

6

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.

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.

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) and Guerraoui and Rodrigues (2006). More formal approaches to a wealth of algorithms can be found in text books from Attiya and Welch (2004), Lynch (1996), and (Tel, 2000).

6.1 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 runs *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 *input.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. 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. 6-1. *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 go crazy trying to understand what is wrong with the code.

There are many more examples where an accurate account of time is needed. The example above can easily be reformulated to file timestamps in general. In addition, think of application domains such as financial brokerage, security auditing, and collaborative sensing, and it will become clear that accurate timing is important. Since time is so basic to the way people think and the effect of not having all the clocks synchronized can be so dramatic, 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? The answer is surprisingly complicated.

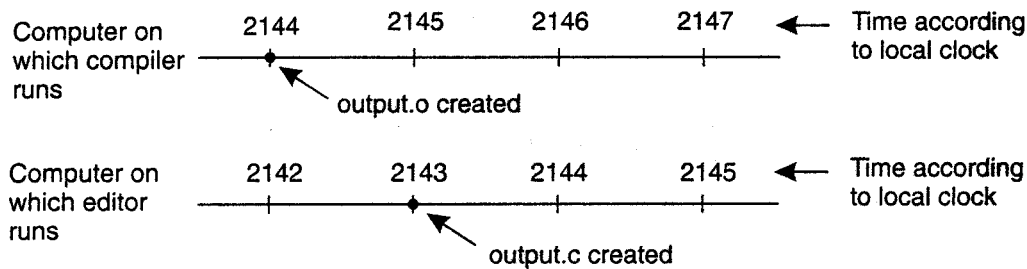


Figure 6-1. When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

6.1.1 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 counter 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, 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. Most 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 radically. 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. Under these circumstances, external physical clocks are needed. 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 easy 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, then climbs to a maximum height in the sky, and finally 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 $1/86400$ th of a solar day. The geometry of the mean solar day calculation is shown in Fig. 6-2.

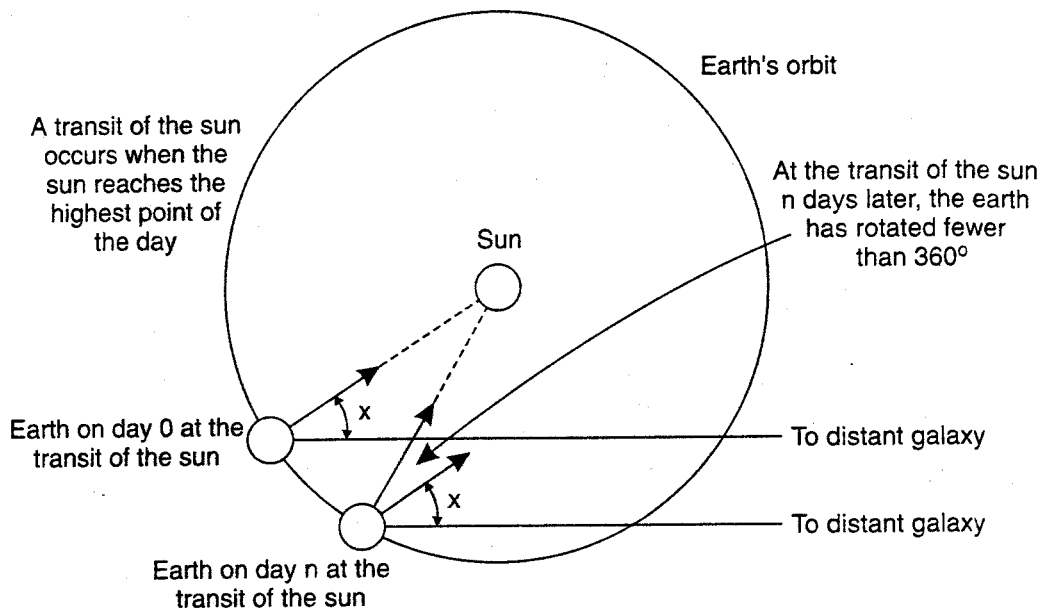


Figure 6-2. Computation of the mean solar day.

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, several laboratories around the world have cesium 133 clocks. Periodically, each laboratory tells the Bureau International de l'Heure (BIR) in Paris how many times its clock has ticked. The BIR 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.

