

# 10

## TIME AND GLOBAL STATES

- 10.1 Introduction
- 10.2 Clocks, events and process states
- 10.3 Synchronizing physical clocks
- 10.4 Logical time and logical clocks
- 10.5 Global states
- 10.6 Distributed debugging
- 10.7 Summary

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 shall 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.

## 10.1 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 'e-commerce' transaction involves events at a merchant's computer and at a bank 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 12.6); checking the authenticity of a request sent to a server (a version of the Kerberos authentication protocol, discussed in Chapter 7, depends on loosely synchronized clocks); and eliminating the processing of duplicate updates (see, for example, Ladin *et al.* [1992]).

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.

Moreover, the relative order of two events can even be reversed for two different observers. But this cannot happen if one event could have caused 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 discredited. 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 non-existent for normal computers (unless one counts computers travelling in spaceships!). 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 that we can appeal to. 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.

## 10.2 Clocks, events and process states

Chapter 2 presented an introductory model of interaction between the processes within a distributed system. We shall 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  $\mathcal{P}$  of  $N$  processes  $p_i, i = 1, 2, \dots, N$ . Each process executes on a single processor, and the processors do not share memory (Chapter 17 considers the case of processes that share memory). Each process  $p_i$  in  $\mathcal{P}$  has a state  $s_i$  which, 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!

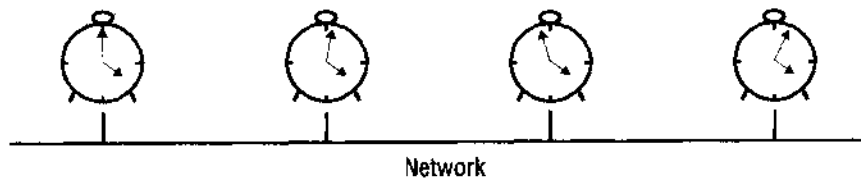
As each process  $p_i$  executes it takes a series of actions, each of which is either a message Send or Receive operation, or an operation that transforms  $p_i$ 's state – one that changes one or more of the values in  $s_i$ . In practice, we may choose to use a high-level description of the actions, according to the application. For example, if the processes in  $\mathcal{P}$  are engaged in an e-commerce 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 shall 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  $\rightarrow_i$ :

$$\text{history}(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$$

**Clocks** ♦ We have seen how to order the events at a process but not how to timestamp them – to assign to them a date and time of day. Computers each contain their own

**Figure 10.1** Skew between computer clocks in a distributed system

physical clock. These clocks are electronic devices that count oscillations occurring in a crystal at a definite frequency, and that 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, and 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 10.1). 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 ( $\text{Cs}^{133}$ ).

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 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 in the order of 0.1–10 milliseconds, depending on the station used. Signals received from GPS are accurate to about 1 microsecond. Computers with receivers attached can synchronize their clocks with these timing signals. Computers may also receive the time to an accuracy of a few milliseconds over a telephone line, from organizations such as the National Institute for Standards and Technology in the USA.

## 10.3 Synchronizing physical clocks

In order to know at what time of day events occur at the processes in our distributed system  $\mathcal{P}$  – 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  $i = 1, 2, \dots, N$  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  $i, j = 1, 2, \dots, N$ , 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  $\mathcal{P}$  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.

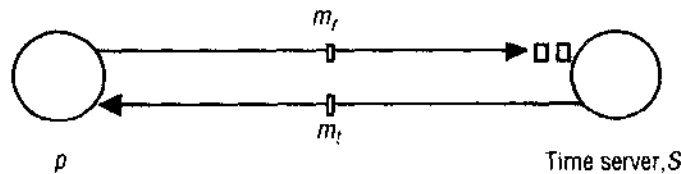
We can achieve monotonicity despite the fact that a clock is found to be running fast. We need only change the rate at which updates are made to the time as given to applications. This can be achieved in software without changing the rate at which the underlying hardware clock ticks – recall that  $C_i(t) = \alpha H_i(t) + \beta$ , where we are free to choose the values of  $\alpha$  and  $\beta$ .

