Chapter Five
Synchronization

5.1. Introduction

On this chapter, we are going to deal with one of the fundamental issues encountered when constructing a system made up of independent communicating processes: dealing with time and making sure that processes do the right thing at the right time. In essence this comes down to allowing processes to synchronize and coordinate their actions. Coordination refers to coordinating the actions of separate processes relative to each other and allowing them to agree on global state (such as values of a shared variable). Synchronization is coordination with respect to time, and refers to the ordering of events and execution of instructions in time. Examples of synchronization include ordering distributed events in a log file and ensuring that a process performs an action at a particular time. Examples of coordination include ensuring that processes agree on what actions will be performed (e.g., money will be withdrawn from the account), who will be performing actions (e.g., which replica will process a request), and the state of the system (e.g., the elevator is stopped).

Synchronization and coordination play an important role in most distributed algorithms (i.e., algorithms intended to work in a distributed environment). In particular, some distributed algorithms are used to achieve synchronization and coordination, while others assume the presence of synchronization or coordination mechanisms. Discussions of distributed algorithms generally assume one of two timing models for distributed systems. The first is a synchronous model, where the time to perform all actions, communication delay, and clock drift on all nodes, are bounded. In asynchronous distributed systems there are no such bounds. Most real distributed systems are asynchronous, however, it is easier to design distributed algorithms for synchronous distributed systems. Algorithms for asynchronous systems are always valid on synchronous systems; however, the converse is not true.

Time & Clocks

As mentioned, time is an important concept when dealing with synchronization and coordination. In particular it is often important to know when events occurred and in what order they occurred. In a non-distributed system, dealing with time is trivial as there is a single shared clock. All processes see the same time. In a distributed system, on the other hand, each computer has its own clock. Because no clock is perfect each of these clocks has its own skew which causes clocks on different computers to drift and eventually become out of sync.

There are several notions of time that are relevant in a distributed system. First of all, internally a computer clock simply keeps track of ticks that can be translated into physical time (hours, minutes, seconds, etc.). This physical time can be global or local. Global time is a universal time that is the same for everyone and is generally based on some form of absolute time. Currently Coordinated Universal Time (UTC), which is based on oscillations of the Cesium-133 atom, is the most accurate global time. Besides global time, processes can also consider local time. In this case the time is only relevant to the processes taking part in the distributed system (or algorithm). This time may be based on physical or logical clocks.

5.2. Physical Clocks and clock synchronization algorithms

Physical clocks keep track of physical time. In distributed systems that rely on actual time it is necessary to keep individual computer clocks synchronized. The clocks can be synchronized to global time (external synchronization), or to each other (internal synchronization). Cristian's algorithm and the Network Time Protocol (NTP) are examples of algorithms developed to synchronize clocks to an external global time source (usually UTC). The Berkeley Algorithm is an example of an algorithm that allows clocks to be synchronized internally.

Cristian's algorithm requires clients to periodically synchronize with a central time server (typically a server with a UTC receiver). One of the problems encountered when synchronizing clocks in a distributed system is that unpredictable communication latencies can affect the synchronization. For example, when a client requests the current time from the time server, by the time the server's reply reaches the client, the time will have changed. The client must, therefore, determine what the communication latency was and adjust the server's response accordingly. Cristian's algorithm deals with this problem by attempting to calculate the communication delay based on the time elapsed between sending a request and receiving a reply.

The Network Time Protocol is similar to Cristian's algorithm in that synchronization is also performed using time servers and an attempt is made to correct for communication latencies. Unlike Cristian's algorithm, however, NTP is not centralized and is designed to work on a wide area scale. As such, the calculation of delay is somewhat more complicated. Furthermore, NTP provides a hierarchy of time servers, with only the top layer containing UTC clocks. The NTP algorithm allows client-server and peer-to-peer

(mostly between time servers) synchronization. It also allows clients and servers to determine the most reliable servers to synchronize with. NTP typically provides accuracies between 1 and 50 msec depending on whether communication is over a LAN or WAN.
Unlike the previous two algorithms, the Berkeley algorithm does not synchronize to a global time. Instead, in this algorithm, a time server polls the clients to determine the average of everyone's time. The server then instructs all clients to set their clocks to this new average time. Note that in all the above algorithms a clock should never be set backward. If time needs to be adjusted backward, clocks are simply slowed down until time 'catches up'.