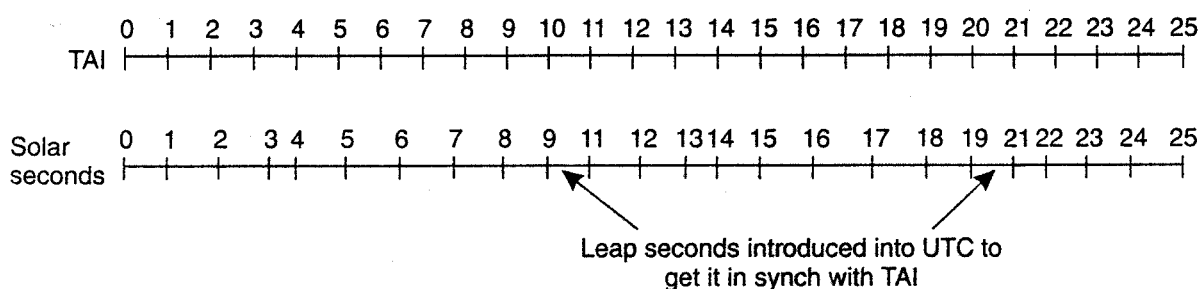


Figure 6-3. TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

BIR 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. 6-3. 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 synchronize the timing of their 60-Hz or 50-Hz clocks to 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.

6.1.2 Global Positioning System

As a step toward actual clock synchronization problems, we first consider a related problem, namely determining one's geographical position anywhere on Earth. This positioning problem is by itself solved through a highly specific, dedicated distributed system, namely GPS, which is an acronym for global positioning system. GPS is a satellite-based distributed system that was launched in 1978. Although it has been used mainly for military applications, in recent years it has found its way to many civilian applications, notably for traffic navigation. However, many more application domains exist. For example, GPS phones now allow to let callers track each other's position, a feature which may show to be extremely handy when you are lost or in trouble. This principle can easily be applied to tracking other things as well, including pets, children, cars, boats, and so on. An excellent overview of GPS is provided by Zogg (2002).

GPS uses 29 satellites each circulating in an orbit at a height of approximately 20,000 km. Each satellite has up to four atomic clocks, which are regularly calibrated from special stations on Earth. A satellite continuously broadcasts its position, and time stamps each message with its local time. This broadcasting allows every receiver on Earth to accurately compute its own position using, in principle, only three satellites. To explain, let us first assume that all clocks, including the receiver's, are synchronized.

In order to compute a position, consider first the two-dimensional case, as shown in Fig. 6-4, in which two satellites are drawn, along with the circles representing points at the same distance from each respective satellite. The y -axis represents the height, while the x -axis represents a straight line along the Earth's surface at sea level. Ignoring the highest point, we see that the intersection of the two circles is a unique point (in this case, perhaps somewhere up a mountain).

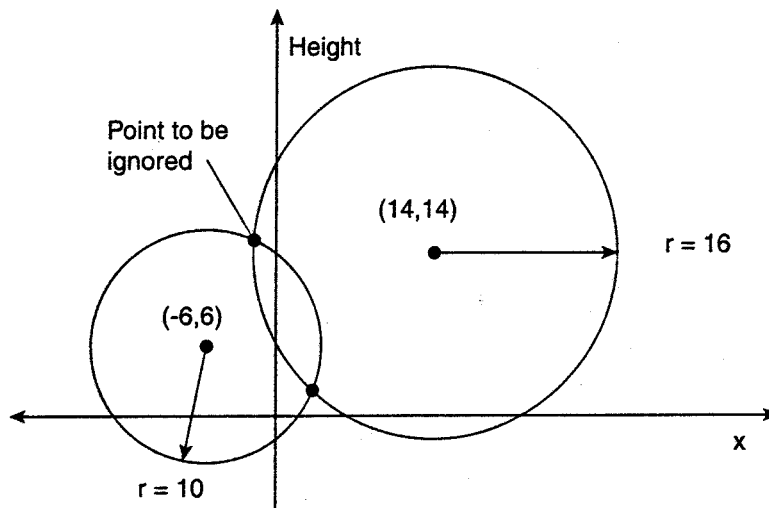


Figure 6-4. Computing a position in a two-dimensional space.

This principle of intersecting circles can be expanded to three dimensions, meaning that we need three satellites to determine the longitude, latitude, and altitude of a receiver on Earth. This positioning is all fairly straightforward, but matters become complicated when we can no longer assume that all clocks are perfectly synchronized.

There are two important real-world facts that we need to take into account:

1. It takes a while before data on a satellite's position reaches the receiver,
2. The receiver's clock is generally not in synch with that of a satellite.

Assume that the timestamp from a satellite is completely accurate. Let A_r denote the deviation of the receiver's clock from the actual time. When a message is

received from satellite i with timestamp T_i , then the measured delay b_i by the receiver consists of two components: the actual delay, along with its own deviation:

$$\Delta_i = (T_{\text{now}} - T_i) + \Delta_r$$

As signals travel with the speed of light, c , the measured distance of the satellite is clearly $c b_i$. With

$$d_i = c(T_{\text{now}} - T_i)$$

being the real distance between the receiver and the satellite, the measured distance can be rewritten to $d_i + c b_r$. The real distance is simply computed as:

$$d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

where x_i , y_i , and z_i denote the coordinates of satellite i . What we see now is that if we have four satellites, we get four equations in four unknowns, allowing us to solve the coordinates x_p , y_p , and z_r for the receiver, but also b_r . In other words, a GPS measurement will also give an account of the actual time. Later in this chapter we will return to determining positions following a similar approach.

