7. Is an identifier allowed to contain information on the entity it refers to?
8. Outline an efficient implementation of globally unique identifiers.
9. Give an example of how the closure mechanism for a URL could work.
10. Explain the difference between a hard link and a soft link in UNIX systems.
11. High-level name servers in DNS, that is, name servers implementing nodes in the DNS name space that are close to the root, generally do not support recursive name resolution. Can we expect much performance improvement if they did?
12. Explain how DNS can be used to implement a home-based approach to locating mobile hosts.
13. A special form of locating an entity is called anycasting, by which a service is identified by means of an IP address (see, for example, Partridge et al., 1993). Sending a request to an anycast address returns a response from a server implementing the service identified by that anycast address. Outline the implementation of an anycast service based on the hierarchical location service described in Sec. 4.2.4.
14. Considering that a two-tiered home-based approach is a specialization of a hierarchical location service, where is the root?
15. Suppose that it is known that a specific mobile entity will almost never move outside domain D, and if it does, it can be expected to return soon. How can this information be used to speed up the lookup operation in a hierarchical location service?
16. In a hierarchical location service with a depth of k, how many location records need to be updated at most when a mobile entity changes its location?
17. Consider an entity moving from location A to B, while passing several intermediate locations where it will reside for only a relatively short time. When arriving at B, it settles down for a while. Changing an address in a hierarchical location service may still take a relatively long time to complete, and should therefore be avoided when visiting an intermediate location. How can the entity be located at an intermediate location?
18. When passing a remote reference from process \( P_1 \) to \( P_2 \) in distributed reference counting, would it help to let \( P_1 \) increment the counter, instead of \( P_2 \)?
19. Make clear that weighted reference counting is more efficient than simple reference counting. Assume communication is reliable.
20. Is it possible in generation reference counting that an object is collected as garbage while there are still references, but which belong to a generation the object does not know of?
21. Is it possible in generation reference counting that an entry \( G[i] \) becomes less than 0?
22. In reference listing, if no response is received after sending a ping message to process \( P \), the process is removed from the object’s reference list. Is it always correct to remove the process?
23. Describe a very simple way to decide that the stabilization step in the tracing-based garbage collector of Lang et al. has been reached.

SYNCHRONIZATION

In the previous chapters, we have looked at processes and communication between processes. While communication is important, it is not the entire story. Closely related is how processes cooperate and synchronize with one another. Cooperation is partly supported by means of naming, which allows processes to at least share resources, or entities in general.

In this chapter, we mainly concentrate on how processes can synchronize. For example, it is important that multiple processes do not simultaneously access a shared resource, such as printer, but instead cooperate in granting each other temporary exclusive access. Another example is that multiple processes may sometimes need to agree on the ordering of events, such as whether message \( m1 \) from process \( P \) was sent before or after message \( m2 \) from process \( Q \).

As it turns out, synchronization in distributed systems is often much more difficult compared to synchronization in uniprocessor or multiprocessor systems. The problems and solutions that are discussed in this chapter are, by their nature, rather general, and occur in many different situations in distributed systems.

We start with a discussion of the issue of synchronization based on actual time, followed by synchronization in which only relative ordering matters rather than ordering in absolute time. We also discuss the notion of a distributed global state, and how, by synchronizing processes, such a state can be recorded.

In many cases, it is important that a group of processes can appoint one process as a coordinator, which can be done by means of election algorithms. We discuss various election algorithms in a separate section.
Two related topics regarding synchronization are mutual exclusion in distributed systems and distributed transactions. Distributed mutual exclusion allows shared resources to be protected against simultaneous access by multiple processes. Distributed transactions do something similar, but optimize access through advanced concurrency control mechanisms. Mutual exclusion and transactions are discussed in separate sections.

Distributed algorithms come in all sorts and flavors and have been developed for very different types of distributed systems. Many examples (and further references) can be found in (Andrews, 2000; Singhal and Shivaratri, 1994; Wu, 1998). A more formal approach to a wealth of algorithms is found in (Lynch, 1996).

5.1 CLOCK SYNCHRONIZATION

In a centralized system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it. If process A asks for the time, and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got. It will certainly not be lower. In a distributed system, achieving agreement on time is not trivial.

Just think, for a moment, about the implications of the lack of global time on the UNIX make program, as a single example. Normally, in UNIX, large programs are split up into multiple source files, so that a change to one source file only requires one file to be recompiled, not all the files. If a program consists of 100 files, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work.

The way make normally works is simple. When the programmer has finished changing all the source files, he starts make, which examines the times at which all the source and object files were last modified. If the source file input.c has time 2151 and the corresponding object file output.o has time 2150, make knows that input.c has been changed since input.o was created, and thus input.c must be recompiled. On the other hand, if output.c has time 2144 and output.o has time 2145, no compilation is needed here. Thus make goes through all the source files to find out which ones need to be recompiled and calls the compiler to recompile them.

Now imagine what could happen in a distributed system in which there were no global agreement on time. Suppose that output.o has time 2144 as above, and shortly thereafter output.c is modified but is assigned time 2143 because the clock on its machine is slightly behind, as shown in Fig. 5-1. Make will not call the compiler. The resulting executable binary program will then contain a mixture of object files from the old sources and the new sources. It will probably crash and the programmer will get crazy trying to understand what is wrong with the code.

Since time is so basic to the way people think and the effect of not having all the clocks synchronized can be so dramatic, as we have just seen, it is fitting that we begin our study of synchronization with the simple question: Is it possible to synchronize all the clocks in a distributed system?

5.1.1 Physical Clocks

Nearly all computers have a circuit for keeping track of time. Despite the widespread use of the word “clock” to refer to these devices, they are not actually clocks in the usual sense. Timer is perhaps a better word. A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension. Associated with each crystal are two registers, a counter and a holding register. Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the timer is reloaded from the holding register. In this way, it is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency. Each interrupt is called one clock tick.

When the system is booted initially, it usually asks the user to enter the date and time, which is then converted to the number of ticks after some known starting date and stored in memory. Many computers have a special battery-backed up CMOS RAM so that the date and time need not be entered on subsequent boots. At every clock tick, the interrupt service procedure adds one to the time stored in memory. In this way, the (software) clock is kept up to date.

With a single computer and a single clock, it does not matter much if this clock is off by a small amount. Since all processes on the machine use the same clock, they will still be internally consistent. For example, if the file input.c has time 2151 and file input.o has time 2150, make will recompile the source file, even if the clock is off by 2 and the true times are 2153 and 2152, respectively.

All that really matters are the relative times.

As soon as multiple CPUs are introduced, each with its own clock, the situation changes. Although the frequency at which a crystal oscillator runs is usually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency. In practice, when a system has n computers,
all n crystals will run at slightly different rates, causing the (software) clocks gradually to get out of synch and give different values when read out. This difference in time values is called clock skew. As a consequence of this clock skew, programs that expect the time associated with a file, object, process, or message to be correct and independent of the machine on which it was generated (i.e., which clock it used) can fail, as we saw in the make example above.

In some systems (e.g., real-time systems), the actual clock time is important. For these systems external physical clocks are required. For reasons of efficiency and redundancy, multiple physical clocks are generally considered desirable, which yields two problems: (1) How do we synchronize them with real-world clocks, and (2) How do we synchronize the clocks with each other?

Before answering these questions, let us digress slightly to see how time is actually measured. It is not nearly as simple as one might think, especially when high accuracy is required. Since the invention of mechanical clocks in the 17th century, time has been measured astronomically. Every day, the sun appears to rise on the eastern horizon, climbs to a maximum height in the sky, and sinks in the west. The event of the sun’s reaching its highest apparent point in the sky is called the transit of the sun. This event occurs at about noon each day. The interval between two consecutive transits of the sun is called the solar day. Since there are 24 hours in a day, each containing 3600 seconds, the solar second is defined as exactly 1/86400th of a solar day. The geometry of the mean solar day calculation is shown in Fig. 5-2.

Figure 5-2. Computation of the mean solar day.

In the 1940s, it was established that the period of the earth’s rotation is not constant. The earth is slowing down due to tidal friction and atmospheric drag. Based on studies of growth patterns in ancient coral, geologists now believe that 300 million years ago there were about 400 days per year. The length of the year (the time for one trip around the sun) is not thought to have changed; the day has simply become longer. In addition to this long-term trend, short-term variations in the length of the day also occur, probably caused by turbulence deep in the earth’s core of molten iron. These revelations led astronomers to compute the length of the day by measuring a large number of days and taking the average before dividing by 86,400. The resulting quantity was called the mean solar second.

With the invention of the atomic clock in 1948, it became possible to measure time much more accurately, and independent of the wiggling and wobbling of the earth, by counting transitions of the cesium 133 atom. The physicists took over the job of timekeeping from the astronomers and defined the second to be the time it takes the cesium 133 atom to make exactly 9,192,631,770 transitions. The choice of 9,192,631,770 was made to make the atomic second equal to the mean solar second in the year of its introduction. Currently, about 50 laboratories around the world have cesium 133 clocks. Periodically, each laboratory tells the Bureau International de l’Heure (BIH) in Paris how many times its clock has ticked. The BIH averages these to produce International Atomic Time, which is abbreviated TAI. Thus TAI is just the mean number of ticks of the cesium 133 clocks since midnight on Jan. 1, 1958 (the beginning of time) divided by 9,192,631,770.

Although TAI is highly stable and available to anyone who wants to go to the trouble of buying a cesium clock, there is a serious problem with it; 86,400 TAI seconds is now about 3 msec less than a mean solar day (because the mean solar day is getting longer all the time). Using TAI for keeping time would mean that over the course of the years, noon would get earlier and earlier, until it would eventually occur in the wee hours of the morning. People might notice this and we could have the same kind of situation as occurred in 1582 when Pope Gregory XIII decreed that 10 days be omitted from the calendar. This event caused riots in the streets because landlords demanded a full month’s rent and bankers a full month’s interest, while employers refused to pay workers for the 10 days they did not work, to mention only a few of the conflicts. The Protestant countries, as a matter of principle, refused to have anything to do with papal decrees and did not accept the Gregorian calendar: for 170 years.

