6 Time in a Distributed System

6.1 INTRODUCTION

Time is an important parameter in a distributed system. Consistency maintenance among replicated data often relies on identifying which update is the most recent one. Real-time systems like air-traffic control must have accurate knowledge of time to provide useful service and avoid catastrophe. Important authentication services (like Kerberos) rely on synchronized clocks. Wireless sensor networks rely on accurately synchronized clocks to compute the trajectory of fast-moving objects. Before addressing the challenges related to time in distributed systems, let us briefly review the prevalent standards of physical time.

6.1.1 THE PHYSICAL TIME

The notion of time and its relation to space have intrigued scientists and philosophers since the ancient days. According to the laws of physics and astronomy, real time is defined in terms of the rotation of earth in the solar system. A solar second equals 1/86,400th part of a solar day, which is the amount of time that the earth takes to complete one revolution around its own axis. This measure of time is called the real time (also known as Newtonian time), and is the primary standard of time. Our watches or other timekeeping devices are secondary standards that have to be calibrated with respect to the primary standard.

Modern timekeepers use atomic clocks as a de-facto primary standard of time. As per this standard, a second is precisely the time for **9,192,631,770** orbital transitions of the **Cesium 133** atom. In actual practice, there is a slight discrepancy — 86,400 atomic seconds is approximately 3 msec less than a solar day, so when the discrepancy grows to about 1 sec, a leap second is added to the atomic clock.

International Atomic Time (TAI) is an accurate time scale that reflects the weighted average of the readings of nearly 300 atomic clocks in over fifty national laboratories worldwide. It has been available since 1955, and became the international standard on which UTC (Coordinated Universal Time) is based. UTC was introduced on January 1, 1972, following a decision taken by the 14th General Conference on Weights and Measures (CGPM). The International Bureau of Weights and Measures is in charge of the realization of TAI.

Coordinated Universal Time popularly known as GMT (Greenwich Mean Time) or Zulu time differs from the local time by the number of hours of your time zone. In the US, the use of a central server receiving the WWV shortwave signals from Fort Collins, Colorado and periodically broadcasting the UTC-based local time to other timekeepers is quite common. In fact, inexpensive clocks driven by these signals are now commercially available.

Another source of precise time is GPS (Global Positioning System). A system of 24 satellites deployed in the earth's orbit maintains accurate spatial coordinates, and provides precise time reference almost everywhere on earth where GPS signals can be received. Each satellite broadcasts the value of an on-board atomic clock. To use the GPS, a receiver must be able to receive signals from at least four different satellites. While the clock values from the different satellites help obtain the precise time, the spatial coordinates (latitude, longitude, and the elevation of the receiver) are computed from the distances of the satellites estimated by the propagation delay of the signals. The clocks on the satellites are physically moving at a fast pace, and as per the theory of relativity, this causes the on-board clocks to run at a slightly slower rate than the corresponding clocks on the earth. The cumulative delay per day is approximately 38 msec, which is compensated using additional circuits. The atomic clocks that define GPS time record the number of seconds elapsed since January 6, 1980. Today (i.e., in 2006), GPS time is nearly 14 sec ahead of UTC, because it does not use the leap second correction. Receivers thus apply a clock-correction offset (which is periodically transmitted along with the other data) in order to display UTC correctly, and optionally adjust for a local time zone.

6.1.2 Sequential and Concurrent Events

Despite technological advances, the clocks commonly available at processors distributed across a system do not exactly show the same time. Built-in atomic clocks are not yet cost-effective — for example, wireless sensor networks cannot (yet) afford an atomic clock at each sensor node, although accurate timekeeping is crucial to detecting and tracking fast-moving objects. Certain regions in the world cannot receive such time broadcasts from reliable timekeeping sources. GPS signals are difficult to receive inside a building. The lack of a consistent notion of system-wide global time leads to several difficulties. One difficulty is the computation of the global state of a distributed system (defined as the set of local states of all processes at a given time). The special theory of relativity tells us that simultaneity has no absolute meaning — it is relative to the location of the observers. If the timestamps of a pair of events are nanoseconds apart, then we cannot convincingly say which one happened earlier, although this knowledge may be useful to establish a cause–effect relationship between the events. However, causality is a basic issue in distributed computing — the ordering of events based on causality is more fundamental than that obtained using physical clocks. Causal order is the basis of logical clocks, introduced by Lamport [L78]. In this chapter, we will address how distributed systems cope with uncertainties in physical time.