So far, we have assumed that measurements are perfectly accurate. Of course, they are not. For one thing, GPS does not take leap seconds into account. In other words, there is a systematic deviation from UTC, which by January 1, 2006 is 14 seconds. Such an error can be easily compensated for in software. However, there are many other sources of errors, starting with the fact that the atomic clocks in the satellites are not always in perfect synch, the position of a satellite is not known precisely, the receiver's clock has a finite accuracy, the signal propagation speed is not constant (as signals slow down when entering, e.g., the ionosphere), and so on. Moreover, we all know that the earth is not a perfect sphere, leading to further corrections.

By and large, computing an accurate position is far from a trivial undertaking and requires going down into many gory details. Nevertheless, even with relatively cheap GPS receivers, positioning can be precise within a range of 1-5 meters. Moreover, professional receivers (which can easily be hooked up in a computer network) have a claimed error of less than 20-35 nanoseconds. Again, we refer to the excellent overview by Zogg (2002) as a first step toward getting acquainted with the details.

6.1.3 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. A survey is given in Ramanathan et al. (1990).

All the algorithms have the same underlying model of the system. Each machine is assumed to have a timer that causes an interrupt H 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, $C_p'(t) = dC_p/dt$ ideally should be 1. $C_p'(t)$ is called the frequency of p 's clock at time t . The skew of the clock is defined as $C_p'(t) - 1$ and denotes the extent to which the frequency differs from that of a perfect clock. The offset relative to a specific time t is $C_p(t) - t$.

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 modem 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 p such that

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

the timer can be said to be working within its specification. The constant p is specified by the manufacturer and is known as the maximum drift rate. Note that the maximum drift rate specifies to what extent a clock's skew is allowed to fluctuate. Slow, perfect, and fast clocks are shown in Fig. 6-5.

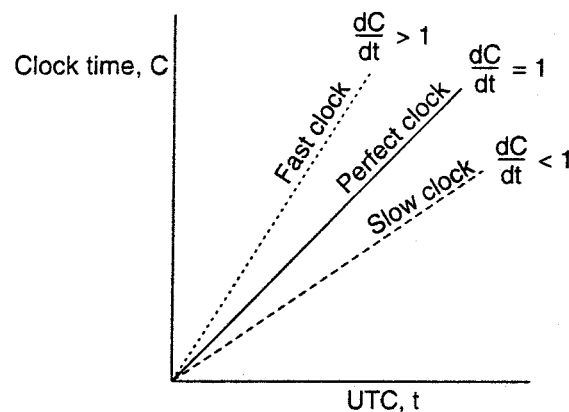


Figure 6-5. The relation between clock time and UTC when clocks tick at different rates.

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

Network Time Protocol

A common approach in many protocols and originally proposed by Cristian (1989) is to let clients contact a time server. The latter can accurately provide the current time, for example, because it is equipped with a WWV receiver or an accurate clock. The problem, of course, is that when contacting the server, message delays will have outdated the reported time. The trick is to find a good estimation for these delays. Consider the situation sketched in Fig. 6-6.

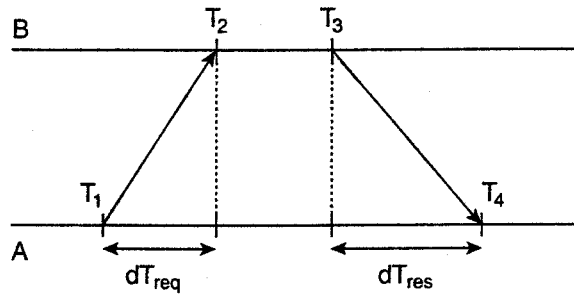


Figure 6-6. Getting the current time from a time server.

In this case, *A* will send a request to *B*, timestamped with value T_1 . *B*, in turn, will record the time of receipt T_2 (taken from its own local clock), and returns a response timestamped with value T_3 , and piggybacking the previously recorded value T_2 . Finally, *A* records the time of the response's arrival, T_4 . Let us assume that the propagation delays from *A* to *B* is roughly the same as *B* to *A*, meaning that $T_2 - T_1 \approx T_4 - T_3$. In that case, *A* can estimate its offset relative to *B* as

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Of course, time is not allowed to run backward. If *A*'s clock is fast, $\theta < 0$, meaning that *A* should, in principle, set its clock backward. This is not allowed as it 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 9 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.

