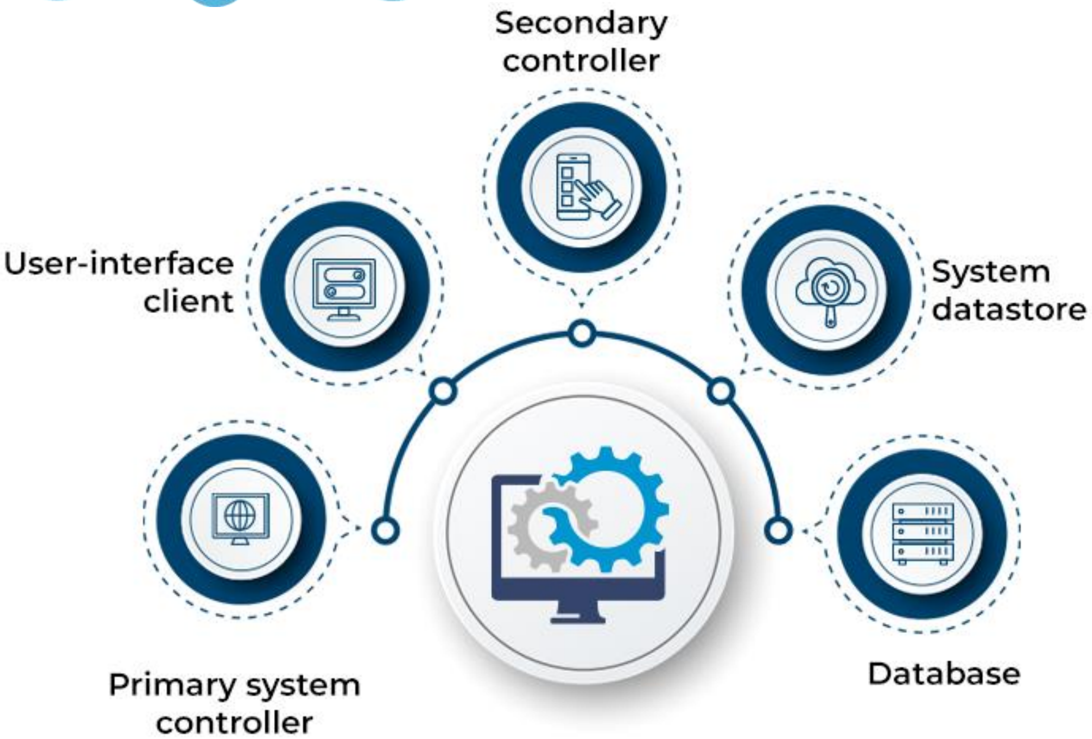
Sistem Terdistribusi

IF2222



06-07: Sinkronisasi

2022-23 Ganjil

Teknik Informatika
Universitas Trunojoyo Madura

Sistem Terdistribusi 2022

1. Mengenal Sistem Terdistribusi
2. Review Jaringan Komputer (layer 2, 3, dan 4)
3. Arsitektur Sistem Terdistribusi
4. *Remote Procedure Calls* (RPC)
5. Layanan Penamaan
6. **Sinkronisasi Data (2 pekan)**
7. *Message Passing Interface* (MPI)
8. Contoh Arsitektur: Hadoop, Pregel, Blockchain
9. Teknik *Caching*
10. Teknik Replikasi Data (2 pekan)
11. Basis Data Terdistribusi
12. Toleransi Kegagalan

Capaian Pembelajaran

Kuliah ini bertujuan memberikan pemahaman mendalam dan pengalaman langsung tentang:

- Prinsip yang menjadi basis dari sistem terdistribusi
- Prinsip untuk optimalisasi sistem terdistribusi
- Model pemrograman sistem terdistribusi dan mesin analitika
- Bagaimana sistem terdistribusi modern memenuhi tuntutan aplikasi terdistribusi kontemporer

