Unit 6: Coordination

Contents:

Coordination mainly concentrate on how processes can synchronize and coordinate their actions. For example, it is important that multiple processes do not simultaneously access a shared resource, such as a file, 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.

Synchronization and coordination are two closely related phenomena. In process synchronization we make sure that one process waits for another to complete its operation. When dealing with data synchronization, the problem is to ensure that two sets of data are the same. When it comes to coordination, the goal is to manage the interactions and dependencies between activities in a distributed system. From this perspective, one could state that coordination encapsulates synchronization.

As it turns out, coordination in distributed systems is often much more difficult compared to that in uniprocessor or multiprocessor systems.

6.1 Clock Synchronization

In a centralized system, time is unambiguous. When a process wants to know the time, it simply makes a call to the operating system. 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 simple example. Normally, in Unix large programs are split up into multiple source files, so that a change to one source file requires only 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.

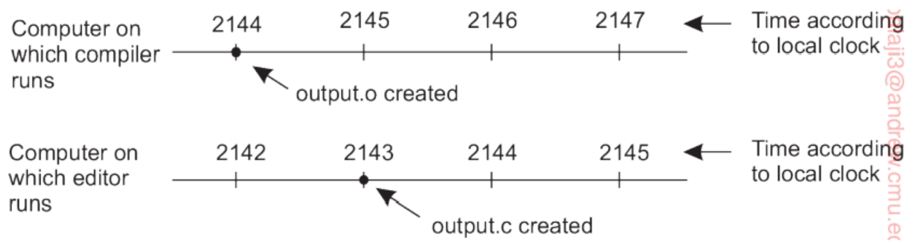

Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

Figure 6.1: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

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.

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.

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 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.

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?

The basis for keeping global time is a called **Universal Coordinated Time**, but is abbreviated as UTC. UTC is the basis of all modern civil timekeeping and is a worldwide standard. To provide UTC to people who need precise time, some 40 shortwave radio stations around the world broadcast a short pulse at the start of each UTC second. The accuracy of these stations 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. 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. By combining receptions from several satellites, ground time servers can be built offering an accuracy of 50 nsec. UTC receivers are commercially available and many computers are equipped with one.

Clock synchronization

If one machine has a UTC receiver, the goal becomes keeping all the other machines synchronized to it. If no machines have UTC 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.

All clocks are based on some harmonic oscillator: an object that resonates at a certain frequency and from which we can subsequently derive time. Atomic clocks are based on the transitions of the cesium 133 atom, which is not only very high, but also very constant. Hardware clocks in most computers use a crystal oscillator based on quartz, which is also capable of producing a very high, stable frequency, although not as stable as that of atomic clocks. A software clock in a computer is derived from that computer's hardware clock. In particular, the hardware clock is assumed to cause an interrupt f times per second. When this timer goes off, the interrupt handler adds 1 to a counter that keeps track of the number of ticks (interrupts) since some agreed upon time in the past. This counter acts as a software clock C, resonating at frequency F.

The whole idea of clock synchronization is that we keep clocks precise.

Network Time Protocol

A common approach in many protocols is to let clients contact a time server. The latter can accurately provide the current time, for example, because it is equipped with a UTC 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.

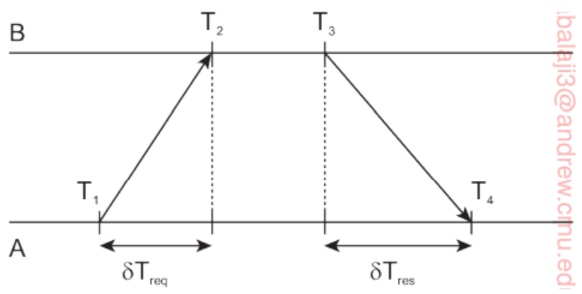


Figure 6.5: Getting the current time from a time server.

Consider a case, A will send a request to B, timestamped with value T1. B, in turn, will record the time of receipt T2 (taken from its own local clock), and returns a response timestamped with value T3, and piggybacking the previously recorded value T2. Finally, A records the time of the response's arrival, T4.

The Berkeley algorithm

In many clock synchronization algorithms 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. 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 UTC receiver. The time daemon's time must be set manually by the operator periodically. The method is illustrated in Figure 6.6.

