# 3

# Synchronization in Distributed Systems

In Chap. 2, we saw how processes in a distributed system communicate with one another. The methods used include layered protocols, request/reply message passing (including RPC), and group communication. While communication is important, it is not the entire story. Closely related is how processes cooperate and synchronize with one another. For example, how are critical regions implemented in a distributed system, and how are resources allocated? In this chapter we will study these and other issues related to interprocess cooperation and synchronization in distributed systems.

In single CPU systems, critical regions, mutual exclusion, and other synchronization problems are generally solved using methods such as semaphores and monitors. These methods are not well suited to use in distributed systems because they invariably rely (implicitly) on the existence of shared memory. For example, two processes that are interacting using a semaphore must both be able to access the semaphore. If they are running on the same machine, they can share the semaphore by having it stored in the kernel, and execute system calls to access it. If, however, they are running on different machines, this method no longer works, and other techniques are needed. Even seemingly simple matters, such as determining whether event $A$ happened before or after event $B$, require careful thought.

We will start out by looking at time and how it can be measured, because time plays a major role in some synchronization methods. Then we will look at

mutual exclusion and election algorithms. After that we will study a high-level synchronization technique called atomic transactions. Finally, we will look at deadlock in distributed systems.

## 3.1. CLOCK SYNCHRONIZATION

Synchronization in distributed systems is more complicated than in centralized ones because the former have to use distributed algorithms. It is usually not possible (or desirable) to collect all the information about the system in one place, and then let some process examine it and make a decision as is done in the centralized case. In general, distributed algorithms have the following properties:

1. The relevant information is scattered among multiple machines.
2. Processes make decisions based only on local information.
3. A single point of failure in the system should be avoided.
4. No common clock or other precise global time source exists.

The first three points all say that it is unacceptable to collect all the information in a single place for processing. For example, to do resource allocation (assigning I/O devices in a deadlock-free way), it is generally not acceptable to send all the requests to a single manager process, which examines them all and grants or denies requests based on information in its tables. In a large system, such a solution puts a heavy burden on that one process.

Furthermore, having a single point of failure like this makes the system unreliable. Ideally, a distributed system should be more reliable than the individual machines. If one goes down, the rest should be able to continue to function. Having the failure of one machine (e.g., the resource allocator) bring a large number of other machines (its customers) to a grinding halt is the last thing we want. Achieving synchronization without centralization requires doing things in a different way from traditional operating systems.

The last point in the list is also crucial. 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 *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 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 is 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 slow, as shown in Fig. 3-1. *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 not work, and the programmer will go crazy trying to understand what is wrong with the code.

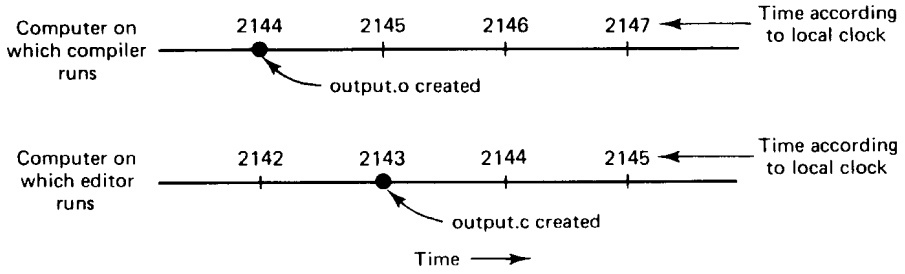


**Fig. 3-1.** When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

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?

### 3.1.1. Logical 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 initially, it usually asks the operator to enter the date and time, which is then converted to the number of ticks after some known starting date and stored in memory. 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 sync 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.

This brings us back to our original question, whether it is possible to synchronize all the clocks to produce a single, unambiguous time standard. In a classic paper, Lamport (1978) showed that clock synchronization is possible and presented an algorithm for achieving it. He extended his work in (Lamport, 1990).

Lamport pointed out that clock synchronization 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 above, what counts is whether *input.c* is older or newer than *input.o*, not their absolute creation times.

For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agree 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**.

When the additional constraint is present that the clocks must not only be the same, but also must not deviate from the real time by more than a certain amount, the clocks are called **physical clocks**. In this section we will discuss Lamport's algorithm, which synchronizes logical clocks. In the following sections we will introduce the concept of physical time and show how physical clocks can be synchronized.

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 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 they happened or which is 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. 3-2(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.