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*. An example of an arbitrary failure is that of a clock with the 'Y2K bug', which breaks 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.

**Figure 10.2** Clock synchronization using a time server

### 10.3.1 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 to execute each step of a process (see Section 2.3.1).

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. 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 half-way 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.

### 10.3.2 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 10.2.

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 in 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 is the accuracy required, the smaller is 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**  $\diamond$  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 impossible temporarily. 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.

### 10.3.3 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 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 algorithm eliminates readings from faulty clocks. Such clocks could have a significant adverse effect if an ordinary average was taken. The master takes a *fault-tolerant average*. That is, a subset of clocks is chosen 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 rate was measured to be less than  $2 \times 10^{-5}$ , and the maximum round-trip time was taken to be 10 milliseconds.

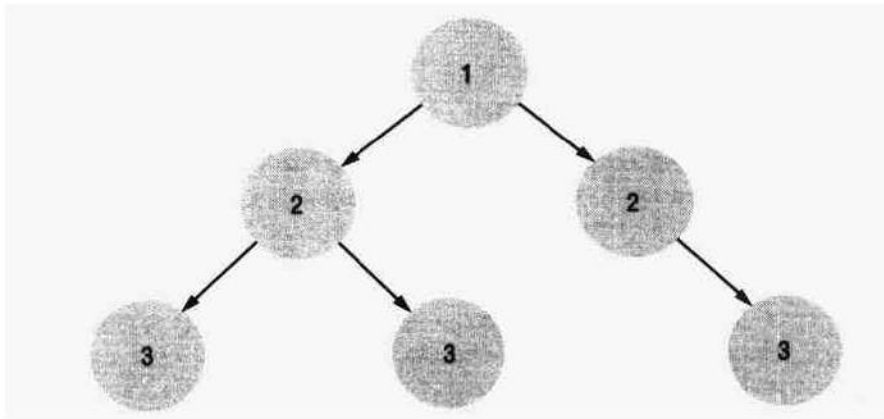
Should the master fail, then another can be elected to take over and function exactly as its predecessor. Section 11.3 discusses some general-purpose election algorithms. Note that these are not guaranteed to elect a new master in bounded time – and so the difference between two clocks would be unbounded if they were used.

### 10.3.4 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:* Despite the large and variable message delays 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 10.3** An example synchronization subnet in an NTP implementation

Note: 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.

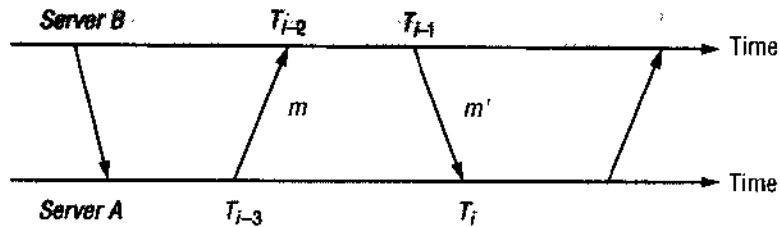
*To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers:* The service is designed to scale to large numbers of clients and servers.

*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 10.3), 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 10.4** 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 above. 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, which 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 10.4 for the messages  $m$  and  $m'$  sent between servers A and B. Note that in symmetric mode, unlike Cristian's algorithm described above, 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:

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

Also

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

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.

