Distributed Synchronization

Overview

- Synchronization of distributed processes requires new concepts in addition to synchronization of processes in single multi-core systems.

Overview

- Synchronization of distributed processes requires new concepts in addition to synchronization of processes in single multi-core systems.
- Topics:
  - Notion of *global time*: absolute time versus relative time
  - *Election algorithms*: for electing a *coordinator* on-the-fly
  - *Distributed mutual exclusion*

Clocks on Computing Devices

- A computer timer is a quartz crystal that oscillates at a well defined frequency.

Clocks on Computing Devices

- A computer timer is a quartz crystal that oscillates at a well defined frequency.
- Each oscillation decrements a counter by one. When the counter goes down to zero, an interrupt is generated and the counter is reloaded from a holding register.

Clocks on Computing Devices

- A computer timer is a quartz crystal that oscillates at a well defined frequency.
- Each oscillation decrements a counter by one. When the counter goes down to zero, an interrupt is generated and the counter is reloaded from a holding register.
- Each interrupt is called a clock tick (and can be set to interrupt certain number of times per second).

Clocks on Computing Devices

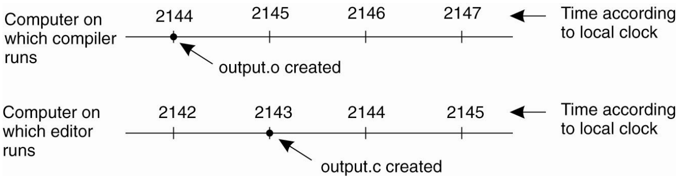
- A computer timer is a quartz crystal that oscillates at a well defined frequency.
- Each oscillation decrements a counter by one. When the counter goes down to zero, an interrupt is generated and the counter is reloaded from a holding register.
- Each interrupt is called a clock tick (and can be set to interrupt certain number of times per second).
- The time is stored on a battery-backed CMOS RAM. At every clock tick, the interrupt service procedure adds one to the stored time.

Clocks on Computing Devices

- A computer timer is a quartz crystal that oscillates at a well defined frequency.
- Each oscillation decrements a counter by one. When the counter goes down to zero, an interrupt is generated and the counter is reloaded from a holding register.
- Each interrupt is called a clock tick (and can be set to interrupt certain number of times per second).
- The time is stored on a battery-backed CMOS RAM. At every clock tick, the interrupt service procedure adds one to the stored time.
- With one computer even if the time is off it is usually not a problem. With n computers, all n crystals will run at slightly different rates, causing the software clocks to gradually get out of sync. This difference in time values is called clock skew.

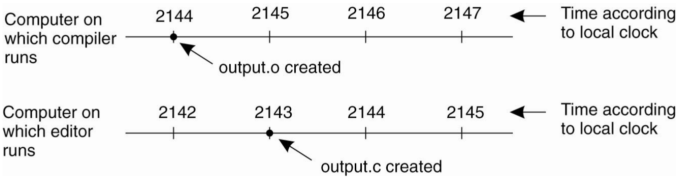
Clock Skew

- Clock skew illustrated on a shared file system:



Clock Skew

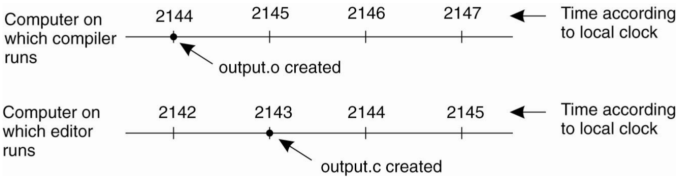
- Clock skew illustrated on a shared file system:



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

Clock Skew

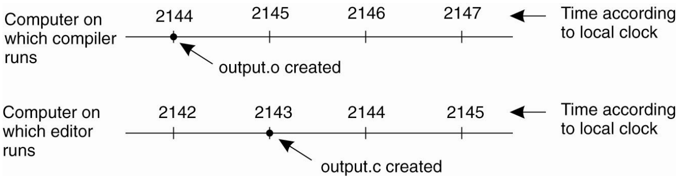
- Clock skew illustrated on a shared file system:



- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
- **In-class Exercise**: Come up with an example where clock skew causes a build system to compile a file unnecessarily.

Clock Skew

- Clock skew illustrated on a shared file system:



- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
- **In-class Exercise**: Come up with an example where clock skew causes a build system to compile a file unnecessarily.
- **In-class Exercise**: Come up with another scenario where a build system misses that a file has changed due to clock skew.

How to Measure Real Time?

- Measure a large number of days and average them and then divide by 86400 to obtain mean solar second. However, this value is changing as Earth's rotation is slowing over time.

How to Measure Real Time?

- Measure a large number of days and average them and then divide by 86400 to obtain mean solar second. However, this value is changing as Earth's rotation is slowing over time.
- Atomic time: one second = time taken for Cesium 133 atom to make 9,192,631,770 transitions. The International Atomic Time (TAI) is the average of about 50 Cesium clocks around the world. TAI is the mean number of ticks of the Cesium 133 atom since midnight, 1st Jan, 1958 reported by the *Bureau International de l'Heure* in Paris.

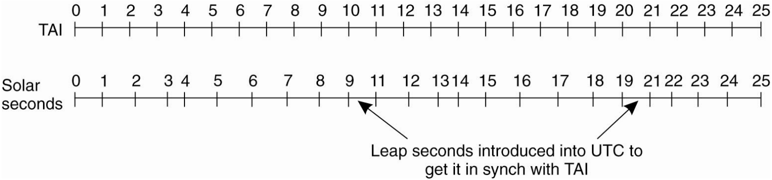
How to Measure Real Time?

- Measure a large number of days and average them and then divide by 86400 to obtain mean solar second. However, this value is changing as Earth's rotation is slowing over time.
- Atomic time: one second = time taken for Cesium 133 atom to make 9,192,631,770 transitions. The International Atomic Time (TAI) is the average of about 50 Cesium clocks around the world. TAI is the mean number of ticks of the Cesium 133 atom since midnight, 1st Jan, 1958 reported by the *Bureau International de l'Heure* in Paris.
- A leap second is introduced whenever the discrepancy between TAI and solar time grows to 800 msec. About 30 leap seconds have been introduced since 1958. This is known as the Universal Coordinated Time (UTC).

How to Measure Real Time?

- Measure a large number of days and average them and then divide by 86400 to obtain mean solar second. However, this value is changing as Earth's rotation is slowing over time.
- Atomic time: one second = time taken for Cesium 133 atom to make 9,192,631,770 transitions. The International Atomic Time (TAI) is the average of about 50 Cesium clocks around the world. TAI is the mean number of ticks of the Cesium 133 atom since midnight, 1st Jan, 1958 reported by the *Bureau International de l'Heure* in Paris.
- A leap second is introduced whenever the discrepancy between TAI and solar time grows to 800 msec. About 30 leap seconds have been introduced since 1958. This is known as the Universal Coordinated Time (UTC).
- UTC is broadcast over short-wave by NIST on station WWV (and from satellites). See `http://www.nist.gov`.

