process A is in when process B is in a certain state, but we cannot rely on physical clocks to know what is true at the same time. The second half of the chapter examines algorithms to determine global states of distributed computations despite the lack of global time.

14.1 Introduction

This chapter introduces fundamental concepts and algorithms related to monitoring distributed systems as their execution unfolds, and to timing the events that occur in their executions.

Time is an important and interesting issue in distributed systems, for several reasons. First, time is a quantity we often want to measure accurately. In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronize its clock with an authoritative, external source of time. For example, an eCommerce transaction involves events at a merchant's computer and at a bank's computer. It is important, for auditing purposes, that those events are timestamped accurately.

Second, algorithms that depend upon clock synchronization have been developed for several problems in distribution [Liskov 1993]. These include maintaining the consistency of distributed data (the use of timestamps to serialize transactions is discussed in Section 16.6), checking the authenticity of a request sent to a server (a version of the Kerberos authentication protocol, discussed in Chapter 11, depends on loosely synchronized clocks) and eliminating the processing of duplicate updates (see, for example, Ladin *et al*. [1992]).

Measuring time can be problematic due to the existence of multiple frames of reference. Einstein demonstrated, in his Special Theory of Relativity, the intriguing consequences that follow from the observation that the speed of light is constant for all observers, regardless of their relative velocity. He proved from this assumption, among other things, that two events that are judged to be simultaneous in one frame of reference are not necessarily simultaneous according to observers in other frames of reference that are moving relative to it. For example, an observer on the Earth and an observer travelling away from the Earth in a spaceship will disagree on the time interval between events, the more so as their relative speed increases.

The relative order of two events can even be reversed for two different observers. But this cannot happen if one event causes the other to occur. In that case, the physical effect follows the physical cause for all observers, although the time elapsed between cause and effect can vary. The timing of physical events was thus proved to be relative to the observer, and Newton's notion of absolute physical time was shown to be without foundation. There is no special physical clock in the universe to which we can appeal when we want to measure intervals of time.

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 (unless one counts computers travelling in spaceships!). Rather, the problem is based on a similar limitation in our ability to timestamp events at different nodes sufficiently accurately to know the order in which any pair of events occurred, or whether they occurred simultaneously. There is no absolute, global time to which we can appeal. And yet we sometimes need to observe distributed systems and establish whether certain states of affairs occurred at the same time. For example, in object-oriented systems we need to be able to establish whether references to a particular object no longer exist – whether the object has become garbage (in which case we can free its memory). Establishing this requires observations of the states of processes (to find out whether they contain references) and of the communication channels between processes (in case messages containing references are in transit).

In the first half of this chapter, we examine methods whereby computer clocks can be approximately synchronized, using message passing. We go on to introduce logical clocks, including vector clocks, which are used to define an order of events without measuring the physical time at which they occurred.

In the second half, we describe algorithms whose purpose is to capture global states of distributed systems as they execute.

14.2 Clocks, events and process states

Chapter 2 presented an introductory model of interaction between the processes within a distributed system. Here we refine that model in order to help us to understand how to characterize the system's evolution as it executes, and how to timestamp the events in a system's execution that interest users. We begin by considering how to order and timestamp the events that occur at a single process.

We take a distributed system to consist of a collection $\wp$ of N processes p_i, $i = 1, 2, \ldots N$. Each process executes on a single processor, and the processors do not share memory (Chapter 6 briefly considered the case of processes that share memory). Each process p_i in $\wp$ has a state s_i that, in general, it transforms as it executes. The process's state includes the values of all the variables within it. Its state may also include the values of any objects in its local operating system environment that it affects, such as files. We assume that processes cannot communicate with one another in any way except by sending messages through the network. So, for example, if the processes operate robot arms connected to their respective nodes in the system, then they are not allowed to communicate by shaking one another's robot hands!

As each process p_i executes it takes a series of actions, each of which is either a message *send* or *receive* operation, or an operation that transforms p_i's state – one that changes one or more of the values in s_i. In practice, we may choose to use a high-level description of the actions, according to the application. For example, if the processes in $\wp$ are engaged in an eCommerce application, then the actions may be ones such as 'client dispatched order message' or 'merchant server recorded transaction to log'.

