Synchronization

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Based on slides by Maarten Van Steen

What is the problem?

Two Generals' Problem (1/3)

- Two generals need to coordinate an attack.
  - Must agree on time to attack.
  - They will win only if they attack simultaneously.
  - Communicate through messengers.
  - Messengers may be killed on their way.

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.
- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.
- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.
  - Problem: how to ensure g1 received ack?

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.
  - Problem: how to ensure g1 received ack?
  - Solution: let g1 ack the receipt of the ack.

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.
- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.
  - Problem: how to ensure g1 received ack?
  - Solution: let g1 ack the receipt of the ack.
  - ...

Two Generals' Problem (2/3)

- Lets try to solve it for general g1 and g2.

- g1 sends time of attack to g2.
  - Problem: how to ensure g2 received message?
  - Solution: let g2 ack receipt of message.
  - Problem: how to ensure g1 received ack?
  - Solution: let g1 ack the receipt of the ack.
  - ...

- This problem is impossible to solve!

Two Generals' Problem (3/3)

- Applicability to distributed systems:
  - Two nodes need to agree on a value.
  - Communicate by messages using an unreliable channel.
- Agreement is a core problem.

Clock Synchronization

Clock Synchronization

- Physical clocks
- Logical clocks

Physical Clock

Physical Clock (1/3)

- Sometimes we simply need the exact time, not just an ordering.

Physical Clock (1/3)

- Sometimes we simply need the exact time, not just an ordering.
- A solution: Universal Coordinated Time (UTC)

Physical Clock (1/3)

- Sometimes we simply need the exact time, not just an ordering.
- A solution: Universal Coordinated Time (UTC)
  - Based on the number of transitions per second of the cesium 133 atom.
  - At present, the real time is taken as the average of some 50 cesium-clocks around the world.

Physical Clock (1/3)

- Sometimes we simply need the exact time, not just an ordering.
- A solution: Universal Coordinated Time (UTC)
  - Based on the number of transitions per second of the cesium 133 atom.
  - At present, the real time is taken as the average of some 50 cesium-clocks around the world.
- UTC is broadcast through short wave radio and satellite.

Physical Clock (2/3)

- Suppose we have a distributed system with a UTC-receiver somewhere in it $\Rightarrow$ we still have to distribute its time to each machine.

Physical Clock (2/3)

- Suppose we have a distributed system with a UTC-receiver somewhere in it ⇒ we still have to distribute its time to each machine.

- Basic principle
    - Every machine has a timer that generates an interrupt H times per second.

Physical Clock (2/3)

- Suppose we have a distributed system with a UTC-receiver somewhere in it $\Rightarrow$ we still have to distribute its time to each machine.

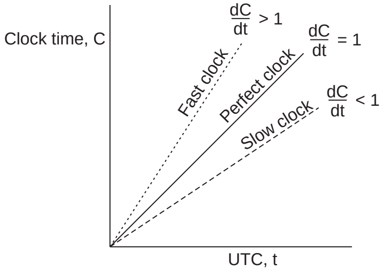
- Basic principle
  - Every machine has a timer that generates an interrupt H times per second.
  - There is a clock in machine p that ticks on each timer interrupt. Denote the value of that clock by $C_p(t)$, where t is UTC time.

Physical Clock (2/3)

- Suppose we have a distributed system with a UTC-receiver somewhere in it ⇒ we still have to distribute its time to each machine.
- Basic principle
  - Every machine has a timer that generates an interrupt H times per second.
  - There is a clock in machine p that ticks on each timer interrupt. Denote the value of that clock by $C_p(t)$, where t is UTC time.
  - Ideally, we have that for each machine p, $C_p(t) = t$, or, in other words, $\frac{dC}{dt} = 1$.

Physical Clock (3/3)

- In practice: $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$.
- Never let two clocks in any system differ by more than δ time units $\Rightarrow$ synchronize at least every $\delta/(2\rho)$ seconds.