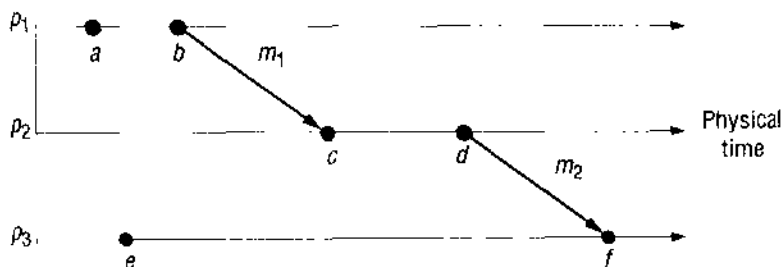
NTP servers apply a data filtering algorithm to successive pairs  $\langle o_i, d_i \rangle$ , which estimates the offset  $o$  and calculates the quality of this estimate as a statistical quantity called the *filter dispersion*. A relatively high filter dispersion represents relatively unreliable data. The eight most recent pairs  $\langle o_i, d_i \rangle$  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 in the order of tens of milliseconds over Internet paths, and one millisecond on LANs.

**Figure 10.5** Events occurring at three processes

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

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

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

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

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

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

Thus, if  $e$  and  $e'$  are events, and if  $e \rightarrow e'$ , then we can find a series of events  $e_1, e_2, \dots, e_n$  occurring at one or more processes such that  $e = e_1$  and  $e' = e_n$ , and for  $i = 1, 2, \dots, N - 1$  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  $e_1, e_2, \dots, e_n$  need not be unique.

The relation  $\rightarrow$  is illustrated for the case of three processes  $p_1, p_2$  and  $p_3$  in Figure 10.5. It can be seen that  $a \rightarrow b$ , since the events occur in this order at process  $p_1$  ( $a \rightarrow_1 b$ ) 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 10.5 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, then 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 bears no specific relation to the first message. No actual causality has been involved, but the relation  $\rightarrow$  would order these events.

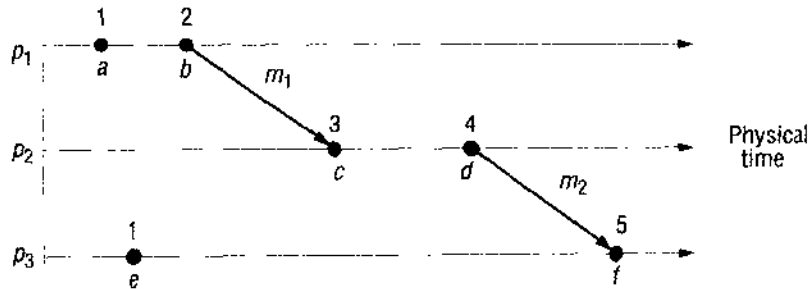
**Logical clocks**  $\diamond$  Lamport invented a simple mechanism by which the happened-before ordering can be captured numerically, called a *logical clock*. A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. 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.

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  $\text{receive}(m)$ .

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 10.6 we illustrate the use of logical clocks for the example given in Figure 10.5. 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  $b \parallel e$ .

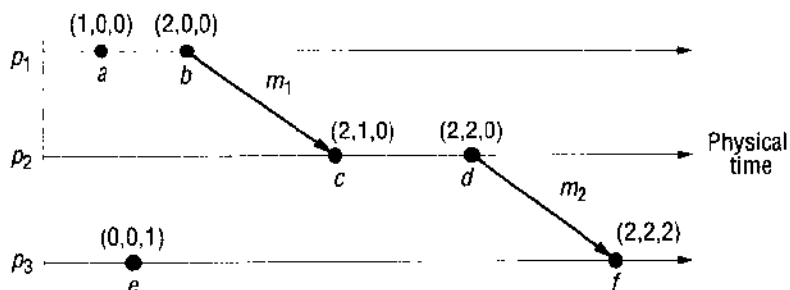
**Figure 10.6** Lamport timestamps for the events shown in Figure 10.5.

**Totally ordered logical clocks**  $\diamond$  Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps. However, we can create a total order on 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**  $\diamond$  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 as follows:

- VC1: Initially,  $V_i[j] = 0$ , for  $i, j = 1, 2, \dots, N$ .
- 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  $j = 1, 2, \dots, N$ . Taking the component-wise maximum of two vector timestamps in this way is known as a *merge operation*.