5.3. Logical Clocks

For many applications, the relative ordering of events is more important than actual physical time. In a single process the ordering of events (e.g., state changes) is trivial. In a distributed system, however, besides local ordering of events, all processes must also agree on ordering of causally related events (e.g., sending and receiving of a single message). Given a system consisting of N processes P_i, i ∈ {1, . . . , N }, we define the local event ordering $\rightarrow_i$ as a binary relation, such that, if P_i observes e before e', we have $e \rightarrow_i e'$. Based on this local ordering, we define a global ordering as a happened before relation →, as proposed by Lamport. The relation → is the smallest relation, such that

1. $e \rightarrow_i e'$ implies $e \rightarrow e'$,
2. for every message m, send (m) → receive (m), and
3. $e \rightarrow e'$ and $e' \rightarrow e''$ implies $e \rightarrow e''$ (transitivity).

The relation → is almost a partial order (it lacks reflexivity). If a → b, then we say a causally affects b. We consider unordered events to be concurrent if they are unordered; i.e. $a \nrightarrow b$ and $b \nrightarrow a$ implies $a \parallel b$.

5.3.1. Lamport clocks

Lamport's logical clocks can be implemented as a software counter that locally computes the happened-before relation →. This means that each process P_i maintains a logical clock L_i. Given such a clock, $L_i(e)$ denotes a Lamport timestamp of event **e** at P_i and L(e) denotes a timestamp of event **e** at the process it occurred at. Processes now proceed as follows:

1. Before time stamping a local event, a process P_i executes $L_i := L_i + 1$.
2. Whenever a message m is sent from P_i to P_j :

- Process P_i executes $L_i := L_i + 1$ and sends the new L_i with m.

- Process P_j receives L_i with m and executes $L_j := \max(L_j , L_i) + 1$. receive(m) is annotated with the new L_j.

In this scheme, $a \rightarrow b$ implies $L(a) < L(b)$, but $L(a) < L(b)$ does not necessarily imply $a \rightarrow b$.

As an example, consider Figure 5.1. In this figure $E12 \rightarrow E23$ and $L_1(E12) < L_2(E23)$ (i.e., $2 < 3$), however we also have $E13 \nrightarrow E24$ while $L1(E13) < L2(E24)$ (i.e., $3 < 4$).

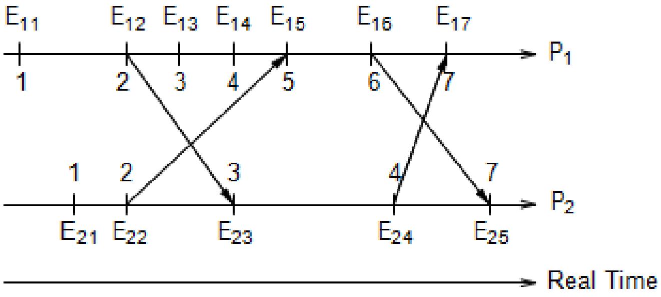

Figure 5.1: Example of the use of a Lamport's clocks

In some situations (e.g., to implement distributed locks), a partial ordering on events is not sufficient and a total ordering is required. In these cases, the partial ordering can be completed to total ordering by including process identifiers. Given local time stamps $L_i(e)$ and $L_j(e')$, we define global time stamps $\langle L_i(e), i\rangle$ and $\langle L_j(e), j\rangle$. We, then, use standard lexicographical ordering, where $\langle L_i(e), i\rangle < \langle L_j(e), j\rangle$ iff $L_i(e) < L_j(e)$, or $L_i(e) = L_j(e)$ and $i < j$.