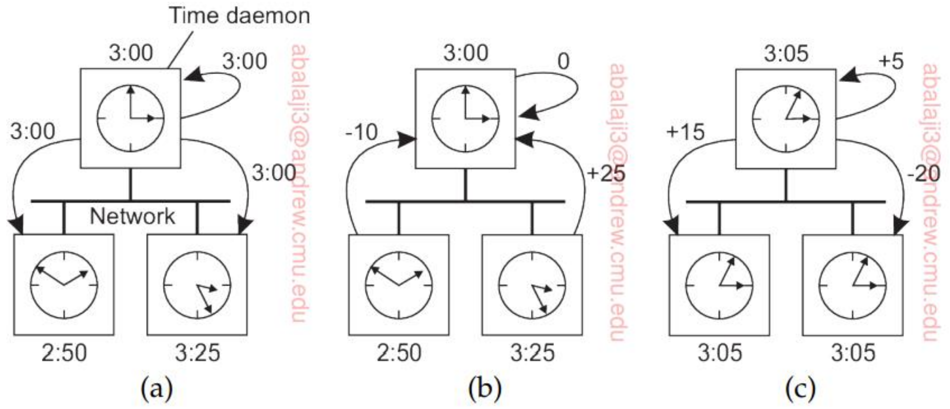


Figure 6.6: (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

In Figure 6.6(a) at 3:00, the time daemon tells the other machines its time and asks for theirs. In Figure 6.6(b) they respond with how far ahead or behind the time daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock [see Figure 6.6(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. The Berkeley algorithm is thus typically an internal clock synchronization algorithm.

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.

Reference broadcast synchronization (RBS) is a clock synchronization protocol that is quite different from other proposals. 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 UTC 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 sync, 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.

6.2 Logical Clocks

Lamport defines the logical clock by keeping track of events and the order in which the events occur. He showed that although clock synchronization is possible, it need not be complete. 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.

Lamport's logical clocks

To synchronize logical clocks, Lamport defined a relation called **happens-before**. The expression a → b is read "event a happens before event 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 → 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 → 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 → b and b → c, then a → c.

If two events, x and y, happen in different processes that do not exchange messages (not even indirectly via third parties), then x → y is not true, but neither is y → 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.

Now let us look at the algorithm Lamport proposed for assigning times to events. Consider the three processes depicted in Figure 6.8. The processes run on different machines, each with its own clock. For the sake of argument, we assume that a clock is implemented as a software counter: the counter is incremented by a specific value every T time units. However, the value by which a clock is incremented differs per process. The clock in process P1 is incremented by 6 units, 8 units in process P2, and 10 units in process P3, respectively. (Below, we explain that Lamport clocks are, in fact, event counters, which explains why their value may differ between processes.)

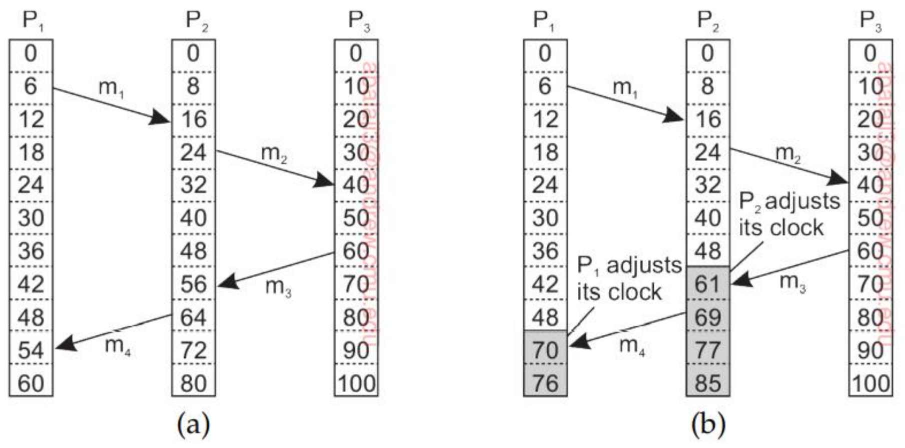


Figure 6.8: (a) Three processes, each with its own (logical) clock. The clocks run at different rates. (b) Lamport's algorithm corrects their values.

At time 6, process P1 sends message m1 to process P2. How long this message takes to arrive depends on whose clock you believe. In any event, the clock in process P2 reads 16 when it arrives. If the message carries the starting time, 6, in it, process P2 will conclude that it took 10 ticks to make the journey. This value is certainly possible. According to this reasoning, message m2 from P2 to P3 takes 16 ticks, again a plausible value.