BIH solves the problem by introducing leap seconds whenever the discrepancy between TAI and solar time grows to 800 msec. The use of leap seconds is illustrated in Fig. 5-3. This correction gives rise to a time system based on constant TAI seconds but which stays in phase with the apparent motion of the sun. It is called Universal Coordinated Time, but is abbreviated as UTC. UTC is the basis of all modern civil timekeeping. It has essentially replaced the old standard, Greenwich Mean Time, which is astronomical time.

Most electric power companies base the timing of their 60-Hz or 50-Hz clocks on UTC, so when BIH announces a leap second, the power companies raise their frequency to 61 Hz or 51 Hz for 60 or 50 sec, to advance all the clocks
in their distribution area. Since 1 sec is a noticeable interval for a computer, an operating system that needs to keep accurate time over a period of years must have special software to account for leap seconds as they are announced (unless they use the power line for time, which is usually too crude). The total number of leap seconds introduced into UTC so far is about 30.

To provide UTC to people who need precise time, the National Institute of Standard Time (NIST) operates a shortwave radio station with call letters WWV from Fort Collins, Colorado. WWV broadcasts a short pulse at the start of each UTC second. The accuracy of WWV itself is about ±1 msec, but due to random atmospheric fluctuations that can affect the length of the signal path, in practice the accuracy is no better than ±10 msec. In England, the station MSF, operating from Rugby, Warwickshire, provides a similar service, as do stations in several other countries.

Several earth satellites also offer a UTC service. The Geostationary Environment Operational Satellite can provide UTC accurately to 0.5 msec, and some other satellites do even better.

Using either shortwave radio or satellite services requires an accurate knowledge of the relative position of the sender and receiver, in order to compensate for the signal propagation delay. Radio receivers for WWV, GEOS, and the other UTC sources are commercially available.

5.1.2 Clock Synchronization Algorithms

If one machine has a WWV receiver, the goal becomes keeping all the other machines synchronized to it. If no machines have WWV receivers, each machine keeps track of its own time, and the goal is to keep all the machines together as well as possible. Many algorithms have been proposed for doing this synchronization (e.g., Cristian, 1989; Drummond and Baboagli, 1989, and Kopetz and Ochsenreiter, 1987). A survey is given in (Ramaathan et al., 1990).

All the algorithms have the same underlying model of the system, which we will now describe. Each machine is assumed to have a timer that causes an interrupt at times a second. When this timer goes off, the interrupt handler adds 1 to a software clock that keeps track of the number of ticks (interrupts) since some agreed-upon time in the past. Let us call the value of this clock $C$. More specifically, when the UTC time is $t$, the value of the clock on machine $p$ is $C_p(t)$. In a perfect world, we would have $C_p(t) = t$ for all $p$ and all $t$. In other words, $dC/dt$ ideally should be 1.

Real timers do not interrupt exactly $H$ times a second. Theoretically, a timer with $H = 60$ should generate 216,000 ticks per hour. In practice, the relative error obtainable with modern timer chips is about $10^{-5}$, meaning that a particular machine can get a value in the range 215,998 to 216,002 ticks per hour. More precisely, if there exists some constant $\rho$ such that

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

the timer can be said to be working within its specification. The constant $\rho$ is specified by the manufacturer and is known as the maximum drift rate. Slow, perfect, and fast clocks are shown in Fig. 5-4.

![Figure 5-4. The relation between clock time and UTC when clocks tick at different rates.](image)

If two clocks are drifting from UTC in the opposite direction, at a time $\Delta t$ after they were synchronized, they may be as much as $2\rho \Delta t$ apart. If the operating system designers want to guarantee that no two clocks ever differ by more than $\delta$, clocks must be resynchronized (in software) at least every $\delta/2\rho$ seconds. The various algorithms differ in precisely how this resynchronization is done.

Cristian’s Algorithm

Let us start with an algorithm that is well suited to systems in which one machine has a WWV receiver and the goal is to have all the other machines stay synchronized with it. Let us call the machine with the WWV receiver a time server. Our algorithm is based on the work of Cristian (1989) and prior work. Periodically, certainly no more than every $\delta/2\rho$ seconds, each machine sends a
message to the time server asking for the current time. That machine responds as fast as it can with a message containing its current time, \( C_{UTC} \), as shown in Fig. 5-5.

![Figure 5-5. Getting the current time from a time server.](image)

As a first approximation, when the sender gets the reply, it can just set its clock to \( C_{UTC} \). However, this algorithm has two problems, one major and one minor. The major problem is that time must never run backward. If the sender’s clock is fast, \( C_{UTC} \) will be smaller than the sender’s current value of \( C \). Just taking over \( C_{UTC} \) could cause serious problems such as an object file compiled just after the clock change having a time earlier than the source which was modified just before the clock change.

Such a change must be introduced gradually. One way is as follows. Suppose that the timer is set to generate 100 interrupts per second. Normally, each interrupt would add 10 msec to the time. When slowing down, the interrupt routine adds only 10 msec each time until the correction has been made. Similarly, the clock can be advanced gradually by adding 11 msec at each interrupt instead of jumping it forward all at once.

The minor problem is that it takes a nonzero amount of time for the time server’s reply to get back to the sender. Worse yet, this delay may be large and vary with the network load. Cristian’s way of dealing with it is to attempt to measure it. It is simple enough for the sender to record accurately the interval between sending the request to the time server and the arrival of the reply. Both the starting time, \( T_0 \), and the ending time, \( T_1 \), are measured using the same clock, so the interval will be relatively accurate even if the sender’s clock is off from UTC by a substantial amount.

In the absence of any other information, the best estimate of the message propagation time is \( (T_1 - T_0)/2 \). When the reply comes in, the value in the message can be increased by this amount to give an estimate of the server’s current time. If the theoretical minimum propagation time is known, other properties of the time estimate can be calculated.

This estimate can be improved if it is known approximately how long it takes the time server to handle the interrupt and process the incoming message. Let us call the interrupt handling time \( I \). Then the amount of the interval from \( T_0 \) to \( T_1 \) that was devoted to message propagation is \( T_1 - T_0 - I \), so the best estimate of the one-way propagation time is half this. Systems do exist in which messages from \( A \) to \( B \) systematically take a different route than messages from \( B \) to \( A \), and thus have a different propagation time, but we will not consider such systems here.

To improve the accuracy, Cristian suggested making not one measurement but a series of them. Any measurements in which \( T_1 - T_0 \) exceeds some threshold value are discarded as being victims of network congestion and thus are unreliable. The estimates derived from the remaining probes can then be averaged to get a better value. Alternatively, the message that came back fastest can be taken to be the most accurate since it presumably encountered the least traffic underway and therefore is the most representative of the pure propagation time.

The Berkeley Algorithm

In Cristian’s algorithm, the time server is passive. Other machines periodically ask it for the time. All it does is respond to their queries. In Berkeley UNIX, exactly the opposite approach is taken (Gusella and Zatti, 1989). Here the time server (actually, a time daemon) is active, polling every machine from time to time to ask what time it is there. Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved. This method is suitable for a system in which no machine has a WW receiver. The time daemon’s time must be set manually by the operator periodically. The method is illustrated in Fig. 5-6.

![Figure 5-6. (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.](image)

In Fig. 5-6(a), at 3:00, the time daemon tells the other machines its time and asks for theirs. In Fig. 5-6(b), they respond with how far ahead or behind the time
daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock [see Fig. 5-6(c)].

Averaging Algorithms

Both of the methods described above are highly centralized with the usual disadvantages. Decentralized algorithms are also known. One class of decentralized clock synchronization algorithms works by dividing time into fixed-length resynchronization intervals. The ith interval starts at $T_0 + iR$ and runs until $T_0 + (i+1)R$, where $T_0$ is an agreed-upon moment in the past, and $R$ is a system parameter. At the beginning of each interval, every machine broadcasts the current time according to its clock. Because the clocks on different machines do not run at exactly the same speed, these broadcasts will not happen precisely simultaneously.

After a machine broadcasts its time, it starts a local timer to collect all other broadcasts that arrive during some interval $S$. When all the broadcasts arrive, an algorithm is run to compute a new time from them. The simplest algorithm is just to average the values from all the other machines. A slight variation on this theme is first to discard the $m$ highest and $m$ lowest values, and average the rest. Discarding the extreme values can be regarded as self defense against up to $m$ faulty clocks sending out nonsense.

Another variation is to try to correct each message by adding to it an estimate of the propagation time from the source. This estimate can be made from the known topology of the network, or by timing how long it takes for probe messages to be echoed.

Additional clock synchronization algorithms are discussed in the literature (e.g., Lundelius-Welch and Lynch, 1998; Ramanathan et al., 1989; and Srikanth and Toueg, 1987). One of the most widely used algorithms in the Internet is the Network Time Protocol (NTP), described in (Mills, 1992). NTP is known to achieve (worldwide) accuracy in the range of 1–50 msec. It achieves this accuracy through the use of advanced clock synchronization algorithms; further improvements are described in (Mills, 1995).

Multiple External Time Sources

For systems in which extremely accurate synchronization with UTC is required, it is possible to equip the system with multiple receivers for WWV, GEOS, or other UTC sources. However, due to inherent inaccuracy in the time source itself as well as fluctuations in the signal path, the best the operating system can do is to establish a range (time interval) in which UTC falls. In general, the various time sources will produce different ranges, which thus requires that the machines attached to them come to a general agreement.

SEC. 5.1

CLOCK SYNCHRONIZATION

To reach this agreement, each processor with a UTC source can broadcast its range periodically, for instance, at the precise start of each UTC minute. None of the processors will get the time packets instantaneously. Worse yet, the delay between transmission and reception depends on the cable distance and number of routers that the packets have to traverse, which is different for each (UTC source, processor) pair. Other factors can also play a role, such as delays due to collisions when multiple machines try to transmit on an Ethernet at the same instant. Furthermore, if a processor is busy handling a previous packet, it may not even look at the time packet for a considerable number of milliseconds, introducing additional uncertainty into the time.