Leap Seconds



- TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

Global Positioning System (GPS)

- GPS is a satellite-based distributed geographical positioning system consisting of 31 satellites at an orbit of around 20,000 km above Earth.

Global Positioning System (GPS)

- GPS is a satellite-based distributed geographical positioning system consisting of 31 satellites at an orbit of around 20,000 km above Earth.
- Each satellite has four atomic clocks, which are regularly calibrated from Earth.

Global Positioning System (GPS)

- GPS is a satellite-based distributed geographical positioning system consisting of 31 satellites at an orbit of around 20,000 km above Earth.
- Each satellite has four atomic clocks, which are regularly calibrated from Earth.
- Each satellite continuously broadcasts its position and timestamps its message with its local time.

Global Positioning System (GPS)

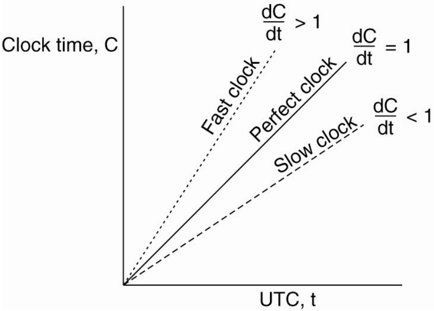
- GPS is a satellite-based distributed geographical positioning system consisting of 31 satellites at an orbit of around 20,000 km above Earth.
- Each satellite has four atomic clocks, which are regularly calibrated from Earth.
- Each satellite continuously broadcasts its position and timestamps its message with its local time.
- A GPS receiver can compute its own position using three satellites, assuming that the receiver has accurate time. Otherwise it requires four satellites.

Global Positioning System (GPS)

- GPS is a satellite-based distributed geographical positioning system consisting of 31 satellites at an orbit of around 20,000 km above Earth.
- Each satellite has four atomic clocks, which are regularly calibrated from Earth.
- Each satellite continuously broadcasts its position and timestamps its message with its local time.
- A GPS receiver can compute its own position using three satellites, assuming that the receiver has accurate time. Otherwise it requires four satellites.
- Typical accuracy is 1-5m but can be as good as less than one foot

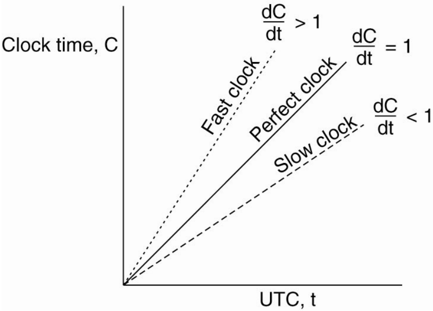
GPS Animation

Clock Synchronization Algorithms



- The relation between clock time and UTC when clocks tick at different rates.

Clock Synchronization Algorithms



- The relation between clock time and UTC when clocks tick at different rates.
- Let maximum drift rate be ρ. Then $1 - \rho \leq dC/dt \leq 1 + \rho$ where dC/dt is the rate of drift of the clock relative to UTC. Ideally, we want dC/dt to be 1.To ensure two clocks never differ more than δ, the clocks must be synchronized at least every $\delta/2\rho$ seconds.

Network Time Protocol (1)

- NTP can achieve worldwide accuracy in the range 1-50 msec. Widely used on the Internet.

Network Time Protocol (1)

- NTP can achieve worldwide accuracy in the range 1-50 msec. Widely used on the Internet.
- Uses combination of various advanced clock synchronization algorithms (RFC1305).

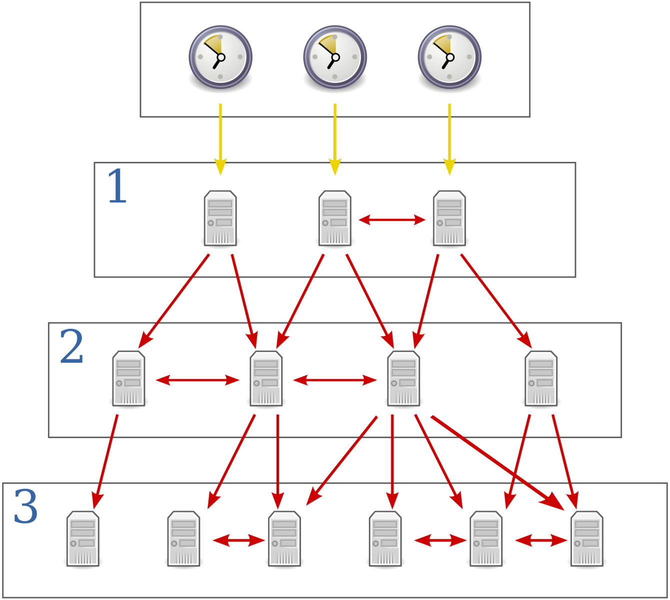
Network Time Protocol (1)

- NTP can achieve worldwide accuracy in the range 1-50 msec. Widely used on the Internet.
- Uses combination of various advanced clock synchronization algorithms (RFC1305).
- Uses a distributed shortest paths algorithm to determine who gets served by whom. Has mechanisms for dealing gracefully with servers being down.

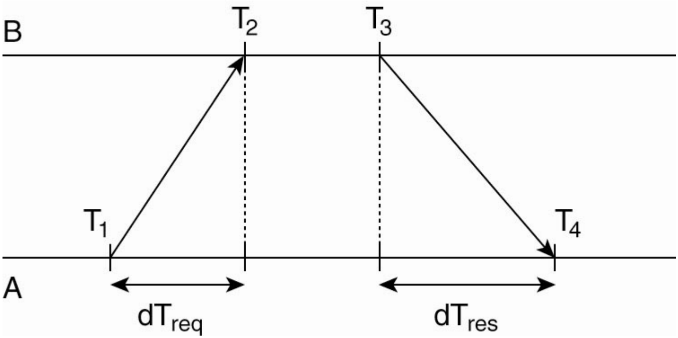
Network Time Protocol (1)

- NTP can achieve worldwide accuracy in the range 1-50 msec. Widely used on the Internet.
- Uses combination of various advanced clock synchronization algorithms (RFC1305).
- Uses a distributed shortest paths algorithm to determine who gets served by whom. Has mechanisms for dealing gracefully with servers being down.
- Clients need to slow down or speed up local clocks to sync up gradually with a server.