Today…

- Last Session
  - Layanan Penamaan
- Today's Session
  - Synchronization
    - Coordinated Universal Time (UTC)
    - Tracking Time on a Computer
    - Clock Synchronization: Cristian's Algorithm, Berkeley Algorithm and Network Time Protocol (NTP)
- Announcements

Synchronization

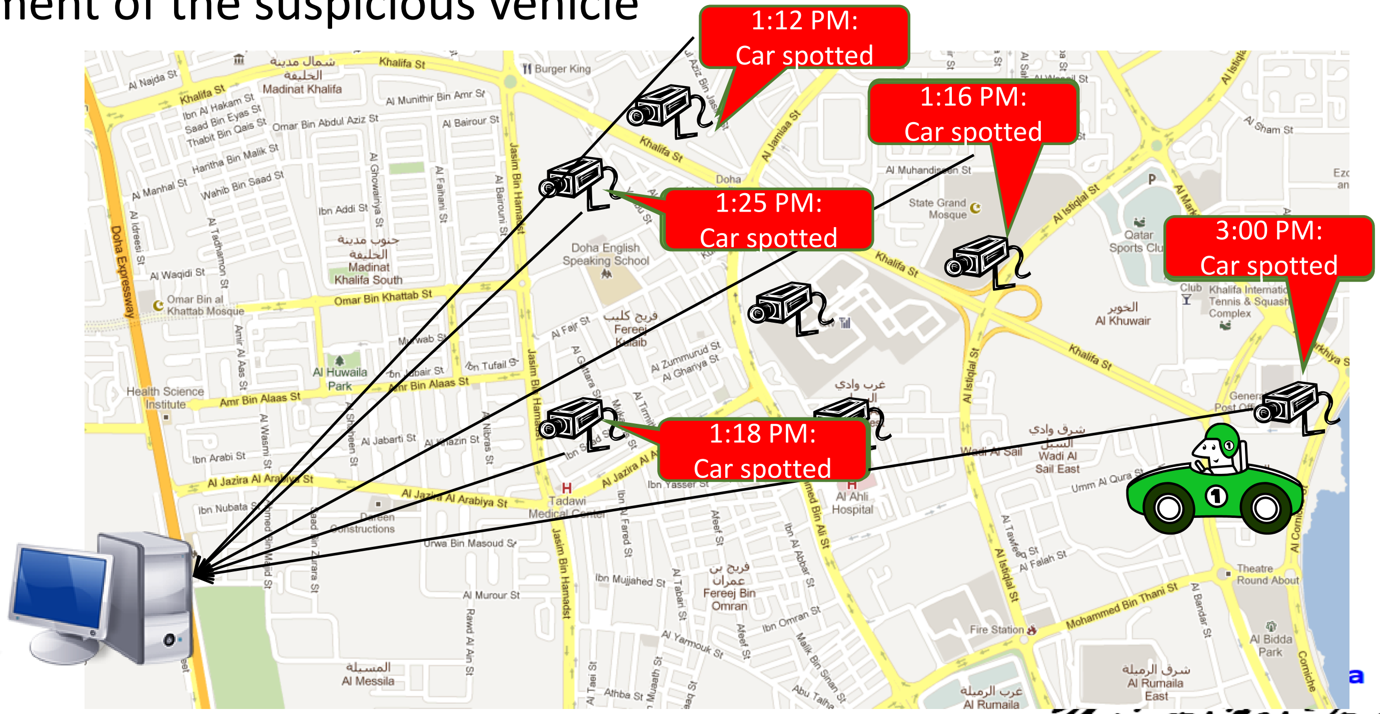
- Until now, we have looked at:
  - How entities can be organized and communicate with each other
  - How entities are named and identified

- In addition to the above requirements, entities in DSs often have to *cooperate* and *synchronize* to solve a given problem correctly
  - E.g., In a distributed file system, processes have to synchronize and cooperate such that two processes are not allowed to write to the same part of a file