6.2 LOGICAL CLOCKS

An event corresponds to the occurrence of an action. A set of events {**a, b, c, d, …**} in a single process is called sequential, and their occurrences can be totally ordered in time using the clock at that process. For example, if Bob returns home at 5:40 P.M., answers the phone at 5:50 P.M., and eats dinner at 6:00 P.M. then the events (**return home, answer phone, eat dinner**) correspond to an ascending order. This total order is based on a single and consistent notion of time that Bob believes in accordance with his clock.

In the absence of reliable timekeepers, two different physical clocks at two different locations will always drift. Even if they are periodically resynchronized, some inaccuracy in the interim period is unavoidable. The accuracy of synchronization depends on clock drift, as well as on the resynchronization interval. Thus, 6:00 P.M. for Bob is not necessarily exactly 6:00 P.M. for Alice at a different location, even if they live in the same time zone. Events at a single point can easily be totally ordered on the basis of their times of occurrences at that point. But how do we decide if an event with Bob happened before another event with Alice? How do we decide if two events are concurrent?

To settle such issues, we depend on an obvious law of nature: No message can be received before it is sent. This is an example of *causality*. Thus if Bob sends a message to Alice, then the event of sending the message must have happened before the event of receiving that message regardless of the clock readings. Causal relationship can be traced in many applications. For example, during a chat, let Bob send a message **M** to Carol and Alice, and Alice post a reply **Re:M** back to Bob and Carol. Clearly **M** happened before **Re:M**. To make any sense of the chat, Carol should always receive **M**

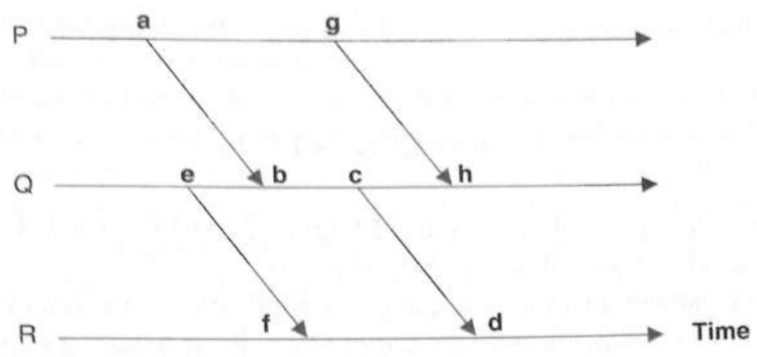

FIGURE 6.1 A space–time view of events in a distributed system consisting of three processes P, Q, R: the horizontal lines indicate the timelines of the individual processes, and the diagonal lines represent the flow of messages between processes.

before **Re:M**. If two events are not causally related, we do not care about their relative orders, and call them *concurrent*.

The above observations lead to three basic rules about the causal ordering of events, and they collectively define the happened before relationship $\prec$ in a distributed system:

Rule 1. Let each process have a physical clock whose value is monotonically increasing. If **a**, **b** are two events within a single process **P**, and the time of occurrence of **a** is earlier than the time of occurrence of **b**, then $\mathbf{a} \prec \mathbf{b}$.

Rule 2. If **a** is the event of sending a message by process **P**, and **b** is the event of receiving the same message by another process **Q**, then $\mathbf{a} \prec \mathbf{b}$.

Rule 3. $(\mathbf{a} \prec \mathbf{b}) \wedge (\mathbf{b} \prec \mathbf{c}) \Rightarrow \mathbf{a} \prec \mathbf{c}$.

$\mathbf{a} \prec \mathbf{b}$ is synonymous with "**a** happened before **b**" or "**a** is causally ordered before **b**." An application of these rules is illustrated using the space–time diagram of Figure 6.1. Here, **P**, **Q**, and **R** are three different sequential processes at three different sites. At each site, there is a separate physical clock, and these clocks tick at an unknown pace. The horizontal lines at each site indicate the passage of time. Based on the rules mentioned before, the following results hold:

$$\mathbf{e} \prec \mathbf{d} \quad \text{since } (\mathbf{e} \prec \mathbf{c} \text{ and } \mathbf{c} \prec \mathbf{d})$$
$$\mathbf{a} \prec \mathbf{d} \quad \text{since } (\mathbf{a} \prec \mathbf{b} \text{ and } \mathbf{b} \prec \mathbf{c} \text{ and } \mathbf{c} \prec \mathbf{d})$$

However, it is impossible to determine any ordering between the events **a, e** — neither $\mathbf{a} \prec \mathbf{e}$ holds, nor $\mathbf{e} \prec \mathbf{a}$ holds. In such a case, we call the two events concurrent. This shows that events in a distributed system cannot always be totally ordered. The happened before relationship defines a *partial order* and concurrency corresponds to the absence of causal ordering.