Network Time Protocol (2)



Network Time Protocol (3)



- Getting the current time from a time server. Relative offset $\theta = T3 - ((T2 - T1) + (T4 - T3))/2$

Network Time Protocol (4)

- NTP can be setup pair-wise between servers. Both servers ask each other for time and calculate the θ and δ, where

$$\delta = ((T2 - T1) + (T4 - T3))/2$$

Network Time Protocol (4)

- NTP can be setup pair-wise between servers. Both servers ask each other for time and calculate the θ and δ, where

$$\delta = ((T2 - T1) + (T4 - T3))/2$$

- Eight pairs of θ and δ are buffered and the minimal value is taken as the delay between the servers

- NTP can be setup pair-wise between servers. Both servers ask each other for time and calculate the θ and δ, where

$$\delta = ((T2 - T1) + (T4 - T3))/2$$

- Eight pairs of θ and δ are buffered and the minimal value is taken as the delay between the servers
- A server with a reference clock such as a WWV receiver or an atomic clock is a stratum-1 server. When A contacts B it will only adjusts its clock if its stratum number if higher than B. Moreover, after the synchronization, A's stratum level becomes one more than B's level

Synchronized Time in the Lab

- The command pdsh runs a parallel/distributed shell across the nodes.

```
[amit@onyx ~]$ pdsh -w node[01-63] date | sort
node01: Wed Mar  9 06:18:26 MST 2016
node02: Wed Mar  9 06:18:26 MST 2016
node03: Wed Mar  9 06:18:26 MST 2016
node04: Wed Mar  9 06:18:26 MST 2016
node05: Wed Mar  9 06:18:26 MST 2016
node06: Wed Mar  9 06:18:26 MST 2016
node07: Wed Mar  9 06:18:26 MST 2016
node08: Wed Mar  9 06:18:26 MST 2016
node11: Wed Mar  9 06:18:26 MST 2016
node12: Wed Mar  9 06:18:26 MST 2016
.
.
.
node62: Wed Mar  9 06:18:26 MST 2016
node63: Wed Mar  9 06:18:26 MST 2016
[amit@onyx ~]$
```

- Try `pdsh -w node[01-63] date -rfc-3339=ns | sort` to see time in nanoseconds resolution.

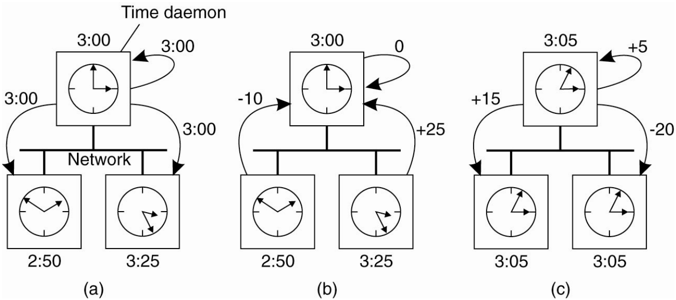
Synchronized Time in the Lab

- The command pdsh runs a parallel/distributed shell across the nodes.

```
[amit@onyx ~]$ pdsh -w node[01-63] date | sort
node01: Wed Mar  9 06:18:26 MST 2016
node02: Wed Mar  9 06:18:26 MST 2016
node03: Wed Mar  9 06:18:26 MST 2016
node04: Wed Mar  9 06:18:26 MST 2016
node05: Wed Mar  9 06:18:26 MST 2016
node06: Wed Mar  9 06:18:26 MST 2016
node07: Wed Mar  9 06:18:26 MST 2016
node08: Wed Mar  9 06:18:26 MST 2016
node11: Wed Mar  9 06:18:26 MST 2016
node12: Wed Mar  9 06:18:26 MST 2016
.
.
.
node62: Wed Mar  9 06:18:26 MST 2016
node63: Wed Mar  9 06:18:26 MST 2016
[amit@onyx ~]$
```

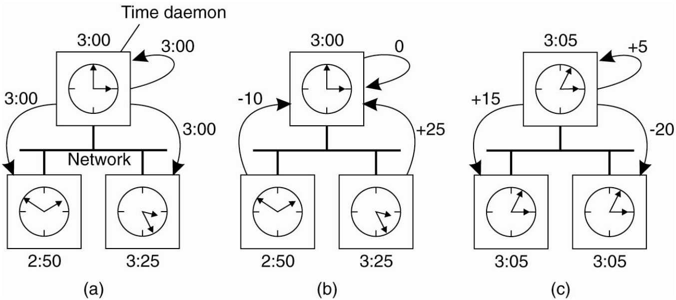
- Try `pdsh -w node[01-63] date -rfc-3339=ns | sort` to see time in nanoseconds resolution.
- The cluster uses NTP (Network Time Protocol) daemons on each node to keep the machines synchronized.

Berkeley Time Algorithm



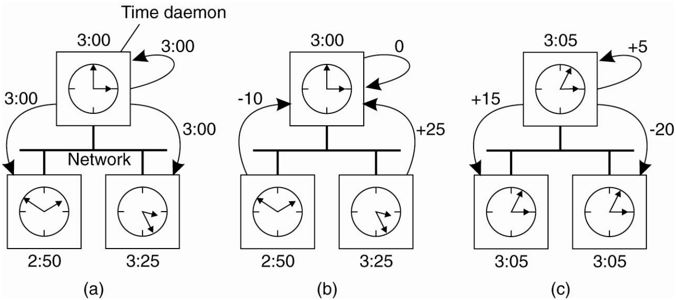
- The time daemon asks all other machines for their clock values.

Berkeley Time Algorithm



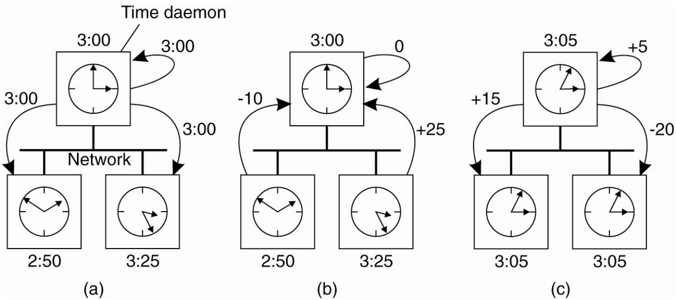
- The time daemon asks all other machines for their clock values.
- The machines answer.

Berkeley Time Algorithm



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

Berkeley Time Algorithm



- The time daemon asks all other machines for their clock values.
- The machines answer.
- The time daemon tells everyone how to adjust their clock.
- **In-class Exercise**. How would you implement the Berkeley Time Algorithm?

Logical Clocks

Logical Clocks

- For many purposes, it is sufficient that machine agree on the same time even though that time may not agree with real world.

Logical Clocks

- For many purposes, it is sufficient that machine agree on the same time even though that time may not agree with real world.
- If two process do not interact, it is not necessary that their clocks be synchronized. Furthermore, if all processes agree on the order in which events occur, then they need not agree on the time.

