Process Coordination

Motivation

- How can 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.

System Models

A *system model* encodes expectations about the behavior of processes, communication links, and timing;

think of it as a set of assumptions that allow us to reason about distributed systems by ignoring the complexity of the actual technologies used to implement them.

Common Models for Communication

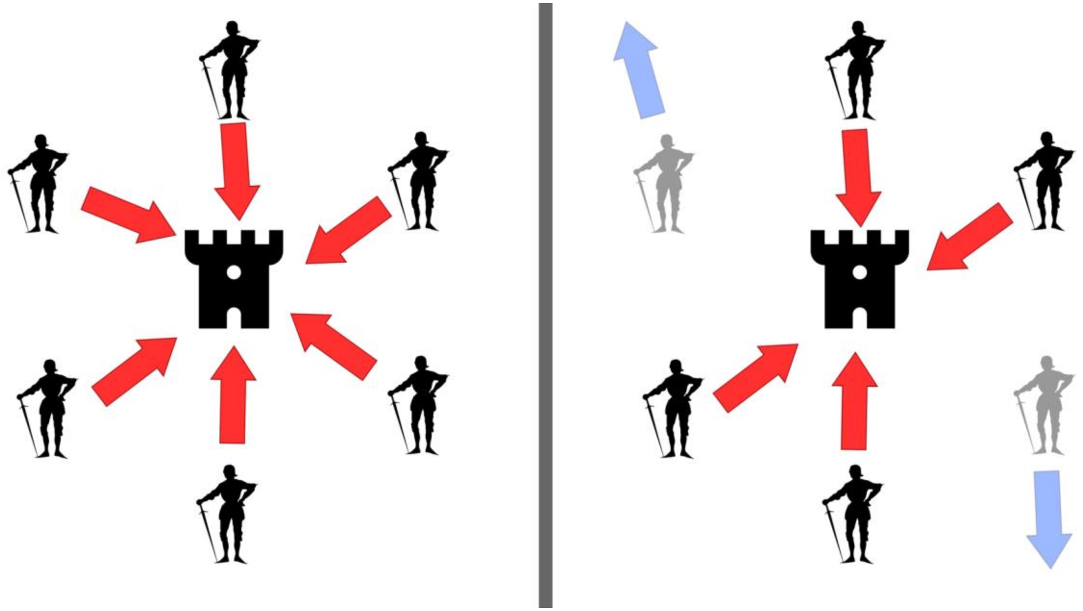
- The *fair-loss link* model assumes that messages may be lost and duplicated, but if the sender keeps retransmitting a message, eventually it will be delivered to the destination.
- The *reliable link* model assumes that a message is delivered exactly once, without loss or duplication. A reliable link can be implemented on top of a fair-loss one by de-duplicating messages at the receiving side.
- The *authenticated reliable link* model makes the same assumptions as the reliable link but additionally assumes that the receiver can authenticate the sender.

Process Model

- The *arbitrary-fault* model assumes that a process can deviate from its algorithm in arbitrary ways, leading to crashes or unexpected behaviors caused by bugs or malicious activity. For historical reasons, this model is also referred to as the **"Byzantine" model**. More interestingly, it can be theoretically proven that a system using this model can tolerate up to 1/3 of faulty processes and still operate correctly.
- The ***crash-recovery*** **model** assumes that a process doesn't deviate from its algorithm but can crash and restart at any time, losing its in-memory state.
- The *crash-stop* model assumes that a process doesn't deviate from its algorithm but doesn't come back online if it crashes. Although this seems unrealistic for software crashes, it models unrecoverable hardware faults and generally makes the algorithms simpler.

Arbitrary Fault Model

A Byzantine fault is any fault presenting different symptoms to different observers. A Byzantine failure is the loss of a system service due to a Byzantine fault in systems that require consensus among distributed nodes.



The arbitrary-fault model is typically used to model safety-critical systems like airplane engines, nuclear power plants, and systems where a single entity doesn’t fully control all the processes (e.g.,digital cryptocurrencies such as Bitcoin).

These use cases are outside the scope for the course, and the algorithms presented here will generally assume a crash-recovery model.

Timing Model

- The *synchronous* model assumes that sending a message or executing an operation never takes more than a certain amount of time.
  - Not very realistic for the type of systems we care about, where we know that sending messages over the network can potentially take a very long time, and processes can be slowed down by, e.g., garbage collection cycles or page faults.
- The *asynchronous* model assumes that sending a message or executing an operation on a process can take an unbounded amount of time.
  - Many problems can't be solved under this assumption; if sending messages can take an infinite amount of time, algorithms can get stuck and not make any progress at all.
  - Nevertheless, this model is useful because it's simpler than models that make timing assumptions, and therefore algorithms based on it are also easier to implement.
- **The *partially synchronous*** model assumes that the system behaves synchronously most of the time. This model is typically representative enough of real-world systems.

Failure Detection

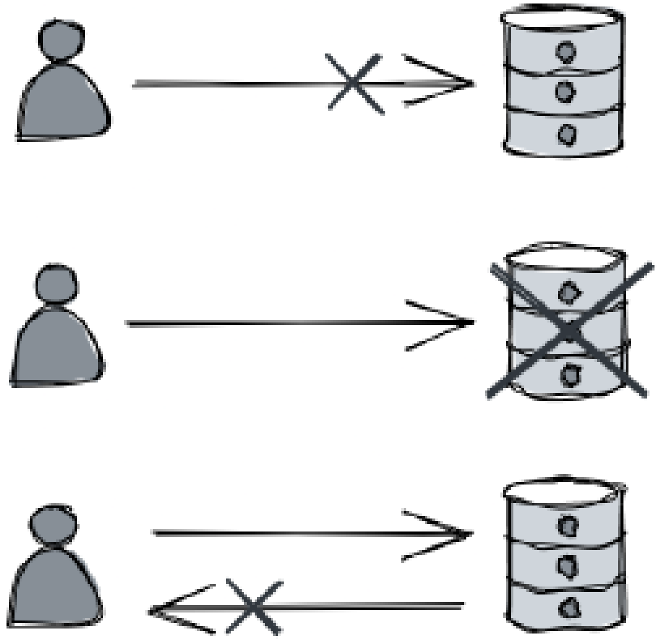


Figure 7.1: The client can't tell whether the server is slow, it crashed or a message was delayed/dropped because of a network issue.

It is impossible to tell whether the server is just very slow, it crashed, or a message couldn't be delivered because of a network issue

The client can configure a timeout to trigger if it hasn't received a response from the server after a certain amount of time.

If and when the timeout triggers, the client considers the server unavailable and either throws an error or retries the request.

Pro-active detection

- A *ping* is a periodic request that a process sends to another to check whether it’s still available.
  - The process expects a response to the ping within a specific time frame. If no response is received, a timeout triggers and the destination is considered unavailable.
  - However, the process will continue to send pings to it to detect if and when it comes back online.
- A *heartbeat* is a message that a process periodically sends to another.
  - If the destination doesn’t receive a heartbeat within a specific time frame, it triggers a timeout and considers the process unavailable.
  - But if the process comes back to life later and starts sending out heartbeats, it will eventually be considered to be available again.

Time and Order

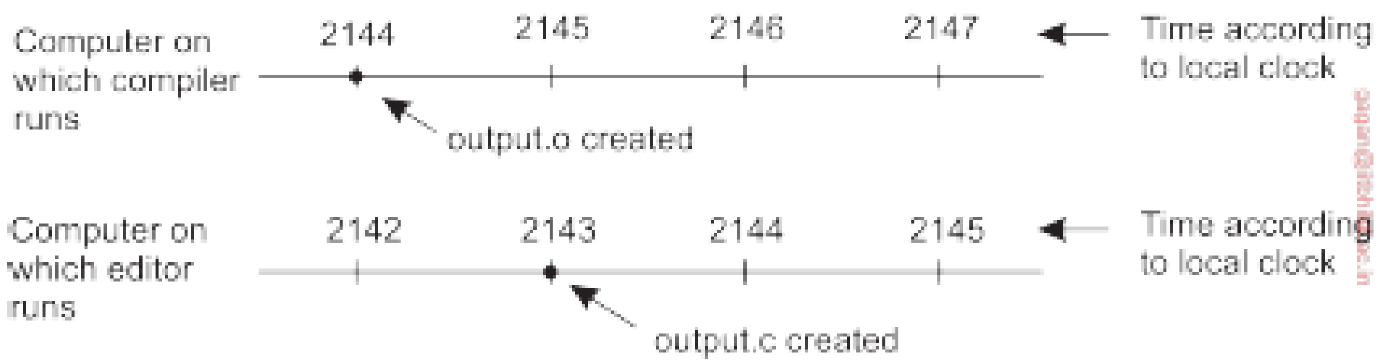
- The flow of execution of a single-threaded application is simple to understand because every operation executes sequentially in time, one after the other.
- But in a distributed system, there is no shared global clock that all processes agree on that can be used to order operations. And, to make matters worse, processes can run concurrently.
- We will learn about a family of clocks that can be used to work out the order of operations across processes in a distributed system.