Now consider message m3. It leaves process P3 at 60 and arrives at P2 at 56. Similarly, message m4 from P2 to P1 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 m3 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 Figure 6.8, we see that m3 now arrives at 61. Similarly, m4 arrives at 70.

To implement Lamport's logical clocks, each process Pi maintains a local counter Ci. These counters are updated according to the following steps:

1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), Pi increments Ci : Ci ← Ci + 1.
2. When process Pi sends a message m to process Pj , it sets m's timestamp ts(m) equal to Ci after having executed the previous step.
3. Upon the receipt of a message m, process Pj adjusts its own local counter as Cj ← max{Cj , ts(m)} after which it then executes the first step and delivers the message to the application.

Vector clocks

Vector Clock is an algorithm that generates partial ordering of events and detects causality violations in a distributed system. These clocks expands on scalar time to facilitate a causally consistent view of the distributed system, they detect whether a contributed event has caused another event in the distributed system. It eventually captures all the casual relationships. This algorithm helps us label every process with a vector (a list of integers) with and integer for each local clock of every process within the system. So, for N given process, there will be vector/array of size N.

1. Initially, all the clocks are set to zero.
2. Every time, an internal event occurs in a process, the value of the processor's logical clock in the vector is incremented by 1.
3. Also, every time a process sends a message, the value of the processor's logical clock in the vector is incremented by 1.
4. Every time, a process receives a message, the value of the processes' logical clock in the vector is incremented by 1, and moreover, each element is updated by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).

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.

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 passes it on.

The next one is a **permission-based approach**. In this case, a process wanting to access the resource first requires the permission from other processes. There are many different ways toward granting such a permission.

A centralized algorithm

A straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator. Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission. If no other process is currently accessing that resource, the coordinator sends back a reply granting permission. When the reply arrives, the requester can go ahead.

Now suppose that another process, P2 asks for permission to access the resource. The coordinator knows that a different process is already at the resource, so it cannot grant permission. The exact method used to deny permission is system dependent. The coordinator just refrains from replying, thus blocking process P2, which is waiting for a reply. Alternatively, it could send a reply saying “permission denied.” Either way, it queues the request from P2 for the time being and waits for more messages.

When process P1 is finished with the resource, it sends a message to the coordinator releasing its exclusive access. The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and accesses the resource.

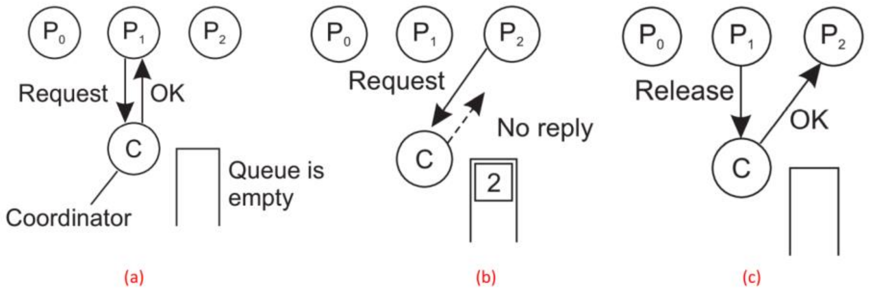


Figure: (a) Process P_1 asks for permission to access a shared resource. Permission is granted. (b) Process P_2 asks permission to access the same resource, but receives no reply. (c) When P_1 releases the resource, the coordinator replies to P_2.

It is easy to see that the algorithm guarantees mutual exclusion: the coordinator lets only one process at a time access the resource. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of resource (request, grant, release). Its simplicity makes it an attractive solution for many practical situations.

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

A distributed algorithm

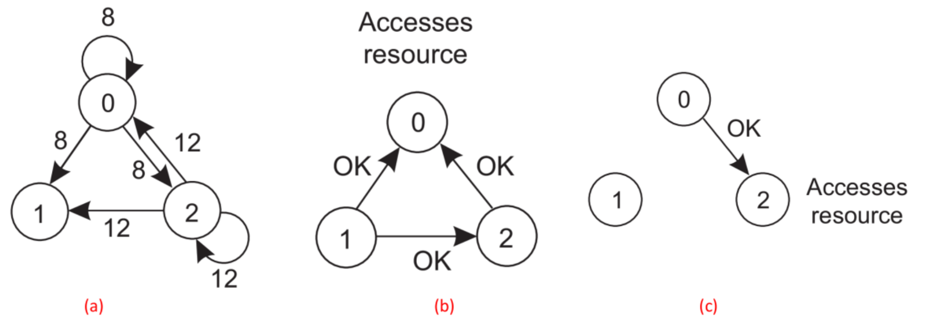
Using Lamport's logical clocks, and inspired by Lamport's original solution for distributed mutual exclusion. Their solution requires a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one actually happened first.