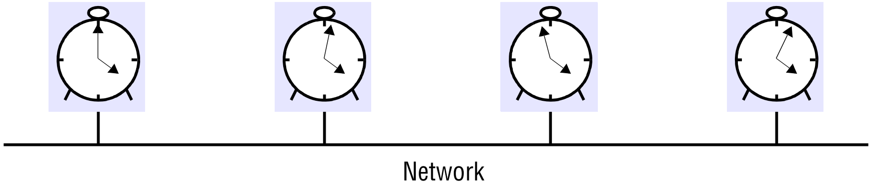
We define an event to be the occurrence of a single action that a process carries out as it executes – a communication action or a state-transforming action. The sequence of events within a single process p_i can be placed in a single, total ordering, which we denote by the relation $\rightarrow_i$ between the events. That is, $e \rightarrow_i e'$ if and only if the event e occurs before e' at p_i. This ordering is well defined, whether or not the process is multi-threaded, since we have assumed that the process executes on a single processor.

Now we can define the *history* of process p_i to be the series of events that take place within it, ordered as we have described by the relation $\rightarrow_i$:

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

Clocks • We have seen how to order the events at a process, but not how to timestamp them – i.e., to assign to them a date and time of day. Computers each contain their own physical clocks. These clocks are electronic devices that count oscillations occurring in

Figure 14.1 Skew between computer clocks in a distributed system



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 in order that, for example, timeslicing can be implemented; however, we shall not concern ourselves with this aspect of clock operation.

The operating system reads the node's hardware clock value, $H_i(t)$, scales it and adds an offset so as to produce a software clock $C_i(t) = \alpha H_i(t) + \beta$ that approximately measures real, physical time *t* for process p_i. In other words, when the real time in an absolute frame of reference is *t*, $C_i(t)$ is the reading on the software clock. For example, $C_i(t)$ could be the 64-bit value of the number of nanoseconds that have elapsed at time *t* since a convenient reference time. In general, the clock is not completely accurate, so $C_i(t)$ will differ from *t*. Nonetheless, if C_i behaves sufficiently well (we shall examine the notion of clock correctness shortly), we can use its value to timestamp any event at p_i. 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 (Figure 14.1). 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. Moreover, even the same clock's frequency varies with temperature. Designs exist that attempt to compensate for this variation, but they cannot eliminate it. The difference in the oscillation period between two clocks might be extremely small, but the difference accumulated over many oscillations leads to an observable difference in the counters registered by two clocks, no matter how accurately they were initialized to the same value. A clock's *drift rate* is the change in the offset (difference in reading) between the clock and a nominal perfect reference clock per unit of time measured by the reference clock. For ordinary clocks based on a quartz crystal this is about 10^{-6} seconds/second, giving a difference of 1 second every 1,000,000 seconds, or 11.6 days. The drift rate of 'high-precision' quartz clocks is about 10^{-7} or 10^{-8}.

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 10^{13}. 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 (Cs^{133}).

Seconds and years and other time units that we use are rooted in astronomical time. They were originally defined in terms of the rotation of the Earth on its axis and its rotation about the Sun. However, the period of the Earth's rotation about its axis is gradually getting longer, primarily because of tidal friction; atmospheric effects and convection currents within the Earth's core also cause short-term increases and decreases in the period. So astronomical time and atomic time have a tendency to get out of step.

Coordinated Universal Time – abbreviated as UTC (from the French equivalent) – 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. UTC signals are synchronized and broadcast regularly from land-based radio stations and satellites covering many parts of the world. For example, in the USA, the radio station WWV broadcasts time signals on several shortwave frequencies. Satellite sources include the *Global Positioning System* (GPS).

Receivers are available commercially. Compared with 'perfect' UTC, the signals received from land-based stations have an accuracy on the order of 0.1–10 milliseconds, depending on the station used. Signals received from GPS satellites are accurate to about 1 microsecond. Computers with receivers attached can synchronize their clocks with these timing signals.

14.3 Synchronizing physical clocks

In order to know at what time of day events occur at the processes in our distributed system $\wp$ – for example, for accountancy purposes – it is necessary to synchronize the processes' clocks, C_i, with an authoritative, external source of time. This is *external synchronization*. And if the clocks C_i 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, $|S(t) - C_i(t)| < D$, for $i = 1, 2, \ldots N$ and for all real times t in I. Another way of saying this is that the clocks C_i are *accurate* to within the bound D.
>
> *Internal synchronization*: For a synchronization bound $D > 0$, $|C_i(t) - C_j(t)| < D$ for $i, j = 1, 2, \ldots N$, and for all real times t in I. Another way of saying this is that the clocks C_i *agree* within the bound D.

Clocks that are internally synchronized are not necessarily externally synchronized, since they may drift collectively from an external source of time even though they agree with one another. However, it follows from the definitions that if the system $\wp$ is externally synchronized with a bound D, then the same system is internally synchronized with a bound of $2D$.