Physical Clocks

- A process has access to a physical wall-time clock. The most common type is based on a vibrating quartz crystal, which is cheap but not very accurate.
- The rate at which a clock runs faster or slower is also called *clock drift*.
- In contrast, the difference between two clocks at a specific point in time is referred to as *clock skew*.
- Because quartz clocks drift, they need to be synced periodically with machines that have access to higher-accuracy clocks, like atomic ones.
- Atomic clocks measure time based on quantum-mechanical properties of atoms. They are significantly more expensive than quartz clocks and accurate to 1 second in 3 million years.

Motivation (make)

- When the programmer has finished changing all the source files, she runs **make**, which examines the times at which all the source and object files were last modified.
- If the source file input.c has time 2151 and the corresponding object file input.o has time 2150, make knows that input.c has been changed since input.o was created, and thus input.c must be recompiled.
- On the other hand, if output.c has time 2144 and output.o has time 2145, no compilation is needed.
- Thus make goes through all the source files to find out which ones need to be recompiled and calls the compiler to recompile them.

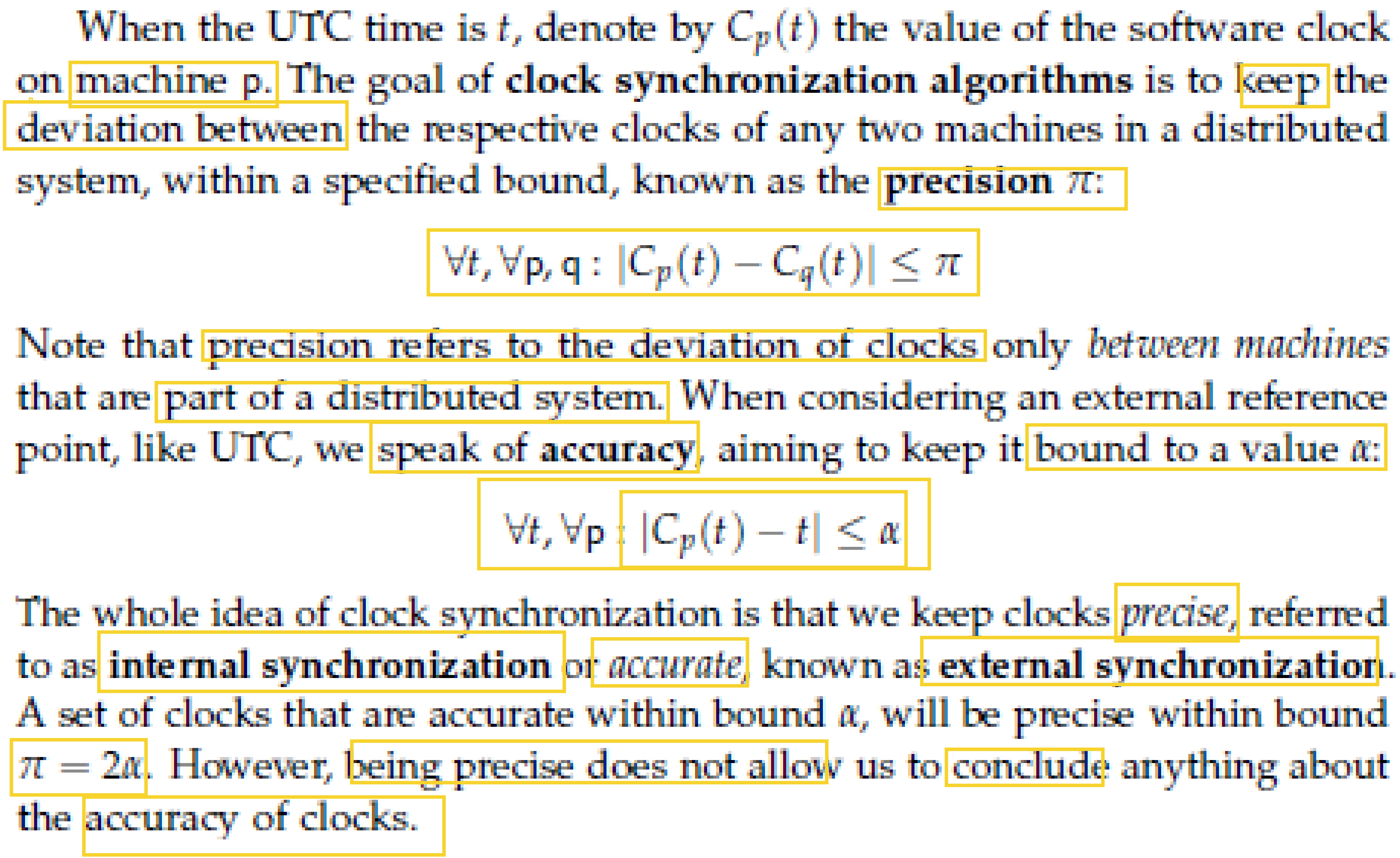


Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

Figure 5.1: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

- In a distributed system in which there was no global agreement on time.
- Suppose that output.o has time 2144 as above, and shortly thereafter output.c is modified but is assigned time 2143 because the clock on its machine is slightly behind, as shown in Figure 5.1.
- Make will not call the compiler. The resulting executable binary program will then contain a mixture of object files from the old sources and the new sources.
- It may crash, and the programmer will go crazy trying to understand what is wrong with the code.

Clock Synchronization



When the UTC time is t, denote by $C_p(t)$ the value of the software clock on machine p. The goal of **clock synchronization algorithms** is to keep the deviation between the respective clocks of any two machines in a distributed system, within a specified bound, known as the **precision** π:

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

Note that precision refers to the deviation of clocks only *between machines* that are part of a distributed system. When considering an external reference point, like UTC, we speak of **accuracy**, aiming to keep it bound to a value α:

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

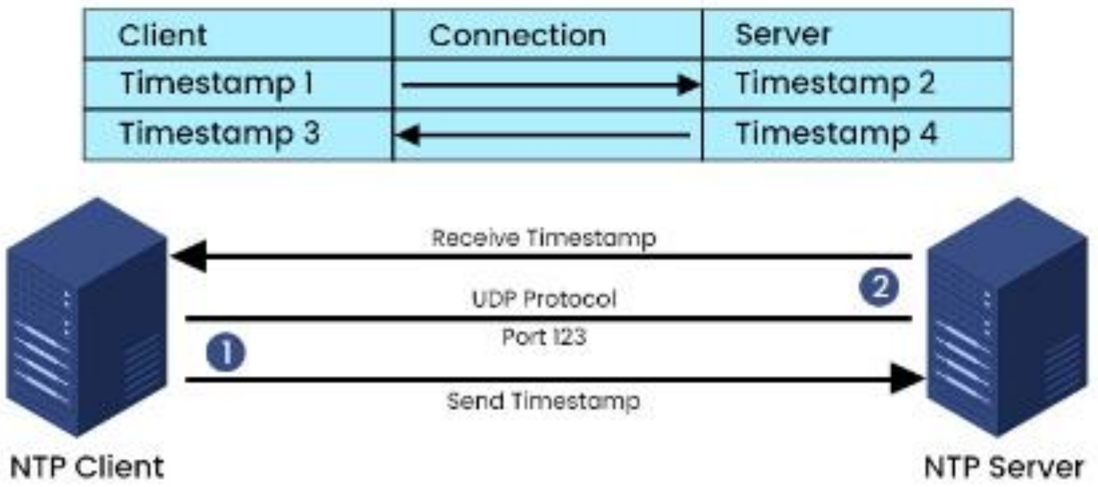
The whole idea of clock synchronization is that we keep clocks *precise*, referred to as **internal synchronization** or *accurate*, known as **external synchronization**. A set of clocks that are accurate within bound α, will be precise within bound $\pi = 2\alpha$. However, being precise does not allow us to conclude anything about the accuracy of clocks.