In the case of the **network time protocol** (NTP), this protocol is set up pairwise between servers. In other words, *B* will also probe *A* for its current time. The offset θ is computed as given above, along with the estimation δ for the delay:

$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

Eight pairs of (8,8) values are buffered, finally taking the minimal value found for δ as the best estimation for the delay between the two servers, and subsequently the associated value ϵ as the most reliable estimation of the offset.

Applying NTP symmetrically should, in principle, also let B adjust its clock to that of A . However, if B 's clock is known to be more accurate, then such an adjustment would be foolish. To solve this problem, NTP divides servers into strata. A server with a reference clock such as a WWV receiver or an atomic clock, is known to be a stratum-I server (the clock itself is said to operate at stratum 0). When A contacts B , it will only adjust its time if its own stratum level is higher than that of B . Moreover, after the synchronization, A 's stratum level will become one higher than that of B . In other words, if B is a stratum- k server, then A will become a stratum- $(k+1)$ server if its original stratum level was already larger than k . Due to the symmetry of NTP, if A 's stratum level was lower than that of B , B will adjust itself to A .

There are many important features about NTP, of which many relate to identifying and masking errors, but also security attacks. NTP is described in Mills (1992) and is known to achieve (worldwide) accuracy in the range of 1-50 msec. The newest version (NTPv4) was initially documented only by means of its implementation, but a detailed description can now be found in Mills (2006).

The Berkeley Algorithm

In many algorithms such as NTP, 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 WWV receiver. The time daemon's time must be set manually by the operator periodically. The method is illustrated in Fig. 6-7.

In Fig. 6-7(a), at 3:00, the time daemon tells the other machines its time and asks for theirs. In Fig. 6-7(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. 6-7(c)].

Note that 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. If in our example of Fig. 6-7 the time daemon's clock would never be manually calibrated, no harm is done provided

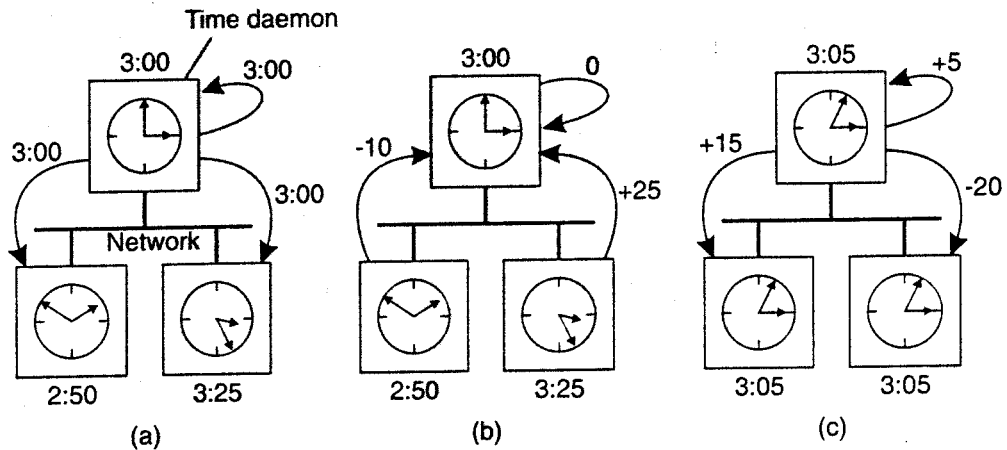


Figure 6.7. (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

none of the other nodes communicates with external computers. Everyone will just happily agree on a current time, without that value having any relation with reality.

Clock Synchronization in Wireless Networks

An important advantage of more traditional distributed systems is that we can easily and efficiently deploy time servers. Moreover, most machines can contact each other, allowing for a relatively simple dissemination of information. These assumptions are no longer valid in many wireless networks, notably sensor networks. Nodes are resource constrained, and multihop routing is expensive. In addition, it is often important to optimize algorithms for energy consumption. These and other observations have led to the design of very different clock synchronization algorithms for wireless networks. In the following, we consider one specific solution. Sivrikaya and Yener (2004) provide a brief overview of other solutions. An extensive survey can be found in Sundararaman et al. (2005).

Reference broadcast synchronization (RBS) is a clock synchronization protocol that is quite different from other proposals (Elson et al., 2002). First, the protocol does not assume that there is a single node with an accurate account of the actual time available. Instead of aiming to provide all nodes with the time, it aims at merely internally synchronizing the clocks, just as the Berkeley algorithm does. Second, the solutions we have discussed so far are designed to bring the sender and receiver into synch, essentially following a two-way protocol. RBS deviates from this pattern by letting only the receivers synchronize, keeping the sender out of the loop.