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 $\rho > 0$ (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' $(t' > t)$ is bounded:

$$(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$$

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:

$$t' > t \Rightarrow C(t') > C(t)$$

For example, the UNIX *make* facility is a tool that is used to compile only those source files that have been modified since they were last compiled. The modification dates of each corresponding pair of source and object files are compared to determine this condition. If a computer whose clock was running fast set its clock back after compiling a source file but before the file was changed, the source file might appear to have been modified prior to the compilation. Erroneously, *make* will not recompile the source file.

We can achieve monotonicity despite the fact that a clock is found to be running fast. 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 $C_i(t) = \alpha H_i(t) + \beta$, where we are free to choose the values of α and β.

A hybrid correctness condition that is sometimes applied is to require that a clock obeys the monotonicity condition, and that its drift rate is bounded between synchronization points, but to allow the clock value to jump ahead at synchronization points.

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*. A historical example of an arbitrary failure is that of a clock with the 'Y2K bug', which broke the monotonicity condition by registering the date after 31 December 1999 as 1 January 1900 instead of 2000; another example is a clock whose batteries are very low and whose drift rate suddenly becomes very large.

Note that clocks do not have to be accurate to be correct, according to the definitions. Since the goal may be internal rather than external synchronization, the criteria for correctness are only concerned with the proper functioning of the clock's 'mechanism', not its absolute setting.

We now describe algorithms for external synchronization and for internal synchronization.

14.3.1 Synchronization in a synchronous system

We begin by considering the simplest possible case: of internal synchronization between two processes in a synchronous distributed system. In a synchronous system, bounds are known for the drift rate of clocks, the maximum message transmission delay, and the time required to execute each step of a process (see Section 2.4.1).

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 $t + T_{trans}$, where T_{trans} is the time taken to transmit m between them. The two clocks would then agree (since the aim is internal synchronization, it does not matter whether the sending process's clock is accurate).

Unfortunately, T_{trans} is subject to variation and is unknown. In general, other processes are competing for resources with the processes to be synchronized at their respective nodes, and other messages compete with m for the network resources. Nonetheless, there is always a minimum transmission time, *min*, that would be obtained if no other processes executed and no other network traffic existed; *min* can be measured or conservatively estimated.

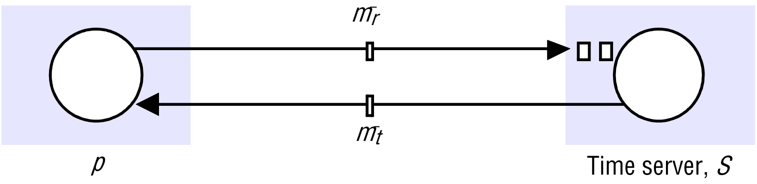
In a synchronous system, by definition, there is also an upper bound *max* on the time taken to transmit any message. Let the uncertainty in the message transmission time be u, so that $u = (max - min)$. If the receiver sets its clock to be $t + min$, then the clock skew may be as much as u, since the 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 $u(1 - 1/N)$ [Lundelius and Lynch 1984].

Most distributed systems found in practice are asynchronous: the factors leading to message delays are not bounded in their effect, and there is no upper bound *max* on message transmission delays. This is particularly so for the Internet. For an asynchronous system, we may say only that $T_{trans} = min + x$, where $x \geq 0$. The value of x is not known in a particular case, although a distribution of values may be measurable for a particular installation.

14.3.2 Cristian's method for synchronizing clocks

Cristian [1989] suggested the use of a time server, connected to a device that receives signals from a source of UTC, to synchronize computers externally. Upon request, the server process S supplies the time according to its clock, as shown in Figure 14.2.

Figure 14.2 Clock synchronization using a time server



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 m_r, and receives the time value *t* in a message m_t (*t* is inserted in m_t at the last possible point before transmission from *S*'s computer). Process *p* records the total round-trip time T_{round} taken to send the request m_r and receive the reply m_t. 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 $t + T_{round}/2$, which assumes that the elapsed time is split equally before and after *S* placed *t* in m_t. 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 m_t was *min* after *p* dispatched m_r. The latest point at which it could have done this was *min* before m_t arrived at *p*. The time by *S*'s clock when the reply message arrives is therefore in the range $[t + min, t + T_{round} - min]$. The width of this range is $T_{round} - 2min$, so the accuracy is $\pm(T_{round}/2 - min)$.

Variability can be dealt with to some extent by making several requests to *S* (spacing the requests so that transitory congestion can clear) and taking the minimum value of T_{round} to give the most accurate estimate. The greater the accuracy required, the smaller the probability of achieving it. This is because the most accurate results are those in which both messages are transmitted in a time close to *min* – an unlikely event in a busy network.