A *logical clock* is an event counter that respects causal ordering. Consider the sequence of events in a single sequential process. Each process has a counter **LC** that represents its logical clock. Initially, for every process, **LC = 0.** The occurrences of events correspond to the ticks of the logical clock local to that process. Every time an event takes place, **LC** is incremented. Logical clocks can be implemented using three simple rules:

LC1. Each time a local or internal event takes place, increment **LC** by **1**.

LC2. When sending a message, append the value of **LC** to the message.

LC3. When receiving a message, set the value of **LC** to **1+ max (local LC, message LC)**, where **local LC** is the local value of **LC**, and **message LC** is the **LC** value appended with the incoming message.

The above implementation of logical clocks provides the following limited guarantee for a pair of events **a** and **b**:

$$\mathbf{a} \prec \mathbf{b} \Rightarrow \mathbf{LC(a)} < \mathbf{LC(b)}$$

However, the converse is not true. In Figure 6.1 **LC(g)** = 2 and **LC(e)** = 1, but there is no causal order between **a** and **g**. This is a weakness of logical clocks.

Although causality induces a partial order, in many applications, it is important to define a total order among events. For example, consider the server of an airline handling requests for reservation from geographically dispersed customers. A fair policy for the server will be to allocate the next seat to the customer who sent the request ahead of others. If the physical clocks are perfectly synchronized and the message propagation delay is zero, then it is trivial to determine the order of the requests using physical clocks. However, if the physical clocks are not synchronized and the message propagation delays are arbitrary, then determining a total order among the incoming requests is a challenge.

One way to evolve a consistent notion of total ordering across an entire distributed system is to strengthen the notion of logical clocks. If **a** and **b** are two events in processes **i** and **j** (not necessarily distinct) respectively, then define total ordering ($\ll$) as follows:

$$\begin{array}{lll} \mathbf{a} \ll \mathbf{b} \ \text{iff} & \text{either } \mathbf{LC(a)} < \mathbf{LC(b)} & \\ & \text{or} \quad \mathbf{LC(a)} = \mathbf{LC(b)} & \text{and} \quad \mathbf{i} < \mathbf{j} \end{array}$$

where $\mathbf{i} < \mathbf{j}$ is determined either by the relative values of the numeric process identifiers, or by the lexicographic ordering of their names. Thus, whenever the logical clock values of two distinct events are equal, the process numbers or names will be used to break the tie. The **(id, LC)** value associated with an event is called its timestamp.

It should be obvious that $\mathbf{a} \prec \mathbf{b} \Rightarrow \mathbf{a} \nleftrightarrow b$. However, its converse is not necessarily true, unless **a**, **b** are events in the same process.

While the definition of causal order is quite intuitive, the definition of concurrency as the absence of causal order leads to tricky situations that may appear counter-intuitive. For example, the concurrency relation is not transitive. Consider Figure 6.1 again. Here, **e** is concurrent with **g**, and **g** is concurrent with **f**, but **e** if not concurrent with **f**.

Even after the introduction of causal and total order, some operational aspects in message ordering remain unresolved. One such problem is the implementation of FIFO communication across a network. Let **m, n** be two messages sent successively by process **P** to process **R** (via arbitrary routes). To preserve the FIFO property, the following must hold:

$$\mathbf{send(m)} \prec \mathbf{send(n)} \Rightarrow \mathbf{receive(m)} \prec \mathbf{receive(n)}$$

In Figure 6.2, **P** first sends the first message **m** directly to **R**, and then sends the next message **n** to **Q**, who forwards it to **R**. Although **send(m)** $\prec$ **send(n)** holds, and each channel individually exhibits

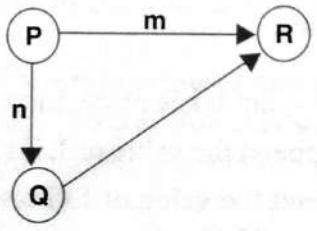

FIGURE 6.2 A network of three processes connected by FIFO channels.

FIFO behavior, there is no guarantee that **m** will reach **R** before **n**. Thus, **send(m)** $\prec$ **send(n)** does not necessarily imply **receive(m)** $\prec$ **receive(n)**.

This anomalous behavior can sometimes lead to difficult situations. For example, even if a server uses the policy of servicing requests based on timestamps, it may not always be able to do so, because the request bearing a smaller timestamp may not have reached the server before another request with a larger timestamp. At the same time, no server is clairvoyant, that is, it is impossible for the server to know whether a request with a lower timestamp will ever arrive in future. This is an important issue in distributed simulation, which requires that the temporal order in the simulated environment to have a one-to-one correspondence with that in the real system. We will address this in Chapter 18.