Clock synchronization

- The synchronization between clocks can be implemented with a protocol, and the challenge is to do so despite the unpredictable latencies introduced by the network.
- The most commonly used protocol is the *Network Time Protocol* (NTP). In NTP, a client estimates the clock skew by receiving a timestamp from a NTP server and correcting it with the estimated network latency.
- With an estimate of the clock skew, the client can adjust its clock. However, this causes the clock to jump forward or backward in time, which creates a problem when comparing timestamps.
- For example, an operation that runs after another could have an earlier timestamp because the clock jumped back in time between the two operations.

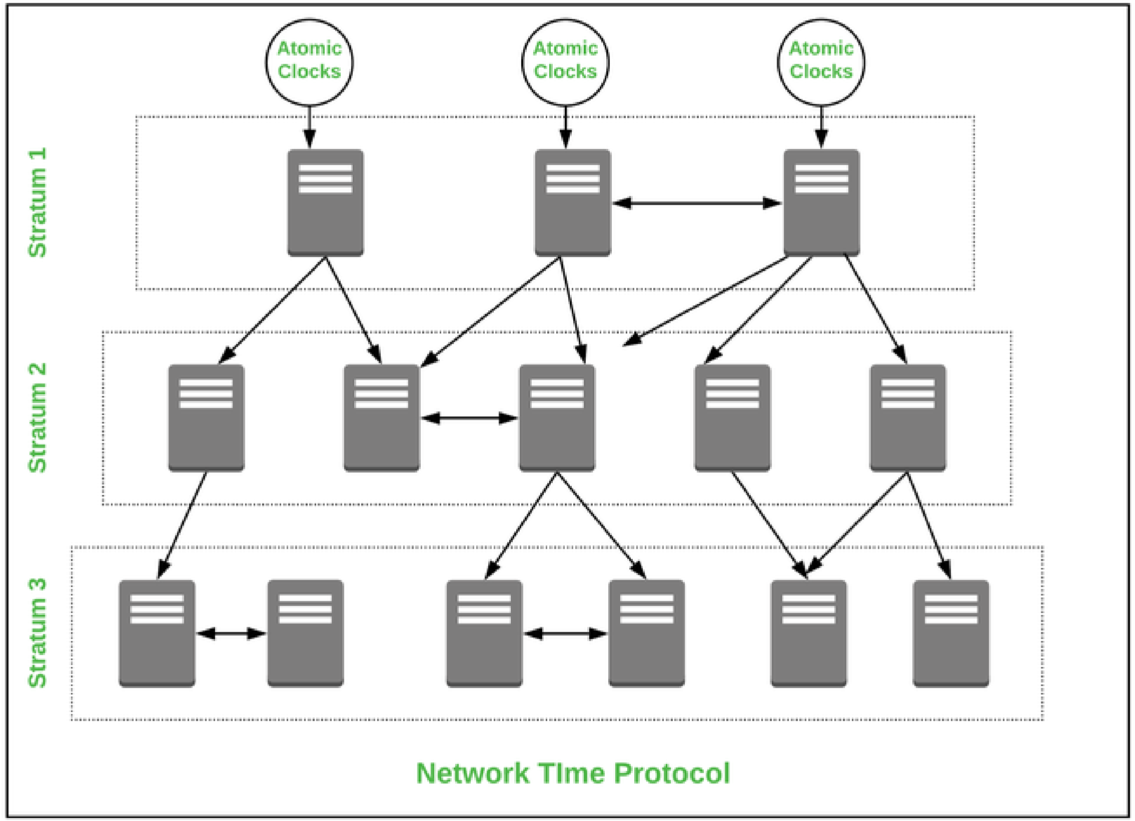
https://www.pynetlabs.com/what-is-ntp-network-time-protocol/
https://datatracker.ietf.org/doc/html/rfc5905

NTP Protocol



Client	Connection	Server
Timestamp 1	→	Timestamp 2
Timestamp 3	←	Timestamp 4

Offset = ((T2 - T1) + (T3 - T4)) / 2
Delay = (T4 - T1) - (T3 - T2)



- NTP is a protocol that works over the application layer, it uses a hierarchical system of time resources and provides synchronization within the stratum servers.
- First, at the topmost level, there is highly accurate time resources' ex. atomic or GPS clocks. These clock resources are called stratum 0 servers, and they are linked to the below NTP server called Stratum 1,2 or 3 and so on.
- These servers then provide the accurate date and time so that communicating hosts are synced to each other.

NTP Example

- Client considers its time as the wrong time and server time as the true time.
- Client must adjust their timing based on the response provided by the NTP server.
- Client sends the request at the wrong time, i.e. (**T1 = 50 sec**)
- Server gets the request, at **T2 = 80 sec** because of some connectivity issues.
- The server might get busy at the time of sending back the accurate time, so it sends back the request to the client at a time T3; let's say **T3 = 90 sec**.
- The client receives the time from the server; let's assume at **T4 = 70 sec**.

Q: What should the client set its time to?

Solved example

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

- Delay can be calculated as:

  (T4-T1) – (T3-T2) = (70 – 50) – (90 – 80) = 10 seconds.

- The client has estimated the elapsed time it took for the server's answer to reach the client as: 10 / 2 = 5 seconds

  (We divided by 2 as it is a two-way communication).

- Client adds 5 sec to the time when the server sends the accurate time, i.e., **T3 (90 + 5 = 95).**
- To calculate the delay, the client now knows he has to subtract the time at which he gets the response from the server and the time at which the server sends the response, i.e., T3 – T4 = 95 – 70 = 25.
- Hence 25 is the delay that the client faces. So, the **client will add 25 seconds to its clock.**

Logical Clocks

- A *logical clock* measures the passing of time in terms of logical operations, not wall-clock time.
  - The simplest possible logical clock is a counter, incremented before an operation is executed.
  - Doing so ensures that each operation has a distinct *logical timestamp*.
  - If two operations execute on the same process, then necessarily one must come before the other, and their logical timestamps will reflect that.
- Operations executed on different processes?
  - Imagine sending an email to a friend. Any actions you did before sending that email, like drinking juice, must have happened before the actions your friend took after receiving the email.
  - Similarly, when one process sends a message to another, a so-called *synchronization point* is created.
  - The operations executed by the sender before the message was sent *must* have happened before the operations that the receiver executed after receiving it.

Lamport Clock

- A logical clock based on the idea in previous slide
- Each process in the systems has a local counter that follows:
  - The counter is initialized with 0.
  - The process increments its counter by 1 before executing an operation.
  - When the process sends a message, it increments its counter by 1 and sends a copy of the counter in the message.
  - When the process receives a message, it merges the counter it received with its local counter by taking the maximum of the two. Finally, it increments the counter by 1.

Although the Lamport clock assumes a crash-stop model, a crash-recovery one can be supported by e.g., persisting the clock's state on disk.