5.4. Distributed Concurrency Control

Some of the issues encountered when looking at concurrency in distributed systems are familiar from the study of operating systems and multithreaded applications in particular dealing with race conditions that occur when concurrent processes access shared resources. In non-distributed system, these problems are solved by implementing mutual exclusion using local primitives such as locks, semaphores, and monitors. In distributed systems, dealing with concurrency becomes more complicated due to the lack of directly shared resources (such as memory, CPU registers, etc.), the lack of a global clock, the lack of a single global program state, and the presence of communication delays.

5.4.1. Distributed Mutual Exclusion

When concurrent access to distributed resources is required, we need to have mechanisms to prevent race conditions while processes are within critical sections. These mechanisms must fulfill the following three requirements:

1. **Safety**: At most one process may execute the critical section at a time
2. **Liveness**: Requests to enter and exit the critical section eventually succeed
3. **Ordering**: Requests are processed in happened-before ordering

1. Central Server

The simplest approach is to use a central server that controls the entering and exiting of critical sections. Processes must send requests to enter and exit a critical section to a lock server (or coordinator), which grants permission to enter by sending a token to the requesting process. Upon leaving the critical section, the token is returned to the server. Processes that wish to enter a critical section while another process is holding the token are put in a queue. When the token is returned the process at the head of the queue is given the token and allowed to enter the critical section.

This scheme is easy to implement, but it does not scale well due to the central authority. Moreover, it is vulnerable to failure of the central server.

2. Token Ring

More sophisticated is a setup that organizes all processes in a logical ring structure, along which a token message is continuously forwarded. Before entering the critical section, a process has to wait until the token comes by and then retain the token until it exits the critical section.

A disadvantage of this approach is that the ring imposes an average delay of N/2 hops, which again limits scalability. Moreover, the token messages consume bandwidth and failing nodes or channels can break the ring. Another problem is that failures may cause the token to be lost. In addition, if new processes join the network or wish to leave, further management logic is needed.

3. Using Multicast and Logical Clocks

Each participating process P_i maintains a Lamport clock and all processes must be able to communicate pairwise. At any moment, each process is in one of three states:

1. Released: Outside of critical section
2. Wanted: Waiting to enter critical section
3. Held: Inside critical section

If a process wants to enter a critical section, it multicasts a message (L_i, P_i) and waits until it has received a reply from every other process. The processes operate as follows:

- If a process is in Released state, it immediately replies to any request to enter the critical section.

- If a process is in Held state, it delays replying until it is finished with the critical section.
- If a process is in Wanted state, it replies to a request immediately only if the requesting timestamp is smaller than the one in its own request.

The only hurdle to scalability is the use of multicasts (i.e., all processes have to be contacted in order to enter a critical section). More scalable variants of this algorithm require each individual process to only contact subsets of its peers when wanting to enter a critical section. Unfortunately, failure of any peer process can deny all other processes entry to the critical section.

5.5. Election Algorithm

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. In this section we will look at algorithms for electing a coordinator (using this as a generic name for the special process). If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process has a unique number, for example, its network address (for simplicity, we will assume one process per machine). In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator. The algorithms differ in the way they do the location.

Furthermore, we also assume that every process knows the process number of every other process. What the processes do not know is which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

5.5.1. The Bully Algorithm

When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows:

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

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator. If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job.