Need for Synchronization – Example 1

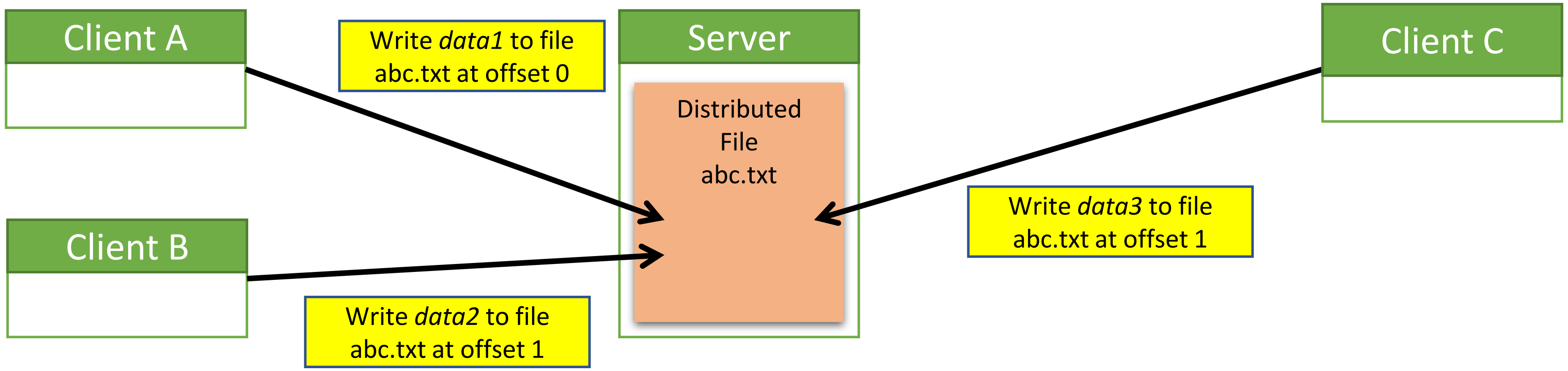
- Vehicle tracking in a City Surveillance System using a Distributed Sensor Network of Cameras
  - **<u>Objective:</u> To keep track of suspicious vehicles**
  - Camera Sensor Nodes are deployed over the city
  - Each Camera Sensor that detects a vehicle reports the time to a central server
  - Server tracks the movement of the suspicious vehicle

If the sensor nodes do not have a consistent version of the time, the vehicle cannot be reliably tracked



Need for Synchronization – Example 2

- Writing a file in a Distributed File System



If the distributed clients do not synchronize their write operations to the distributed file, then the data in the file can be corrupted

A Broad Taxonomy of Synchronization

Reason for synchronization and cooperation	Entities have to agree on ordering of events	Entities have to share common resources
Examples	E.g., Vehicle tracking in a Camera Sensor Network; Financial transactions in Distributed E-commerce Systems	E.g., Reading and writing in a Distributed File System
Requirement for entities	Entities should have a common understanding of time across different computers	Entities should coordinate and agree on when and how to access resources
Topics we will study	Time Synchronization	Mutual Exclusion

Overview

Today's lecture

- Time Synchronization
  - Physical Clock Synchronization (or, simply, Clock Synchronization)
    - Here, actual time on computers are synchronized
  - Logical Clock Synchronization
    - Computers are synchronized based on relative ordering of events
- Mutual Exclusion
  - How to coordinate between processes that access the same resource?
- Election Algorithms
  - Here, a group of entities elect one entity as the coordinator for solving a problem

Next two lectures

Overview

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
- Election Algorithms

Clock Synchronization

- Clock synchronization is a mechanism to synchronize the time of all the computers in a DS

- We will study:
  - Coordinated Universal Time
  - Tracking Time on a Computer
  - Clock Synchronization Algorithms
    - Cristian's Algorithm
    - Berkeley Algorithm
    - Network Time Protocol

Clock Synchronization

- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
    - Cristian's Algorithm
    - Berkeley Algorithm
    - Network Time Protocol

Coordinated Universal Time (UTC)

- All the computers are generally synchronized to a standard time called Coordinated Universal Time (UTC)
  - UTC is the primary time standard by which the world regulates clocks and time