For a vector clock  $V_i$ ,  $V_i[i]$  is the number of events that  $p_i$  has timestamped, and  $V_i[j]$  ( $j \neq i$ ) is the number of events that have occurred at  $p_j$  that  $p_i$  has potentially been affected by. (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 10.7** Vector timestamps for the events shown in Figure 10.5

We may compare vector timestamps as follows:

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

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

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

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  $e \rightarrow e'$ .

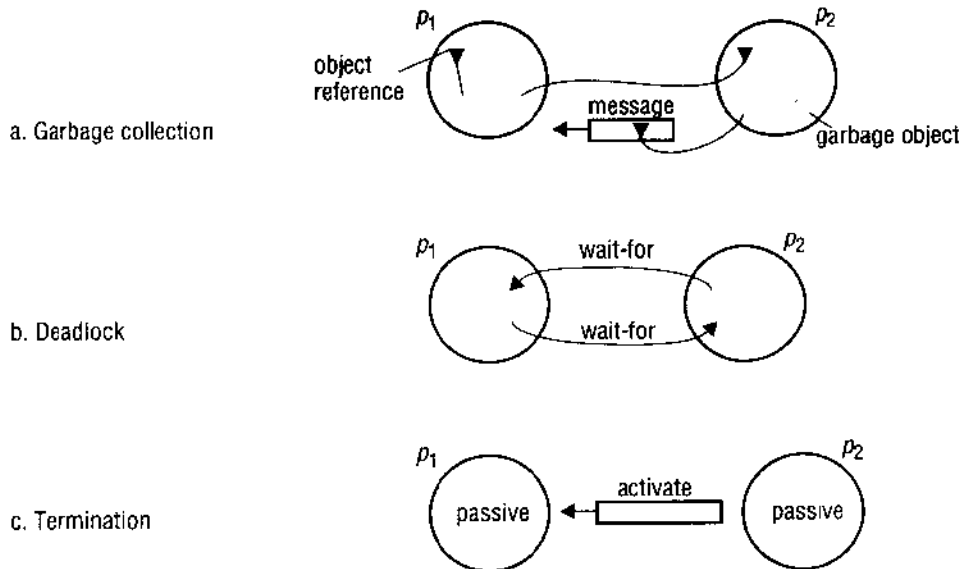
Figure 10.7 shows the vector timestamps of the events of Figure 10.5. It can be seen, for example, that  $V(a) < V(f)$ , which reflects the fact that  $a \rightarrow f$ . Similarly, we can tell when two events are concurrent by comparing their timestamps. For example, that  $c \parallel e$  can be seen from the facts that neither  $V(c) \leq V(e)$  nor  $V(e) \leq V(c)$ .

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

## 10.5 Global states

In this and the next section we shall 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 10.8** 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 10.8a, 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 10.8b shows that each of processes  $p_1$  and  $p_2$  waits for a message from the other, so this system will never make progress.

*Distributed termination detection:* The problem here is 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 10.8c).

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 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, Smith has written an application in which each process  $p_i$  contains a variable  $x_i$  ( $i = 1, 2, \dots, N$ ). The variables change as the program executes, but they are required always to be within a value  $\delta$  of one another. Unfortunately, there is a bug in the program, and she 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.

### 10.5.1 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 first introduce some definitions.

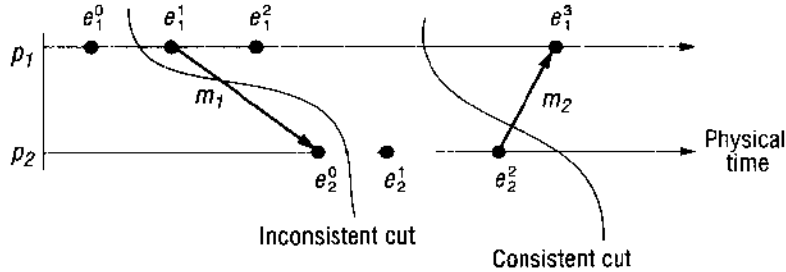
Let us return to our general system  $\mathcal{P}$  of  $N$  processes  $p_i$  ( $i = 1, 2, \dots, N$ ), whose execution we wish to study. We said above that a series of events occurs at each process, and that we may characterize the execution of each process by its history:

$$\text{history}(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$$

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

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

Figure 10.9 Cuts



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

In principle, we can record what occurred in  $\mathcal{P}$ '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  $k$ th 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  $\mathcal{P}$  as the union of the individual process histories:

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

Mathematically, we can take any set of states of the individual processes to form a global state  $S = (s_1, s_2, \dots, s_N)$ . 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:

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

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 –  $e_i^{c_i}$  ( $i = 1, 2, \dots, N$ ). The set of events  $\{e_i^{c_i} : i = 1, 2, \dots, N\}$  is called the *frontier* of the cut.

Consider the events occurring at processes  $p_1$  and  $p_2$  shown in Figure 10.9. The figure shows two cuts, one with frontier  $\langle e_1^0, e_2^0 \rangle$  and another with frontier  $\langle e_1^2, e_2^2 \rangle$ . The leftmost cut is *inconsistent*. This is because at  $p_2$  it includes the receipt of the message  $m_1$ , but at  $p_1$  it does not include the sending of that message. This is showing an 'effect' without a 'cause'. The actual execution never was in a global state corresponding to the process states at that frontier, and we can in principle tell this by examining the  $\rightarrow$  relation between events. By contrast, the rightmost cut is *consistent*.

It includes both the sending and the receipt of message  $m_1$ . It includes 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:

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

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:

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots$$

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,  $\rightarrow_i$  ( $i = 1, 2, \dots, N$ ). 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.

## 10.5.2 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  $\{\text{True}, \text{False}\}$ . 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  $\alpha$  that is a predicate of the system's global state – for example,  $\alpha$  could be the property of being deadlocked. Let

**Figure 10.10** Chandy and Lamport's 'snapshot' algorithm

*Marker receiving rule for process  $p_i$*

On  $p_i$ 's receipt of a *marker* message 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  $c$ ).

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

### 10.5.3 The 'snapshot' algorithm of Chandy and Lamport

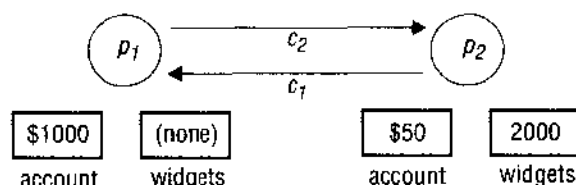
Chandy and Lamport [1985] describe a 'snapshot' algorithm for determining global states of distributed systems, which we now present. The goal of the algorithm is to record a set of process and channel states (a 'snapshot') for a set of processes  $p_i$  ( $i = 1, 2, \dots, N$ ) such that, even though 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);

**Figure 10.11** Two processes and their initial states

- 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,  $p_i$ 's *outgoing channels* 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.