Discussion of Cristian's algorithm • As described, Cristian's method suffers from the problem associated with all services implemented by a single server: that the single time server might fail and thus render synchronization temporarily impossible. Cristian suggested, for this reason, that time should be provided by a group of synchronized time servers, each with a receiver for UTC time signals. For example, a client could multicast its request to all servers and use only the first reply obtained.

Note that a faulty time server that replied with spurious time values, or an imposter time server that replied with deliberately incorrect times, could wreak havoc in a computer system. These problems were beyond the scope of the work described by Cristian [1989], which assumes that sources of external time signals are self-checking. Cristian and Fetzer [1994] describe a family of probabilistic protocols for internal clock synchronization, each of which tolerates certain failures. Srikanth and Toueg [1987] first described an algorithm that is optimal with respect to the accuracy of the synchronized clocks, while tolerating some failures. Dolev *et al.* [1986] showed that if *f* is the number of faulty clocks out of a total of *N*, then we must have $N > 3f$ if the other, correct, clocks are still to be able to achieve agreement. The problem of dealing with faulty clocks is partially addressed by the Berkeley algorithm, which is described next. The problem of malicious interference with time synchronization can be dealt with by authentication techniques.

14.3.3 The Berkeley algorithm

Gusella and Zatti [1989] 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 (similarly to Cristian's technique), and it averages the values obtained (including its own clock's reading). The balance of probabilities is that this average cancels out the individual clocks' tendencies to run fast or slow. The accuracy of the protocol depends upon a nominal maximum round-trip time between the master and the slaves. The master eliminates any occasional readings associated with larger times than this maximum.

Instead of sending the updated current time back to the other computers – which would introduce further uncertainty due to the message transmission time – the master sends the amount by which each individual slave's clock requires adjustment. This can be a positive or negative value.

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*. That is, a subset is chosen of clocks that do not differ from one another by more than a specified amount, and the average is taken of readings from only these clocks.

Gusella and Zatti describe an experiment involving 15 computers whose clocks were synchronized to within about 20–25 milliseconds using their protocol. The local clocks' drift rates were measured to be less than 2×10^{-5}, and the maximum round-trip time was taken to be 10 milliseconds.

Should the master fail, then another can be elected to take over and function exactly as its predecessor. Section 15.3 discusses some general-purpose election algorithms. Note that these are not guaranteed to elect a new master in bounded time, so the difference between two clocks would be unbounded if they were used.

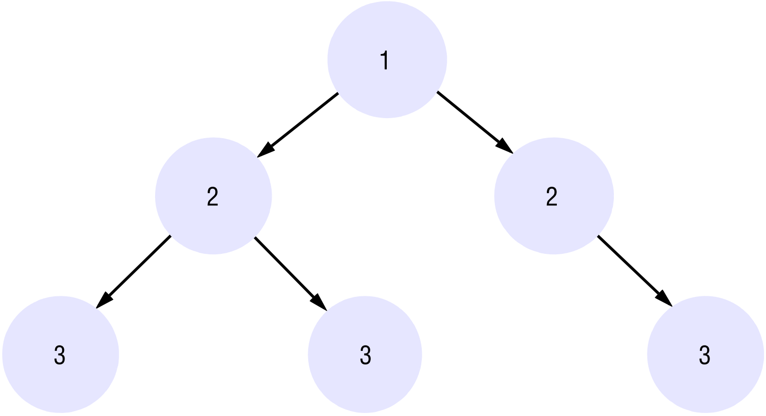
14.3.4 The Network Time Protocol

Cristian's method and the Berkeley algorithm are intended primarily for use within intranets. The Network Time Protocol (NTP) [Mills 1995] defines an 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: Although large and variable message delays are encountered in Internet communication, NTP employs statistical techniques for the filtering of timing data and it discriminates between the quality of timing data from different servers.

Figure 14.3 An example synchronization subnet in an NTP implementation



Arrows denote synchronization control, numbers denote strata.

To provide a reliable service that can survive lengthy losses of connectivity: There are redundant servers and redundant paths between the servers. The servers can reconfigure so as to continue to provide the service if one of them becomes unreachable.

To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers: The service is designed to scale to large numbers of clients and servers.

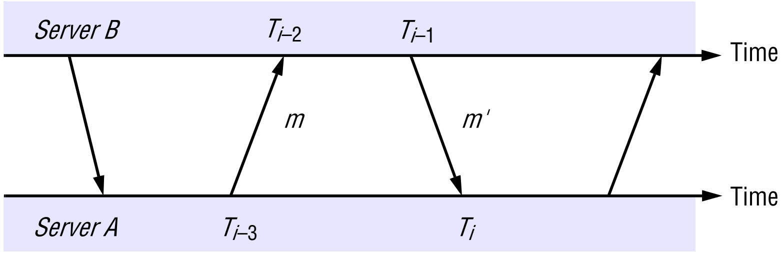
To provide protection against interference with the time service, whether malicious or accidental: The time service uses authentication techniques to check that timing data originate from the claimed trusted sources. It also validates the return addresses of messages sent to it.