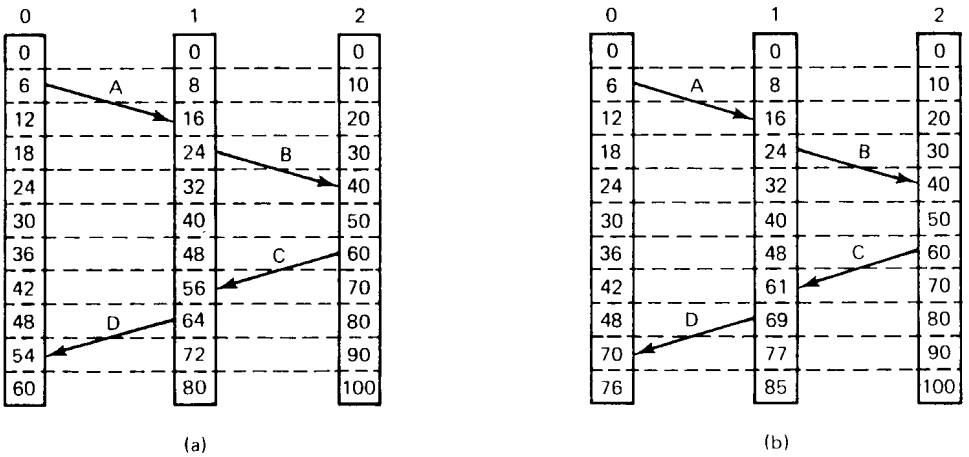


**Fig. 3-2.** (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.

At time 6, process 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 happened-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. 3-2(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, $C(a) < C(b)$.
3. For all 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.

### 3.1.2. Physical Clocks

Although Lamport's algorithm gives an unambiguous event ordering, the time values assigned to events are not necessarily close to the actual times at which they occur. 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. 3-3.

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

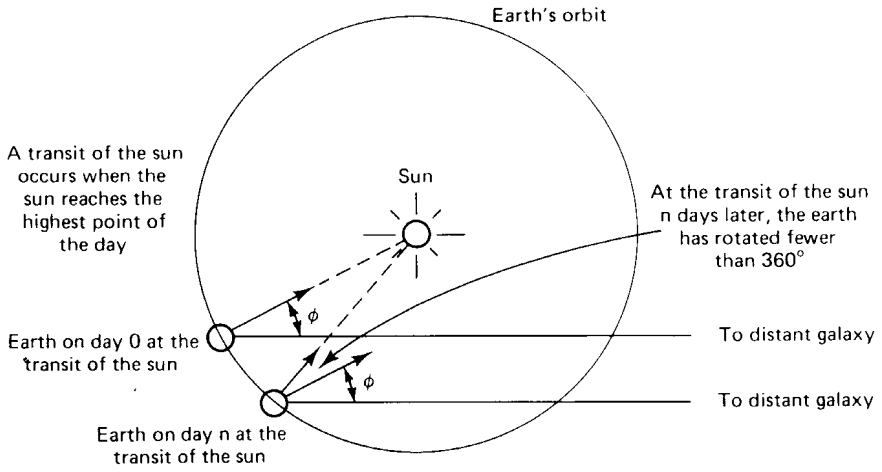


**Fig. 3-3.** Computation of the mean solar day.

300 million years ago there were about 400 days per year. The length of the year, that is, 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.

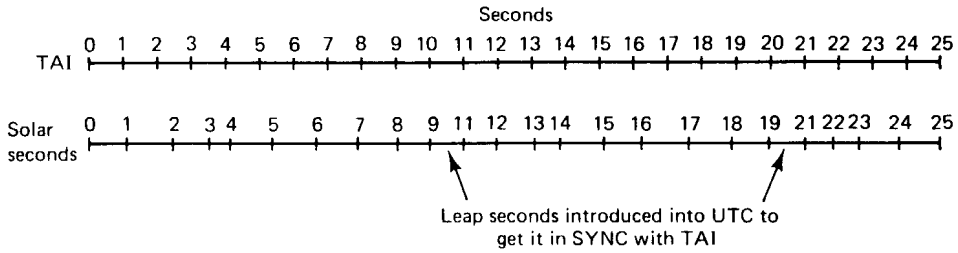


**Fig. 3-4.** TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

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. 3-4. 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. The cost varies from a few thousand dollars each to tens of thousands of dollars each, being more for the better sources. UTC can also be obtained more cheaply, but less accurately, by telephone from NIST in Fort Collins, but here too, a correction must be made for the signal path and modem speed. This correction introduces some uncertainty, making it difficult to obtain the time with extremely high accuracy.

### 3.1.3. 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 (e.g., Cristian, 1989; Drummond and Babaoglu, 1993; and Kopetz and Ochsenreiter, 1987). A survey is given in (Ramanathan et al., 1990b).

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 $H$ 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, $dC/dt$ 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