6.3 VECTOR CLOCKS

One major weakness of logical clocks is that, the LC values of two events cannot reveal if they are causally ordered. Vector clocks, independently discovered by Fidge [F88] and Mattern [M89] overcome this weakness. The goal of vector clocks is to detect causality. They define a mapping **VC** from events to integer arrays, and an order $<$ such that for any pair of events $\mathbf{a}, \mathbf{b} : \mathbf{a} \prec \mathbf{b} \Leftrightarrow \mathbf{VC(a)} < \mathbf{VC(b)}$

In a distributed system containing **N** processes $\mathbf{0, 1, 2, \ldots, N-1}$, for every process **i**, the vector clock **VC** is a (nonnegative) integer vector of length **N**. Like the logical clock, the vector clock is also event-driven. Each element of **VC** is a logical clock that is updated by the events local to that process only.

Let **a**, **b** be a pair of events. Denote the **k**th element of the vector clock for the event **a** by **VC(a)[k]**. Define **VC(a)** $<$ **VC(b)** (read **VC(a)** is dominated by **VC(b)**), if and only if the following two conditions hold:

- $\forall \mathbf{i} : \mathbf{0 \le i \le N-1} : \mathbf{VC(a)[i] \le VC(b)[i]}$
- $\exists \mathbf{i} : \mathbf{0 \le i \le N-1} : \mathbf{VC(a)[i] < VC(b)[i]}$

To implement a system of vector clocks, initialize the vector clock of each process to **0, 0, 0, …, 0** (**N** components). The implementation is based on the following three rules:

Rule 1. Each local event at process **i** increments the **i**th component of its vector clock (i.e., **VC[i]**) by 1.

Rule 2. The sender appends the vector timestamp to every message that it sends.

Rule 3. When process **j** receives a message with a vector timestamp **T** from another process, it first increments the **j**th component **VC[j]** of its own vector clock, (i.e., $\mathbf{VC[j] := VC[j] + 1}$) and then updates its vector clock as follows:

$$\forall \mathbf{k : 0 \le k \le N-1 :: VC[k] := max(T[k], VC[k])}$$

When the vector clock values of two events are incomparable, the events are concurrent. An example is shown in Figure 6.3. The event with vector timestamp **(2, 1, 0)** is causally ordered before the event with the vector timestamp **(2, 1, 4),** but is concurrent with the event having timestamp **(0, 0, 2)**.

Although vector clocks detect causal ordering or the lack of it, a problem is their poor scalability. As the size of the system increases, so does the size of the clock. Consequently, the addition of a process requires a reorganization of the state space across the entire system. Even if the topology is static, communication bandwidth suffers when message are stamped with the value of the vector clock. In Chapter 15, we will discuss the use of vector timestamps in solving the problem of causally ordered group communication.

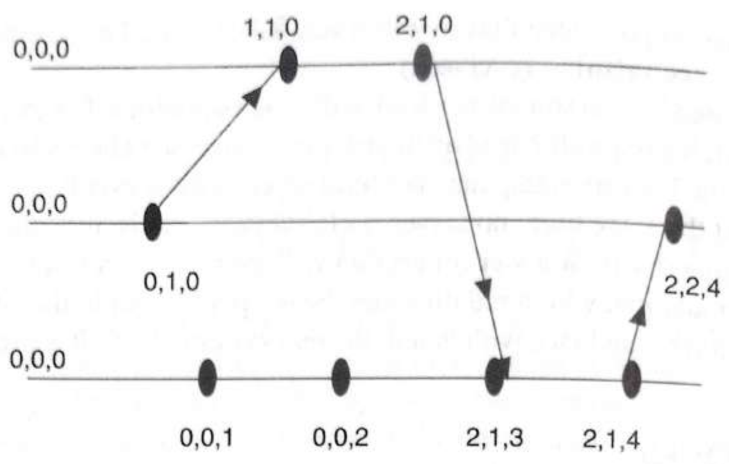

FIGURE 6.3 Example of vector timestamps.