Lamport Clock (cont)

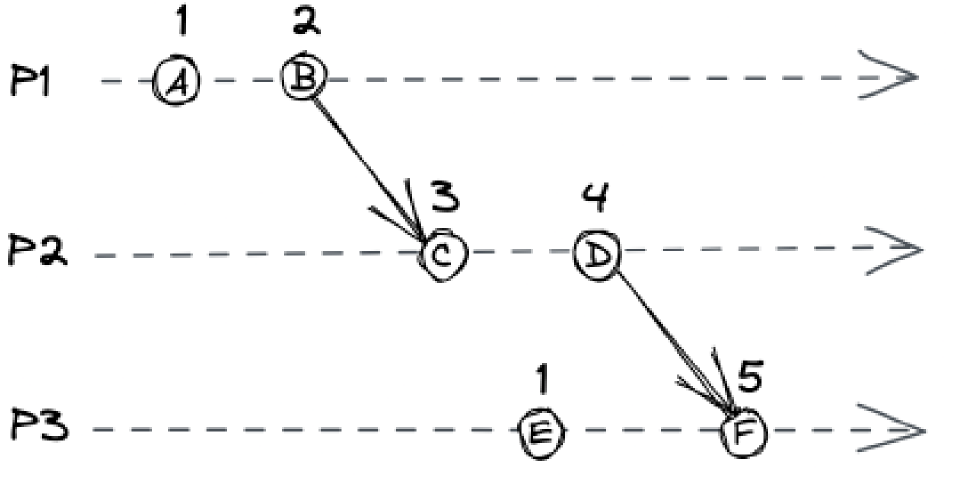


Figure 8.1: Three processes using Lamport clocks. For example, because D happened before F, D's logical timestamp is less than F's.

- The rules guarantee that if operation *O*1 **happened-before** operation *O*2, the logical timestamp of *O*1 is less than the one of *O*2.
  - Operation D happened-before F, and their logical timestamps, 4 and 5, reflect that.
- However, two unrelated operations can have the same logical timestamp.
  - Logical timestamps of operations A and E are equal to 1.
- To create a strict total order, we can arbitrarily order the processes to break ties.
  - If we used the process IDs to break ties (1, 2, and 3), E's timestamp would be greater than A's.
- The order of logical timestamps doesn't imply a causal relationship.
  - Operation E didn't happen-before C, even if their timestamps appear to imply it.
  - To guarantee this relationship, we have to use vector clocks.

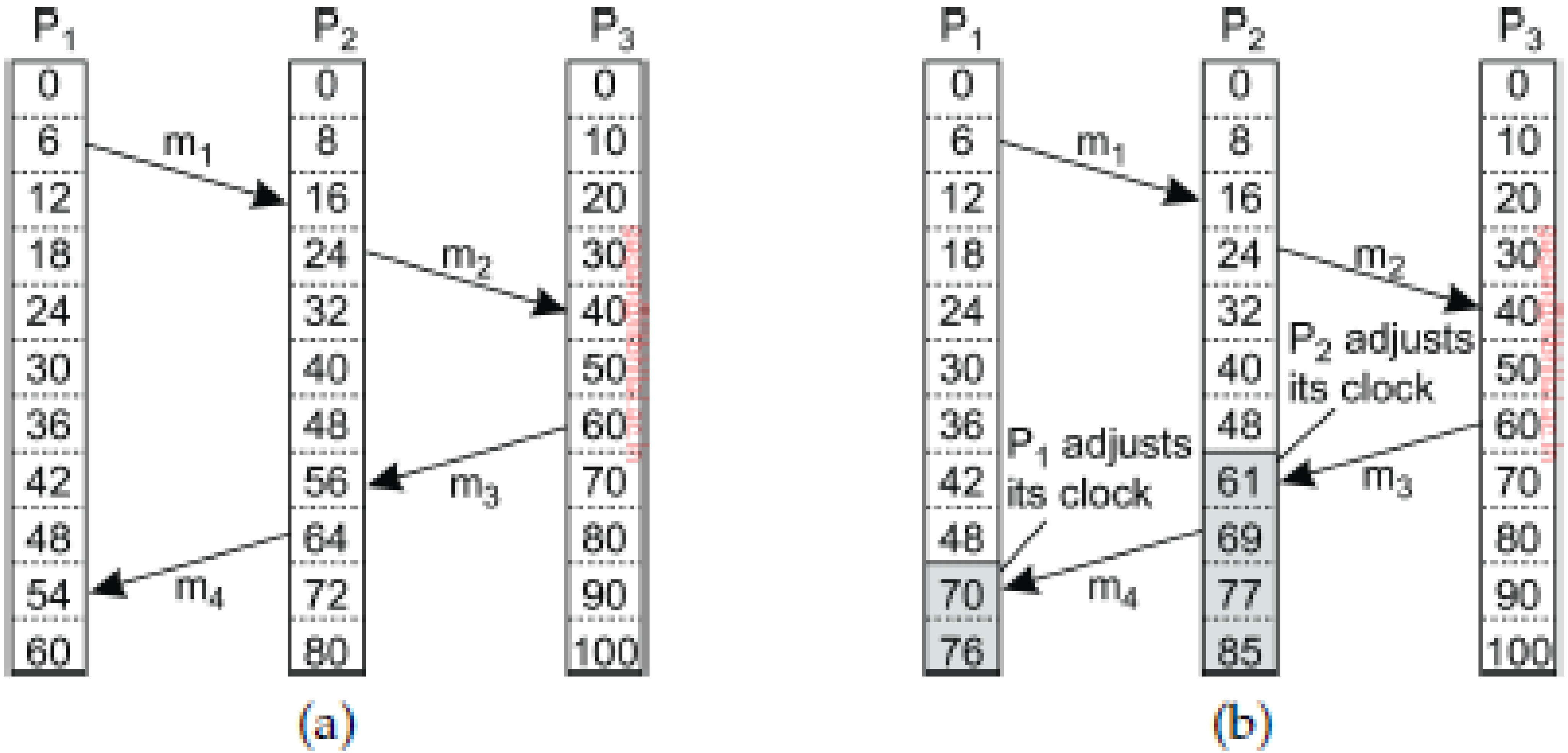
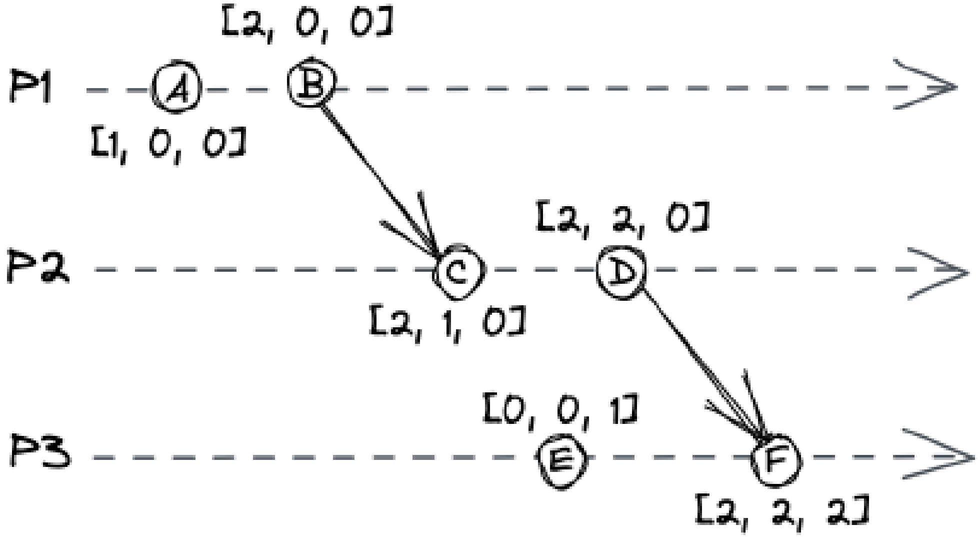


Figure 5.7: (a) Three processes, each with its own (logical) clock. The clocks run at different rates. (b) Lamport's algorithm corrects their values.

Vector Clocks

- A *vector clock* is a logical clock that guarantees that if a logical timestamp is less than another, then the former must have happened-before the latter.
- A vector clock is implemented with an array of counters, one for each process in the system. And, as with Lamport clocks, each process has its local copy.
- For example, suppose the system is composed of three processes, $P\ 1$, $P\ 2$, and $P\ 3$.
- In this case, each process has a local vector clock implemented with an array of three counters [$CP\ 1$, $CP\ 2$, $CP\ 3$].
- The first counter in the array is associated with $P\ 1$, the second with $P\ 2$, and the third with $P\ 3$.