Logical Clocks

- For many purposes, it is sufficient that machine agree on the same time even though that time may not agree with real world.
- If two process do not interact, it is not necessary that their clocks be synchronized. Furthermore, if all processes agree on the order in which events occur, then they need not agree on the time.
- A logical clock is a mechanism for capturing chronological and causal relationships in a distributed system.

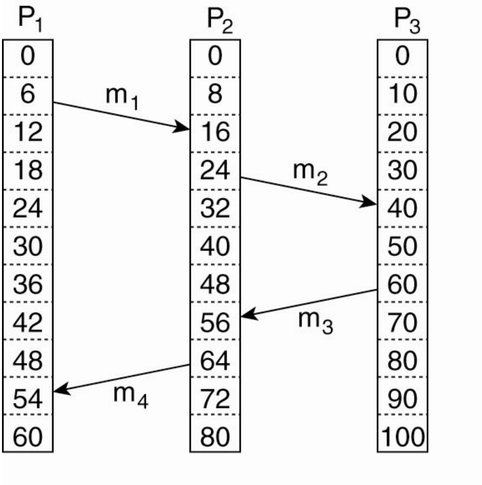
Logical Clocks

- For many purposes, it is sufficient that machine agree on the same time even though that time may not agree with real world.
- If two process do not interact, it is not necessary that their clocks be synchronized. Furthermore, if all processes agree on the order in which events occur, then they need not agree on the time.
- A logical clock is a mechanism for capturing chronological and causal relationships in a distributed system.
  - The first implementation, the Lamport timestamps, was proposed by Leslie Lamport in 1978 (Turing Award in 2013).

Logical Clocks

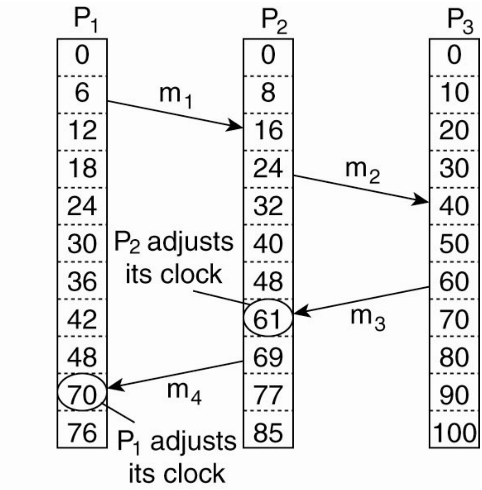
- For many purposes, it is sufficient that machine agree on the same time even though that time may not agree with real world.
- If two process do not interact, it is not necessary that their clocks be synchronized. Furthermore, if all processes agree on the order in which events occur, then they need not agree on the time.
- A logical clock is a mechanism for capturing chronological and causal relationships in a distributed system.
  - The first implementation, the Lamport timestamps, was proposed by Leslie Lamport in 1978 (Turing Award in 2013).
- Some noteworthy logical clock algorithms:
  - Lamport timestamps, which are monotonically increasing software counters.
  - Vector clocks, that allow for partial ordering of events in a distributed system.
  - Version vectors, order replicas, according to updates, in an optimistic replicated system.
  - Matrix clocks, an extension of vector clocks that also contains information about other processes' views of the system.

Lamport's Logical Clocks (1)



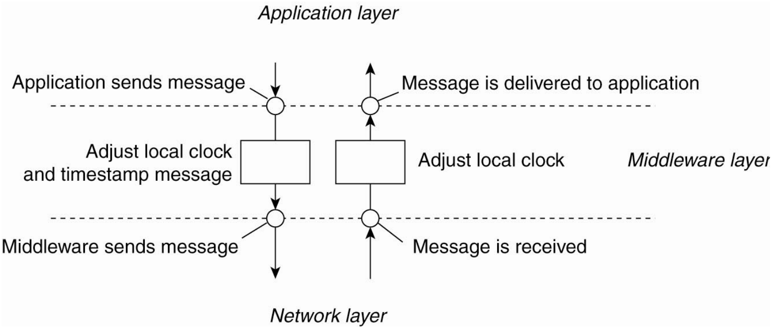
- Three processes, each with its own clock.

Lamport's Logical Clocks (2)



- Lamport's algorithm corrects the clock by adjusting the timestamps.

Lamport's Logical Clocks (3)



- The positioning of Lamport's logical clocks in a distributed system.

Lamport's Logical Clocks (4)

- C is the timestamp function that is defined as follows:

Lamport's Logical Clocks (4)

- C is the timestamp function that is defined as follows:
  1. If a happens before b in the same process, $C(\mathtt{a}) < C(\mathtt{b})$

Lamport's Logical Clocks (4)

- C is the timestamp function that is defined as follows:
  1. If a happens before b in the same process, $C(\mathtt{a}) < C(\mathtt{b})$
  2. If a represents sending and b receiving of a message, then $C(\mathtt{a}) < C(\mathtt{b})$

Lamport's Logical Clocks (4)

- C is the timestamp function that is defined as follows:
  1. If a happens before b in the same process, $C(\mathtt{a}) < C(\mathtt{b})$
  2. If a represents sending and b receiving of a message, then $C(\mathtt{a}) < C(\mathtt{b})$
  3. For all distinct events a and b, $C(\mathtt{a}) \neq C(\mathtt{b})$

Lamport's Logical Clocks (4)

- C is the timestamp function that is defined as follows:
  1. If a happens before b in the same process, $C(\mathtt{a}) < C(\mathtt{b})$
  2. If a represents sending and b receiving of a message, then $C(\mathtt{a}) < C(\mathtt{b})$
  3. For all distinct events a and b, $C(\mathtt{a}) \neq C(\mathtt{b})$
- The time is always adjusted forward. Each message carries the sending time according to the sender's clock. If receiver's time is prior to the sending time, the receiver fast forwards its clock to be 1 more than the sending time.

Lamport's Logical Clocks (4)

- C is the timestamp function that is defined as follows:
  1. If a happens before b in the same process, $C(\texttt{a}) < C(\texttt{b})$
  2. If a represents sending and b receiving of a message, then $C(\texttt{a}) < C(\texttt{b})$
  3. For all distinct events a and b, $C(\texttt{a}) \neq C(\texttt{b})$
- The time is always adjusted forward. Each message carries the sending time according to the sender's clock. If receiver's time is prior to the sending time, the receiver fast forwards its clock to be 1 more than the sending time.
- In addition, between every two events the clock must tick at least once.

Lamport's Logical Clocks (4)

- C is the timestamp function that is defined as follows:
  1. If a happens before b in the same process, $C(\mathtt{a}) < C(\mathtt{b})$
  2. If a represents sending and b receiving of a message, then $C(\mathtt{a}) < C(\mathtt{b})$
  3. For all distinct events a and b, $C(\mathtt{a}) \neq C(\mathtt{b})$
- The time is always adjusted forward. Each message carries the sending time according to the sender's clock. If receiver's time is prior to the sending time, the receiver fast forwards its clock to be 1 more than the sending time.
- In addition, between every two events the clock must tick at least once.
- No two events ever occur at exactly the same time.