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* (see Figure 14.3), 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.

The synchronization subnet can reconfigure as servers become unreachable or failures occur. If, for example, a primary server's UTC source fails, then it can become

Figure 14.4 Messages exchanged between a pair of NTP peers



a stratum 2 secondary server. If a secondary server's normal source of synchronization fails or becomes unreachable, then it may synchronize with another 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. This mode can achieve only relatively low accuracies, but ones that nonetheless are considered sufficient for many purposes.

Procedure-call mode is similar to the operation of Cristian's algorithm, described in Section 14.3.2. In this mode, one server accepts requests from other computers, which it processes by replying with its timestamp (current clock reading). This mode is suitable where higher accuracies are required than can be achieved with multicast, or where multicast is not supported in hardware. For example, file servers on the same or a neighbouring LAN that need to keep accurate timing information for file accesses could contact a local server in procedure-call mode.

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. A pair of servers operating in symmetric mode exchange messages bearing timing information. Timing data are retained as part of an association between the servers that is maintained in order to improve the accuracy of their synchronization over time.

In all modes, messages are delivered unreliably, using the standard UDP Internet transport protocol. In procedure-call mode and symmetric mode, processes exchange pairs of messages. Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted. The recipient of the NTP message notes the local time when it receives the message. The four times T_{i-3}, T_{i-2}, T_{i-1} and T_i are shown in Figure 14.4 for the messages *m* and *m'* sent between servers *A* and *B*. Note that in symmetric mode, unlike in Cristian's algorithm, there can be a non-negligible delay between the arrival of one message and the dispatch of the next. Also, messages may be lost, but the three timestamps carried by each message are nonetheless valid.

For each pair of messages sent between two servers the NTP calculates an *offset* o_i, which is an estimate of the actual offset between the two clocks, and a *delay* d_i, 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:

$$T_{i-2} = T_{i-3} + t + o \text{ and } T_i = T_{i-1} + t' - o$$

This leads to:

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

and:

$$o = o_i + (t' - t)/2 \text{, where } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

Using the fact that $t, t' \geq 0$, it can be shown that $o_i - d_i/2 \leq o \leq o_i + d_i/2$. Thus o_i is an estimate of the offset, and d_i is a measure of the accuracy of this estimate.

NTP servers apply a data filtering algorithm to successive pairs $<o_i, d_i>$, which estimates the offset *o* and calculates the quality of this estimate as a statistical quantity called the *filter dispersion*. A relatively high filter dispersion represents relatively unreliable data. The eight most recent pairs $<o_i, d_i>$ are retained. As with Cristian's algorithm, the value of o_j that corresponds to the minimum value d_j is chosen to estimate *o*.

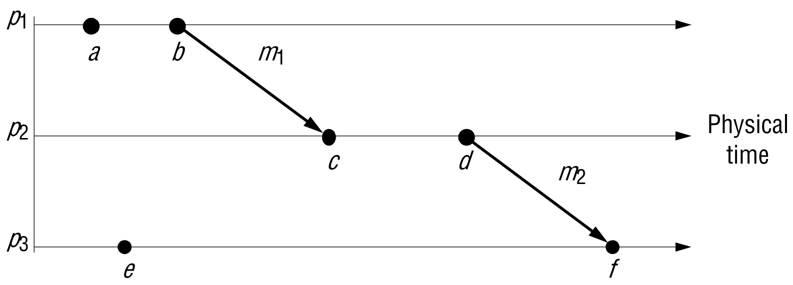
The value of the offset derived from communication with a single source is not necessarily used by itself to control the local clock, however. In general, an NTP server engages in message exchanges with several of its peers. In addition to data filtering applied to exchanges with each single peer, NTP applies a peer-selection algorithm. This examines the values obtained from exchanges with each of several peers, looking for relatively unreliable values. The output from this algorithm may cause a server to change the peer that it primarily uses for synchronization.

Peers with lower stratum numbers are more favoured than those in higher strata because they are 'closer' to the primary time sources. Also, those with the lowest *synchronization dispersion* are relatively favoured. This is the sum of the filter dispersions measured between the server and the root of the synchronization subnet. (Peers exchange synchronization dispersions in messages, allowing this total to be calculated.)

NTP employs a phase lock loop model [Mills 1995], which modifies the local clock's update frequency in accordance with observations of its drift rate. To take a simple example, if a clock is discovered always to gain time at the rate of, say, four seconds per hour, then its frequency can be reduced slightly (in software or hardware) to compensate for this. The clock's drift in the intervals between synchronization is thus reduced.