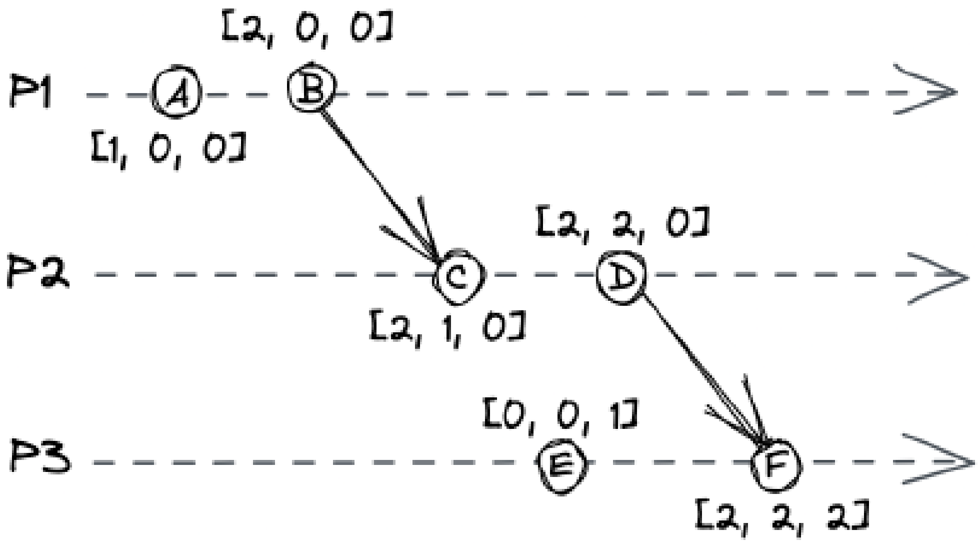
How it works?



A process updates its local vector clock based on the following rules:

- Initially, the counters in the array are set to 0.
- When an operation occurs, the process increments its counter in the array by 1.
- When the process sends a message, it increments its counter in the array by 1 and sends a copy of the array with the message.
- When the process receives a message, it merges the array it received with the local one by taking the maximum of the two arrays element-wise. Finally, it increments its counter in the array by 1.

Benefits



The beauty of vector clock timestamps is that they can be partially ordered; given two operations $O1$ and $O2$ with timestamps $T1$ and $T2$, if:

- every counter in $T1$ is less than or equal to the corresponding counter in $T2$,
- and there is at least one counter in $T1$ that is strictly less than the corresponding counter in $T2$,

then $O1$ happened-before $O2$. For example, in Fig, B happened-before C.

- If $O1$ didn't happen-before $O2$ and $O2$ didn't happen-before $O1$, then the timestamps can't be ordered, and the operations are considered to be concurrent. So, for example, operations E and C in Fig can't be ordered, and therefore are concurrent.

- One problem with vector clocks is that the storage requirement on each process grows linearly with the number of processes, which becomes a problem for applications with many clients.
- However, there are other types of logical clocks that solve this issue, like dotted version vectors
- What's important to internalize at this point is that, in general, we can't use physical clocks to accurately derive the order of events that happened on different processes.
- That being said, sometimes physical clocks are good enough. For example, using physical clocks to timestamp logs may be fine if they are only used for debugging purposes

Leader Election

- **Need:** A single process in the system needs to have special powers, like accessing a shared resource or assigning work to others.
- To grant a process these powers, the system needs to elect a *leader* among a set of *candidate processes*, which remains in charge until it relinquishes its role or becomes otherwise unavailable.
  - When that happens, the remaining processes can elect a new leader among themselves.
- A leader election algorithm needs to guarantee that there is at most one leader at any given time and that an election eventually completes even in the presence of failures.
  - These two properties are also referred to as *safety* and *liveness*, respectively, and they are general properties of distributed algorithms.
  - Informally, safety guarantees that nothing bad happens and liveness that something good eventually does happen.

Raft Leader Election

Raft's leader election algorithm is implemented as a state machine in which any process is in one of three states (see Fig):

- the *follower state,* where the process recognizes another one as the leader;
- the *candidate state*, where the process starts a new election proposing itself as a leader;
- or the *leader state,* where the process is the leader.

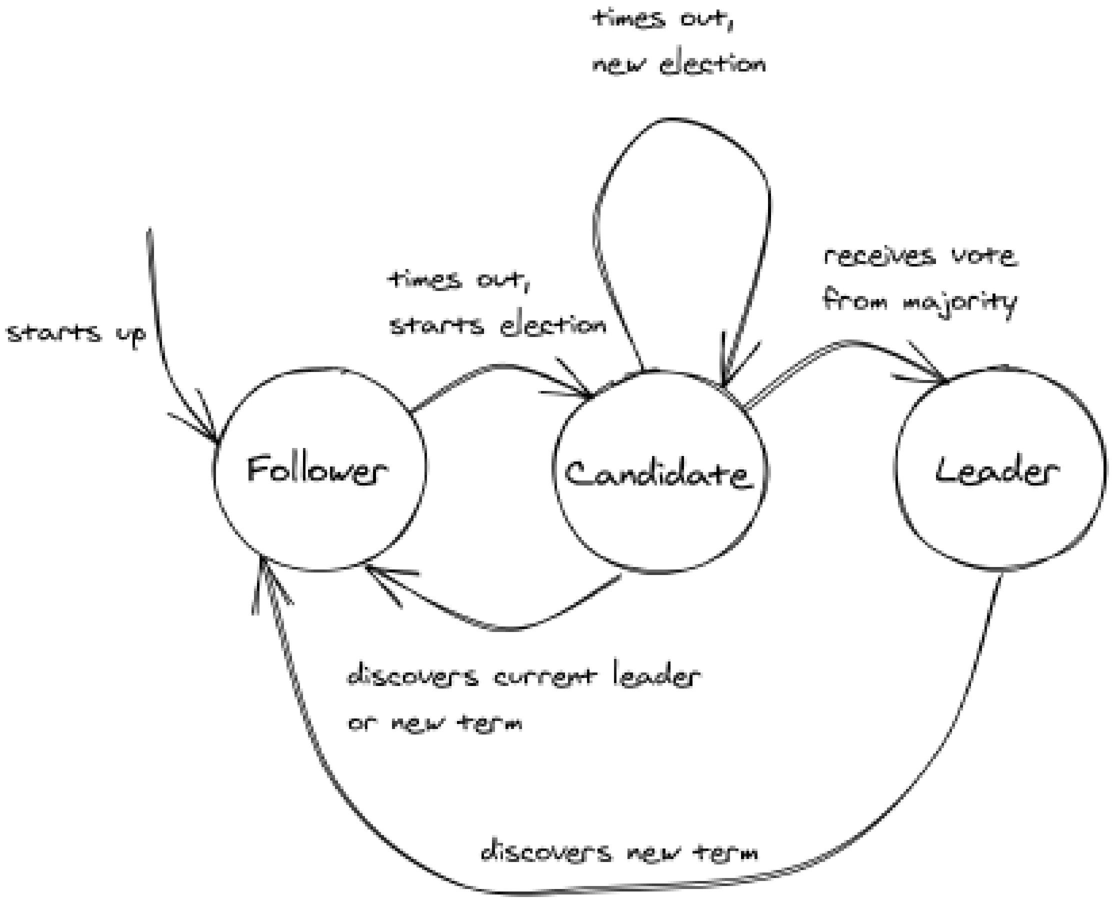


Figure 9.1: Raft's leader election algorithm represented as a state machine.

Election terms

- In Raft, time is divided into *election terms* of arbitrary length that are numbered with consecutive integers (i.e., logical timestamps).
- A term begins with a new election, during which one or more candidates attempt to become the leader. The algorithm guarantees that there is at most one leader for any term. But what triggers an election in the first place?
- When the system starts up, all processes begin their journey as followers.
- A follower expects to receive a periodic heartbeat from the leader containing the election term the leader was elected in.
- If the follower doesn’t receive a heartbeat within a certain period of time, a timeout fires and the leader is presumed dead.
- At that point, the follower starts a new election by incrementing the current term and transitioning to the candidate state. It then votes for itself and sends a request to all the processes in the system to vote for it, stamping the request with the current election term.

Till when process remains a candidate?