In RBS, a sender broadcasts a reference message that will allow its receivers to adjust their clocks. A key observation is that in a sensor network the time to

propagate a signal to other nodes is roughly constant, provided no multi-hop routing is assumed. Propagation time in this case is measured from the moment that a message leaves the network interface of the sender. As a consequence, two important sources for variation in message transfer no longer play a role in estimating delays: the time spent to construct a message, and the time spent to access the network. This principle is shown in Fig. 6-8.

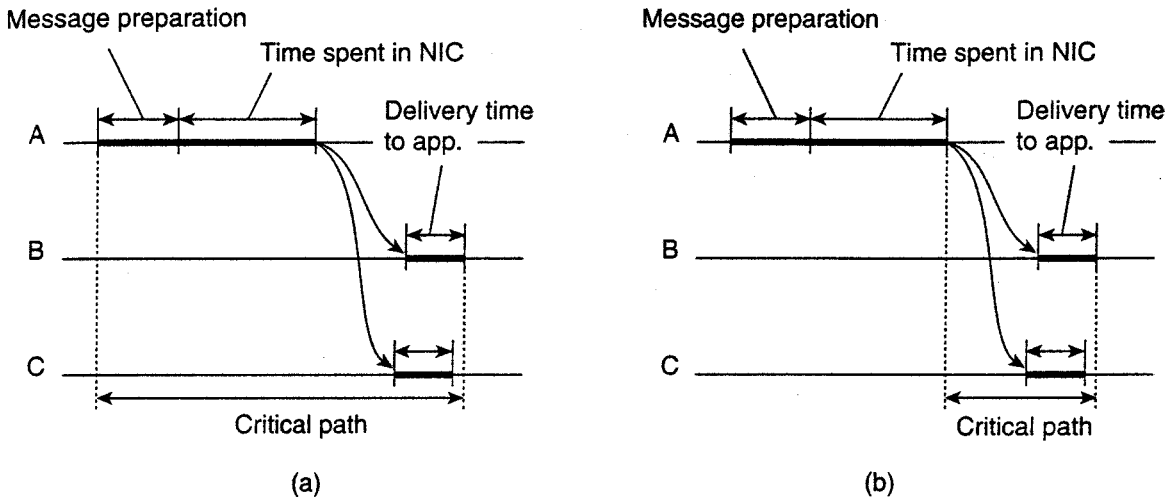


Figure 6-8. (a) The usual critical path in determining network delays. (b) The critical path in the case of RBS.

Note that in protocols such as NTP, a timestamp is added to the message before it is passed on the network interface. Furthermore, as wireless networks are based on a contention protocol, there is generally no saying how long it will take before a message can actually be transmitted. These factors of nondeterminism are eliminated in RBS. What remains is the delivery time at the receiver, but this time varies considerably less than the network-access time.

The idea underlying RBS is simple: when a node broadcasts a reference message m , each node p simply records the time $T_{p,m}$ that it received m . Note that $T_{p,m}$ is read from p 's local clock. Ignoring clock skew, two nodes p and q can exchange each other's delivery times in order to estimate their mutual, relative offset:

$$\text{Offset}[p, q] = \frac{\sum_{k=1}^M (T_{p,k} - T_{q,k})}{M}$$

where M is the total number of reference messages sent. This information is important: node p will know the value of q 's clock relative to its own value. Moreover, if it simply stores these offsets, there is no need to adjust its own clock, which saves energy.

Unfortunately, clocks can drift apart. The effect is that simply computing the average offset as done above will not work: the last values sent are simply less

accurate than the first ones. Moreover, as time goes by, the offset will presumably increase. Elson et al. use a very simple algorithm to compensate for this: instead of computing an average they apply standard linear regression to compute the offset as a function:

$$\text{Offset}[p,q](t) = \alpha t + \beta$$

The constants α and β are computed from the pairs $(T_{p,k}, T_{q,k})$. This new form will allow a much more accurate computation of q 's current clock value by node p , and vice versa.

6.2 LOGICAL CLOCKS

So far, we have assumed that clock synchronization is naturally related to real time. However, we have also seen that it may be sufficient that every node agrees on a current time, without that time necessarily being the same as the real time. We can go one step further. For running *make*, for example, it is adequate that two nodes agree that *input.o* is outdated by a new version of *input.c*. In this case, keeping track of each other's events (such as a producing a new version of *input.c*) is what matters. 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, what counts is whether *input.c* is older or newer than *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.

6.2.1 Lamport's Logical Clocks

To synchronize logical clocks, Lamport defined a relation called happens-before. The expression $a \sim b$ 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 $a \sim b$ 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 $a \sim b$

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 $a \sim b$ and $b \sim c$, then $a \sim c$. If two events, x and y , happen in different processes that do not exchange messages (not even indirectly via third parties), then $x \sim y$ is not true, but neither is $y \sim x$. 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 a notion of 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 $a \sim b$, 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. 6-9(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 P_1 , it has ticked 8 times in process P_2 and 10 times in process P_3 . Each clock runs at a constant rate, but the rates are different due to differences in the crystals.