Mills quotes synchronization accuracies on the order of tens of milliseconds over Internet paths, and one millisecond on LANs.

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 →, 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 → 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:
$L_i := L_i + 1$.

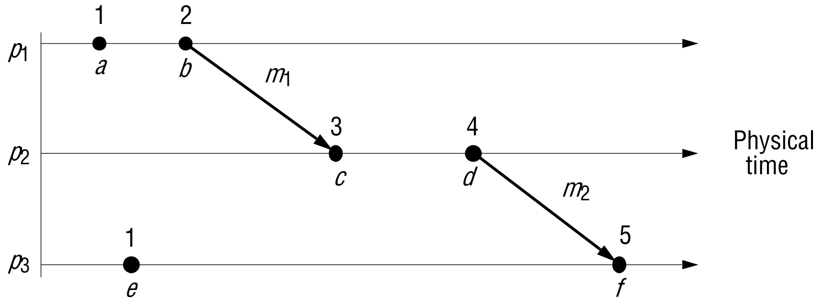
LC2: (a) When a process p_i sends a message m, it piggybacks on m the value $t = L_i$.

(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 $i, j = 1, 2 \ldots, N$.

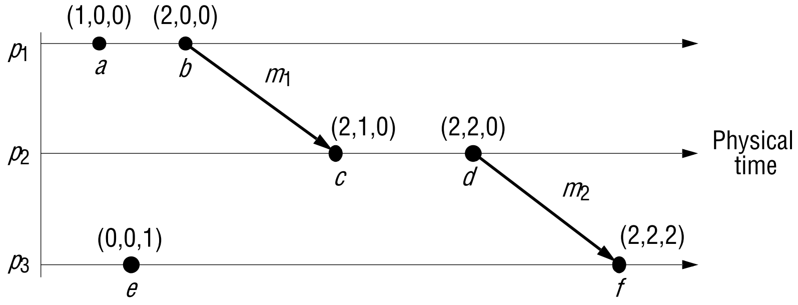
VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.

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 componentwise 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:

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

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

$$V < V' \text{ iff } V \leq V' \wedge V \neq V'$$

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 $e \rightarrow e'$.

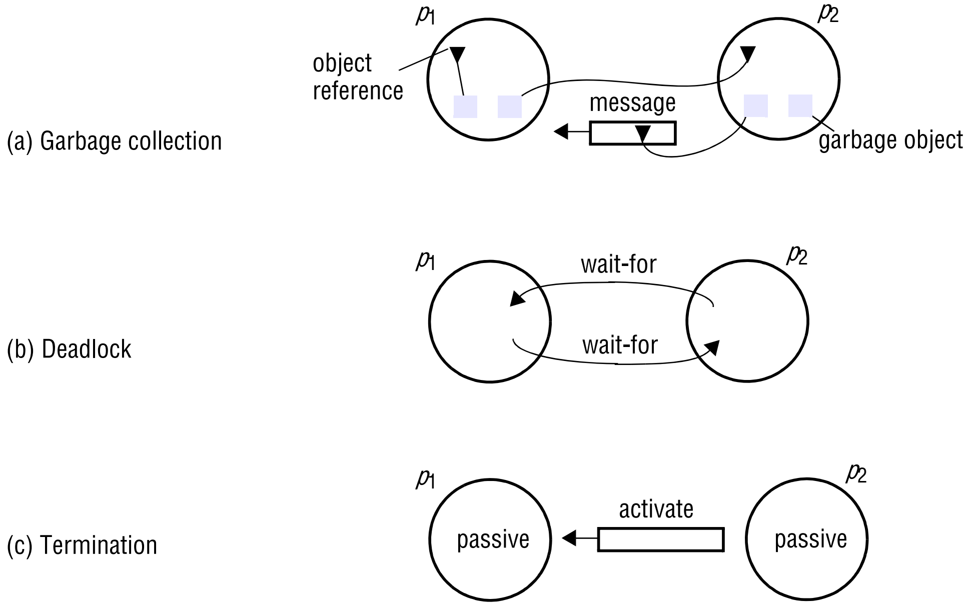
Figure 14.7 shows the vector timestamps of the events of Figure 14.5. It can be seen, for example, that $V(a) < V(f)$, which reflects the fact that $a \rightarrow f$. Similarly, we can tell when two events are concurrent by comparing their timestamps. For example, that $c \| e$ can be seen from the facts 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.