Clock Synchronization

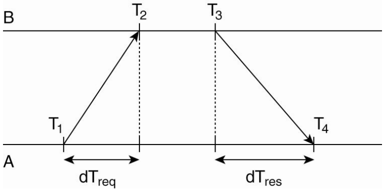
- Network Time Protocol (NTP)
- The Berkeley algorithm

Clock Synchronization - NTP

- Network Time Protocol (NTP)

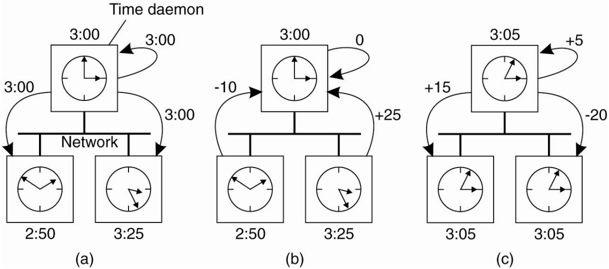
- Every machine asks a time server for the accurate time at least once every $\delta/(2\rho)$ seconds.

- You need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.



Clock Synchronization - The Berkeley Algorithm

- The time daemon asks all the other machines for their clock values.
- The machines answer.
- The time daemon tells everyone how to adjust their clock.



Logical Clock

The Happened-Before Relationship

- The happened-before relation:

The Happened-Before Relationship

- The happened-before relation:
  - If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$.

The Happened-Before Relationship

- The happened-before relation:
  - If a and b are two events in the same process, and a comes before b, then $a \to b$.
  - If a is the sending of a message, and b is the receipt of that message, then $a \to b$.

The Happened-Before Relationship

- The happened-before relation:
  - If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$.
  - If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$.
  - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

The Happened-Before Relationship

- The happened-before relation:
  - If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$.
  - If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$.
  - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.
- If two events, a and b, happen in different processes that do not exchange messages, then $a \rightarrow b$ is not true, but neither is $b \rightarrow a$. These events are said to be concurrent.

Lamport Logical Clocks (1/4)

- How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Lamport Logical Clocks (1/4)

- How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?
- Solution: attach a time-stamp $C(e)$ to each event e, satisfying the following properties:

Lamport Logical Clocks (1/4)

- How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

- Solution: attach a time-stamp $C(e)$ to each event e, satisfying the following properties:
  - P1: if a and b are two events in the same process, and $a \to b$, then we demand that $C(a) < C(b)$.

Lamport Logical Clocks (1/4)

- How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

- Solution: attach a time-stamp $C(e)$ to each event e, satisfying the following properties:
  - P1: if a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.
  - P2: if a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

Lamport Logical Clocks (2/4)

- How to attach a time-stamp to an event when there's no global clock.

Lamport Logical Clocks (2/4)

- How to attach a time-stamp to an event when there's no global clock.
- Solution: each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:

Lamport Logical Clocks (2/4)

- How to attach a time-stamp to an event when there's no global clock.
- Solution: each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:
  1. For any two successive events that take place within P_i, C_i is incremented by 1.

Lamport Logical Clocks (2/4)

- How to attach a time-stamp to an event when there's no global clock.

- Solution: each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:
  1. For any two successive events that take place within P_i, C_i is incremented by 1.
  2. Each time a message m is sent by process P_i, the message receives a time-stamp $ts(m) = C_i$.

Lamport Logical Clocks (2/4)

- How to attach a time-stamp to an event when there's no global clock.
- Solution: each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:
  1. For any two successive events that take place within P_i, C_i is incremented by 1.
  2. Each time a message m is sent by process P_i, the message receives a time-stamp $ts(m) = C_i$.
  3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

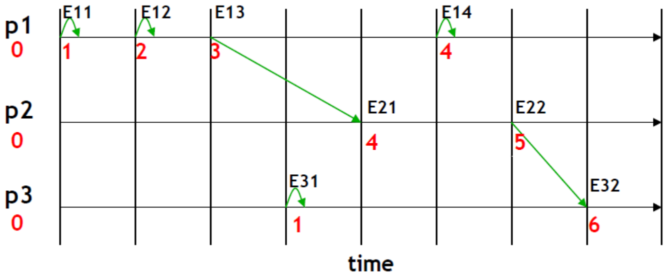
Lamport Logical Clocks (2/4)

- How to attach a time-stamp to an event when there's no global clock.