5.1.3 Use of Synchronized Clocks

In the past few years, the necessary hardware and software for synchronizing clocks on a wide scale (e.g., over the entire Internet) has become easily available. With this new technology, it is possible to keep millions of clocks synchronized to within a few milliseconds of UTC. New algorithms that utilize synchronized clocks are just starting to appear. One example, discussed in (Liskov, 1993), concerns how to enforce at-most-once message delivery to a server, even in the face of crashes. The traditional approach is for each message to bear a unique message number, and have each server store all the numbers of the messages it has seen so that it can detect new messages from retransmissions. The problem with this algorithm is that if a server crashes and reboots, it loses its table of message numbers. Also, for how long should message numbers be saved?

Using time, the algorithm can be modified as follows. Now, every message carries a connection identifier (chosen by the sender) and a timestamp. For each connection, the server records in a table the most recent timestamp it has seen. If any incoming message for a connection is lower than the timestamp stored for that connection, the message is rejected as a duplicate.

To make it possible to remove old timestamps, each server continuously maintains a global variable

$$G = \text{CurrentTime} - \text{MaxLifetime} - \text{MaxClockSkew}$$

where $MaxLifetime$ is the maximum time a message can live and $MaxClockSkew$ is how far from UTC the clock might be at worst. Any timestamp older than $G$ can safely be removed from the table because all messages that old have already died out. If an incoming message has an unknown connection identifier, it is accepted if its timestamp is more recent than $G$ and rejected if its timestamp is older than $G$ because anything that old surely is a duplicate. In effect, $G$ is a summary of the message numbers of all old messages. Every $\Delta T$, the current time is written to disk.

When a server crashes and then reboots, it reloads $G$ from the time stored on disk and increments it by the update period, $\Delta T$. Any incoming message with a
timestamp older than \( G \) is rejected as a duplicate. As a consequence, every message that might have been accepted before the crash is rejected. Some new messages may be incorrectly rejected, but under all conditions the algorithm maintains at-most-once semantics.

In addition to this algorithm, Liskov (1993) also describes how synchronized clocks can be used to achieve cache consistency, how to use time-out tickets in distributed system authentication, and how to handle commitment in atomic transactions. We will discuss some of these algorithms in later sections. As timer synchronization improves, no doubt new applications for it will be found.

5.2 LOGICAL CLOCKS

For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agrees with the real time as announced on the radio every hour. For running make, for example, it is adequate that all machines agree that it is 10:00, even if it is really 10:02. Thus for a certain class of algorithms, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time. For these algorithms, it is conventional to speak of the clocks as logical clocks.

In a classic paper, Lamport (1978) showed that although clock synchronization is possible, it need not be absolute. If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems. Furthermore, he pointed out that what usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order in which events occur. In the make example given in the previous section, what counts is whether \( \text{input.c} \) is older or newer than \( \text{input.o} \), not their absolute creation times.

In this section we will discuss Lamport’s algorithm, which synchronizes logical clocks. Also, we discuss an extension to Lamport’s approach, called vector timestamps. Lamport extended his own work in (Lamport, 1990).

5.2.1 Lamport timestamps

To synchronize logical clocks, Lamport defined a relation called happens-before. The expression \( a \rightarrow b \) is read “\( a \) happens before \( b \)” and means that all processes agree that first event \( a \) occurs, then afterward, event \( b \) occurs. The happens-before relation can be observed directly in two situations:

1. If \( a \) and \( b \) are events in the same process, and \( a \) occurs before \( b \), then \( a \rightarrow b \) is true.
2. If \( a \) is the event of a message being sent by one process, and \( b \) is the event of the message being received by another process, then \( a \rightarrow b \) is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite, nonzero amount of time to arrive.

Happens-before is a transitive relation, so if \( a \rightarrow b \) and \( b \rightarrow c \), then \( a \rightarrow c \). If two events, \( x \) and \( y \), happen in different processes that do not exchange messages (not even indirectly via third parties), then \( x \rightarrow y \) is not true, but neither is \( y \rightarrow x \). These events are said to be concurrent, which simply means that nothing can be said (or need be said) about when the events happened or which event happened first.

What we need is a way of measuring time such that for every event, \( a \), we can assign it a time value \( C(a) \) on which all processes agree. These time values must have the property that if \( a \rightarrow b \), then \( C(a) < C(b) \). To rephrase the conditions we stated earlier, if \( a \) and \( b \) are two events within the same process and \( a \) occurs before \( b \), then \( C(a) < C(b) \). Similarly, if \( a \) is the sending of a message by one process and \( b \) is the reception of that message by another process, then \( C(a) \) and \( C(b) \) must be assigned in such a way that everyone agrees on the values of \( C(a) \) and \( C(b) \) with \( C(a) < C(b) \). In addition, the clock time, \( C \), must always go forward (increasing), never backward (decreasing). Corrections to time can be made by adding a positive value, never by subtracting one.

Now let us look at the algorithm Lamport proposed for assigning times to events. Consider the three processes depicted in Fig. 5.7(a). The processes run on different machines, each with its own clock, running at its own speed. As can be seen from the figure, when the clock has ticked 6 times in process 0, it has ticked 8 times in process 1 and 10 times in process 2. Each clock runs at a constant rate, but the rates are different due to differences in the crystals.

At time 6, process 0 sends message \( A \) to process 1. How long this message takes to arrive depends on whose clock you believe. In any event, the clock in process 1 reads 16 when it arrives. If the message carries the starting time, 6, in it, process 1 will conclude that it took 10 ticks to make the journey. This value is certainly possible. According to this reasoning, message \( B \) from 1 to 2 takes 16 ticks, again a plausible value.

Now comes the fun part. Message \( C \) from 2 to 1 leaves at 60 and arrives at 56. Similarly, message \( D \) from 1 to 0 leaves at 64 and arrives at 54. These values are clearly impossible. It is this situation that must be prevented.

Lamport’s solution follows directly from the happens-before relation. Since \( C \) left at 60, it must arrive at 61 or later. Therefore, each message carries the sending time according to the sender’s clock. When a message arrives and the receiver’s clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time. In Fig. 5.7(b) we see that \( C \) now arrives at 61. Similarly, \( D \) arrives at 70.

With one small addition, this algorithm meets our requirements for global time. The addition is that between every two events, the clock must tick at least
once. If a process sends or receives two messages in quick succession, it must advance its clock by (at least) one tick in between them.

In some situations, an additional requirement is desirable: no two events ever occur at exactly the same time. To achieve this goal, we can attach the number of the process in which the event occurs to the low-order end of the time, separated by a decimal point. Thus if events happen in processes 1 and 2, both with time 40, the former becomes 40.1 and the latter becomes 40.2.

Using this method, we now have a way to assign time to all events in a distributed system subject to the following conditions:

1. If $a$ happens before $b$ in the same process, $C(a) < C(b)$.
2. If $a$ and $b$ represent the sending and receiving of a message, respectively, $C(a) < C(b)$.
3. For all distinctive events $a$ and $b$, $C(a) \neq C(b)$.

This algorithm gives us a way to provide a total ordering of all events in the system. Many other distributed algorithms need such an ordering to avoid ambiguities, so the algorithm is widely cited in the literature.

**Example: Totally-Ordered Multicasting**

As an application of Lamport timestamps, consider the situation in which a database has been replicated across several sites. For example, to improve query performance, a bank may place copies of an account database in two different cities, say New York and San Francisco. A query is always forwarded to the nearest copy. The price for a fast response to a query is partly paid in higher update costs, because each update operation must be carried out at each replica.

In fact, there is a more stringent requirement with respect to updates. Assume a customer in San Francisco wants to add $100 to his account, which currently contains $1,000. At the same time, a bank employee in New York initiates an update by which the customer’s account is to be increased with 1 percent interest. Both updates should be carried out at both copies of the database. However, due to communication delays in the underlying network, the updates may arrive in the order as shown in Fig. 5-8.

The customer’s update operation is performed in San Francisco before the interest update. In contrast, the copy of the account in the New York replica is first updated with the 1 percent interest, and after that with the $100 deposit. Consequently, the San Francisco database will record a total amount of $1,112, whereas the New York database records $1,110.

The problem that we are faced with is that the two update operations should have been performed in the same order at each copy. Although it makes a difference whether the deposit is $100 or 1%, the order in which the operations are performed is important for a consistent view of the database. The important issue is that both copies should be exactly the same. In general, situations such as these require a **totally-ordered multicast**, that is, a multicast operation by which all messages are delivered in the same order to each receiver. Lamport timestamps can be used to implement totally-ordered multicasts in a completely distributed fashion.

Consider a group of processes multicasting messages to each other. Each message is always timestamped with the current (logical) time of its sender. When a message is multicast, it is conceptually sent to the sender. In addition, we assume that messages from the same sender are received in the order they were sent, but if no messages are lost.

When a process receives a message, it is put into a local queue, ordered according to its timestamp. The receiver multicasts an acknowledgement to the
other processes. Note that if we follow Lamport’s algorithm for adjusting local clocks, the timestamp of the received message is lower than the timestamp of the acknowledgement.

The interesting aspect of this approach, is that all processes will eventually have the same copy of the local queue. Each message is multicast to all processes, including acknowledgements, and is assumed to be received by all processes. Recall also that we assume that messages are delivered in the order that they are sent. Each process puts a received message in its local queue according to the timestamp in that message. Lamport’s clocks ensure that no two messages have the same timestamp, but also that the timestamps reflect a consistent global ordering of events.

A process can deliver a queued message to the application it is running only when that message is at the head of the queue and has been acknowledged by each other process. At that point, the message is removed from the queue and handed over to the application; the associated acknowledgements can simply be removed. Because each process has the same copy of the queue, all messages are delivered in the same order everywhere. In other words, we have established totally-ordered multicasting.

5.2.2 Vector timestamps

Lamport timestamps lead to a situation where all events in a distributed system are totally ordered with the property that if event a happened before event b, then a will also be positioned in that ordering before b, that is, \( C(a) < C(b) \).