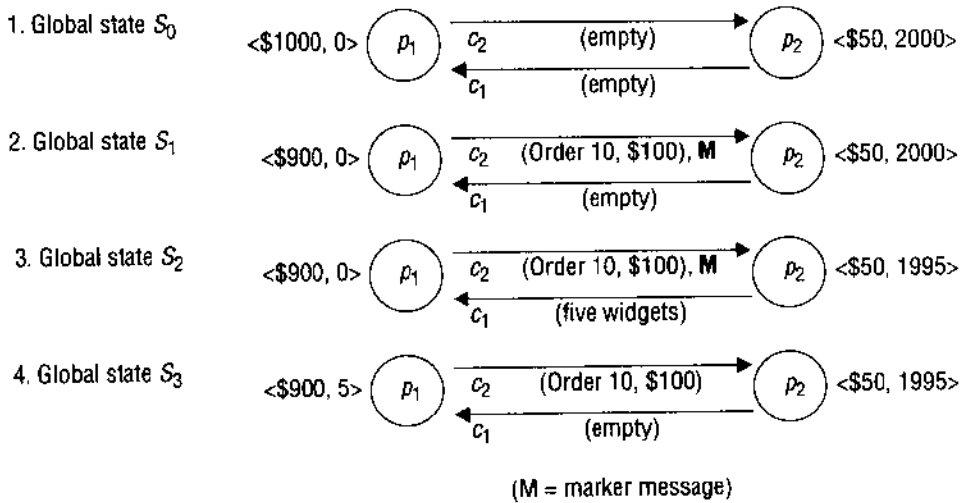
The algorithm proceeds through use of special *marker* messages, which are distinct from any other messages the processes send, and which the processes may send and receive while they proceed with their normal execution. The marker has a dual role: as a prompt for the receiver to save its own state,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

The algorithm is defined through two rules, the *marker receiving rule* and the *marker sending rule* (Figure 10.10). 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 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 non-existent 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 10.12** The execution of the processes in Figure 10.11

processes have the initial states shown in Figure 10.11. Process  $p_2$  has already received an order for five widgets, which it will shortly dispatch to  $p_1$ .

Figure 10.12 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  $p_1$ 's state is  $\langle \$1000, 0 \rangle$ . 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:  $(\text{Order } 10, \$100)$  over channel  $c_2$ . The system enters actual global state  $S_1$ .

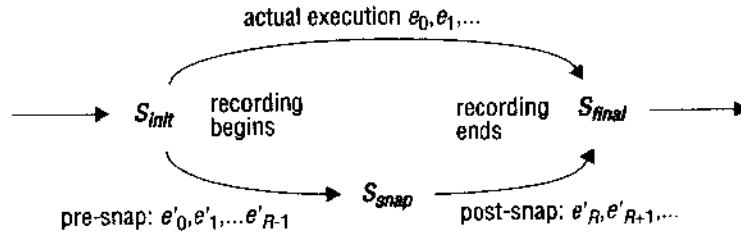
Before  $p_2$  receives the marker, it emits an application message (five widgets) over  $c_1$  in response to  $p_1$ 's previous order, yielding a new actual global state  $S_2$ .

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  $\langle \$50, 1995 \rangle$  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  $p_1: \langle \$1000, 0 \rangle$ ;  $p_2: \langle \$50, 1995 \rangle$ ;  $c_1: \langle (\text{five widgets}) \rangle$ ;  $c_2: \langle \rangle$ . Note that this state differs from all the global states through which the system actually passed.

**Termination of the snapshot algorithm**  $\diamond$  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 10.13** Reachability between states in the snapshot algorithm

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

**Characterising the observed state**  $\diamond$  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  $m_1, m_2, \dots, m_H$  ( $H \geq 1$ ), 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  $m_1, m_2, \dots, m_H$ . 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  $Sys = e_0, e_1, \dots$  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  $Sys$ ,  $Sys' = e'_0, e'_1, e'_2, \dots$  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 10.13 shows this situation, in which the upper linearization is  $Sys$ , and the lower linearization is  $Sys'$ .

We derive  $Sys'$  from  $Sys$  by first categorising 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  $e'_0, e'_1, e'_2, \dots, e'_{R-1}$  prior to all post-snap events  $e'_R, e'_{R+1}, e'_{R+2}, \dots$  with  $Sys'$  the resulting execution. For each process, the set of events in  $e'_0, e'_1, e'_2, \dots, e'_{R-1}$  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**  $\diamond$  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  $S_{init}$ .

## 10.6 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  $|x_i - x_j| \leq \delta$  ( $i, j = 1, 2, \dots, N$ ); 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 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 shall describe (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.