- Solution: each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:
  1. For any two successive events that take place within P_i, C_i is incremented by 1.
  2. Each time a message m is sent by process P_i, the message receives a time-stamp $ts(m) = C_i$.
  3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

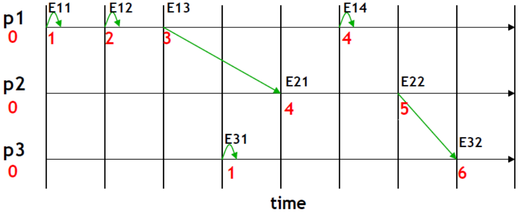
- Property P1 is satisfied by (1); Property P2 by (2) and (3).

Lamport Logical Clocks (3/4)



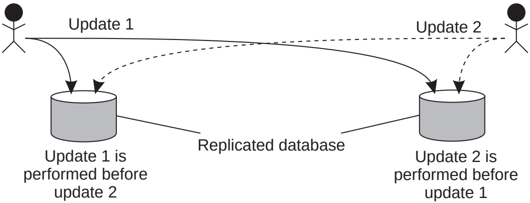
Lamport Logical Clocks (4/4)

- $a \to b$ implies $C(a) < C(b)$.
- $C(a) < C(b)$ does not necessarily imply $a \to b$.
- $C(E31) < C(E13)$, but $E31 \not\to E13$.



Example: Totally Ordered Multicast (1/2)

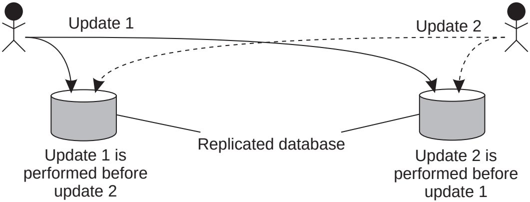
- We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere:
  - P_1 adds \$100 to an account (initial value: \$1000)
  - P_2 increments account by 1%
  - There are two replicas



Example: Totally Ordered Multicast (1/2)

- We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere:
  - P_1 adds \$100 to an account (initial value: \$1000)
  - P_2 increments account by 1%
  - There are two replicas

- In absence of proper synchronization:
  replica #1 $\leftarrow$ \$1111, while replica #2 $\leftarrow$ \$1110.



Example: Totally Ordered Multicast (2/2)

- Solution:
  - Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
  - Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

Example: Totally Ordered Multicast (2/2)

- Solution:
  - Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
  - Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

- P_j passes a message msg_i to its application if:
  - msg_i is at the head of $queue_j$.
  - for each process P_k, there is a message msg_k in $queue_j$ with a larger timestamp.

Example: Totally Ordered Multicast (2/2)

- Solution:
  - Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
  - Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

- P_j passes a message msg_i to its application if:
  - msg_i is at the head of $queue_j$.
  - for each process P_k, there is a message msg_k in $queue_j$ with a larger timestamp.

- We are assuming that communication is reliable and FIFO ordered.

Shortcoming of Lamport Clocks

- Main shortcoming of Lamport's clocks:
  - $C(a) < C(b)$ does not necessarily imply $a \rightarrow b$
  - We cannot deduce causal dependencies from time stamps.

- Why?
  - Clocks advance independently or via messages.
  - There is no history as to where advances come from.

Vector Clocks (1/2)

- Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$ denotes the number of events that process P_i knows have taken place at P_j.

Vector Clocks (1/2)

- Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$ denotes the number of events that process P_i knows have taken place at P_j.

- When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i along with m as vector time-stamp $vt(m)$. Result: upon arrival, recipient knows P_i's time-stamp.