Lamport's Logical Clocks (4)

- C is the timestamp function that is defined as follows:
  1. If a happens before b in the same process, $C(\mathtt{a}) < C(\mathtt{b})$
  2. If a represents sending and b receiving of a message, then $C(\mathtt{a}) < C(\mathtt{b})$
  3. For all distinct events a and b, $C(\mathtt{a}) \neq C(\mathtt{b})$
- The time is always adjusted forward. Each message carries the sending time according to the sender's clock. If receiver's time is prior to the sending time, the receiver fast forwards its clock to be 1 more than the sending time.
- In addition, between every two events the clock must tick at least once.
- No two events ever occur at exactly the same time. Tag process ids to low bits of time to make time be unique since process ids are unique.

Lamport's Logical Clocks (5)

To implement Lamport's logical clocks, each process P_i maintains a *local counter* C_i that is updated as follows:

Lamport's Logical Clocks (5)

To implement Lamport's logical clocks, each process P_i maintains a *local counter* C_i that is updated as follows:

Step 1. Before executing an event, P_i executes

$$C_i \leftarrow C_i + 1$$

Lamport's Logical Clocks (5)

To implement Lamport's logical clocks, each process P_i maintains a *local counter* C_i that is updated as follows:

Step 1. Before executing an event, P_i executes

$$C_i \leftarrow C_i + 1$$

Step 2. When process P_i sends a message m to P_j, it sets m's timestamp $ts(m)$ equal to C_i after having executed the previous step.

Lamport's Logical Clocks (5)

To implement Lamport's logical clocks, each process P_i maintains a *local counter* C_i that is updated as follows:

Step 1. Before executing an event, P_i executes

$$C_i \leftarrow C_i + 1$$

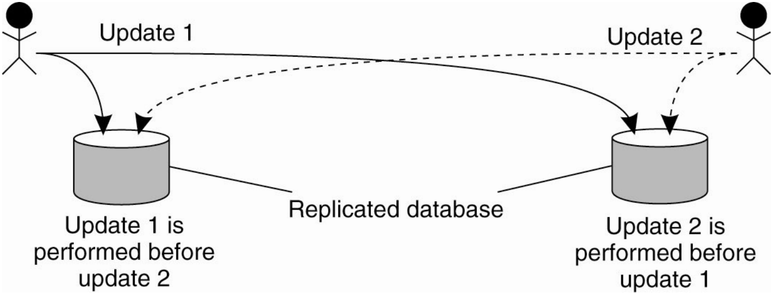
Step 2. When process P_i sends a message m to P_j, it sets m's timestamp $ts(m)$ equal to C_i after having executed the previous step.

Step 3. Upon the receipt of a message m, process P_j adjusts its own local counter as

$$C_j \leftarrow \max\{C_j, ts(m)\}$$

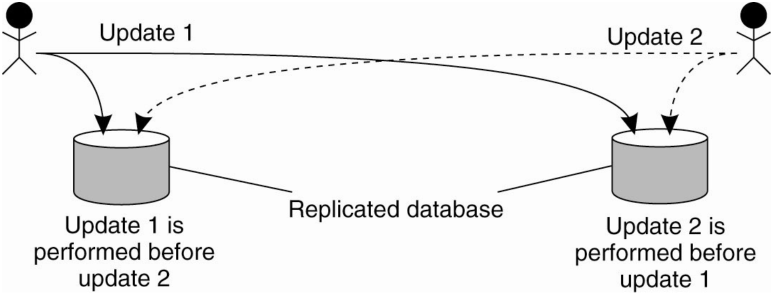
after which it then executes the first step and delivers the message to the application.

Example: Totally Ordered Multicasting (1)



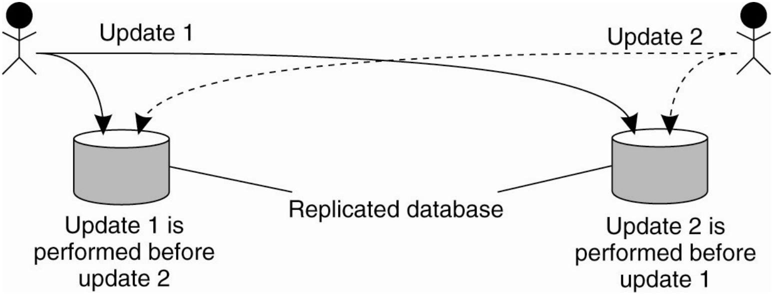
- Updating a database and leaving it in an inconsistent state.

Example: Totally Ordered Multicasting (1)



- Updating a database and leaving it in an inconsistent state. Consider two transactions on an account with a balance of \$1,000.
  - *Transaction A*: Customer wants to deposit \$100 to the account in Boise.
  - *Transaction B*: A bank employee initiates a 10% interest payment to the same account.

Example: Totally Ordered Multicasting (1)



- Updating a database and leaving it in an inconsistent state. Consider two transactions on an account with a balance of \$1,000.
  - *Transaction A*: Customer wants to deposit \$100 to the account in Boise.
  - *Transaction B*: A bank employee initiates a 10% interest payment to the same account.
- Lamport timestamps can be used to fix this problem.

Example: Totally Ordered Multicasting (2)

- Totally Ordered Multicast: A multicast operation by which all messages are delivered in the same order to each receiver.

Example: Totally Ordered Multicasting (2)

- Totally Ordered Multicast: A multicast operation by which all messages are delivered in the same order to each receiver.
  - Can be implemented using Lamport's logical clock algorithm.

Example: Totally Ordered Multicasting (2)

- Totally Ordered Multicast: A multicast operation by which all messages are delivered in the same order to each receiver.
  - Can be implemented using Lamport's logical clock algorithm.
- Each message is time-stamped with the current (logical) time of the sender.

Example: Totally Ordered Multicasting (2)

- Totally Ordered Multicast: A multicast operation by which all messages are delivered in the same order to each receiver.
  - Can be implemented using Lamport's logical clock algorithm.
- Each message is time-stamped with the current (logical) time of the sender.
  - Assumption. Messages from one receiver are ordered and messages aren't lost.