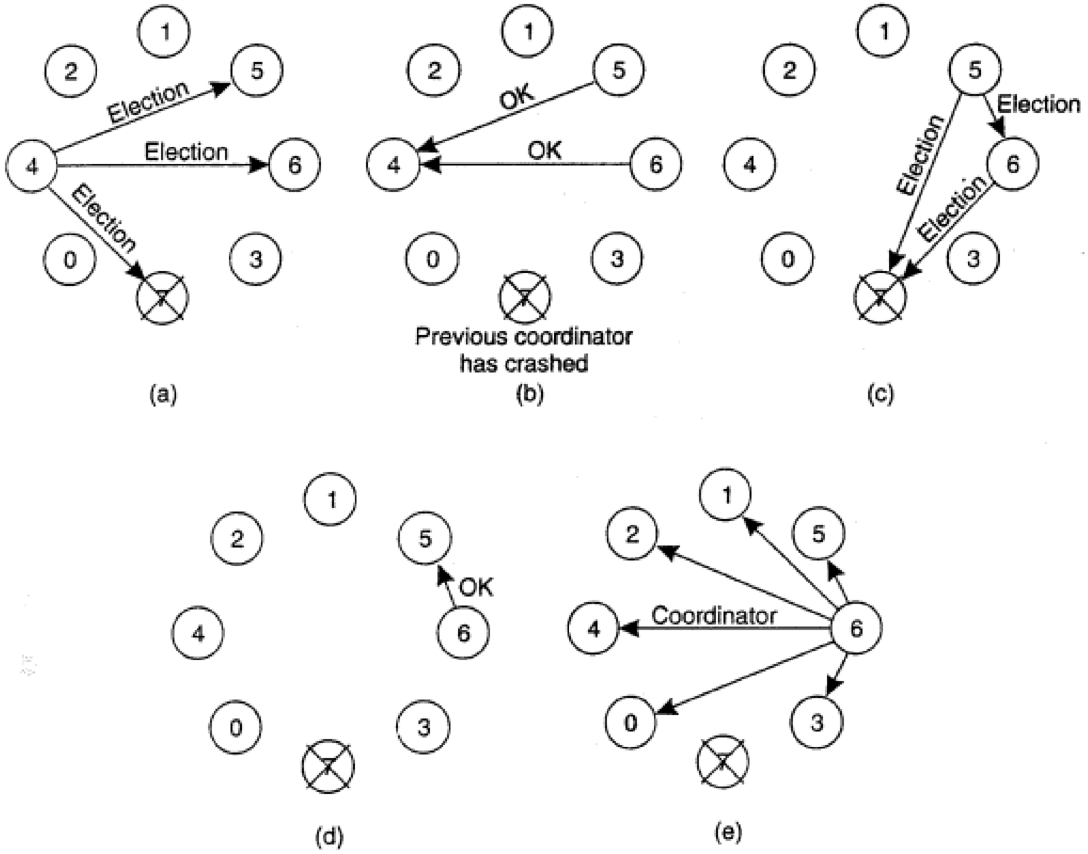

Figure 5.2 The bully election algorithm. (Details found in your text book)

5.5.2. A Ring Algorithm

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

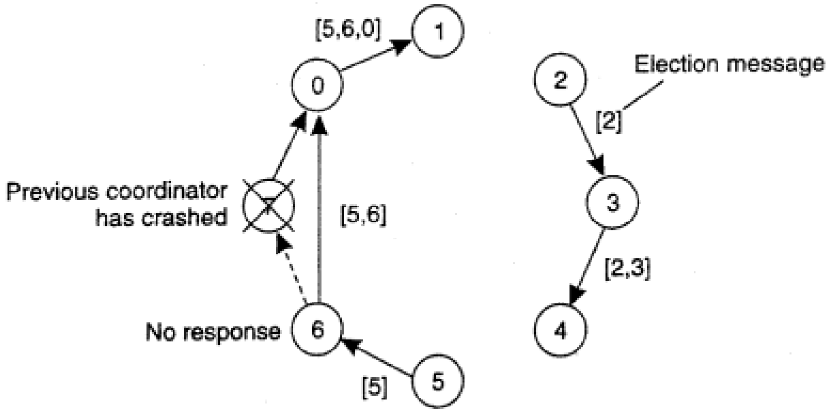

Figure 5.3 Election algorithm using a ring

Further reading assignment

1. Vector clock
2. How decentralized algorithm achieves mutual exclusion in a distributed system?
3. Compare distributed mutual exclusion algorithms (based on number of entry/wait message, delay, and reliability)
4. Election in a wireless network and large scale systems
5. How Global Positioning System (GPS) works?