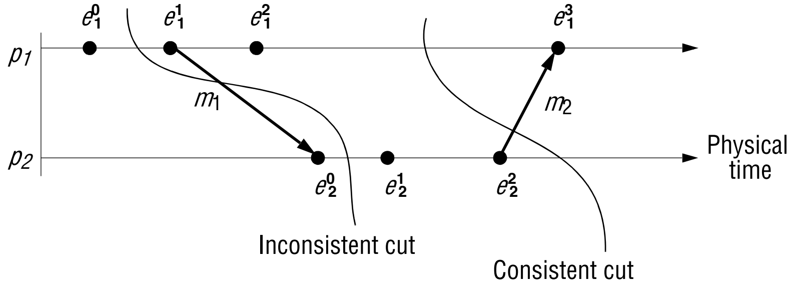
Let us return to our general system $\wp$ of N processes p_i $(i = 1, 2, \ldots, N)$, whose execution we wish to study. We said above that a series of events occurs at each process, and that we may characterize the execution of each process by its history:

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

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 → 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

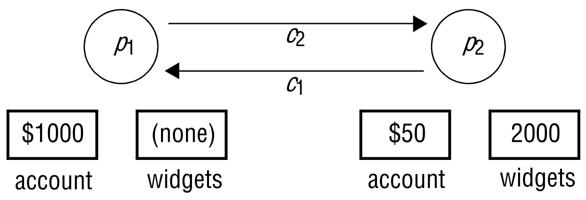
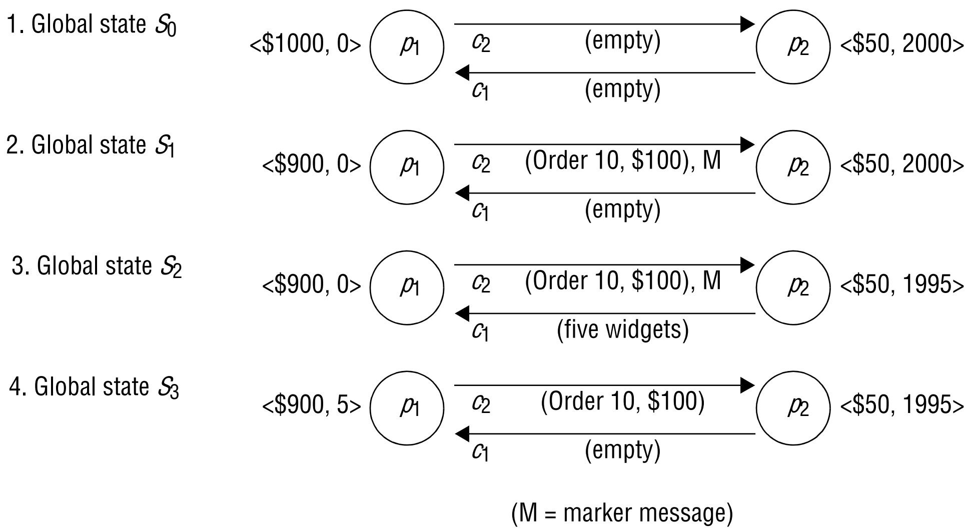


Figure 14.12 The execution of the processes in Figure 14.11



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.

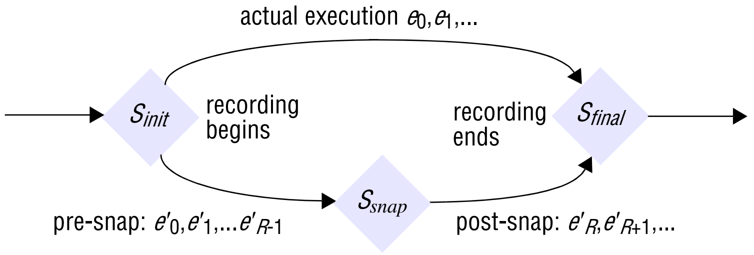
Now process p_1 receives p_2's message (five widgets), and p_2 receives the marker. Following the marker receiving rule, p_2 records its state as <$50, 1995> and that of channel c_2 as the empty sequence. Following the marker sending rule, it sends a marker message over c_1.

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


actual execution $e_0, e_1, ...$
S_{init}
recording begins
recording ends
S_{final}
S_{snap}
pre-snap: $e'_0, e'_1, ...e'_{R-1}$
post-snap: $e'_R, e'_{R+1}, ...$

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 $m_1, m_2 ..., m_H$ ($H \geq 1$), 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 $m_1, m_2 ..., m_H$. 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.