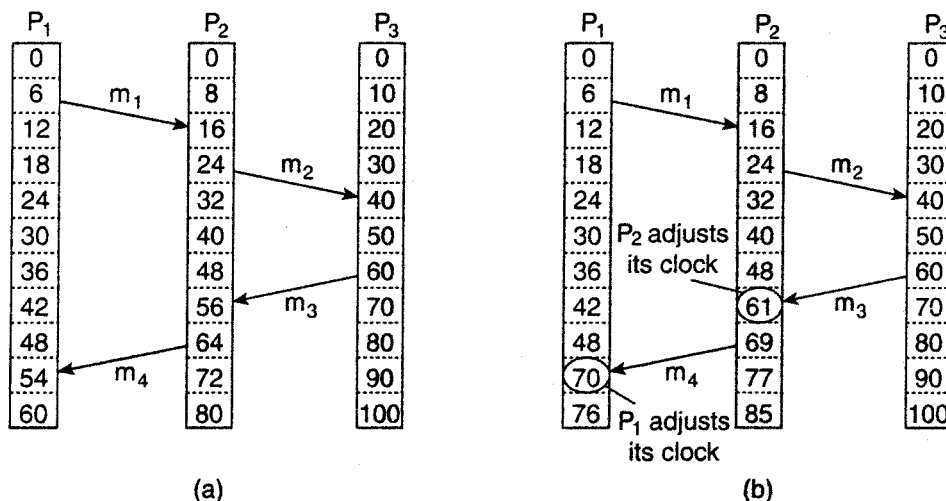


Figure 6-9. (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.

At time 6, process P_1 sends message m_1 to process P_2 . How long this message takes to arrive depends on whose clock you believe. In any event, the clock in process P_2 reads 16 when it arrives. If the message carries the starting time, 6, in it, process P_2 will conclude that it took 10 ticks to make the journey. This value is certainly possible. According to this reasoning, message m_2 from P_2 to P_1 takes 16 ticks, again a plausible value.

Now consider message m_3 . It leaves process P_3 at 60 and arrives at P_2 at 56. Similarly, message m_4 from P_2 to P_1 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 m_3 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. 6-9(b) we see that m_3 now arrives at 61. Similarly, m_4 arrives at 70.

To prepare for our discussion on vector clocks, let us formulate this procedure more precisely. At this point, it is important to distinguish three different layers of software as we already encountered in Chap. 1: the network, a middleware layer, and an application layer, as shown in Fig. 6-10. What follows is typically part of the middleware layer.

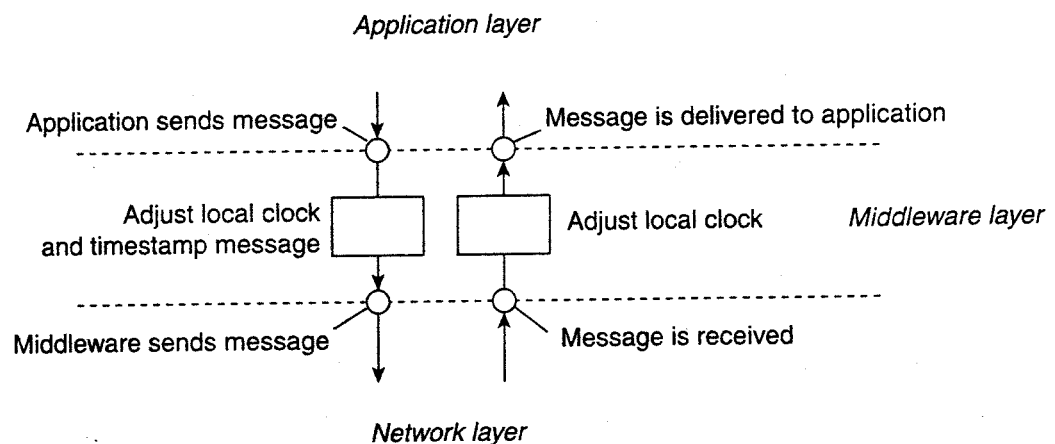


Figure 6-10. The positioning of Lamport's logical clocks in distributed systems.

To implement Lamport's logical clocks, each process P_i maintains a local counter G . These counters are updated as follows steps (Raynal and Singhal, 1996):

1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), P_i executes $G \leftarrow G + 1$.
2. When process P_i sends a message m to P_j it sets m 's timestamp $ts(m)$ equal to G after having executed the previous step.

3. Upon the receipt of a message m , process P_j adjusts its own local counter as $C_j \leftarrow \max\{C_j, ts(m)\}$, after which it then executes the first step and delivers the message to the application.