However, with Lamport timestamps, nothing can be said about the relationship between two events a and b by merely comparing their time values C(a) and C(b), respectively. In other words, if \( C(a) < C(b) \), then this does not necessarily imply that a indeed happened before b. Something more is needed for that.

To understand what is going on, consider a messaging system in which processes post articles and react to posted articles. One of the most popular examples of such a messaging system is the Internet’s electronic bulletin board service, network news (see, for example, Comer, 2000b). Users, and hence processes, join specific discussion groups. Postings within such a group, whether they are articles or reactions, are multicasted to all group members. To ensure that reactions are delivered after their associated postings, we may decide to use a totally-ordered multicasting scheme as described above. However, such a scheme does not imply that if message B is delivered after message A, that B is a reaction to what is posted by means of message A. In fact, the two may be completely independent. Totally-ordered multicasting is too strong in this case.

The problem is that Lamport timestamps do not capture causality. In our example, the receipt of an article always causally precedes the posting of a reaction. Consequently, if causal relationships are to be maintained within a group of processes, then the receipt of the reaction to an article should always follow the receipt of that article. No more, no less. If two articles or reactions are independent, their order of delivery should not matter at all.

Causality can be captured by means of vector timestamps. A vector timestamp \( VT(a) \) assigned to an event a has the property that if \( VT(a) < VT(b) \) for some event b, then event a is known to causally precede event b. Vector timestamps are constructed by letting each process \( P_i \) maintain a vector \( V_i \) with the following two properties:

1. \( V_i[j] \) is the number of events that have occurred so far at \( P_i \).
2. If \( V_i[j] = k \) then \( P_i \) knows that k events have occurred at \( P_j \).

The first property is maintained by incrementing \( V_i[i] \) at the occurrence of each new event that happens at process \( P_i \). The second property is maintained by piggybacking vectors along with messages that are sent. In particular, when \( P_i \) sends message m, it sends along its current vector as a timestamp \( vt \).

In this way, a receiver is informed about the number of events that have occurred at \( P_i \). More important, however, is that the receiver is told how many events at other processes have taken place before \( P_i \) sent message m. In other words, timestamp \( vt \) of m tells the receiver how many events in other processes have preceded m, and on which m may causally depend. When process \( P_i \) receives m, it adjusts its own vector by setting each entry \( V_i[k] \) to \( \max(V_i[k], vt[k]) \). The vector now reflects the number of messages that \( P_i \) must receive to have at least seen the same messages that preceded the sending of m. Hereafter, entry \( V_i[j] \) is incremented by 1 representing the event of receiving a next message (Raynal and Singhal, 1996).

With a slight adjustment, vector timestamps can be used to guarantee causal message delivery. Assume that \( V_i[i] \) is incremented only when process \( P_i \) sends a message. Consider again the example of an electronic bulletin board. When a process \( P_i \) posts an article, it multicasts that article as a message a with timestamp \( vt(a) \) set equal to \( V_i \).

Now suppose \( P_j \) posts a reaction to the article. It does this by multicasting a message r with a timestamp \( vt(r) \) set equal to \( V_j \). Note that \( vt(r) > vt(a) \). Assuming communication is reliable, both the message a containing the article, and the message r containing the reaction will eventually arrive at another process \( P_k \). As we have made no assumptions concerning the ordering of messages, message r may arrive at \( P_k \) before message a. When receiving r, \( P_k \) inspects timestamp \( vt(r) \) and will decide to postpone delivery until all messages that causally precede r have been received as well. In particular, message r is delivered only if the following conditions are met:

1. \( vt(r)[j] = V_i[j]+1 \)
2. \( vt(r)[i] \leq V_i[i] \) for all \( i \neq j \)
The first condition states that \( r \) is the next message that \( P_k \) was expecting from process \( P_j \). The second condition states that \( P_k \) has not seen any messages that were not seen by \( P_j \) when it sent message \( r \). In particular, this means that \( P_k \) has already seen message \( a \).

**A Note on Ordered Message Delivery**

Some middleware systems, notably ISIS and its successor Horus (Birman and van Renesse, 1994), provide support for totally-ordered and causally-ordered (reliable) multicasting. There has been some controversy whether such support should be provided as part of the message-communication layer, or whether applications should handle ordering (see, e.g., Cheriton and Skeen, 1993; and Birman, 1994).

There are two main problems with letting the communication layer deal with message ordering. First, because the communication layer cannot tell what a message actually contains, only potential causality is captured. For example, two messages from the same sender that are completely independent will always be marked as causally related by the communication layer. This approach is overly restrictive and may lead to efficiency problems.

A second problem is that not all causality may be captured. Consider again the news system. Suppose Alice posts an article. If she then phones Bob telling about what she just wrote, Bob may post another article as a reaction without having seen Alice’s posting on the news. In other words, there is a causality between Bob’s posting and that of Alice due to external communication. This causality is not captured by the network news system.

In essence, ordering issues, like many other application-specific communication issues, can be adequately solved by looking at the application for which communication is taking place. This is also known as the end-to-end argument in systems design (Saltzer et al., 1984). A drawback of having only application-level solutions, is that a developer is forced to concentrate on issues that do not immediately relate to the core functionality of the application. For example, ordering may not be the most important problem when developing a messaging system such as network news. In that case, having an underlying communication layer handle ordering may turn out to be convenient. We will come across the end-to-end argument a number of times, notably when dealing with security in distributed systems.

### 5.3 GLOBAL STATE

On many occasions, it is useful to know the global state in which a distributed system is currently residing. The global state of a distributed system consists of the local state of each process, together with the messages that are currently in transit, that is, that have been sent but not delivered. What exactly the local state of a process is depends on what we are interested in (Helary, 1989). In the case of a distributed database system, it may consist of only those records that form part of the database and exclude temporary records used for computations. In our example of tracing-based garbage collection as discussed in the previous chapter, the local state may consist of variables representing markings for those proxies, skeletons, and objects that are contained in the address space of a process.

Knowing the global state of a distributed system may be useful for many reasons. For example, when it is known that local computations have stopped and that there are no more messages in transit, the system has obviously entered a state in which no more progress can be made. By analyzing such a global state, it may be concluded that we are either dealing with a deadlock (see, for example, Bracha and Toueg, 1987), or that a distributed computation has correctly terminated. An example of how such an analysis can actually be done is discussed below.

A simple, straightforward way for recording the global state of a distributed system was proposed by Chandy and Lamport (1985) who introduced the notion of a distributed snapshot. A distributed snapshot reflects a state in which the distributed system might have been. An important property is that such a snapshot reflects a consistent global state. In particular, this means that if we have recorded that a process \( P \) has received a message from another process \( Q \), then we should also have recorded that process \( Q \) had actually sent that message. Otherwise, a snapshot will contain the recording of messages that have been received but never sent, which is obviously not what we want. The reverse condition (\( Q \) has sent a message that \( P \) has not yet received) is allowed, however.

The notion of a global state can be graphically represented by what is called a cut, as shown in Fig. 5.9. In Fig. 5.9(a), a consistent cut is shown by means of the dashed line crossing the time axis of the three processes \( P_1 \), \( P_2 \), and \( P_3 \). The cut represents the last event that has been recorded for each process. In this case, it can be readily verified that all recorded message receipts have a corresponding recorded send event. In contrast, Fig. 5.9(b) shows an inconsistent cut. The receipt of message \( m_2 \) by process \( P_3 \) has been recorded, but the snapshot contains no corresponding send event.

To simplify the explanation of the algorithm for taking a distributed snapshot, we assume that the distributed system can be represented as a collection of processes connected to each other through unidirectional point-to-point communication channels. For example, processes may first set up TCP connections before any further communication takes place.

Any process may initiate the algorithm. The initiating process, say \( P \), starts by recording its own local state. Then, it sends a marker along each of its outgoing channels, indicating that the receiver should participate in recording the global state.

When a process \( Q \) receives a marker through an incoming channel \( C \), its action depends on whether or not it has already saved its local state. If it has not
A process is said to have finished its part of the algorithm when it has received a marker along each of its incoming channels, and processed each one. At that point, its recorded local state, as well as the state it recorded for each incoming channel, can be collected and sent, for example, to the process that initiated the snapshot. The latter can then subsequently analyze the current state. Note that, meanwhile, the distributed system as a whole can continue to run normally.

It should be noted that because any process can initiate the algorithm, the construction of several snapshots may be in progress at the same time. For this reason, a marker is tagged with the identifier (and possibly also a version number) of the process that initiated the snapshot. Only after a process has received that marker through each of its incoming channels, can it finish its part in the construction of the marker’s associated snapshot.

Example: Termination Detection

As an application of taking a snapshot, consider detecting the termination of a distributed computation. If a process $Q$ receives the marker requesting a snapshot for the first time, it considers the process that sent that marker as its predecessor. When $Q$ completes its part of the snapshot, it sends its predecessor a DONE message. By recursion, when the initiator of the distributed snapshot has received a DONE message from all its successors, it knows that the snapshot has been completely taken.

However, a snapshot may show a global state in which messages are still in transit. In particular, suppose a process records that it had received messages along one of its incoming channels between the point where it had recorded its local state, and the point where it received the marker through that channel. Then, clearly, we cannot conclude that the distributed computation is completed, for those messages may have generated other messages that are not part of the snapshot.

What is needed is a snapshot in which all channels are empty. The following is a simple modification to the algorithm described above. When a process $Q$ finishes its part of the snapshot, it either returns a DONE message to its predecessor, or a CONTINUE message. A DONE message is returned only when the following two conditions are met:

1. All of $Q$’s successors have returned a DONE message.
2. $Q$ has not received any message between the point it recorded its state, and the point it had received the marker along each of its incoming channels.

In all other cases $Q$ sends a CONTINUE message to its predecessor.
Eventually, the original initiator of the snapshot, say process $P$, will either receive a \textit{CONTINUE} message, or only \textit{DONE} messages from its successors. When only \textit{DONE} messages are received, it is known that no regular messages are in transit, and thus the computation has terminated. Otherwise, process $P$ initiates another snapshot, and continues to do so until only \textit{DONE} messages are eventually returned.