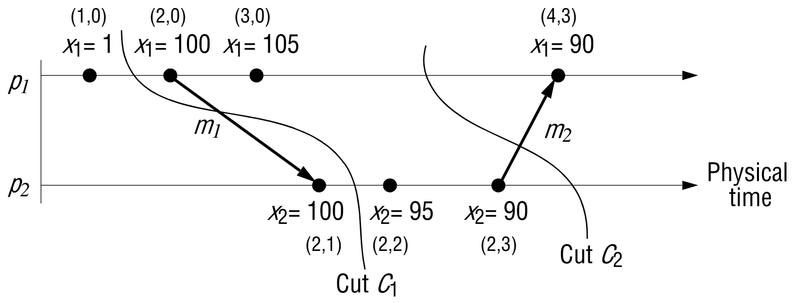
We now describe: how the process states are collected; how the monitor extracts consistent global states; and how the monitor evaluates *possibly* ϕ and *definitely* ϕ in both asynchronous and synchronous systems.

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 $|x_i - x_j| \le \delta$. ($i, j = 1, 2, \ldots, N$), 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, $f \rightarrow e \Rightarrow f \in C$.

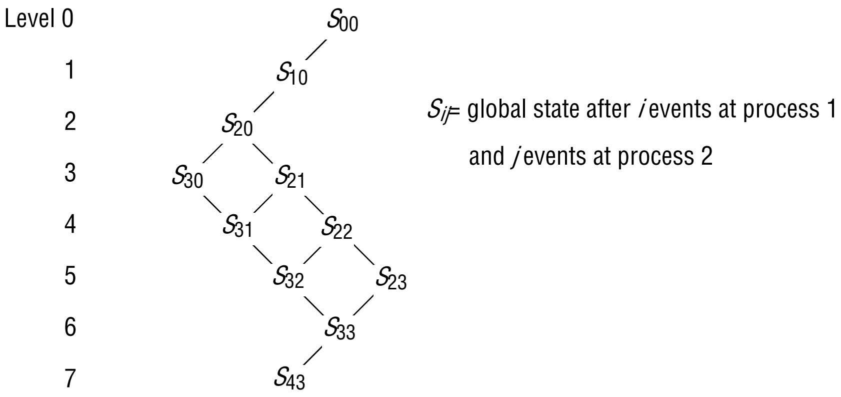
For example, Figure 14.14 shows two processes p_1 and p_2 with variables x_1 and x_2, respectively. The events shown on the timelines (with vector timestamps) are adjustments to the values of the two variables. Initially, $x_1 = x_2 = 0$. The requirement is $|x_1 - x_2| \le 50$. The processes make adjustments 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 $|x_1 - x_2| \le 50$. 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 $\phi$ for global history $H$ of $N$ processes
   $L := 0$;
   $States := \{ (s_1^0, s_2^0, \ldots, s_N^0) \}$;
   while ($\phi(S) = False$ for all $S \in States$)
       $L := L + 1$;
       $Reachable := \{ S' : S'$ reachable in $H$ from some $S \in States \wedge level(S') = L \}$;
       $States := Reachable$
   end while
   output "possibly $\phi$";

2. Evaluating definitely $\phi$ for global history $H$ of $N$ processes
   $L := 0$;
   if ($\phi(s_1^0, s_2^0, \ldots, s_N^0)$) then $States := \{\}$ else $States := \{ (s_1^0, s_2^0, \ldots, s_N^0) \}$;
   while ($States \neq \{\}$)
       $L := L + 1$;
       $Reachable := \{ S' : S'$ reachable in $H$ from some $S \in States \wedge level(S') = L \}$;
       $States := \{ S \in Reachable : \phi(S) = False \}$
   end while
   output "definitely $\phi$";
```

14.6.3 Evaluating possibly ϕ

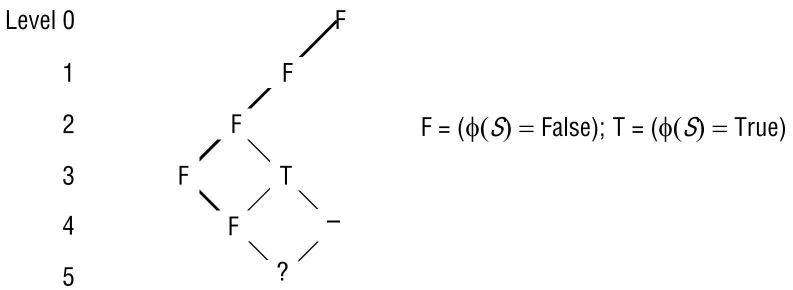
To evaluate *possibly* ϕ, the monitor must traverse the lattice of reachable states, starting from the initial state $(s_1^0, s_2^0 \ldots s_N^0)$. The algorithm is shown in Figure 14.16. The algorithm assumes that the execution is infinite. It may easily be adapted for a finite execution.

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:

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

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 $i, j = 1, 2, \ldots, N$:

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

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:

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

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.