6.4 PHYSICAL CLOCK SYNCHRONIZATION

6.4.1 Preliminary Definitions

Consider a system of **N** physical clocks {**0**, **1**, **2**, **3**, . . ., **N** − **1**} ticking approximately at the same rate. Such clocks may not accurately reflect real time, and despite great care taken in building these clocks, their readings slowly drift apart over time. Many of you know the story of Gregorian reform of the calendar: since Julius Cesar introduced the Julian calendar in 46 B.C. the small discrepancy between one solar day and 1/365 of a year was never corrected until in 1582 Pope Gregory XIII tried a remedy by lopping off ten days from the month of October, leading to a pandemonium. The availability of synchronized clocks simplifies many problems in distributed systems. Air-traffic control systems rely on accurate timekeeping to monitor flight paths and avoid collisions. Some security mechanisms depend on coordinated times across the network, so a loss of synchronization is a potential security lapse. The Unix *make* facility will fall apart unless the clocks are synchronized. Three main problems have been studied in the area of physical clock synchronization:

External synchronization. The goal of external synchronization is to maintain the reading of a clock as close to the UTC as possible. The use of a central server receiving the WWV shortwave signals from Fort Collins, Colorado and periodically broadcasting the time to other timekeepers is well known. In fact, inexpensive clocks driven by these signals are commercially available. However, failures and signal propagation delays can push the clock readings well beyond the comfort zone of some applications.

The NTP (Network Time Protocol) is an external synchronization protocol that runs on the Internet and coordinates a number of time-servers. This enables a large number of local clocks to synchronize themselves within a few milliseconds from the UTC. NTP takes appropriate recovery measures against possible failures of one or more servers, as well as the failure of links connecting the servers.

Internal synchronization. The goal of internal synchronization is to keep the readings of a system of autonomous clocks closely synchronized with one another, despite the failure or malfunction of one or more clocks. These clock readings may not have any connection with UTC or GPS time — mutual consistency is the primary goal. The internal clocks of wireless sensor networks are implemented as a counter driven by an inexpensive oscillator. A good internal synchronization protocol can bring the counter values close enough and make the network usable for certain critical applications.

Phase synchronization. In the clock phase synchronization problem, it is assumed that the clocks are driven by the same source of pulses, so they tick at the same rate, and are not autonomous. However, due to transient failures, clock readings may occasionally differ, so that while all the non-faulty clocks tick as 1, 2, 3, 4, …the faulty clock might tick as 6, 7, 8, 9, …during the same time.

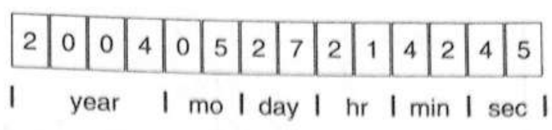

FIGURE 6.4 The clock reading when the drawing of this diagram was completed.

A phase synchronization algorithm guarantees that eventually the readings of all the clocks become identical.

Bounded and unbounded clocks. A clock is bounded, when with every tick, its value **c** is incremented in a **mod-M** field, $\mathbf{M > 1}$. Such a clock has only a finite set of possible values $\{\mathbf{0, 1, 2, \ldots, M-1}\}$. After $\mathbf{M-1}$, the value of **c** rolls back to 0. The value of an unbounded clock on the other hand increases monotonically, and thus such a clock can have an infinite number of possible values.

Due to the finite space available in physical systems, only bounded clocks can be implemented. It may appear that by appending additional information like year and month, a clock reading can look unbounded. Consider Figure 6.4 for an example of a clock reading.

Even this clock will overflow in the year 10,000. Anyone familiar with the Y2K problem knows the potential danger of bounded clocks: as the clock value increases beyond $\mathbf{M-1}$, it changes to **0** instead of **M**, and computer systems consider the corresponding event as an event from the past. As a result, many anticipated events may not be scheduled.

The solution to this problem by allocating additional space for storing the clock values is only temporary. All that we can guarantee is that clocks will not overflow in the foreseeable future. A 64-bit clock that is incremented every microsecond will not overflow for nearly 20 trillion years. However, when we discuss about fault-tolerance, we will find out that a faulty clock can still overflow very easily, and cause the old problem to resurface.

Drift rate. The maximum rate by which two clocks can drift apart is called the drift rate ρ. With ordinary crystal controlled clocks, the drift rate $< 10^{-6}$ (i.e., < 1 in 10^6). With atomic clocks, the drift rate is much smaller (around 1 in 10^{13}). A maximum drift rate ρ guarantees that $(1-\rho) \leq dC/dt \leq (1+\rho)$, where **C** is the clock time and **t** represents the real time.

Clock skew. The maximum difference δ between any two clocks that is allowed by an application is called the clock skew.

Resynchronization interval. Unless driven by the same external source, physical clocks drift with respect to one another, as well as with respect to UTC. To keep the clock readings close to one another, the clocks are periodically resynchronized. The maximum time difference **R** between two consecutive synchronization actions is called the resynchronization interval, and it depends on maximum permissible clock skew in the application (Figure 6.5).