Numerous other solutions to termination detection as discussed in this section have been developed. See (Andrews, 2000; and Singhal and Shivaratri, 1994) for further examples and references. An overview and comparison of different solutions can also be found in (Mattern, 1987; and Raynal, 1988).

5.4 \textbf{ELECTION ALGORITHMS}

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. In this section we will look at algorithms for electing a coordinator (using this as a generic name for the special process).

If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process has a unique number, for example, its network address (for simplicity, we will assume one process per machine). In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator. The algorithms differ in the way they do the location.

Furthermore, we also assume that every process knows the process number of every other process. What the processes do not know is which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be. Various algorithms are known, for example, (Fredrickson and Lynch, 1987; Garcia-Molina, 1982; and Singh and Kurose, 1994).

5.4.1 The Bully Algorithm

As a first example, consider the \textbf{bully algorithm} devised by Garcia-Molina (1982) When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, $P$, holds an election as follows:

1. $P$ sends an \textit{ELECTION} message to all processes with higher numbers.
2. If no one responds, $P$ wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. $P$'s job is done.

At any moment, a process can get an \textit{ELECTION} message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an \textit{OK} message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

In Fig. 5.11 we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends \textit{ELECTION} messages to all the processes higher than it, namely, 5, 6, and 7, as shown in Fig. 5.11(a). Processes 5 and 6 both respond with \textit{OK}, as shown in Fig. 5.11(b). Upon getting the first of these responses, 4 knows that its job is over. It knows that one of these bigwigs will take over and become coordinator. It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).

In Fig. 5.11(c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In Fig. 5.11(d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a \textit{COORDINATOR} message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of 7 is handled and the work can continue.

If process 7 is ever restarted, it will just send all the others a \textit{COORDINATOR} message and bully them into submission.

5.4.2 A Ring Algorithm

Another election algorithm is based on the use of a ring. Unlike some ring algorithms, this one does not use a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an \textit{ELECTION} message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as coordinator.
Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.

In Fig. 5-12 we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed. Each of these builds an ELECTION message and and each of them starts circulating its message, independent of the other one. Eventually, both messages will go all the way around, and both 2 and 5 will convert them into COORDINATOR messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at worst it consumes a little bandwidth, but this not considered wasteful.

Figure 5-12. Election algorithm using a ring.

5.5 MUTUAL EXCLUSION

Systems involving multiple processes are often most easily programmed using critical regions. When a process has to read or update certain shared data structures, it first enters a critical region to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time. In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems. For a taxonomy and bibliography of other methods, see (Raynal, 1991; and Singhal, 1993).

5.5.1 A Centralized Algorithm

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address). Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission, as shown in Fig. 5-13(a). When the reply arrives, the requesting process enters the critical region.

Now suppose that another process, 2 in Fig. 5-13(b), asks for permission to enter the same critical region. The coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. In Fig. 5-13(b), the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying “permission denied.” Either way, it queues the request from 2 for the time being and waits for more messages.
Figure 5.13. (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Fig. 5.13(c). The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. Either way, when it sees the grant, it can enter the critical region.

It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time into each critical region. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release). It can also be used for more general resource allocation rather than just managing critical regions.

The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

5.5.2 A Distributed Algorithm

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport’s 1978 paper on clock synchronization presented the first one. Ricart and Agrawala (1981) made it more efficient. In this section we will describe their method.

Ricart and Agrawala’s algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one actually happened first. Lamport’s algorithm presented in Sec. 5.2.1 is one way to achieve this ordering and can be used to provide timestamps for distributed mutual exclusion.

The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

1. If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends OK messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 5.14(a).

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends OK to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending OK. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 5.14(b). When it is finished, it removes the request from 2 from its queue and sends an OK message to process 2, allowing the latter to enter its critical region, as shown in Fig. 5.14(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

Note that the situation in Fig. 5.14 would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and
one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

Various minor improvements are possible to this algorithm. For example, getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission. Other improvements are also possible, such as proposed by Maekawa (1985), but these easily become more intricate.

Nevertheless, this algorithm is slower, more complicated, more expensive, and less robust that the original centralized one. Why bother studying it under these conditions? For one thing, it shows that a distributed algorithm is at least possible, something that was not obvious when we started. Also, by pointing out the shortcomings, we may stimulate future theoreticians to try to produce algorithms that are actually useful. Finally, like eating spinach and learning Latin in high school, some things are said to be good for you in some abstract way.

5.5.3 A Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in Fig. 5-15. Here we have a bus network, as shown in Fig. 5-15(a), (e.g., Ethernet), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig. 5-15(b). The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process 0 is given a token. The token circulates around the ring. It is passed from process k to process k+1 (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring.

The correctness of this algorithm is easy to see. Only one process has the token at any instant, so only one process can actually be in a critical region. Since
the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

As usual, this algorithm has problems too. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.

The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintains the current ring configuration.

5.5.4 A Comparison of the Three Algorithms

A brief comparison of the three mutual exclusion algorithms we have looked at is instructive. In Fig. 5-16 we have listed the algorithms and three key properties: the number of messages required for a process to enter and exit a critical region, the delay before entry can occur (assuming messages are passed sequentially over a network), and some problems associated with each algorithm.

The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request, a grant to enter, and a release to exit. The distributed algorithm requires \( n - 1 \) request messages, one to each of the other processes, and an additional \( n - 1 \) grant messages, for a total of \( 2(n - 1) \). (We assume that only point-to-point communication channels are used.) With the token ring algorithm, the number is variable. If every process constantly

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed</td>
<td>( 2(n - 1) )</td>
<td>( 2(n - 1) )</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to ( \infty )</td>
<td>0 to ( n - 1 )</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

Figure 5-16. A comparison of three mutual exclusion algorithms.

wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn. In Fig. 5-16 we show the former case. It takes only two message times to enter a critical region in the centralized case, but \( 2(n - 1) \) message times in the distributed case, assuming that messages are sent one after the other. For the token ring, the time varies from 0 (token just arrived) to \( n - 1 \) (token just departed).

Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they might do.

5.6 DISTRIBUTED TRANSACTIONS

A concept that is strongly related to mutual exclusion is that of a transaction. Mutual exclusion algorithms ensure that a shared resource such as a file, printer, and so on, is accessed by at most one process at a time. Transactions have in common that they also protect a shared resource against simultaneous access by several concurrent processes. In particular, transactions are used to protect shared data. However, transactions can do much more. In particular, they allow a process to access and modify multiple data items as a single atomic operation. If the process backs out halfway during the transaction, everything is restored to the point just before the transaction started. In this section we take a closer look at the concept of a transaction, and in particular concentrate on a transaction's capabilities for synchronizing multiple processes to protect shared data.
5.6.1 The Transaction Model

The original model of the transaction comes from the world of business. Suppose that the International Dingbat Corporation needs a batch of widgets. They approach a potential supplier, U.S. Widget, known far and wide for the quality of its widgets, for a quote on 100,000 10-cm purple widgets for June delivery. U.S. Widget makes a bid on 100,000 4-inch mauve widgets to be delivered in December. International Dingbat agrees to the price, but dislikes mauve, wants them by July, and insists on 10 cm for its international customers. U.S. Widget replies by offering 3 15/16 inch lavender widgets in October. After much further negotiation, they finally agree on 3 959/1024 inch violet widgets for delivery on August 15.

Up until this point, both parties are free to terminate the discussion, in which case the world returns to the state it was in before they started talking. However, once both companies have signed a contract, they are both legally bound to complete the sale, come what may. Thus until both parties have signed on the dotted line, either one can back out and it is as if nothing ever happened, but at the moment they both sign, they pass the point of no return and the transaction must be carried out.

The computer model is similar. One process announces that it wants to begin a transaction with one or more other processes. They can negotiate various options, create and delete entities, and perform operations for a while. Then the initiator announces that it wants all the others to commit themselves to the work done so far. If all of them agree, the results are made permanent. If one or more processes refuse (or crash before agreement), the situation reverts to exactly the state it was in before the transaction began, with all side effects on files, databases, and so on, magically wiped out. This all-or-nothing property eases the programmer’s job.

The use of transactions in computer systems goes back to the 1960s. Before there were disks and online databases, all files were kept on magnetic tape. Imagine a supermarket with an automated inventory system. Every day after closing, a computer run was made with two input tapes. The first one contained the complete inventory as of opening time that morning. The second one contained a list of the day’s updates: products sold to customers and products delivered by suppliers. The computer read both input tapes and produced a new master inventory tape, as shown in Fig. 5-17.

The great beauty of this scheme (although the people who actually had to live with it probably did not realize it at the time) is that if a run failed for any reason, all the tapes could be rewound and the job restarted with no harm done. Primitive as it was, the old magnetic tape system had the all-or-nothing property of a transaction.

Now look at a modern banking application that updates an online database in place. The customer calls up the bank using a PC with a modem with the intention of withdrawing money from one account and depositing it in another. The operation is performed in two steps:

1. Withdraw an amount \( a \) from account 1.
2. Deposit amount \( a \) to account 2.

If the telephone connection is broken after the first step but before the second one, the first account will have been debited but the second one will not have been credited. The money vanishes into thin air.

Being able to group these two operations in a transaction would solve the problem. Either both would be completed, or neither would be completed. A key issue is therefore rolling back to the initial state if the transaction fails to complete. What we really want is a way to rewind the database as we were able to do with the magnetic tapes. This ability is what a transaction has to offer.

Programming using transactions requires special primitives that must either be supplied by the underlying distributed system or by the language runtime system. Typical examples of transaction primitives are shown in Fig. 5-18. The exact list of primitives depends on what kinds of objects are being used in the transaction. In a mail system, there might be primitives to send, receive, and forward mail. In an accounting system, they might be quite different. READ and WRITE are typical examples, however. Ordinary statements, procedure calls, and so on, are also allowed inside a transaction.