Vector Clocks (1/2)

- Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$ denotes the number of events that process P_i knows have taken place at P_j.

- When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i along with m as vector time-stamp $vt(m)$. Result: upon arrival, recipient knows P_i's time-stamp.

- When a process P_j delivers a message m that it received from P_i with vector time-stamp $ts(m)$, it

  (1) updates each $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$
  (2) increments $VC_j[j]$ by 1.

Vector Clocks (1/2)

- Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$ denotes the number of events that process P_i knows have taken place at P_j.

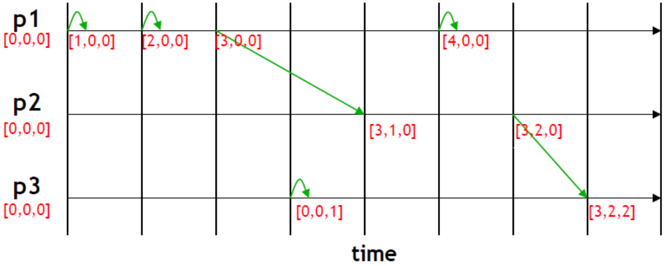
- When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i along with m as vector time-stamp $vt(m)$. Result: upon arrival, recipient knows P_i's time-stamp.

- When a process P_j delivers a message m that it received from P_i with vector time-stamp $ts(m)$, it

  (1) updates each $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$
  (2) increments $VC_j[j]$ by 1.

- $VC_i[j] = k$ tells us that P_i knows that P_j has sent k messages.

Vector Clocks (2/2)



Example: Causally Ordered Multicasting (1/2)

- To ensure that a message is delivered only if all causally preceding messages have already been delivered.

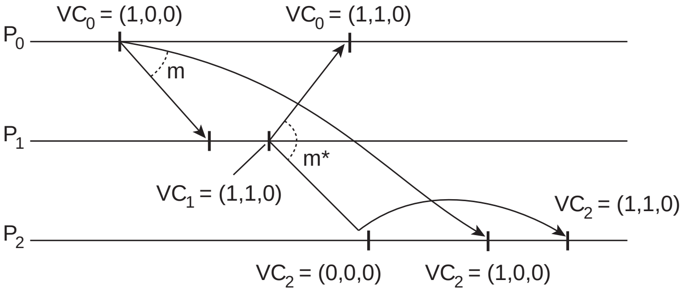
Example: Causally Ordered Multicasting (1/2)

- To ensure that a message is delivered only if all causally preceding messages have already been delivered.
- P_i increments $VC_i[i]$ only when sending a message, and P_j adjusts VC_j when receiving a message.

Example: Causally Ordered Multicasting (1/2)

- To ensure that a message is delivered only if all causally preceding messages have already been delivered.
- P_i increments $VC_i[i]$ only when sending a message, and P_j adjusts VC_j when receiving a message.
- P_j postpones delivery of m until:
  - $ts(m)[i] = VC_j[i] + 1$.
  - $ts(m)[k] \leq VC_j[k]$ for $k \neq i$.

Example: Causally Ordered Multicasting (2/2)



Mutual Exclusion

Mutual Exclusion

- A number of processes in a distributed system want exclusive access to some resource.

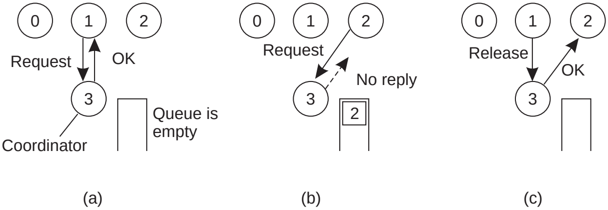
Mutual Exclusion

- A number of processes in a distributed system want exclusive access to some resource.
- Basic solutions:
  - Via a centralized server.
  - Decentralized algorithm.
  - Distributed algorithm, with no topology imposed.
  - Logical ring algorithm.