6.4.2 Clock Reading Error

In physical clock synchronization, there are some unusual sources of error that are specific to time-dependent variables. Some of these are as follows:

Propagation delay. If clock **i** sends its reading to clock **j**, then the accuracy of the received value must depend not only on the value that was sent, but also on the message propagation delay. Note that this is not an issue when other types of values (that are not time-sensitive) are sent across a channel. Even when the propagation delay due to the physical separation between processes is negligible, the operating system at the receiving machine may defer the reading of the incoming clock value, and introduce error. Ideally, we need to simulate a situation when a clock ticks in transit to account for these overheads. A single parameter ε (called the reading error) accounts for the inaccuracy.

Processing overhead. Every computation related to clock synchronization itself takes a finite amount of time that needs to be separately accounted for.

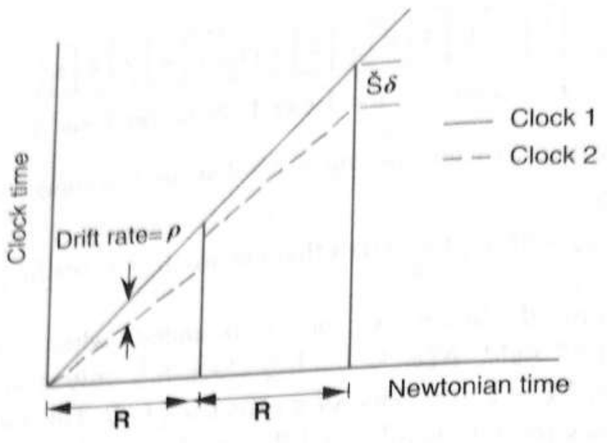

FIGURE 6.5 The cumulative drift between two clocks drifting apart at the rate ρ is brought closer after every resynchronization interval **R**.

6.4.3 Algorithms for Internal Synchronization

Berkeley algorithm. A well-known algorithm for internal synchronization is the Berkeley algorithm, used in Berkeley Unix 4.3 BSD. The basic idea is to periodically fetch the time from all the participant clocks, compute the average of these values, and then report back to the participants the adjustment that needs to be made to their local clocks, so that between any pair of clocks, the maximum difference between their reading never exceeds a predefined skew δ. The algorithm assumes that the condition holds in the initial state. A participant whose clock reading lies outside the predefined limit δ is disregarded when computing the average. This prevents the overall system time from being drastically skewed due to one erroneous clock. The algorithm handles the case where the notion of faults may be a relative one: for example, there may be two disjoint sets of clocks, and in each set the clocks are synchronized with one another, but no clock in one set is synchronized with any other clock in the second set. Here, to every clock in one set, the clocks in the other set are faulty.

Lamport and Melliar-Smith's algorithm. This algorithm is an adaptation of the Berkeley algorithm. It not only handles faulty clocks, but also handles two-faced clocks, an extreme form of faulty behavior, in which two nonfaulty clocks obtain conflicting readings from the same faulty clock. For the sake of simplicity, we disregard the overhead due to propagation delay or computation overhead, and consider clock reading to be an instantaneous action. Let **c(i,k)** denote clock **i**'s reading of clock **k**'s value. Each clock **i** repeatedly executes the following three steps:

Step 1. Read the value of every clock in the system.

Step 2. Discard bad clock values and substitute them by the value of the local clock. Thus if $|c(i,i) - c(i,j)| > \delta$ then $c(i,j) := c(i,i)$.

Step 3. Update the clock reading using the average of these values.

The above algorithm guarantees that if the clocks are initially synchronized, then they remain synchronized when there are at most **t** two-faced clocks, and $n > 3t$. To verify this, consider two distinct nonfaulty clocks **i** and **j** reading a third clock **k**. Two cases are possible:

Case 1. If clock **k** is non-faulty, then $c(i,k) = c(j,k)$.

Case 2. If clock **k** is faulty, then they can produce any reading. However, a faulty clock can make other clocks accept their readings as good values even if they transmit erroneous readings that are at most 3δ apart. Figure 6.6 illustrates such a scenario.

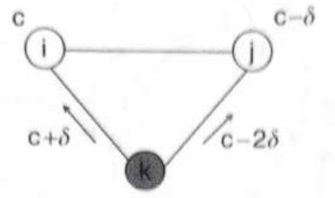

FIGURE 6.6 Two nonfaulty clocks **i** and **j** reading the value of a faulty clock **k**.