- UTC is broadcasted via the satellites
  - UTC broadcasting service provides an accuracy of 0.5 msec

- Computer servers and online services with **UTC receivers** can be synchronized by satellite broadcasts
  - Many popular synchronization protocols in distributed systems use UTC as a reference time to synchronize clocks of computers

Clock Synchronization

- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
  - Cristian’s Algorithm
  - Berkeley Algorithm
  - Network Time Protocol

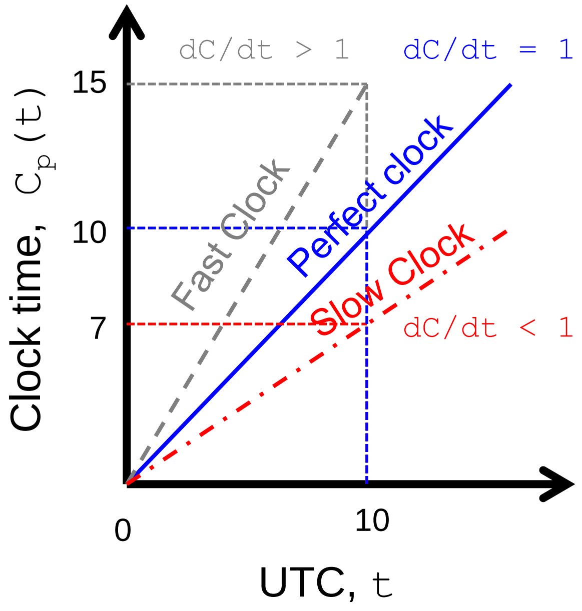
Tracking Time on a Computer

- How does a computer keep track of its time?
  - Each computer has a hardware *timer*
    - The timer causes an interrupt ‘H’ times a second
  - The interrupt handler adds 1 to its Software Clock (C)
- Issues with clocks on a computer
  - In practice, the hardware timer is imprecise
    - It does not interrupt ‘H’ times a second due to material imperfections of the hardware and temperature variations
    - The computer counts the time slower or faster than actual time
  - Loosely speaking, Clock Skew is the skew between:
    - the computer clock and the actual time (e.g., UTC)

Clock Skew

- When the UTC time is `t`, let the clock on the computer have a time `C(t)`

- Three types of clocks are possible
  - Perfect clock:
    - The timer ticks 'H' interrupts a second
      `dC/dt = 1`
  - Fast clock:
    - The timer ticks more than 'H' interrupts a second
      `dC/dt > 1`
  - Slow clock:
    - The timer ticks less than 'H' interrupts a second
      `dC/dt < 1`



Clock Skew (cont'd)

- Frequency of the clock is defined as the ratio of the number of seconds counted by the software clock for every UTC second

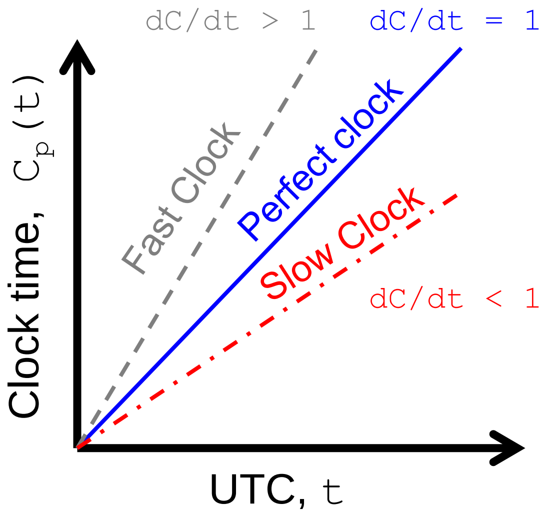
$$\text{Frequency} = dC/dt$$

- Skew of the clock is defined as the extent to which the frequency differs from that of a perfect clock

$$\text{Skew} = dC/dt - 1$$

- Hence,

$$Skew \begin{cases} > 0 & \text{for a fast clock} \\ = 0 & \text{for a perfect clock} \\ < 0 & \text{for a slow clock} \end{cases}$$