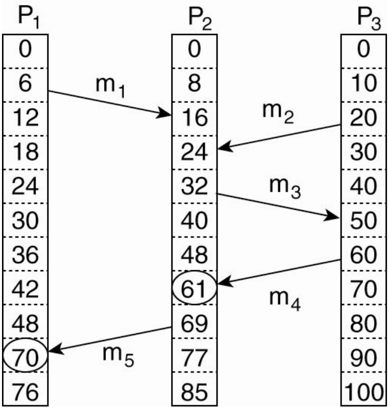
Example: Totally Ordered Multicasting (2)

- Totally Ordered Multicast: A multicast operation by which all messages are delivered in the same order to each receiver.
  - Can be implemented using Lamport's logical clock algorithm.
- Each message is time-stamped with the current (logical) time of the sender.
  - Assumption. Messages from one receiver are ordered and messages aren't lost.
- A process puts received messages into a queue ordered by timestamps. It acknowledges the messages with a multicast to all other processes. Eventually the local queues are the same at all processes.

Example: Totally Ordered Multicasting (2)

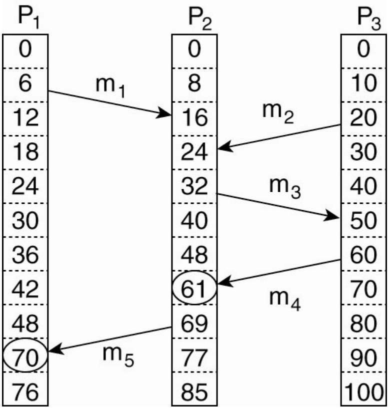
- Totally Ordered Multicast: A multicast operation by which all messages are delivered in the same order to each receiver.
  - Can be implemented using Lamport's logical clock algorithm.
- Each message is time-stamped with the current (logical) time of the sender.
  - Assumption. Messages from one receiver are ordered and messages aren't lost.
- A process puts received messages into a queue ordered by timestamps. It acknowledges the messages with a multicast to all other processes. Eventually the local queues are the same at all processes.
- A process can deliver a queued message to an application only if it is at the head of queue and has been acknowledged by each other process.

Vector Clocks (1)



- Concurrent message transmission using Lamport logical clocks. Knowing that m_1 was received before m_2 doesn't tell us if they are connected.

Vector Clocks (1)



- Concurrent message transmission using Lamport logical clocks. Knowing that m_1 was received before m_2 doesn't tell us if they are connected.
- Lamport clocks do not capture causality. We need *vector clocks* to capture causality.

Vector Clocks (2)

- Vector clock: A vector clock $VC(\texttt{a})$ assigned to an event a has the property that if $VC(\texttt{a}) < VC(\texttt{b})$ for some event b, then event a is known to causally precede event b.

Vector Clocks (2)

- Vector clock: A vector clock $VC(\mathtt{a})$ assigned to an event a has the property that if $VC(\mathtt{a}) < VC(\mathtt{b})$ for some event b, then event a is known to causally precede event b.
- Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

Vector Clocks (2)

- Vector clock: A vector clock $VC(\mathtt{a})$ assigned to an event a has the property that if $VC(\mathtt{a}) < VC(\mathtt{b})$ for some event b, then event a is known to causally precede event b.
- Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

Step 1. $VC_i[i]$ is the number of events that have occurred so far at P_i. In other words, $VC_i[i]$ is the local logical clock at process P_i.

Vector Clocks (2)

- Vector clock: A vector clock $VC(\mathtt{a})$ assigned to an event a has the property that if $VC(\mathtt{a}) < VC(\mathtt{b})$ for some event b, then event a is known to causally precede event b.
- Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

Step 1. $VC_i[i]$ is the number of events that have occurred so far at P_i. In other words, $VC_i[i]$ is the local logical clock at process P_i.

Step 2. If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j

Vector Clocks (3)

Step 2 is carried out by piggybacking vectors along with messages. The details are shown below:

Vector Clocks (3)

Step 2 is carried out by piggybacking vectors along with messages. The details are shown below:

- Before an event (send/receive or internal event), P_i executes

$$VC_i[i] \leftarrow VC_i[i] + 1$$

Vector Clocks (3)

Step 2 is carried out by piggybacking vectors along with messages. The details are shown below:

- Before an event (send/receive or internal event), P_i executes

$$VC_i[i] \leftarrow VC_i[i] + 1$$

- When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(\mathtt{m})$ equal to VC_i after having executed the previous step.

Vector Clocks (3)

Step 2 is carried out by piggybacking vectors along with messages. The details are shown below:

- Before an event (send/receive or internal event), P_i executes

$$VC_i[i] \leftarrow VC_i[i] + 1$$

- When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(\mathtt{m})$ equal to VC_i after having executed the previous step.
- Upon the receipt of a message m, process P_j adjusts its own vector by setting:

$$VC_j[k] \leftarrow \max\{VC_j[k], ts(\mathtt{m})[k]\}, \forall k$$

after which it executes the first step and delivers the message to the application.

Example: Enforcing Causal Communication (1)

- Causally-ordered multicasting: We want to ensure that a message is delivered only if all messages that causally precede it have been delivered. We assume that the message are multicast within the group.

Example: Enforcing Causal Communication (1)

- Causally-ordered multicasting: We want to ensure that a message is delivered only if all messages that causally precede it have been delivered. We assume that the message are multicast within the group.
- Clocks are adjusted only when sending or receiving a message.

Example: Enforcing Causal Communication (1)

- Causally-ordered multicasting: We want to ensure that a message is delivered only if all messages that causally precede it have been delivered. We assume that the message are multicast within the group.
- Clocks are adjusted only when sending or receiving a message.
- Then if P_j receives a message m from P_i with vector timestamp $ts(\mathtt{m})$, the delivery is delayed until the following conditions are met:

Example: Enforcing Causal Communication (1)

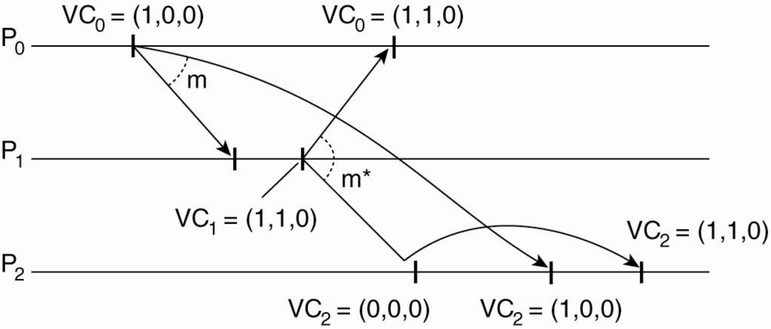
- Causally-ordered multicasting: We want to ensure that a message is delivered only if all messages that causally precede it have been delivered. We assume that the message are multicast within the group.
- Clocks are adjusted only when sending or receiving a message.
- Then if P_j receives a message m from P_i with vector timestamp $ts(\mathtt{m})$, the delivery is delayed until the following conditions are met:
  - $$ts(m)[i] = VC_j[i] + 1$$

    (m is the next process P_j was expecting from P_i)

Example: Enforcing Causal Communication (1)