Mutual Exclusion
Centralized Model

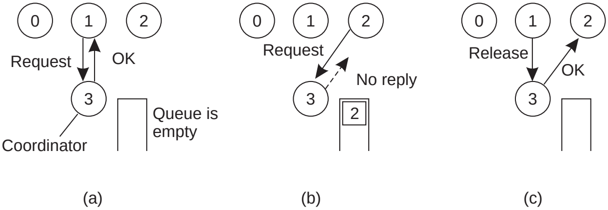
Centralized Algorithm

- Process 1 asks the coordinator for permission to access the shared resource. Permission is granted.



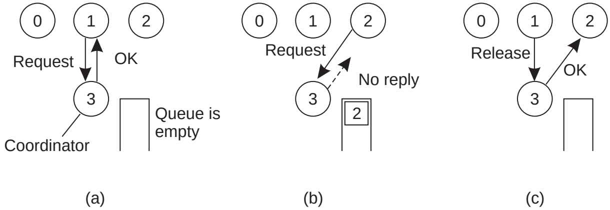
Centralized Algorithm

- Process 1 asks the coordinator for permission to access the shared resource. Permission is granted.
- Process 2 then asks permission to access the same resource. The coordinator does not reply.



Centralized Algorithm

- Process 1 asks the coordinator for permission to access the shared resource. Permission is granted.
- Process 2 then asks permission to access the same resource. The coordinator does not reply.
- When process 1 releases the resource, it tells the coordinator, which then replies to 2.



Mutual Exclusion
Decentralized Algorithm

Decentralized Algorithm (1/2)

- Assume every resource is replicated n times, with each replica having its own coordinator $\Rightarrow$ access requires a majority vote from $m > n/2$ coordinators.
- A coordinator always responds immediately to a request.
- When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

Decentralized Algorithm (2/2)

- How robust is this system?
- Let $p = \Delta t / T$ denote the probability that a coordinator crashes and recovers in a period Δt while having an average lifetime T.
- Probability that k out m coordinators reset:

$$P[\text{violation}] = p_v = \sum_{k=2m-n}^{n} \binom{m}{k} p^k (1-p)^{m-k}$$

With $p = 0.001$, $n = 32$, $m = 0.75n$, $p_v < 10^{-40}$

Mutual Exclusion
Distributed Algorithm

Distributed Algorithm (1/3)

- Ricart and Agrawala's algorithm
- Requires a total ordering of all events: Lamport's algorithm.

Distributed Algorithm (1/3)

- Ricart and Agrawala's algorithm
- Requires a total ordering of all events: Lamport's algorithm.
- 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.

Distributed Algorithm (2/3)

- When a process receives a request message from another process:

Distributed Algorithm (2/3)

- When a process receives a request message from another process:
  - If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.

Distributed Algorithm (2/3)

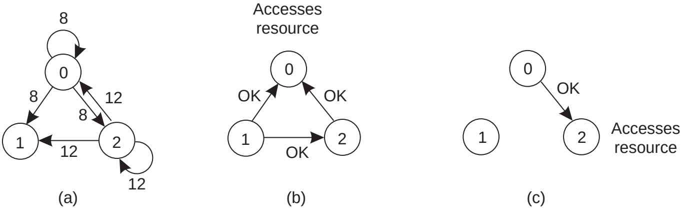
- When a process receives a request message from another process:
  - If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
  - If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.

Distributed Algorithm (2/3)

- When a process receives a request message from another process:
  - If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
  - If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
  - 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.

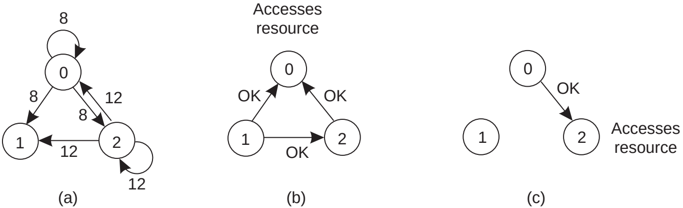
Distributed Algorithm (3/3)

- Two processes want to access a shared resource at the same moment.