Maximum Drift Rate of a Clock

- The manufacturer of the timer specifies the upper and the lower bound that the clock skew may fluctuate. This value is known as *maximum drift rate (ρ)*

$$1 - \rho <= dC/dt <= 1 + \rho$$

- How far can two clocks drift apart?
  - If two clocks are drifting from UTC in the opposite direction, at a time Δt after they were synchronized, they may be as much as $2\rho\Delta t$ seconds apart
- Guaranteeing maximum drift between computers in a DS
  - If maximum drift permissible in a DS is δ seconds, then clocks of every computer must be resynchronized at least every $\delta/2\rho$ seconds

Clock Synchronization

- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
  - Cristian’s Algorithm
  - Berkeley Algorithm
  - Network Time Protocol

Cristian’s Algorithm

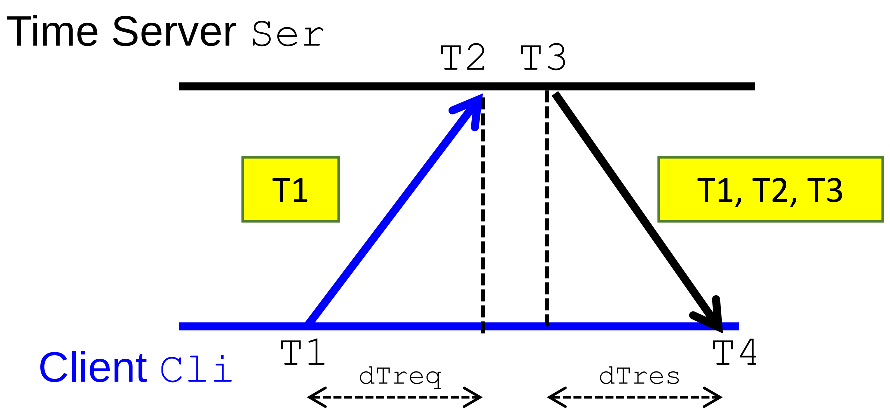
- Flaviu Cristian (in 1989) provided an algorithm to synchronize networked computers with a time server

- The basic idea:
  - Identify a network time server that has an accurate source for time (e.g., the time server has a UTC receiver)
  - All the clients contact the network time server for synchronization

- However, the network delays incurred when the client contacts the time server results in outdated time
  - The algorithm estimates the network delays and compensates for it

Cristian's Algorithm – Approach

+ Client `Cli` sends a request to Time Server `Ser`, time stamped its local clock time `T1`
+ `Ser` will record the time of receipt `T2` according to its local clock
  + `dTreq` is network delay for request transmission



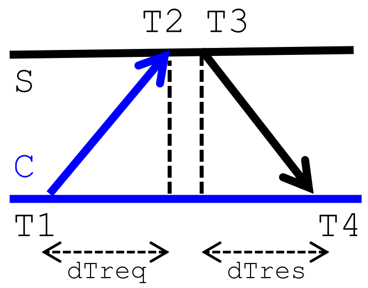
- `Ser` replies to `Cli` at its local time `T3`, piggybacking `T1` and `T2`
- `Cli` receives the reply at its local time `T4`
  - `dTres` is the network delay for response transmission
- Now `Cli` has the information `T1`, `T2`, `T3` and `T4`
- **Assuming that the transmission delay from `Cli`→`Ser` and `Ser`→`Cli` are the same**

$$T2-T1 \approx T4-T3$$

Christian's Algorithm – Synchronizing Client Time

- Client C estimates its offset θ relative to Time Server S

  θ = T3 + dTres – T4
  = T3 + ((T2-T1)+(T4-T3))/2 – T4
  = ((T2-T1)+(T3-T4))/2



- If θ > 0 or θ < 0, then the client time should be incremented or decremented by θ seconds

Gradual Time Synchronization at the client

- Instead of changing the time drastically by θ seconds, typically the time is gradually synchronized
  - The software clock is updated at a lesser/greater rate whenever timer interrupts

Note: Setting clock backward (say, if θ < 0) is not allowed in a DS since decrementing a clock at any computer has adverse effects on several applications (e.g., *make* program)

Cristian's Algorithm – Discussion

1. Assumption about packet transmission delays

- Cristian's algorithm assumes that the round-trip times for messages exchanged over the network are reasonably short
- The algorithm assumes that the delay for the request and response are equal
  - Will the trend of increasing Internet traffic decrease the accuracy of the algorithm?
  - Can the algorithm handle delay asymmetry that is prevalent in the Internet?
  - Can the clients be mobile entities with intermittent connectivity?