The following values constitute a feasible set of readings acceptable to every clock:

- $\mathbf{c(i, i) = c}$
- $\mathbf{c(i, k) = c + \delta}$
- $\mathbf{c(j, j) = c - \delta}$
- $\mathbf{c(j, k) = c - 2\delta}$

Now, assume that at most **t** out of the **n** clocks are faulty, and the remaining clocks are nonfaulty. Then, the maximum difference between the averages computed by any two nonfaulty processes is $(\mathbf{3t}\delta/\mathbf{n})$. If $\mathbf{n > 3t}$, then this difference is $<\delta$, which means that the algorithm brings the readings of the clocks **i** and **j** closer to each other, and within the permissible skew δ. This implies that the nonfaulty clocks remain synchronized.

How often do we need to run the above algorithm? To maintain synchrony, the maximum resynchronization period should be small enough, so that the cumulative drift does not offset the convergence achieved after each round of synchronization. A sample calculation is shown below:

As a result of the synchronization, the maximum difference between two nonfaulty clocks is reduced from δ to $(\mathbf{3t}\delta/\mathbf{n})$. Let $\mathbf{n = 3t + 1}$. Then the amount of correction is

$$\delta - \frac{\mathbf{3t}\delta}{\mathbf{3t+1}} = \frac{\delta}{\mathbf{3t+1}}$$

If ρ is the maximum rate at which two clocks drift apart, then it will take a time $(\delta/\rho(\mathbf{3t+1}))$ before their difference grows to δ again, and resynchronization becomes necessary. By definition, this is the resynchronization period **R**. Therefore, $\mathbf{R} = (\delta/\rho(\mathbf{3t+1}))$. If the resynchronization interval increases, then the system has to tolerate a larger clock skew.

6.4.4 Algorithms for External Synchronization

Cristian's method. In this method, a client obtains the data from a special host (called the time-server) that contains the reference time obtained from some precise external source. Cristian's algorithm compensates for the clock reading error. The client sends requests to the time-server every **R** units of time where $\mathbf{R} < \delta/\mathbf{2}\rho$ (this follows from the fact that in an interval $\mathbf{\Delta t}$ two perfectly synchronized clocks can be $\mathbf{2\Delta t}.\ \rho$ apart, and $\mathbf{R}.\ \mathbf{2\Delta t}.\rho<\delta$) and the server sends a response back to the client with the current time. For an accurate estimate of the current time, the client needs to estimate how long it has been since the time-server replied. This is done by assuming that the client's clock is reasonably accurate over short intervals, and that the latency of the link is approximately symmetric (the request takes as long to get to the server as the reply takes to get back). Given these assumptions, the client measures the round-trip time (RTT) of the request using its local clock, and divide it by half to estimate the propagation delay. As a result, if clock **i** receives the value **c(j)** from the time-server **j**, then **c(i)** corrects itself to **c(j) + RTT/2**.

These estimates can be further improved by adjusting for unusually large **RTT**s. Large **RTT**s are likely to be less symmetric, making the estimated error less accurate. One approach is to keep track of recent **RTT**s, and repeat requests if the **RTT** appears to be an outlier.

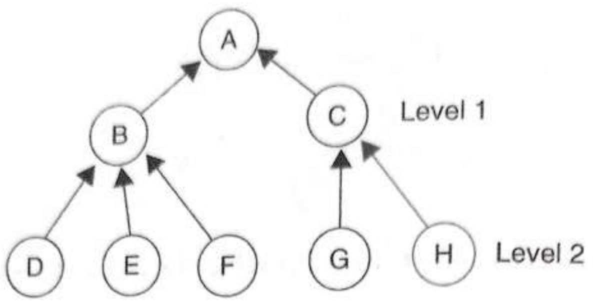

FIGURE 6.7 A network of time-servers used in NTP. The top level Server A (level 0) directly receives the UTC signals.

Network time protocol. Network Time Protocol (NTP), is an elaborate external synchronization mechanism designed to synchronize clocks on the Internet with the UTC despite occasional loss of connectivity, and failure of some of the time-servers, and possibly malicious timing inputs from untrusted sources. These time-servers are located at different sites on the Internet. NTP architecture is a tiered structure of clocks, whose accuracy decreases as its level number increases. The primary time-server that receives UTC from a dedicated transmitter or the GPS satellite belongs to level 0; a computer that is directly linked to the level 0 clock belongs to level 1; a computer that receives its time from level **i** belongs level (**i + 1**), and so on. Figure 6.7 shows the hierarchy with a single level 0 node.

A site can receive UTC using several different methods, including radio and satellite systems. Like the GPS system in the United States, a few other nations have their own systems of signaling accurate time. However, it is not practical to equip every computer with satellite receivers, since these signals are not received inside buildings. Cost is another factor. Instead, computers designated as primary time-servers are equipped with such receivers and they use the NTP to synchronize the clocks of networked computers.