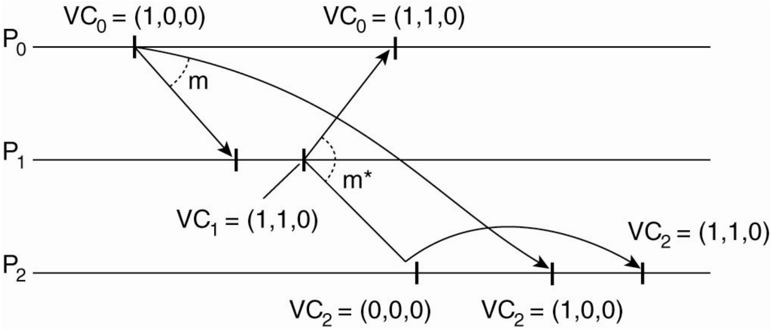
- Causally-ordered multicasting: We want to ensure that a message is delivered only if all messages that causally precede it have been delivered. We assume that the message are multicast within the group.
- Clocks are adjusted only when sending or receiving a message.
- Then if P_j receives a message m from P_i with vector timestamp $ts(\mathtt{m})$, the delivery is delayed until the following conditions are met:
  - $$ts(m)[i] = VC_j[i] + 1$$
    (m is the next process P_j was expecting from P_i)
  - $$ts(m)[k] \leq VC_j[k], \forall k \neq i$$
    (P_j has seen all the messages that have been seen by P_i when it sent message m)

Example: Enforcing Causal Communication (2)



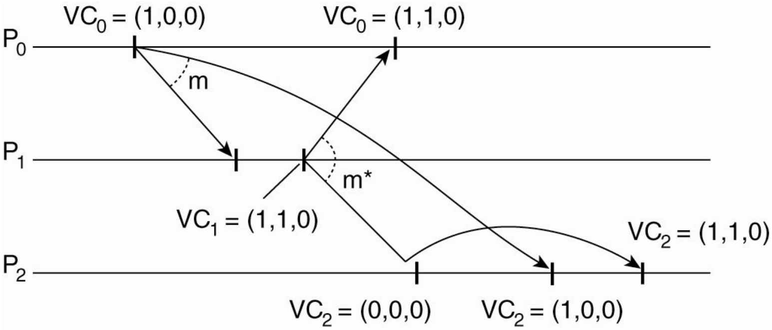
- At time (1,0,0): P_0 sends message *m* to P_1 and P_2.

Example: Enforcing Causal Communication (2)



- At time (1,0,0): P_0 sends message m to P_1 and P_2.
- After receipt by P_1, it decides to send m^* to P_2.

Example: Enforcing Causal Communication (2)



- At time (1,0,0): P_0 sends message m to P_1 and P_2.
- After receipt by P_1, it decides to send m^* to P_2.
- On P_2: The message m^* arrives sooner than m. The delivery of m^* is delayed by P_2 until m has been received and delivered to P_2's application layer.

Whose problem is it?

- Middleware deals with message ordering:

Whose problem is it?

- Middleware deals with message ordering:
  - The middleware cannot tell what the message contains so only potential causality is captured.

Whose problem is it?

- Middleware deals with message ordering:
  - The middleware cannot tell what the message contains so only potential causality is captured.
  - Two messages sent by the same process are always marked as causally related.

Whose problem is it?

- Middleware deals with message ordering:
  - The middleware cannot tell what the message contains so only potential causality is captured.
  - Two messages sent by the same process are always marked as causally related.
- Middleware cannot be aware of external communication. Ordering issues can be adequately solved by looking at the application for which the communication is taking place. This is known as the end-to-end argument.

Election Algorithms

- Many distributed systems need one process to act as a coordinator to initiate actions or to resolve conflicts.

Election Algorithms

- Many distributed systems need one process to act as a coordinator to initiate actions or to resolve conflicts.
- *Assumptions*:
  - We will assume that each process has a unique process number id so we can attempt to locate the highest numbered process to act as the coordinator.

Election Algorithms

- Many distributed systems need one process to act as a coordinator to initiate actions or to resolve conflicts.
- *Assumptions*:
  - We will assume that each process has a unique process number id so we can attempt to locate the highest numbered process to act as the coordinator.
  - Every process knows the id numbers of all other processes.

Election Algorithms

- Many distributed systems need one process to act as a coordinator to initiate actions or to resolve conflicts.
- *Assumptions*:
  - We will assume that each process has a unique process number id so we can attempt to locate the highest numbered process to act as the coordinator.
  - Every process knows the id numbers of all other processes.
  - A process doesn't not know which of the other processes is up or down.

Election Algorithms

- Many distributed systems need one process to act as a coordinator to initiate actions or to resolve conflicts.
- *Assumptions*:
  - We will assume that each process has a unique process number id so we can attempt to locate the highest numbered process to act as the coordinator.
  - Every process knows the id numbers of all other processes.
  - A process doesn't not know which of the other processes is up or down.
- *Goal*: When an election starts, it ends with all processes agreeing on who the coordinator is.

Bully Algorithm (1)

- When a process P notices that the coordinator is no longer responding to requests, it initiates an election as follows:

Bully Algorithm (1)

- When a process P notices that the coordinator is no longer responding to requests, it initiates an election as follows:
  1. P sends an `ELECTION` message to all processes with a higher id numbers.

Bully Algorithm (1)

- When a process P notices that the coordinator is no longer responding to requests, it initiates an election as follows:
  1. P sends an `ELECTION` message to all processes with a higher id numbers.
  2. If no one responds, P wins the election and becomes coordinator.

Bully Algorithm (1)

- When a process P notices that the coordinator is no longer responding to requests, it initiates an election as follows:
  1. P sends an `ELECTION` message to all processes with a higher id numbers.
  2. If no one responds, P wins the election and becomes coordinator.
  3. If one of the higher-ups answers, it takes over. P's job is done.

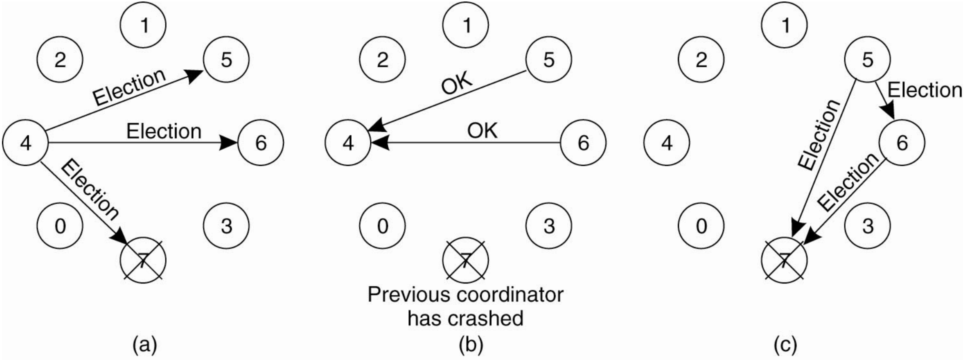
Bully Algorithm (1)

- When a process P notices that the coordinator is no longer responding to requests, it initiates an election as follows:
  1. P sends an `ELECTION` message to all processes with a higher id numbers.
  2. If no one responds, P wins the election and becomes coordinator.
  3. If one of the higher-ups answers, it takes over. P's job is done.
- The new coordinator sends a message to all processes announcing that is is the new coordinator.