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. For example, an event at time 40 at process P_i will be time-stamped with $40.i$.

Note that by assigning the event time $C(a) \leftarrow q(a)$ if a happened at process P_i at time $q(a)$, we have a distributed implementation of the global time value we were initially seeking for.

Example: Totally Ordered Multicasting

As an application of Lamport's logical clocks, 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. 6-11.

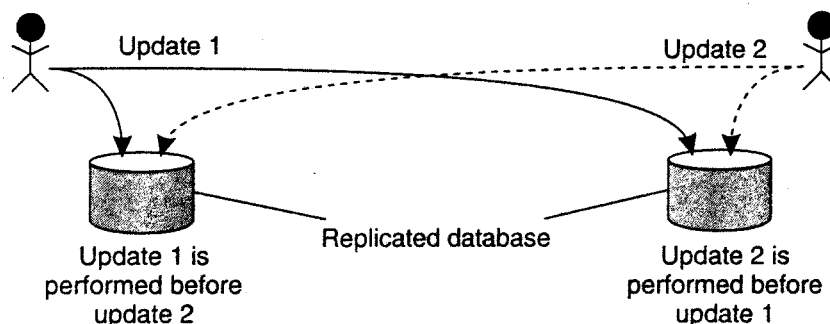


Figure 6-11. Updating a replicated database and leaving it in an inconsistent state.

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 \$1,111, 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 processed before the interest update or the other way around, which order is followed is not important from a consistency point of view. 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's logical clocks 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 also sent to the sender. In addition, we assume that messages from the same sender are received in the order they were sent, and that 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 acknowledgment 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 acknowledgment. The interesting aspect of this approach is that all processes will eventually have the same copy of the local queue (provided no messages are removed).

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 acknowledgments 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.

As we shall see in later chapters, totally-ordered multicasting is an important vehicle for replicated services where the replicas are kept consistent by letting them execute the same operations in the same order everywhere. As the replicas essentially follow the same transitions in the same finite state machine, it is also known as state machine replication (Schneider, 1990).

6.2.2 Vector Clocks

Lamport's logical clocks 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 clocks, 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 explain, consider the messages as sent by the three processes shown in Fig. 6-12. Denote by $I_{\text{snd}}(m_i)$ the logical time at which message m_i was sent, and likewise, by $T_{\text{rcv}}(m_i)$ the time of its receipt. By construction, we know that for each message $I_{\text{snd}}(m_i) < T_{\text{rcv}}(m_i)$. But what can we conclude in general from $T_{\text{rcv}}(m_i) < I_{\text{snd}}(m_j)$?

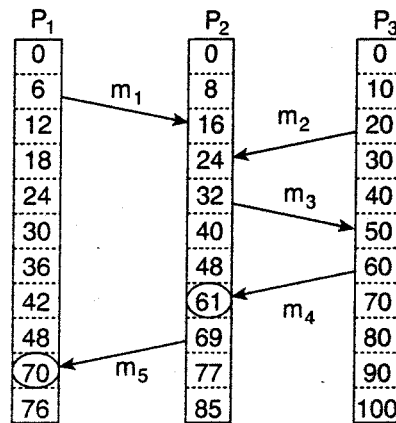


Figure 6-12. Concurrent message transmission using logical clocks.

In the case for which $m_i = m_1$ and $m_j = m_3$, we know that these values correspond to events that took place at process P_2 , meaning that m_j was indeed sent after the receipt of message m_i . This may indicate that the sending of message m_j depended on what was received through message m_i . However, we also know that $T_{\text{rcv}}(m_1) < I_{\text{snd}}(m_3)$. However, the sending of m_3 has nothing to do with the receipt of m_i .

The problem is that Lamport clocks do not capture **causality**. Causality can be captured by means of **vector clocks**. A vector clock $VC(a)$ assigned to an event a has the property that if $VC(a) < VC(b)$ for some event b , then event a is known to causally precede event b . Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

1. $VC_j[i]$ is the number of events that have occurred so far at P_i . In other words, $VC_j[i]$ is the local logical clock at process P_i .
2. If $VC_j[i] = k$ then P_i knows that k events have occurred at P_j . It is thus P_i 's knowledge of the local time at P_j .

The first property is maintained by incrementing $VC_j[i]$ 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, the following steps are performed:

1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), P_i executes $VC_j[l] \sim VC_j[i] + 1$.
2. When process P_i sends a message m to l_j , it sets m 's (vector) timestamp $ts(m)$ equal to VC_j after having executed the previous step.
3. Upon the receipt of a message m , process l_j adjusts its own vector by setting $VC_j[k] \sim \max\{VC_j[k], ts(m)[k]\}$ for each k , after which it executes the first step and delivers the message to the application.