In a sense, NTP is a refinement of Cristian's method. NTP provides time service using the following three mechanisms:

Multicasting. The time-server periodically multicasts the correct time to the client machines. This is the simplest method, and perhaps the least accurate. The readings are not compensated for signal delays.

Procedure call. The client processes send requests to the time-server, and the server responds by providing the current time. Using Cristian's method, each client can compensate for the propagation delay by using an estimate of the round-trip delay. The resulting accuracy is better than that obtained using multicasting.

Peer-to-peer communication. Network time protocol allows for several time-servers to communicate with one another to keep better track of the real time, and thus provide a more accurate time service to the client processes. This has the highest accuracy compared to the previous two methods.

Consider the exchange of a pair of messages between two time-servers as shown in Figure 6.8. Define the offset between two servers **P** and **Q** as the difference between their clock values. For each message, the sending time is stamped on the body of the message, and the receiver records its own time when the message was received. Let T_{PQ} and T_{QP} be the message propagation delays from **P** to **Q** and **Q** to **P**, respectively. If server **Q**'s time is ahead of server **P**'s time by an offset δ

$$T_2 = T_1 + T_{PQ} + \delta \tag{6.1}$$

$$T_4 = T_3 + T_{QP} - \delta \tag{6.2}$$

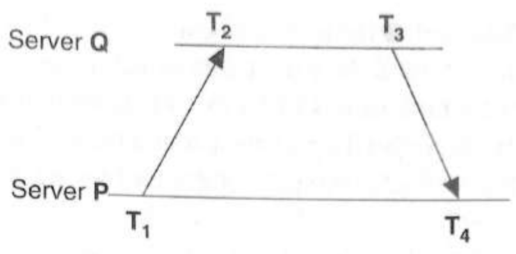

FIGURE 6.8 The exchange of messages between two time-servers.

Adding Equation 6.1 and Equation 6.2,

$$\mathbf{T_2 + T_4 = T_1 + T_3 + (T_{PQ} + T_{QP})}$$

The round-trip delay

$$\mathbf{y = T_{PQ} + T_{QP} = T_2 - T_1 + T_4 - T_3}$$

Subtracting Equation 6.2 from Equation 6.1,

$$\mathbf{2\delta = (T_2 - T_4 - T_1 + T_3) - (T_{PQ} - T_{QP})}$$

So, the offset

$$\mathbf{\delta = (T_2 - T_4 - T_1 + T_3)/2 - (T_{PQ} - T_{QP})/2}$$

where $\mathbf{T_{PQ} = T_{QP}}$, δ can be computed from the above expression. Otherwise, let $\mathbf{x = (T_2 - T_4 - T_1 + T_3)/2}$. Since both $\mathbf{T_{PQ}}$ and $\mathbf{T_{QP}}$ are greater than zero, the value of $\mathbf{(T_{PQ} - T_{QP})}$ must lie between **+y** and $\mathbf{-y}$. Therefore, the actual offset δ must lie between $\mathbf{x + y/2}$ and $\mathbf{x - y/2}$. Therefore, if each server bounces messages back and forth with another server and computes several pairs of **(x, y)**, then a good approximation of the real offset can be obtained from that pair in which **y** is the smallest, since that will minimize the dispersion in the window $\mathbf{[x + y/2, x - y/2]}$.

The multicast mode of communication is considered adequate for most applications. When the accuracy from the multicast mode is considered inadequate, the procedure call mode is used. An example is a file server on a LAN that wants to keep track of when a file was created (by communicating with a time-server at a higher level, i.e., a lower level number). Finally, the peer-to-peer mode is used only with the higher-level time-servers (level 1) for achieving the best possible accuracy. The synchronization subnet reconfigures itself when some servers fail, or become unreachable. NTP can synchronize clocks within an accuracy of 1 to 50 msec.

6.5 CONCLUDING REMARKS

In asynchronous distributed systems, the absolute physical time is not significant, but the temporal order of events is important for some applications. In replicated servers, each server can be viewed as a state machine whose state is modified by receiving and handling the inputs from the clients. In order that all replicas always remain in the same state (so that one can seamlessly switch to a different server if one crashes) all replicas must receive the inputs from clients in the same order.

The performance of a clock synchronization algorithm is determined by how close two distinct clock times can be brought, the time of convergence, and the nature of failures tolerated by such algorithms. The adjustment of clock values may have interesting side effects. For example, if a clock is advanced from **171** to **174** during an adjustment, then the time instants **172** and **173** are lost. This