Bully Algorithm (1)

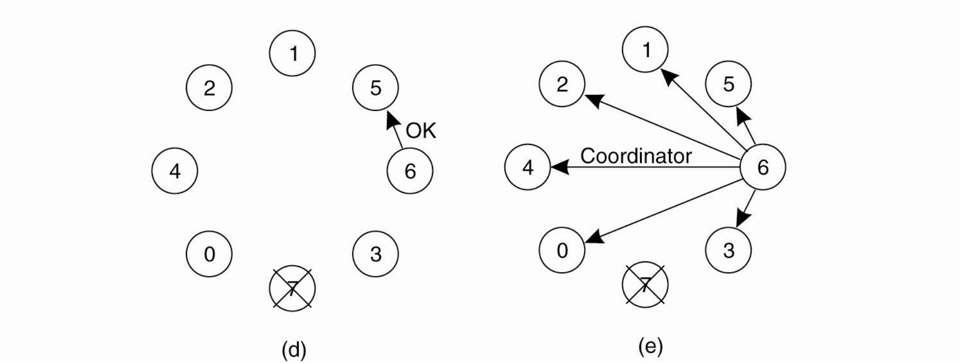
- When a process P notices that the coordinator is no longer responding to requests, it initiates an election as follows:
  1. P sends an `ELECTION` message to all processes with a higher id numbers.
  2. If no one responds, P wins the election and becomes coordinator.
  3. If one of the higher-ups answers, it takes over. P's job is done.
- The new coordinator sends a message to all processes announcing that is is the new coordinator.
- Several elections can be running simultaneously. If a process that was down previously comes back up, it immediately runs an election. The "biggest" process in town always wins, hence the name "bully algorithm."

Bully Algorithm (2)



(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.

Bully Algorithm (3)



(d) Process 6 tells 5 to stop.

(e) Process 6 wins and tells everyone.

Bully Algorithm (4)

- The new coordinator typically has to pick up the state information left off by the old coordinator before it announces the results of the election.

Bully Algorithm (4)

- The new coordinator typically has to pick up the state information left off by the old coordinator before it announces the results of the election.
- If Process 7 comes back up, it just sends a new election message and bullies them into submission.

Bully Algorithm (4)

- The new coordinator typically has to pick up the state information left off by the old coordinator before it announces the results of the election.
- If Process 7 comes back up, it just sends a new election message and bullies them into submission.
- We can use `Are You Alive` messages periodically to speed up detection of absconding coordinators

Ring Algorithm (1)

- The processes are logically arranged in a ring. Each process knows its neighbor in the ring as well who all is in the ring. When any process notices that the coordinator is not responding, it builds an `ELECTION` message containing its own process number and sends it to its successor

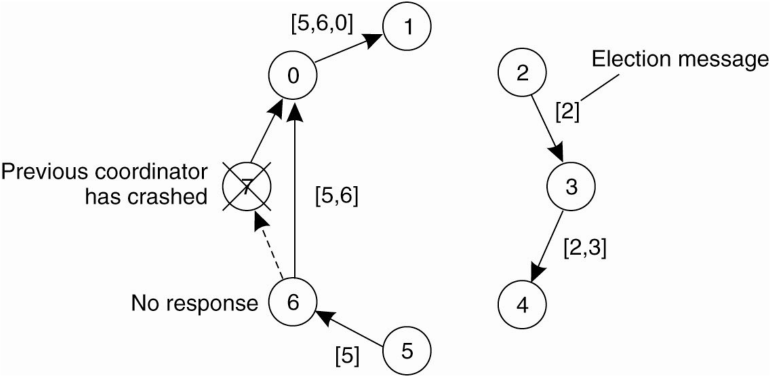
Ring Algorithm (1)

- The processes are logically arranged in a ring. Each process knows its neighbor in the ring as well who all is in the ring. When any process notices that the coordinator is not responding, it builds an `ELECTION` message containing its own process number and sends it to its successor
- At each step, the sender adds its own number to the list in the message thus making itself be a candidate. Eventually, the message reaches the process that started it all. Then it looks through the message and decides which process has the highest number and that becomes the coordinator

Ring Algorithm (1)

- The processes are logically arranged in a ring. Each process knows its neighbor in the ring as well who all is in the ring. When any process notices that the coordinator is not responding, it builds an `ELECTION` message containing its own process number and sends it to its successor
- At each step, the sender adds its own number to the list in the message thus making itself be a candidate. Eventually, the message reaches the process that started it all. Then it looks through the message and decides which process has the highest number and that becomes the coordinator
- Then the message type is changed to `COORDINATOR` and the message circulates once again so everyone knows the new coordinator and the new ring configuration.

Ring Algorithm (2)



- Elections in Wireless Environments. We often want the best leader (the one with most battery life or other relevant resources in mobile environments).

- Elections in Wireless Environments. We often want the best leader (the one with most battery life or other relevant resources in mobile environments).
  - Impose a tree on the network and work our way backwards to determine the best leader.

Wireless and Large Scale Systems

- Elections in Wireless Environments. We often want the best leader (the one with most battery life or other relevant resources in mobile environments).
  - Impose a tree on the network and work our way backwards to determine the best leader.
- Elections in Large Scale Systems. Several superpeer nodes may be selected instead of just one. The superpeers should be evenly distributed across the network. Normal nodes should have low latency access to superpeers. There should be a predefined portion of superpeers and each superpeer should not have to serve more than a fixed number of normal nodes.

Wireless and Large Scale Systems

- Elections in Wireless Environments. We often want the best leader (the one with most battery life or other relevant resources in mobile environments).
  - Impose a tree on the network and work our way backwards to determine the best leader.
- Elections in Large Scale Systems. Several superpeer nodes may be selected instead of just one. The superpeers should be evenly distributed across the network. Normal nodes should have low latency access to superpeers. There should be a predefined portion of superpeers and each superpeer should not have to serve more than a fixed number of normal nodes.
  - Solutions use either DHT (Distributed Hash Tables) or randomly unstructured layouts.

References

- Time, clocks, and the ordering of events in a distributed system. Leslie Lamport. *Communications of the ACM* 21 (7): 558–565, 1978.
- Time is an illusion. Lunchtime doubly so. George V. Neville-Neil. *Communications of the ACM*, January 2016, Vol. 59. No. 1, pages 50–55. [Note: this article is only accessible on campus]