BEGIN_TRANSACTION and END_TRANSACTION are used to delimit the scope of a transaction. The operations between them form the body of the transaction. Either all of these operations are executed or none are executed. These may be system calls, library procedures, or bracketing statements in a language, depending on the implementation.

Consider, as an example, the process of reserving a seat from White Plains, New York, to Malindi, Kenya, in an airline reservation system. One possible route is White Plains to JFK, JFK to Nairobi, and Nairobi to Malindi. In Fig. 5-19(a) we see reservations for these three flights being made as three different operations.
### Table

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN TRANSACTION</td>
<td>Mark the start of a transaction</td>
</tr>
<tr>
<td>END TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

#### Figure 5.18
Example primitives for transactions.

Now suppose that the first two flights have been reserved but the third one is booked solid. The transaction is aborted and the results of the first two bookings are undone—the airline database is restored to the value it had before the transaction started (see Fig. 5.19(b)). It is as though nothing happened.

BEGIN TRANSACTION reserve WP → JFK; reserve JFK → Nairobi; reserve Nairobi → Malindi; END TRANSACTION

#### Figure 5.19
(a) Transaction to reserve three flights commits. (b) Transaction aborts when third flight is unavailable.

The all-or-nothing property of transactions is one of the four characteristic properties that transactions have. More specifically, transactions are:

1. Atomic: To the outside world, the transaction happens indivisibly.
2. Consistent: The transaction does not violate system invariants.
3. Isolated: Concurrent transactions do not interfere with each other.
4. Durable: Once a transaction commits, the changes are permanent.

These properties are often referred to by their initial letters, ACID.

The first key property exhibited by all transactions is that they are atomic. This property ensures that each transaction either happens completely, or not at all, and if it happens, it happens in a single indivisible, instantaneous action. While a transaction is in progress, other processes (whether or not they are themselves involved in transactions) cannot see any of the intermediate states.

Suppose, for example, that some file is 10 bytes long when a transaction starts to append to it. If other processes read the file while the transaction is in progress, they see only the original 10 bytes, no matter how many bytes the transaction has appended. If the transaction commits successfully, the file grows instantaneously to its new size at the moment of commitment, with no intermediate states, no matter how many operations it took to get it there.

The second property says that they are consistent. What this means is that if the system has certain invariants that must always hold, if they held before the transaction, they will hold afterward too. For example, in a banking system, a key invariant is the law of conservation of money. After any internal transfer, the amount of money in the bank must be the same as it was before the transfer, but for a brief moment during the transaction, this invariant may be violated. The violation is not visible outside the transaction, however.

The third property says that transactions are isolated or serializable. What it means is that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as though all transactions ran sequentially in some (system dependent) order. We return to serializability below.

The fourth property says that transactions are durable. It refers to the fact that once a transaction commits, no matter what happens, the transaction goes forward and the results become permanent. No failure after the commit can undo the results or cause them to be lost. Durability is discussed extensively in Chap. 7.

#### 5.6.2 Classification of Transactions

So far, we have basically considered a transaction as a series of operations that satisfy the ACID properties. This type of transaction is also called a flat transaction. Flat transactions are the simplest type of transaction, and are most often used. However, flat transactions have a number of limitations that have led to alternative models. Below we discuss two important classes: nested transactions and distributed transactions. Other classes are discussed extensively in (Gray and Reuter, 1993).

### Some Limitations of Flat Transactions

The main limitation of flat transactions is that they do not allow partial results to be committed or aborted. In other words, the strength of the atomicity property of a flat transaction also is partly its weakness.

Consider again booking a flight from New York to Kenya, as shown in Fig. 5.19. Suppose that the entire trip was being sold as a relatively cheap single package deal, for which reason the three parts were grouped into a single transaction. At the time we discover that only the last part cannot be booked, it may be decided to still confirm the reservations of the first two parts. For example, we may have also found out that it was already hard enough to reserve the flight from JFK to Nairobi. Aborting the entire transaction would mean that we would have to make a second attempt to reserve a seat on that flight, which by then may fail.
Consequently, what we need in this case is to only partially commit the transaction. Flat transactions do not allow this.

As another example, consider a Web site in which a hyperlink is implemented as a bidirectional reference. In other words, if a Web page \( W_i \) contains a URL to a page \( W_j \), then \( W_j \) knows that \( W_i \) refers to it (see, e.g., Kappe, 1999). Now suppose a page \( W \) is moved to another location or replaced by another page. In that case, all hyperlinks to \( W \) should be updated, and preferably in a single atomic operation, or otherwise there will (temporarily) be dangling references to \( W \). In theory, a flat transaction can be used here. The transaction consists of updating \( W \) and a series of operations, where each operation updates a single Web page containing a hyperlink to \( W \).

The problem, however, is that such a transaction may take hours to complete. Not only may pages referring to \( W \) be scattered across the Internet, there may also be thousands of them that need to be updated. Doing each update as a separate transaction is no good, for in that case some Web pages may have correct links, while others will not. A possible solution in this case is to commit updates, but also to keep the old \( W \) for those pages whose link has not yet been updated.

**Nested Transactions**

Some of the limitations mentioned above can be solved by making use of nested transactions. A nested transaction is constructed from a number of subtransactions. The top-level transaction may fork off children that run in parallel with one another, on different machines, to gain performance or simplify programming. Each of these children may also execute one or more subtransactions, or fork off its own children.

Subtransactions give rise to a subtle, but important, problem. Imagine that a transaction starts several subtransactions in parallel, and one of these commits, making its results visible to the parent transaction. After further computation, the parent aborts, restoring the entire system to the state it had before the top-level transaction started. Consequently, the results of the subtransaction that committed must nevertheless be undone. Thus the permanence referred to above applies only to top-level transactions.

Since transactions can be nested arbitrarily deeply, considerable administration is needed to get everything right. The semantics are clear, however. When any transaction or subtransaction starts, it is conceptually given a private copy of all data in the entire system for it to manipulate as it wishes. If it aborts, its private universe just vanishes, as if it had never existed. If it commits, its private universe replaces the parent’s universe. Thus if a subtransaction commits and then later a new subtransaction is started, the second one sees the results produced by the first one. Likewise, if an enclosing (higher-level) transaction aborts, all its underlying subtransactions have to be aborted as well.

**Distributed Transactions**

Nested transactions are important in distributed systems, for they provide a natural way of distributing a transaction across multiple machines. However, nested transactions generally follow a logical division of the work of the original transaction. For example, the transaction by which three different flights needed to be reserved as shown in Fig. 5-19, can be logically split up into three subtransactions. Each of these subtransactions can be managed separately and independent of the other two.

However, a logical division of a nested transaction into subtransactions does not necessarily imply that all distribution is taken care of. For example, the subtransaction handling the seat reservation from New York to Nairobi, may still have to access two databases, one in each city. In this case, the subtransaction can no longer be subdivided into smaller subtransactions, because, logically, there are none; a reservation itself is an indivisible operation.

In this case, the situation that we are faced with is that of a (flat) subtransaction that operates on data that are distributed across multiple machines. Such transactions are known as distributed transactions. The difference between nested and distributed transactions is subtle, but important. A nested transaction is a transaction that is logically decomposed into a hierarchy of subtransactions. In contrast, a distributed transaction is logically a flat, indivisible transaction that operates on distributed data. This difference is illustrated in Fig. 5-20.

![Figure 5-20](image_url)

(a) A nested transaction. (b) A distributed transaction.

The main problem with distributed transactions is that separate distributed algorithms are needed to handle the locking of data and committing the entire transaction. Distributed locking is discussed below. A detailed presentation of distributed commit protocols is deferred until Chap. 7, where we discuss fault tolerance and recovery mechanisms, to which commit protocols belong.
5.6.3 Implementation

Transactions sound like a great idea, but how are they implemented? That is the question we will tackle in this section. To simplify matters, we consider transactions on a file system. It should be clear by now that if each process executing a transaction just updates the file it uses in place, transactions will not be atomic and changes will not vanish magically if the transaction aborts. Clearly, some other implementation method is required. Two methods are commonly used, which are discussed in turn below.

Private Workspace

Conceptually, when a process starts a transaction, it is given a private workspace containing all the files to which it has access. Until the transaction either commits or aborts, all of its reads and writes go to the private workspace, rather than directly to the file system. This observation leads directly to the first implementation method: actually giving a process a private workspace at the instant it begins a transaction.

The problem with this technique is that the cost of copying everything to a private workspace is prohibitive, but various optimizations make it feasible. The first optimization is based on the realization that when a process reads a file but does not modify it, there is no need for a private copy. It can just use the real file (unless it has been changed since the transaction started). Consequently, when a process starts a transaction, it is sufficient to create a private workspace for it that is empty except for a pointer back to its parent’s workspace. When the transaction is at the top level, the parent’s workspace is the file system. When the process opens a file for reading, the back pointers are followed until the file is located in the parent’s (or further ancestor’s) workspace.

When a file is opened for writing, it can be located in the same way as for reading, except that now it is first copied to the private workspace. However, a second optimization removes most of the copying, even here. Instead of copying the entire file, only the file’s index is copied into the private workspace. The index is the block of data associated with each file telling where its disk blocks are. In UNIX, the index is the inode. Using the private index, the file can be read in the usual way, since the disk addresses it contains are for the original disk blocks. However, when a file block is first modified, a copy of the block is made and the address of the copy inserted into the index, as shown in Fig. 5-21. The block can then be updated without affecting the original. Appended blocks are handled this way too. The new blocks are sometimes called shadow blocks.

As can be seen from Fig. 5-21(b), the process running the transaction sees the modified file, but all other processes continue to see the original file. In a more complex transaction, the private workspace might contain a large number of files instead of just one. If the transaction aborts, the private workspace is simply deleted and all the private blocks that it points to are put back on the free list. If the transaction commits, the private indices are moved into the parent’s workspace atomically, as shown in Fig. 5-21(c). The blocks that are no longer reachable are put onto the free list.

This scheme also works for distributed transactions. In that case, a process is started on each machine containing a file that is to be accessed as part of the transaction. Each process is given its own private workspace as described above. If the transaction aborts, all processes simply discard their private workspace. On the other hand, when the transaction commits, updates are propagated locally, at which point the transaction as a whole completes.