The algorithm works as follows. When a process wants to access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, no message is lost.

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

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

After sending out requests asking permission, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may go ahead. When it is finished, it sends OK messages to all processes in its queue and deletes them all from the queue. If there is no conflict, it clearly works. However, suppose that two processes try to simultaneously access the resource.



(a) Two processes want to access a shared resource at the same moment.
(b) P_0 has the lowest timestamp, so it wins.
(c) When process P_0 is done, it sends an *OK* also, so P_2 can now go ahead.

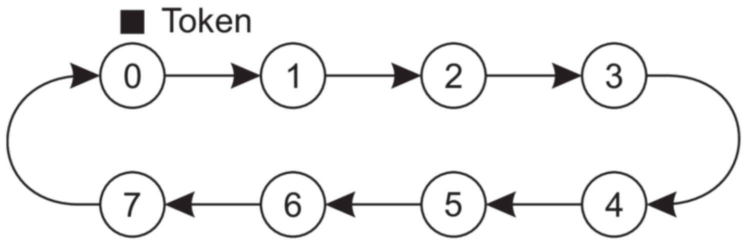
Process P0 sends everyone a request with timestamp 8, while at the same time, process P2 sends everyone a request with timestamp 12. P1 is not interested in the resource, so it sends OK to both senders. Processes P0 and P2 both see the conflict and compare timestamps. P2 sees that it has lost, so it grants permission to P0 by sending OK. Process P0 now queues the request from P2 for later processing and accesses the resource, as shown in Figure 6.16(b). When it is finished, it removes the request from P2 from its queue and sends an OK message to P2, allowing the latter to go ahead, as shown in Figure 6.16(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

With this algorithm, mutual exclusion is guaranteed without deadlock or starvation. Unfortunately, this algorithm has N points of failure, where N is the total number of processes. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter any of their respective critical regions. The algorithm can be patched up as follows. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent OK message.

A token-ring algorithm

A completely different approach to deterministically achieving mutual exclusion in a distributed system is illustrated in Figure below. In software, we construct an overlay network in the form of a logical ring in which each process is assigned a position in the ring. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process P0 is given a token. The token circulates around the ring.



When a process acquires the token from its neighbor, it checks to see if it needs to access the shared resource. If so, the process goes ahead, does all the work it needs to, and releases the resources. After it has finished, it passes the token along the ring. It is not permitted to immediately enter the resource again using the same token.

If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along. As a consequence, when no processes need the resource, the token just circulates around the ring.

This algorithm has its own problems. If the token is ever lost, for example, because its holder crashes or due to a lost message containing the token, it must be regenerated. In fact, detecting that it is lost may be difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.

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

6.4 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.

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 P has a unique identifier id(P). In general, election algorithms attempt to locate the process with the highest identifier and designate it as coordinator. The algorithms differ in the way they locate the coordinator.

Furthermore, we also assume that every process knows the identifier of every other process. In other words, each process has complete knowledge of the process group in which a coordinator must be elected. 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.

The bully algorithm

A well-known solution for electing a coordinator is the bully algorithm devised by Garcia-Molina [1982]. In the following, we consider N processes {P0, . . . , PN−1} and let id(Pk) = k. When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, Pk , holds an election as follows:

1. Pk sends an ELECTION message to all processes with higher identifiers: Pk+1, Pk+2, . . . , PN−1.
2. If no one responds, Pk wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and Pk ’s job is done.