Note that if an event a has timestamp $ts(a)$, then $ts(a)[l] - 1$ denotes the number of events processed at P_i that causally precede a . As a consequence, when l_j receives a message from P_i with timestamp $ts(m)$, it knows about the number of events that have occurred at P_i that causally preceded the sending of m . More important, however, is that l_j is also told how many events at *other* processes have taken place before P_i sent message m . In other words, timestamp $ts(m)$ tells the receiver how many events in other processes have preceded the sending of m , and on which m may causally depend.

Enforcing Causal Communication

Using vector clocks, it is now possible to ensure that a message is delivered only if all messages that causally precede it have also been received as well. To enable such a scheme, we will assume that messages are multicast within a group of processes. Note that this causally-ordered multicasting is weaker than the totally-ordered multicasting we discussed earlier. Specifically, if two messages are not in any way related to each other, we do not care in which order they are delivered to applications. They may even be delivered in different order at different locations.

Furthermore, we assume that clocks are only adjusted when sending and receiving messages. In particular, upon sending a message, process P_i will only increment $VC_j[i]$ by 1. When it receives a message m with timestamp $ts(m)$, it only adjusts $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$ for each k .

Now suppose that l_j receives a message m from P_i with (vector) timestamp $ts(m)$. The delivery of the message to the application layer will then be delayed until the following two conditions are met:

1. $ts(m)[i] = VC_j[i] + 1$
2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$

The first condition states that m is the next message that P_j was expecting from process P_i . The second condition states that P_j has seen all the messages that have been sent by P_i when it sent message m . Note that there is no need for process P_j to delay the delivery of its own messages.

As an example, consider three processes P_0 , P_1 , and P_2 as shown in Fig. 6-13. At local time $(1,0,0)$, P_0 sends message m to the other two processes. After its receipt by P_1 , the latter decides to send m^* , which arrives at P_2 sooner than m . At that point, the delivery of m^* is delayed by P_2 until m has been received and delivered to P_2 's application layer.

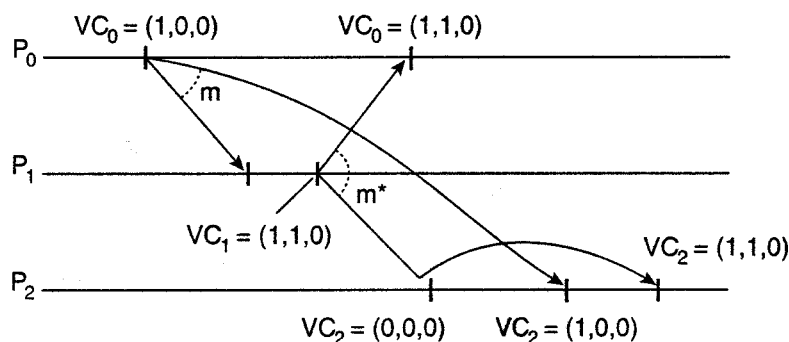


Figure 6-13. Enforcing causal communication.

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). Matters have not been settled, but more important is that the arguments still hold today.

There are two main problems with letting the middle ware deal with message ordering. First, because the middle ware 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 middleware layer. This approach is overly restrictive and may lead to efficiency problems.

A second problem is that not all causality may be captured. Consider an electronic bulletin board. 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 board. 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 bulletin board 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 an electronic bulletin board. 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.

6.3 MUTUAL EXCLUSION

Fundamental to distributed systems is the concurrency and collaboration among multiple processes. In many cases, this also means that processes will need to simultaneously access the same resources. To prevent that such concurrent accesses corrupt the resource, or make it inconsistent, solutions are needed to grant mutual exclusive access by processes. In this section, we take a look at some of the more important distributed algorithms that have been proposed. A recent survey of distributed algorithms for mutual exclusion is provided by Saxena and Rai (2003). Older, but still relevant is Velazquez (1993).

6.3.1 Overview

Distributed mutual exclusion algorithms can be classified into two different categories. In token-based solutions mutual exclusion is achieved by passing a special message between the processes, known as a token. There is only one token available and whoever has that token is allowed to access the shared resource. When finished, the token is passed on to a next process. If a process having the token is not interested in accessing the resource, it simply passes it on.

Token-based solutions have a few important properties. First, depending on how the processes are organized, they can fairly easily ensure that every process will get a chance at accessing the resource. In other words, they avoid starvation. Second, deadlocks by which several processes are waiting for each other to proceed, can easily be avoided, contributing to their simplicity. Unfortunately, the main drawback of token-based solutions is a rather serious one: when the token is lost (e.g., because the process holding it crashed), an intricate distributed procedure needs to be started to ensure that a new token is created, but above all, that it is also the only token.

As an alternative, many distributed mutual exclusion algorithms follow a permission-based approach. In this case, a process wanting to access the re-