Writeahead Log

Another common method of implementing transactions is the writeahead log. With this method, files are actually modified in place, but before any block is changed, a record is written to a log telling which transaction is making the change, which file and block is being changed, and what the old and new values are. Only after the log has been written successfully is the change made to the file.

Fig. 5-22 gives an example of how the log works. In Fig. 5-22(a), we have a simple transaction that uses two shared variables (or other objects), $x$ and $y$, both initialized to 0. For each of the three statements inside the transaction, a log record is written before executing the statement, giving the old and new values. These values are separated by a slash in Fig. 5-22(b)-(d).
x = 0;
y = 0;
BEGIN TRANSACTION;
x = x + 1;
[y = 0/1] [x = 0/1] [x = 0/1]
y = y + 2;
[y = 0/2] [y = 0/2]
x = y * y;
[x = 1/4]
END TRANSACTION;

(a) (b) (c) (d)

Figure 5-22. (a) A transaction. (b)-(d) The log before each statement is executed.

If the transaction succeeds and is committed, a commit record is written to the log, but the data structures do not have to be changed as they have already been updated. If the transaction aborts, the log can be used to back up to the original state. Starting at the end and going backward, each log record is read and the changes described in it undone. This action is called a rollback.

Again, this scheme is also seen to work for distributed transactions. In that case, each machine keeps its own log of changes to its local file system. Rolling back in the case of an abort requires that each machine rolls back separately to restore the original files.

5.6.4 Concurrency Control

So far, we have explained the essence of achieving atomicity of transactions. Achieving atomicity (and durability) in the presence of failures is an important topic that we will discuss in Chap. 7, as it is related to more than only transactions. The properties of consistency and isolation are basically handled by properly controlling the execution of concurrent transactions, that is, transactions that are executed at the same time on shared data.

The goal of concurrency control is to allow several transactions to be executed simultaneously, but in such a way that the collection of data items (e.g., files or database records) that is being manipulated, is left in a consistent state. This consistency is achieved by giving transactions access to data items in a specific order whereby the final result is the same as if all transactions had run sequentially.

Concurrency control is best understood in terms of three different managers which are organized in a layered fashion as shown in Fig. 5-23. The bottom layer consists of a data manager that performs the actual read and write operations on data. The data manager is not concerned about which transaction it is performing a read or write. In fact, it knows nothing about transactions.

The middle layer consists of a scheduler and carries the main responsibility for properly controlling concurrency. It determines which transaction is allowed to pass a read or write operation to the data manager and at which time. It does so by scheduling individual read and write operations in such a way that isolation and consistency of transactions are met. Below, we discuss scheduling based on the use of locks, and scheduling based on the use of timestamps.

The highest layer contains the transaction manager, which is primarily responsible for guaranteeing atomicity of transactions. It processes transaction primitives by transforming them into scheduling requests for the scheduler.

The model shown in Fig. 5-23 can be adopted for the distributed case as shown in Fig. 5-24. Each site has its own scheduler and data manager, together responsible for ensuring that local data remain consistent. Each transaction is handled by a single transaction manager. The latter communicates with the scheduler of individual sites. Depending on the concurrency control algorithm, a scheduler may also communicate with remote data managers. We return to the distribution of concurrency control below.

Serializability

The main purpose of concurrency control algorithms is to guarantee that multiple transactions can be executed simultaneously while still being isolated at the same time. This means that the final result should be the same as if the transactions were executed one after the other in some specific order.

In Fig. 5-25(a)-(c) we have three transactions that are executed simultaneously by three separate processes. If they were to be run sequentially, the final value of x would be 1, 2, or 3, depending upon which one ran last (x could be a shared variable, a file, or some other kind of entity). In Fig. 5-25(d) we see various orders, called schedules, in which they might be interleaved. Schedule 1 is actually serialized. In other words, the transactions run strictly sequentially, so it meets the serializability condition by definition. Schedule 2 is not serialized, but is still legal because it results in a value for x that could have been achieved by
running the transactions strictly sequentially. The third one is illegal since it sets x to 5, something that no sequential order of the transactions could produce. It is up to the system to ensure that individual operations are interleaved correctly. By allowing the system the freedom to choose any ordering of the operations it wants to, provided that it gets the answer correct, we eliminate the need for programmers to do their own mutual exclusion, thus simplifying the programming.

BEGIN_TRANSACTION
x = 0;
END_TRANSACTION

BEGIN_TRANSACTION
x = x + 1;
END_TRANSACTION

BEGIN_TRANSACTION
x = x + 2;
END_TRANSACTION

BEGIN_TRANSACTION
x = x + 3;
END_TRANSACTION

(a)  
(b)  
(c)  

Figure 5-25. (a)–(c) Three transactions $T_1$, $T_2$, and $T_3$. (d) Possible schedules.

To understand schedules and concurrency control, it is not necessary to know exactly what is being computed. In other words, it does not matter whether the value of $x$ is incremented by 2 or 3. What does matter is that the value of $x$ is being changed. Consequently, we can represent transactions as a series of read and write operations on specific data items. For example, each of the three transactions $T_1$, $T_2$, and $T_3$ shown in Fig. 5-25(a)–(c), respectively, can be represented as the series

write($T_1$, x); read($T_1$, x); write($T_1$, x)

The whole idea behind concurrency control is to properly schedule conflicting operations. Two operations conflict if they operate on the same data item, and if at least one of them is a write operation. In a read-write conflict exactly one of the operations is a write. Otherwise, we are dealing with a write-write conflict. Note that it does not matter whether conflicting operations are from the same transaction or from different transactions. It is important to note that two read operations never conflict.

Concurrency control algorithms can generally be classified by looking at the way read and write operations are synchronized. Synchronization can take place either through mutual exclusion mechanisms on shared data (i.e., locking), or explicitly ordering operations using timestamps.

A further distinction can be made between pessimistic and optimistic concurrency control. Fundamental to pessimistic approaches is Murphy’s law: if something can go wrong, it will. In pessimistic approaches, operations are synchronized before they are carried out, meaning that conflicts are resolved before they are allowed to happen. In contrast, optimistic approaches are based on the idea that, in general, nothing will go wrong. Operations are therefore simply carried out and synchronization takes place at the end of a transaction. If at that point it turns out that conflicts occurred, one or more transactions are forced to abort. In the following pages, we study two pessimistic and one optimistic approach. An excellent overview of various mechanisms is given in (Bernstein and Goodman, 1981).

Two-Phase Locking

The oldest and most widely used concurrency control algorithm is locking. In the simplest form, when a process needs to read or write a data item as part of a transaction, it requests the scheduler to grant it a lock for that data item. Likewise, when a data item is no longer needed, the scheduler is requested to release the lock. The task of the scheduler is to grant and release locks in such a way that only valid schedules result. In other words, it needs to apply an algorithm that provides only serializable schedules. One such algorithm is two-phase locking.

In two-phase locking (2PL), which is illustrated in Fig. 5-26, the scheduler first acquires all the locks it needs during the growing phase, and then releases them during the shrinking phase. More specifically, the following three rules are obeyed, as explained in (Bernstein et al., 1987):
1. When the scheduler receives an operation \( \text{opert}(T,x) \) from the transaction manager, it tests whether that operation conflicts with any other operation for which it already granted a lock. If there is a conflict, operation \( \text{opert}(T,x) \) is delayed (and thus also transaction \( T \)). If there is no conflict, the scheduler grants a lock for data item \( x \), and passes the operation to the data manager.

2. The scheduler will never release a lock for data item \( x \), until the data manager acknowledges it has performed the operation for which the lock was set.

3. Once the scheduler has released a lock on behalf of a transaction \( T \), it will never grant another lock on behalf of \( T \), no matter for which data item \( T \) is requesting a lock. Any attempt by \( T \) to acquire another lock is a programming error that aborts \( T \).

It can be proven (Eswaran et al., 1976) that if all transactions use two-phase locking, all schedules formed by interleaving them are serializable. This is why two-phase locking is widely used.

In many systems, the shrinking phase does not take place until the transaction has finished running and has either committed or aborted, leading to the release of locks as shown in Fig. 5-27. This policy, called \textit{strict two-phase locking}, has two main advantages. First, a transaction always reads a value written by a committed transaction; therefore, one never has to abort a transaction because its calculations were based on a data item it should not have seen. Second, all lock acquisitions and releases can be handled by the system without the transaction being aware of them: locks are acquired whenever a data item is to be accessed and released when the transaction has finished. This policy eliminates \textit{cascaded aborts}: having to undo a committed transaction because it saw a data item it should not have seen.

Both two-phase locking and strict two-phase locking can lead to deadlocks. If two processes each try to acquire the same pair of locks but in the opposite order, a deadlock may result. The usual techniques apply here, such as acquiring all locks in some canonical order to prevent hold-and-wait cycles. Also possible is deadlock detection by maintaining an explicit graph of which process has which locks and wants which locks, and checking the graph for cycles. Finally, when it is known in advance that a lock will never be held longer than \( t \) sec, a timeout scheme can be used: if a lock remains continuously under the same ownership for longer than \( t \) sec, there must be a deadlock.

There are several ways that the basic two-phase locking scheme can be implemented in a distributed system. The assumption is that the data that is operated on are distributed across multiple machines. In \textit{centralized 2PL}, a single site is responsible for granting and releasing locks. Each transaction manager communicates with this centralized lock manager, from which it receives lock grants. When a lock has been granted, the transaction manager subsequently communicates directly with the data managers. Note that in this scheme, data items may also be replicated possibly across multiple machines. When the operation has completed, the transaction manager returns the lock to the lock manager.

In \textit{primary 2PL}, each data item is assigned a primary copy. The lock manager on that copy's machine is responsible for granting and releasing locks. Primary 2PL works essentially the same as centralized 2PL, except that locking has been distributed across multiple machines.