- **The candidate wins the election** — The candidate wins the election if the majority of processes in the system vote for it. Each process can vote for at most one candidate in a term on a first-come-first-served basis. This majority rule enforces that at most one candidate can win a term. If the candidate wins the election, it transitions to the leader state and starts sending heartbeats to the other processes.
- **Another process wins the election** — If the candidate receives a heartbeat from a process that claims to be the leader with a term greater than or equal to the candidate's term, it accepts the new leader and returns to the follower state. If not, it continues in the candidate state.
  - How that could happen? Eg. if the candidate process was to stop for any reason, like for a long garbage collection pause, by the time it resumes another process could have won the election.
- **A period of time goes by with no winner** — It's unlikely but possible that multiple followers become candidates simultaneously, and none manages to receive a majority of votes; this is referred to as a split vote. The candidate will eventually time out and start a new election when that happens. The election timeout is picked randomly from a fixed interval to reduce the likelihood of another split vote in the next election.

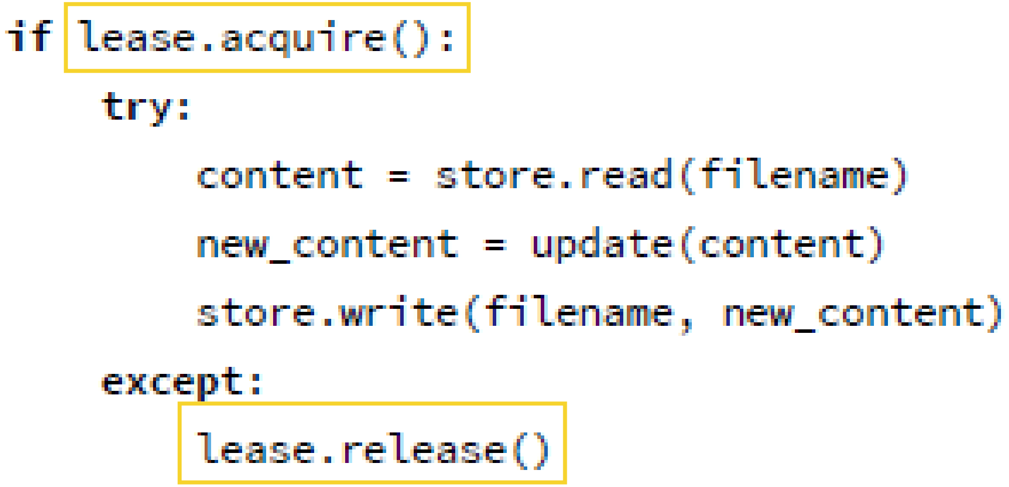
Compare-And-Swap based election

- The compare-and-swap operation atomically updates the value of a key if and only if the process attempting to update the value correctly identifies the current value.
- The operation takes three parameters: K, Vo, and Vn, where K is a key, and Vo and Vn are values referred to as the old and new value, respectively.
- The operation atomically compares the current value of K with Vo, and if they match, it updates the value of K to Vn. If the values don't match, then K is not modified, and the operation fails.
- The expiration time defines the time to live for a key, after which the key expires and is removed from the store unless the expiration time is extended.
- The idea is that each competing process tries to acquire a *lease* by creating a new key with compare-and-swap.
- The first process to succeed becomes the leader and remains such until it stops renewing the lease, after which another process can become the leader.

Is this enough?

Suppose multiple processes need to update a file on a shared file store, and we want to guarantee that only one at a time can access it to avoid race conditions.

Now, suppose we use a lease to lock the critical section. Each process tries to acquire the lease, and the one that does so successfully reads the file, updates it in memory, and writes it back to the store:



```
if lease.acquire():
    try:
        content = store.read(filename)
        new_content = update(content)
        store.write(filename, new_content)
    except:
        lease.release()
```

Issue

- The issue is that by the time the process gets to write to the file, it might no longer hold the lease. For example, the operating system might have preempted and stopped the process for long enough for the lease to expire.
- The process could try to detect that by comparing the lease expiration time to its local clock before writing to the store, assuming clocks are synchronized.
- However, clock synchronization isn’t perfectly accurate. On top of that, the lease could expire while the request to the store is in-flight because of a network delay.
- To account for these problems, the process could check that the lease expiration is far enough in the future before writing to the file. Unfortunately, this workaround isn’t foolproof, and the lease can’t guarantee mutual exclusion by itself.

Solution

- To solve this problem, we can assign a version number to each file that is incremented every time the file is updated.
- The process holding the lease can then read the file and its version number from the file store, do some local computation, and finally update the file (and increment the version number) conditional on the version number not having changed.
- The process can perform this validation atomically using a compare-and-swap operation, which many file stores support.

Design considerations

- Although having a leader can simplify the design of a system as it eliminates concurrency, it can also become a scalability bottleneck if the number of operations performed by it increases to the point where it can no longer keep up.
- Also, a leader is a single point of failure with a large blast radius; if the election process stops working or the leader isn't working as expected, it can bring down the entire system with it.
- We can mitigate some of these downsides by introducing partitions and assigning a different leader per partition, but that comes with additional complexity.
- This is the solution many distributed data stores use since they need to use partitioning anyway to store data that doesn't fit in a single node.
- As a rule of thumb, if we must have a leader, we have to minimize the work it performs and be prepared to occasionally have more than one.
- Requirement: Data store that holds leases must be fault-tolerant

Election algorithms

Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.

Note

In many systems, the coordinator is chosen manually (e.g., file servers). This leads to centralized solutions ⇒ single point of failure.

Election algorithms

Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.

Note

In many systems, the coordinator is chosen manually (e.g., file servers). This leads to centralized solutions ⇒ single point of failure.

Teasers

1. If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
2. Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

Basic assumptions

- All processes have unique id’s
- All processes know id’s of all processes in the system (but not if they are up or down)
- Election means identifying the process with the highest id that is up