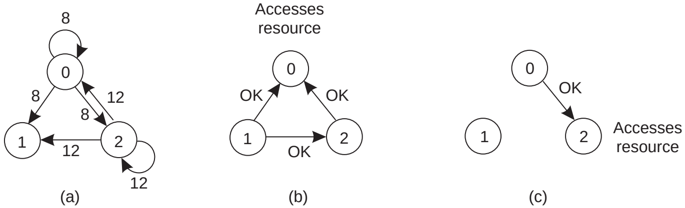
Distributed Algorithm (3/3)

- Two processes want to access a shared resource at the same moment.
- Process 0 has the lowest timestamp, so it wins.



Distributed Algorithm (3/3)

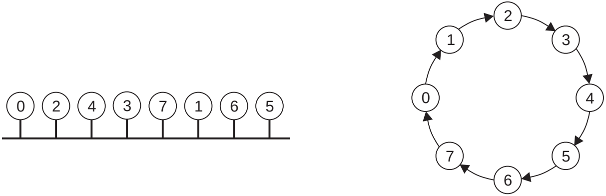
- Two processes want to access a shared resource at the same moment.
- Process 0 has the lowest timestamp, so it wins.
- When process 0 is done, it sends an OK also, so 2 can now go ahead.



Mutual Exclusion
Token Ring Algorithm

Token Ring Algorithm

- Organize processes in a logical ring.
- Let a token be passed between them.
- The one that holds the token is allowed to enter the critical region (if it wants to).



(a) (b)

Election Algorithms

Election Algorithms

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

Election by Bullying (1/3)

- Each process has an associated priority (weight), and the process with the highest priority should always be elected as the coordinator.

Election by Bullying (1/3)

- Each process has an associated priority (weight), and the process with the highest priority should always be elected as the coordinator.
- How do we find the heaviest process?

Election by Bullying (2/3)

- Any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).

Election by Bullying (2/3)

- Any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).

- If a process P_{heavy} receives an election message from a lighter process P_{light}, it sends a take-over message to P_{light}. P_{light} is out of the race.

Election by Bullying (2/3)

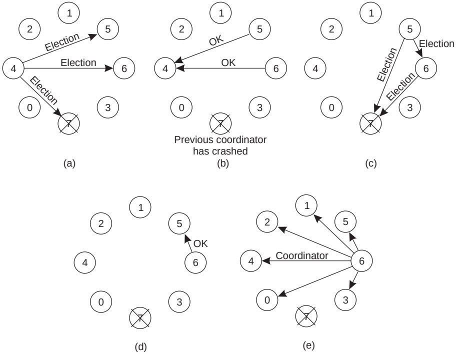
- Any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).

- If a process P_{heavy} receives an election message from a lighter process P_{light}, it sends a take-over message to P_{light}. P_{light} is out of the race.

- If a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

Election by Bullying (3/3)

- Process 4 holds an election.
- Processes 5 and 6 respond, telling 4 to stop.
- Now 5 and 6 each hold an election.
- Process 6 tells 5 to stop.
- Process 6 wins and tells everyone.



Election in a Ring (1/3)

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

Election in a Ring (2/3)

- 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.

Election in a Ring (2/3)

- 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.

Election in a Ring (2/3)

- 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 (3/3)

- Does it matter if two processes initiate an election?
  - There is no problem with having two concurrent initiators.

Election in a Ring (3/3)

- Does it matter if two processes initiate an election?
  - There is no problem with having two concurrent initiators.
- What happens if a process crashes during the election?
  - Crashes during elections are permitted: you just start over again.

Summary

Summary

- Agreement in a distribute system
- Clocks: physical vs. logical
- Physical clocks: NTP, Berkeley
- Logical clocks: Happened-Before, Lamport, vector clocks
- Mutual exclusion: centralized, decentralized, distributed, ring-based
- Election: bullying, ring

Reading

- Chapter 6 of the Distributed Systems: Principles and Paradigms.

Questions?