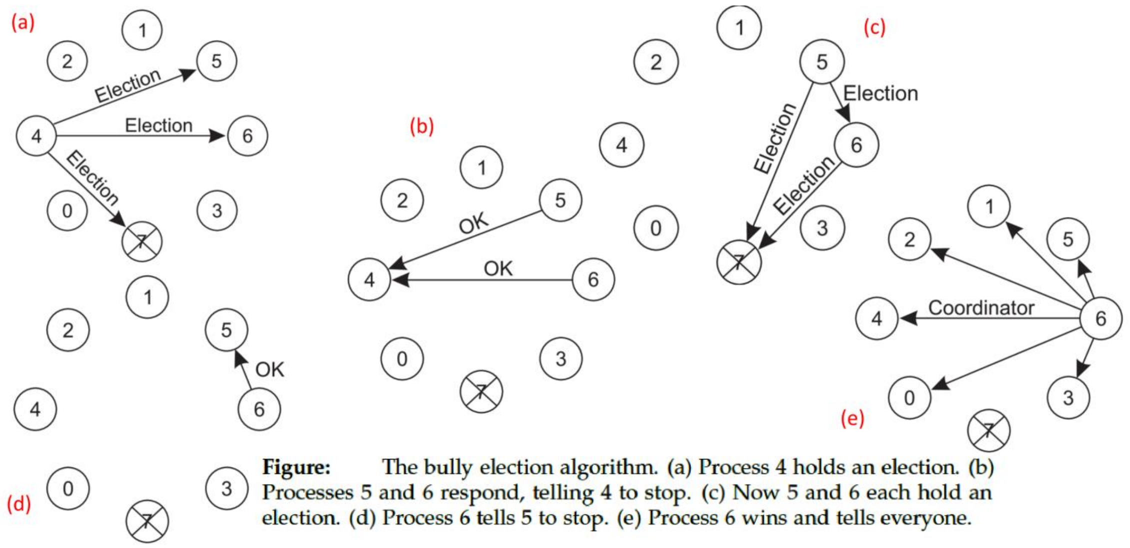


Figure: The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

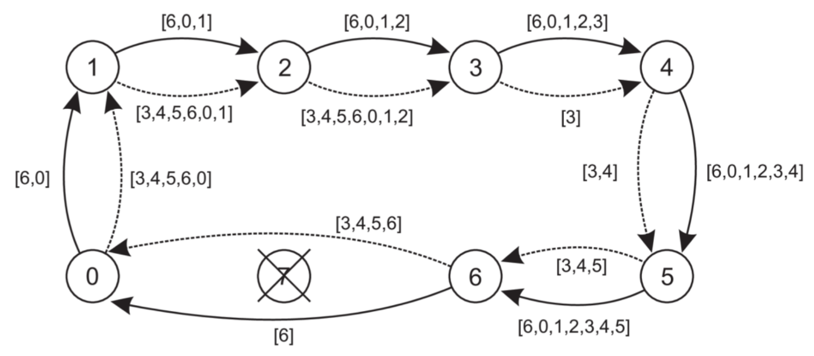
At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, 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. Thus the biggest guy in town always wins, hence the name "bully algorithm."

A ring algorithm

Consider the following election algorithm that is based on the use of a (logical) ring. Unlike some ring algorithms, this one does not use a token. We assume 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 identifier 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 identifier 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 identifier. 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 identifier) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.



- The solid line shows the election messages initiated by P_6
- The dashed one the messages by P_3

6.5 Location System

When looking at very large distributed systems that are dispersed across a wide-area network, it is often necessary to take proximity into account. Just imagine a distributed system organized as an overlay network in which two processes are neighbors in the overlay network, but are actually placed far apart in the underlying network. If these two processes communicate a lot, it may have been better to ensure that they are also physically placed in each other's proximity.

GPS: Global Positioning System

Let us start by considering how to determine your 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 initially was used mainly for military applications, it by now has found its way to many civilian applications, notably for traffic navigation. However, many more application domains exist. For example, modern smartphones now allow owners to track each other's position. This principle can easily be applied to tracking other things as well, including pets, children, cars, boats, and so on.

GPS uses up to 72 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 four satellites. To explain, let us first assume that all clocks, including the receiver's, are synchronized. We need to know the distance to four satellites to determine the longitude, latitude, and altitude of a receiver on Earth. This positioning is all fairly straightforward, but determining the distance to a satellite becomes complicated when we move from theory to practice. There are at least 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 sync with that of a satellite.

<u>When GPS is not an option?</u>

A major drawback of GPS is that it can generally not be used indoors. For that purpose, other techniques are necessary. An increasingly popular technique is to make use of the numerous WiFi access points available. The basic idea is simple: if we have a database of known access points along with their coordinates, and we can estimate our distance to an access point, then with only three detected access points, we should be able to compute our position. Of course, it really is not that simple at all.