Election by bullying

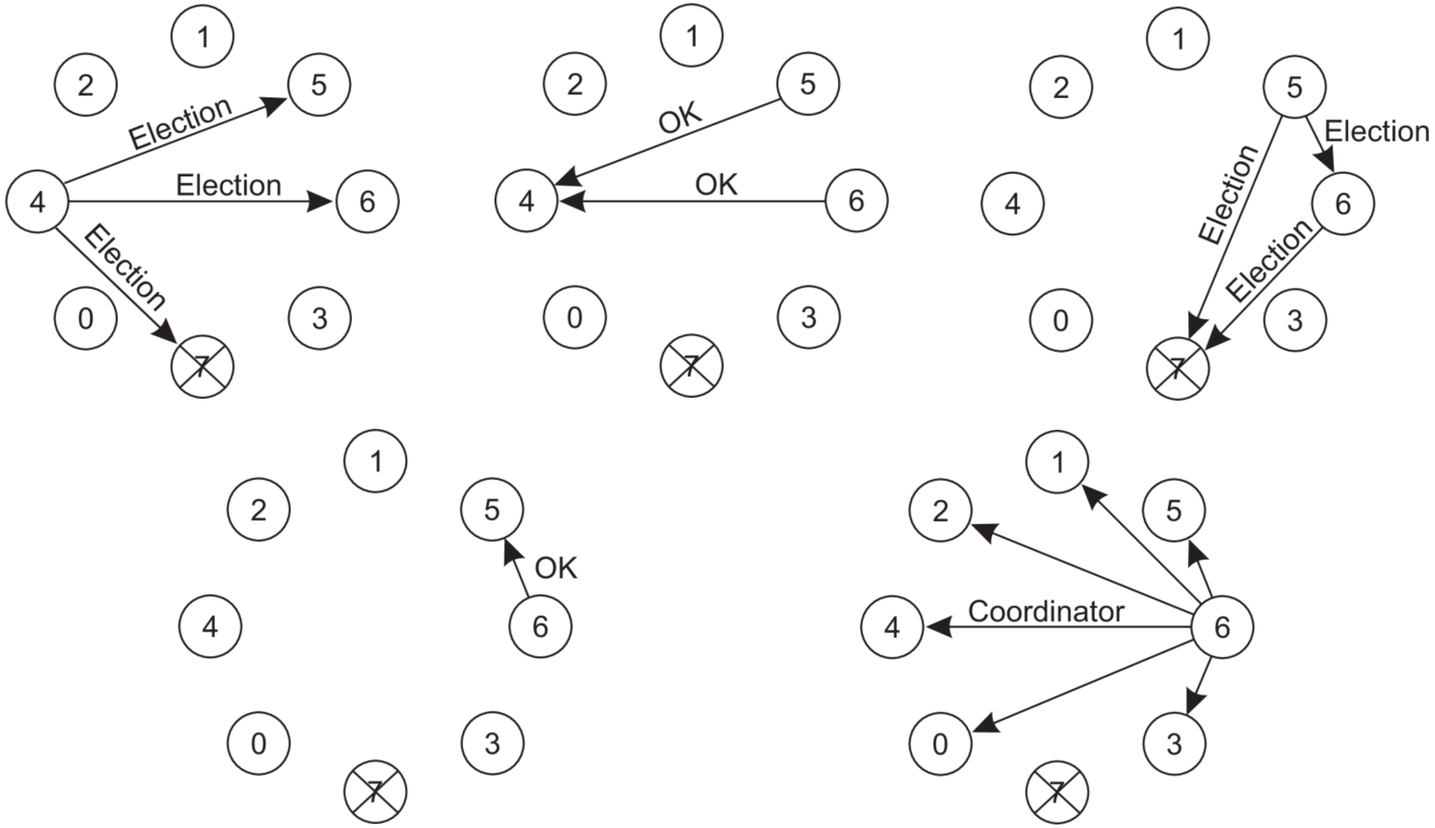
Principle

Consider N processes $\{P_0, \ldots, P_{N-1}\}$ and let $id(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:

1. P_k sends an *ELECTION* message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
2. If no one responds, P_k wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and P_k's job is done.

Election by bullying

The bully election algorithm



Election in a ring

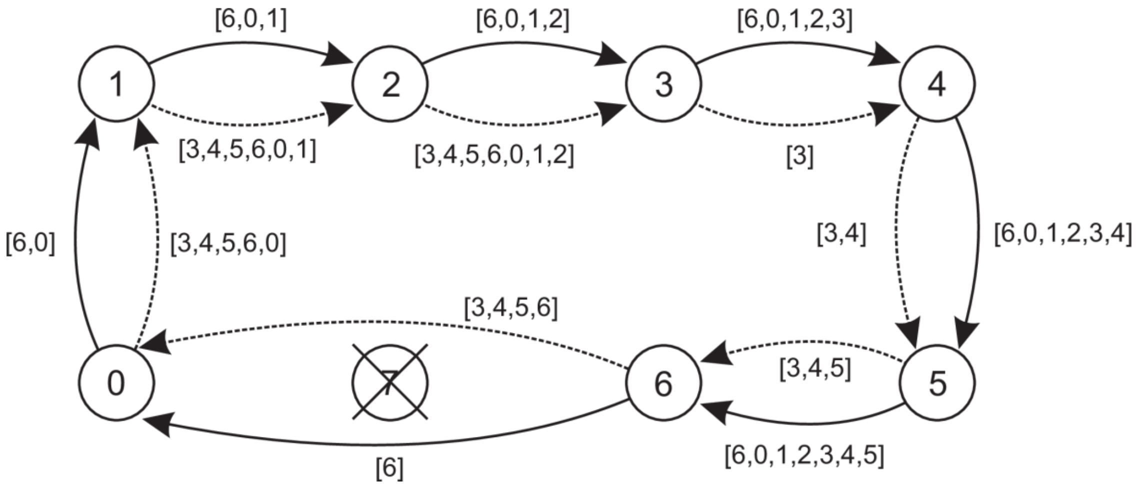
Principle

Process priority is obtained by organizing processes into a (logical) ring. The process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

Election in a ring

Election algorithm using a ring



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

Example: Leader election in ZooKeeper server group

Basics

- Each server *s* in the server group has an identifier *id*(*s*)
- Each server has a monotonically increasing counter *tx*(*s*) of the latest transaction it handled (i.e., series of operations on the namespace).
- When follower *s* suspects leader crashed, it broadcasts an *ELECTION* message, along with the pair (*voteID*,*voteTX*). Initially,
    - *voteID* ← *id*(*s*)
    - *voteTX* ← *tx*(*s*)
- Each server *s* maintains two variables:
    - *leader*(*s*): records the server that *s* believes may be final leader. Initially, *leader*(*s*) ← *id*(*s*).
    - *lastTX*(*s*): what *s* knows to be the most recent transaction. Initially, *lastTX*(*s*) ← *tx*(*s*).

Example: Leader election in ZooKeeper server group

When s^* receives ($voteID$,$voteTX$)

- If $lastTX(s^*) < voteTX$, then s^* just received more up-to-date information on the most recent transaction, and sets
  - $leader(s^*) \leftarrow voteID$
  - $lastTX(s^*) \leftarrow voteTX$
- If $lastTX(s^*) = voteTX$ *and* $leader(s^*) < voteID$, then s^* knows as much about the most recent transaction as what it was just sent, but its perspective on which server will be the next leader needs to be updated:
  - $leader(s^*) \leftarrow voteID$

Note

When s^* believes it should be the leader, it broadcasts $\langle id(s^*), tx(s^*) \rangle$.

Essentially, we're bullying.

Example: Leader election in Raft

Basics

- We have a (relatively small) group of servers
- A server is in one of three states: *follower*, *candidate*, or *leader*
- The protocol works in terms, starting with term 0
- Each server starts in the *follower* state.
- A leader is to regularly broadcast messages (perhaps just a simple heartbeat)

Example: Leader election in Raft

Selecting a new leader

When follower s^* hasn't received anything from the alleged leader s for some time, s^* broadcasts that it volunteers to be the next leader, increasing the term by 1. s^* enters the candidate state. Then:

- If leader s receives the message, it responds by acknowledging that it is still the leader. s^* returns to the follower state.
- If another follower s^{**} gets the election message from s^*, and it is the first election message during the current term, s^{**} votes for s^*. Otherwise, it simply ignores the election message from s^*. When s^* has collected a majority of votes, a new term starts with a new leader.

Example: Leader election in Raft

Selecting a new leader

When follower s^* hasn't received anything from the alleged leader s for some time, s^* broadcasts that it volunteers to be the next leader, increasing the term by 1. s^* enters the candidate state. Then:

- If leader s receives the message, it responds by acknowledging that it is still the leader. s^* returns to the follower state.
- If another follower s^{**} gets the election message from s^*, and it is the first election message during the current term, s^{**} votes for s^*. Otherwise, it simply ignores the election message from s^*. When s^* has collected a majority of votes, a new term starts with a new leader.

Observation

By slightly differing the timeout values per follower for deciding when to start an election, we can avoid concurrent elections, and the election will rapidly converge.

Elections by proof of work

Basics

- Consider a potentially large group of processes
- Each process is required to solve a computational puzzle
- When a process solves the puzzle, it broadcasts its victory to the group
- We assume there is a conflict resolution procedure when more than one process claims victory

Solving a computational puzzle

- Make use of a secure hashing function $H(m)$:
  - m is some data; $H(m)$ returns a fixed-length bit string
  - computing $h = H(m)$ is computationally efficient
  - finding a function H^{-1} such that $m = H^{-1}(H(m))$ is computationally extremely difficult
- Practice: finding H^{-1} boils down to an extensive trial-and-error procedure

Elections by proof of work

Controlled race

- Assume a globally known secure hash function H^*. Let H_i be the hash function used by process P_i.
- Task: given a bit string $h = H_i(m)$, find a bit string $\tilde{h}$ such that $h^* = H^*(H_i(\tilde{h} \odot h))$ where:
  - h^* is a bit string with K leading zeroes
  - $\tilde{h} \odot h$ denotes some predetermined bitwise operation on $\tilde{h}$ and h

Elections by proof of work

Controlled race

- Assume a globally known secure hash function H^*. Let H_i be the hash function used by process P_i.
- Task: given a bit string $h = H_i(m)$, find a bit string $\tilde{h}$ such that $h^* = H^*(H_i(\tilde{h} \odot h))$ where:
  - h^* is a bit string with K leading zeroes
  - $\tilde{h} \odot h$ denotes some predetermined bitwise operation on $\tilde{h}$ and h

Observation

By controlling K, we control the difficulty of finding $\tilde{h}$. If p is the probability that a random guess for $\tilde{h}$ will suffice: $p = (1/2)^K$.