Finally, in \textit{distributed 2PL}, it is assumed that data may be replicated across multiple machines. In contrast to primary 2PL and centralized 2PL, the schedulers on each machine only take care that locks are granted and released, but also that the operation is forwarded to the (local) data manager. In this sense, distributed 2PL comes much closer to the basic 2PL scheme, but which is now executed at each site where the data reside.

A classical treatment of two-phase locking for database systems and concurrency control in general can be found in (Bernstein et al., 1987).
Pessimistic Timestamp Ordering

A completely different approach to concurrency control is to assign each transaction \( T \) a timestamp \( ts(T) \) at the moment it starts. Using Lamport’s algorithm, we can ensure that the timestamps are unique, which is important here. Every operation that is part of a transaction \( T \), is timestamped with \( ts(T) \). Furthermore, every data item \( x \) in the system has a read timestamp \( ts_{RP}(x) \) and a write timestamp \( ts_{WR}(x) \) associated with it. The read timestamp is set to the timestamp of the transaction that most recently read \( x \), whereas the write timestamp is that of the transaction that most recently changed \( x \). Using timestamp ordering, if two operations conflict, the data manager processes the one with the lowest timestamp first.

Now suppose that the scheduler receives an operation \( \text{read}(T,x) \) from transaction \( T \) with timestamp \( ts \) but that \( ts < ts_{RP}(x) \). In other words, it notices that a write operation on \( x \) has been performed after \( T \) started. In that case, transaction \( T \) is simply aborted. On the other hand, if \( ts > ts_{RP}(x) \), it is correct to let the read operation take place. In addition, \( ts_{RD}(x) \) is set to \( \max(ts, ts_{RD}(x)) \).

Likewise, assume the scheduler receives a write operation \( \text{write}(T,x) \) as part of transaction \( T \) with timestamp \( ts \). If \( ts < ts_{RD}(x) \), it can only abort transaction \( T \), because the current value of \( x \) has been read by a more recent transaction. Transaction \( T \) is simply too late. On the other hand, if \( ts > ts_{RD}(x) \), it is in order to change the value of \( x \), as no younger transaction has yet read it. Also, \( ts_{WR}(x) \) is set to \( \max(ts, ts_{WR}(x)) \).

To better understand timestamp ordering, consider the following example. Imagine that there are three transactions, \( T_1 \), \( T_2 \), and \( T_3 \). \( T_1 \) ran a long time ago, and used every data item needed by \( T_2 \) and \( T_3 \), so all their data items have read and write timestamps set to \( ts(T_1) \). Transactions \( T_2 \) and \( T_3 \) start concurrently, with \( ts(T_2) < ts(T_3) \).

Let us first consider \( T_2 \) writing a data item \( x \). Unless \( T_2 \) has snuck in already and committed, both \( ts_{RP}(x) \) and \( ts_{WR}(x) \) will have been set to \( ts(T_1) \), and thus less than \( ts(T_2) \). In Fig. 5.28(a) and (b) we see that \( ts(T_2) \) is larger than both \( ts_{RP}(x) \) and \( ts_{WR}(x) \), so the write is accepted and done tentatively. It will become permanent when \( T_2 \) commits. \( T_2 \)’s timestamp is now recorded in the data item as a tentative write.

In Fig. 5.28(c) and (d) \( T_2 \) is out of luck. \( T_3 \) has either read (c) or written (d) \( x \) and committed. \( T_3 \)’s transaction is aborted. However, it can apply for a new timestamp and start all over again.

Now look at reads. In Fig. 5.28(e), there is no conflict, so the read can happen immediately. In Fig. 5.28(f), some interpoler has gotten in there and is trying to write \( x \). The interpoler’s timestamp is lower than \( T_3 \)’s, so \( T_3 \) simply waits until the interpoler commits, at which time it can read the new file and continue.

In Fig. 5.28(g), \( T_3 \) has changed \( x \) and already committed. Again \( T_2 \) must abort. In Fig. 5.28(h), \( T_3 \) is in the process of changing \( x \), although it has not committed yet. Still, \( T_3 \) is too late and must abort.

Different properties than locking. When a transaction encounters a larger (later) timestamp, it aborts, whereas under the same circumstances with locking it would either wait or be able to proceed immediately. On the other hand, it is deadlock free, which is a big plus.

The basic timestamp ordering has several variants, notably conservative timestamp ordering and multiversion timestamp ordering. Details can be found in (Gray and Reuter, 1993; and Ozsu and Valduriez, 1999).

Optimistic Timestamp Ordering

A third approach to handling multiple transactions at the same time is optimistic concurrency control (Kung and Robinson, 1981). The idea behind this technique is surprisingly simple: just go ahead and do whatever you want to without paying attention to what anybody else is doing. If there is a problem, worry about it later. (Many politicians use this algorithm, too.) In practice, conflicts are relatively rare, so most of the time it works all right.

Although conflicts may be rare, they are not impossible, so some way is needed to handle them. What optimistic concurrency control does is keep track of which data items have been read and written. At the point of committing, it checks all other transactions to see if any of its items have been changed since the transaction started. If so, the transaction is aborted. If not, it is committed.

Optimistic concurrency control fits best with the implementation based on private workspaces. That way, each transaction changes its data privately, without interference from the others. At the end, the new data are either committed or released, leading to a relatively simple and straightforward scheme.
5.7 SUMMARY

Strongly related to communication between processes is the issue of how processes in distributed systems synchronize. Synchronization is all about doing the right thing at the right time. A problem in distributed systems, and computer networks in general, is that there is no notion of a globally shared clock. In other words, processes on different machines have their own idea of what time it is.

There are various ways to synchronize clocks in a distributed system, but all methods are essentially based on exchanging clock values, while taking into account the time it takes to send and receive messages. Variations in communication delays and the way those variations are dealt with, largely determine the accuracy of clock synchronization algorithms.

In many cases, knowing the absolute time is not necessary. What counts is that related events at different processes happen in the correct order. Lamport showed that by introducing a notion of logical clocks, it is possible for a collection of processes to reach global agreement on the correct ordering of events. In essence, each event e, such as sending or receiving a message, is assigned a globally unique logical timestamp C(e) such that when event a happened before b, C(a) < C(b). Lamport timestamps can be extended to vector timestamps: if C(a) < C(b), we even know that event a causally preceded b.

Because there is no notion of shared memory in a distributed system, it is often hard to determine exactly what a system's current state is. Determining the global state of a distributed system can be done by synchronizing all processes so that each collects its own local state, along with the messages that are currently in transit. The synchronization itself can be done without forcing processes to stop and collect their state. Instead, what is called a distributed snapshot, can be collected while the distributed system continues to operate.

Synchronization between processes often requires that one process acts as a coordinator. In those cases where the coordinator is not fixed, it is necessary that processes in a distributed computation decide on who is going to be that coordinator. Such a decision is taken by means of election algorithms. Election algorithms are primarily used in cases where the coordinator can crash.
10. Consider Fig. 5-13 again. Suppose that the coordinator crashes. Does this always bring the system down? If not, under what circumstances does this happen? Is there any way to avoid the problem and make the system able to tolerate coordinator crashes?

11. Ricart and Agrawala’s algorithm has the problem that if a process has crashed and does not reply to a request from another process to enter a critical region, the lack of response will be interpreted as denial of permission. We suggested that all requests be answered immediately, to make it easy to detect crashed processes. Are there any circumstances where even this method is insufficient? Discuss.

12. How do the entries in Fig. 5-16 change if we assume that the algorithms can be implemented on a LAN that supports hardware broadcasts?

13. A distributed system may have multiple, independent critical regions. Imagine that process 0 wants to enter critical region A and process 1 wants to enter critical region B. Can Ricart and Agrawala’s algorithm lead to deadlocks? Explain your answer.

14. In Fig. 5-17 we saw a way to update an inventory list atomically using magnetic tape. Since a tape can easily be simulated on disk (as a file), why do you think this method is not used any more?

15. In Fig. 5-25(d) three schedules are shown, two legal and one illegal. For the same transactions, give a complete list of all values that x might have at the end, and state which are legal and which are illegal.

16. When a private workspace is used to implement transactions on files, it may happen that a large number of file indices must be copied back to the parent’s workspace. How can this be done without introducing race conditions?

17. Give the full algorithm for whether an attempt to lock a file should succeed or fail. Consider both read and write locks, and the possibility that the file was unlocked, read locked, or write locked.

18. Systems that use locking for concurrency control usually distinguish read locks from write locks. What should happen if a process has already acquired a read lock and now wants to change it into a write lock? What about changing a write lock into a read lock?

19. With timestamp ordering in distributed transactions, suppose a write operation write(\(T_x\), x) can be passed to the data manager, because the only possibly conflicting operation write(\(T_y\), x) had a lower timestamp. Why would it make sense to let the scheduler postpone passing write(\(T_x\), x) until transaction \(T_y\) finishes?

20. Is optimistic concurrency control more or less restrictive than using timestamps? Why?


22. We have repeatedly said that when a transaction is aborted, the world is restored to its previous state, as though the transaction had never happened. We lied. Give an example where resetting the world is impossible.

6

CONSISTENCY AND REPLICATION

An important issue in distributed systems is the replication of data. Data are generally replicated to enhance reliability or improve performance. One of the major problems is keeping replicas consistent. Informally, this means that when one copy is updated, we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same. In this chapter, we take a detailed look at what consistency of replicated data actually means, and the various ways that consistency can be achieved.

We start with a general introduction by discussing why replication is useful and how it relates to scalability. Particular attention is paid to object-based replication, which is becoming increasingly important in many distributed systems.

To achieve high performance of operations on shared data, designers of parallel computers have paid much attention to different consistency models for distributed shared memory systems. These models are equally well applicable to other kinds of distributed systems, and are extensively discussed in this chapter.

Consistency models for shared data are often hard to implement efficiently in large-scale distributed systems. Moreover, in many cases simpler models can be used, which are also often easier to implement. One specific class is formed by client-centric consistency models, which concentrate on consistency from the perspective of a single (possibly mobile) client. Client-centric consistency models are discussed in a separate section.

Consistency is only half of the story. We also need to consider how consistency is actually implemented. Two, more or less independent, issues play a