6.6 Distributed Event Matching

Event matching, or more precisely, notification filtering, is at the heart of publish-subscribe systems. The problem boils down to the following:

- A process specifies through a subscription S in which events it is interested.
- When a process publishes a notification N on the occurrence of an event, the system needs to see if S matches N.
- In the case of a match, the system should send the notification N, possibly including the data associated with the event that took place, to the subscriber.

As a consequence, we need to facilitate at least two things:

1. Matching subscriptions against events, and
2. Notifying a subscriber in the case of a match.

The two can be separated, but this need not always be the case. In the following, we assume the existence of a function match(S, N) which returns true when subscription S matches the notification N, and false otherwise.

Centralized implementations

A simple, naive implementation of event matching is to have a fully centralized server that handles all subscriptions and notifications. In such a scheme, a subscriber simply submits a subscription, which is subsequently stored. When a publisher submits a notification, that notification is checked against each and every subscription, and when a match is found, the notification is copied and forwarded to the associated subscriber.

Obviously, this is not a very scalable solution. Nevertheless, provided the matching can be done efficiently and the server itself has enough processing power, the solution is feasible for many cases.

The idea of having a central server can be extended by distributing the matching across multiple servers and dividing the work. The servers, generally referred to as brokers, are organized into an overlay network. The issue then becomes how to route notifications to the appropriate set of subscribers. There can be three different classes:

1. flooding,
2. selective routing, and
3. gossip-based dissemination.

A straightforward way to make sure that notifications reach their subscribers, is to deploy broadcasting. There are essentially two approaches. First, we store each subscription at every broker, while publishing notifications only a single broker. The latter will handle identifying the matching subscriptions and subsequently copy and forward the notification. The alternative is to store a subscription only at one broker while broadcasting notifications to all brokers. In that case, matching is distributed across the brokers which may lead to a more balanced workload among the brokers.

When systems become big, flooding is not the best way to go, if even possible. Instead, routing notifications across the overlay network of brokers may be necessary. This is typically the way to go in information-centric networking, which makes use of name-based routing.

6.7 Gossip-based coordination

A gossip protocol or epidemic protocol is a procedure or process of computer peer-to-peer communication that is based on the way epidemics spread. Some distributed systems use peer-to-peer gossip to ensure that data is disseminated to all members of a group.

Some of the gossip based coordination protocols are:

- **Dissemination protocols (or rumor-mongering protocols):** These use gossip to spread information; they basically work by flooding agents in the network, but in a manner that produces bounded worst-case loads:
    1. **Event dissemination protocols** use gossip to carry out multicasts. They report events, but the gossip occurs periodically and events don't actually trigger the gossip. One concern here is the potentially high latency from when the event occurs until it is delivered.
    2. **Background data dissemination protocols** continuously gossip about information associated with the participating nodes. Typically, propagation latency isn't a concern, perhaps because the information in question changes slowly or there is no significant penalty for acting upon slightly stale data.
- **Protocols that compute aggregates**: These compute a network-wide aggregate by sampling information at the nodes in the network and combining the values to arrive at a system-wide value – the largest value for some measurement nodes are making, smallest, etc. The key requirement is that the aggregate must be computable by fixed-size pairwise information exchanges; these typically terminate after a number of rounds of information exchange logarithmic in the system size, by which time an all-to-all information flow pattern will have been established. As a side effect of aggregation, it is possible to solve other kinds of problems using gossip; for example, there are gossip protocols that can arrange the nodes in a gossip overlay into a list sorted by node-id (or some other attribute) in logarithmic time using aggregation-style exchanges of information. Similarly, there are gossip algorithms that arrange nodes into a tree and compute aggregates such as "sum" or "count" by gossiping in a pattern biased to match the tree structure.

Practice Questions

1. Explain the coordination in distributed system.
2. Why is clock synchronization required in distributed system? Explain.
3. What the logical clocks and physical clocks? Explain.
4. Distinguish between Lamport's and Vector clock? Explain.
5. What is the role of coordinator in distributed system? Explain different approaches of electing a coordinator.
6. What is an election algorithm? Why is it required in distributed system?
7. How is clock synchronization done in Cristian's method? Explain.
8. Explain the Berkley's algorithm. What are the issues resolved by Berkley's algorithm?
9. Explain Network Time Protocol in brief.
10. Write short notes on:
    a. Gossip-based coordination
    b. Clock drift rate
    c. Coordinated Universal Time