Elections by proof of work

Controlled race

- Assume a globally known secure hash function H^*. Let H_i be the hash function used by process P_i.
- Task: given a bit string $h = H_i(m)$, find a bit string $\tilde{h}$ such that $h^* = H^*(H_i(\tilde{h} \odot h))$ where:
  - h^* is a bit string with K leading zeroes
  - $\tilde{h} \odot h$ denotes some predetermined bitwise operation on $\tilde{h}$ and h

Observation

By controlling K, we control the difficulty of finding $\tilde{h}$. If p is the probability that a random guess for $\tilde{h}$ will suffice: $p = (1/2)^K$.

Current practice

In many PoW-based blockchain systems, $K = 64$

- With $K = 64$, it takes about 10 minutes on a supercomputer to find $\tilde{h}$
- With $K = 64$, it takes about 100 years on a laptop to find $\tilde{h}$

Elections by proof of stake

Basics

We assume a blockchain system in which N secure tokens are used:

- Each token has a unique owner
- Each token has a uniquely associated index $1 \leq k \leq N$
- A token cannot be modified or copied without this going unnoticed

Principle

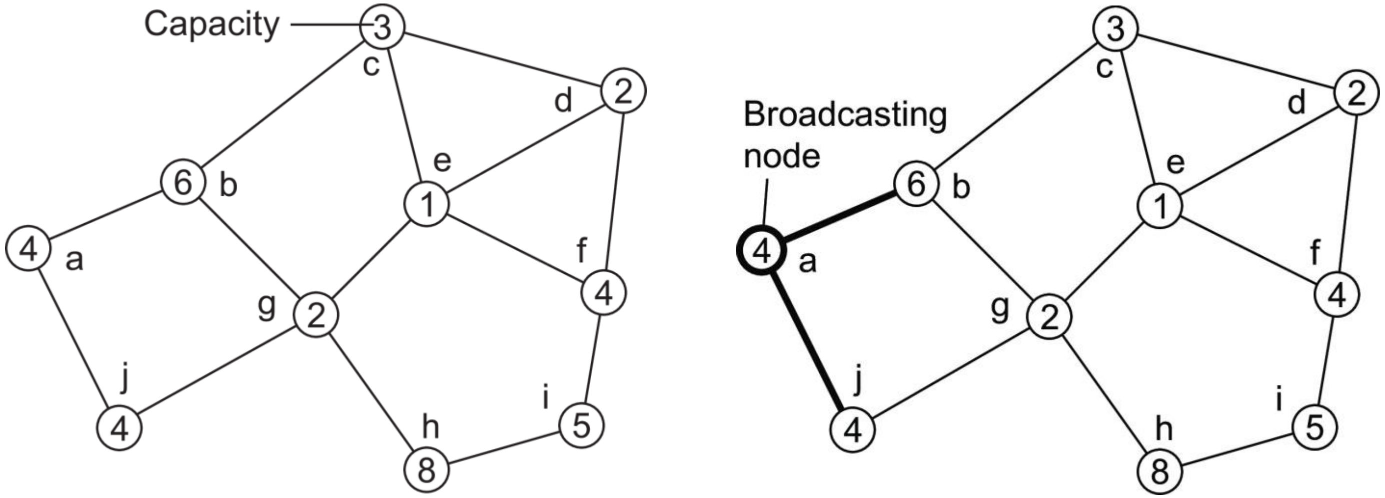
- Draw a random number $k \in \{1, \dots, N\}$
- Look up the process P that owns the token with index k. P is the next leader.

Observation

The more tokens a process owns, the higher the probability it will be selected as leader.

A solution for wireless networks

A sample network

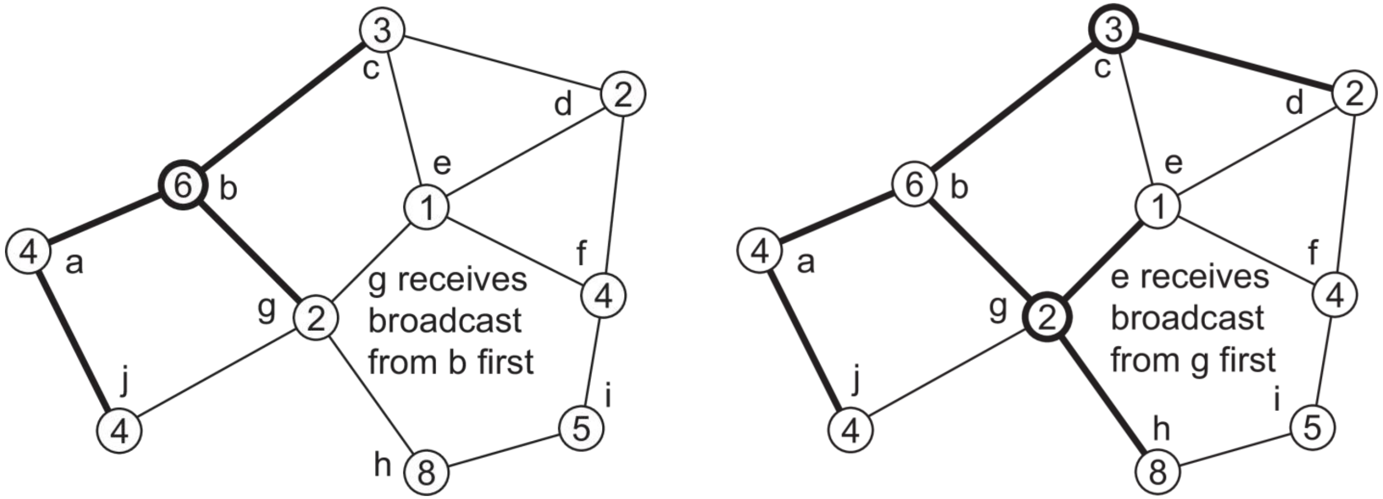


Essence

Find the node with the highest capacity to select as the next leader.

A solution for wireless networks

A sample network

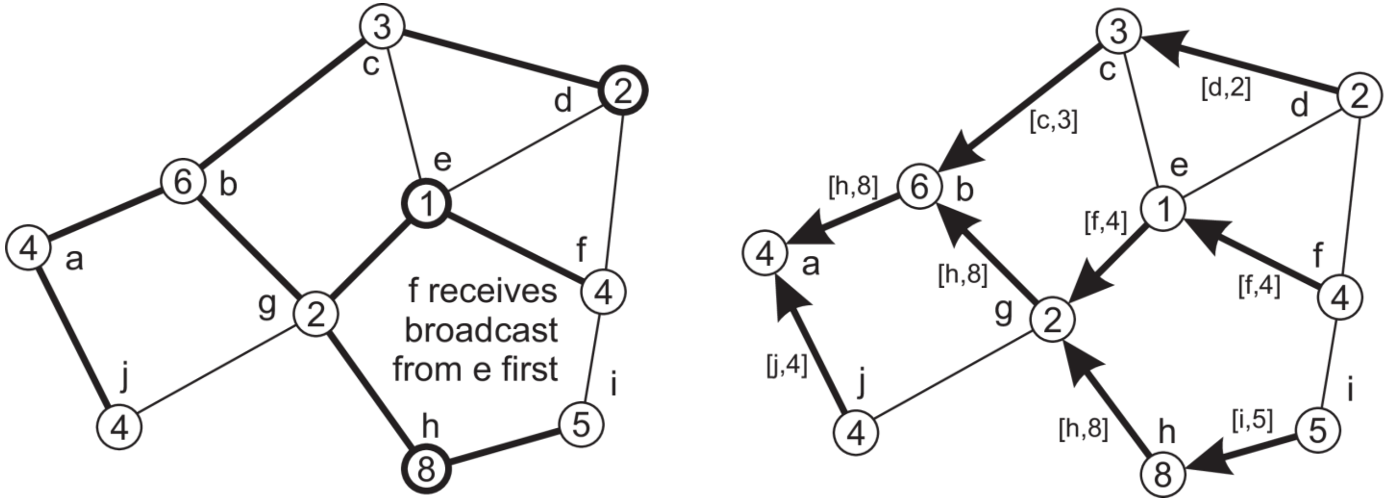


g receives broadcast from b first

e receives broadcast from g first

A solution for wireless networks

A sample network



Essence

A node reports back only the node that it found to have the highest capacity.