Cristian's algorithm is intended for synchronizing computers within intranets

2. A probabilistic approach for calculating delays

- There is no tight bound on the maximum drift between clocks of computers

3. Time server failure or faulty server clock

- Faulty clock on the time server leads to inaccurate clocks in the entire DS
- Failure of the time server will render synchronization impossible

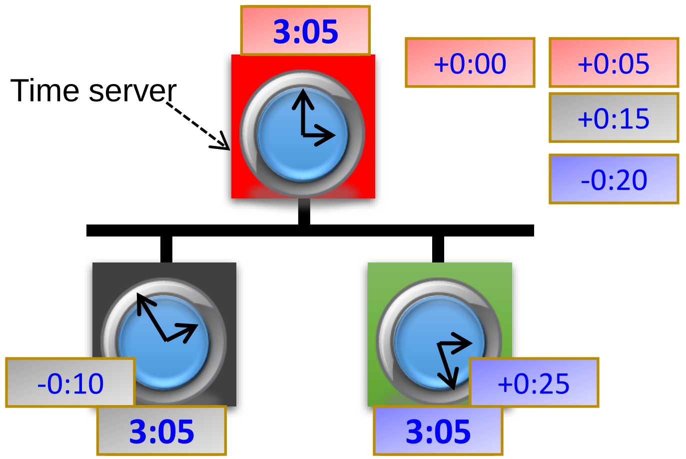
Clock Synchronization

- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
  - Cristian's Algorithm
  - Berkeley Algorithm
  - Network Time Protocol

Berkeley Algorithm

- Berkeley algorithm is a distributed approach for time synchronization

- Approach:
    1. A time server periodically (approx. once in 4 minutes) sends its time to all the computers and polls them for the time difference
    2. The computers compute the time difference and then reply
    3. The server computes an average time difference for each computer
    4. The server commands all the computers to update their time (by gradual time synchronization)



Berkeley Algorithm – Discussion

1. Assumption about packet transmission delays

- Berkeley's algorithm predicts network delay (similar to Cristian's algorithm)
- Hence, it is effective in intranets, and not accurate in wide-area networks

2. No UTC Receiver is necessary

- The clocks in the system synchronize by averaging all the computer's times

3. Decreases the effect of faulty clocks

- Fault-tolerant averaging, where outlier clocks are ignored, can be easily performed in Berkeley Algorithm

4. Time server failures can be masked

- If a time server fails, another computer can be *elected* as a time server

Clock Synchronization

- Coordinated Universal Time
- Tracking Time on a Computer
- Clock Synchronization Algorithms
  - Cristian's Algorithm
  - Berkeley Algorithm
  - Network Time Protocol

Network Time Protocol (NTP)

- NTP defines an architecture for a time service and a protocol to distribute time information over the Internet
- In NTP, servers are connected in a logical hierarchy called *synchronization subnet*
- The levels of synchronization subnet is called *strata*
  - Stratum 1 servers have most accurate time information (connected to a UTC receiver)
  - Servers in each stratum act as time servers to the servers in the lower stratum

Hierarchical organization of NTP Servers

More accurate time

Stratum 1
- This stratum contains the *primary servers* that are directly connected to the UTC receivers

Stratum 2
- Stratum 2 are secondary servers that are synchronized directly with primary servers

Stratum 3
- Stratum 3 synchronizes with Stratum 2 servers

Last stratum
- End user computers synchronize with the servers in the upper layer stratum

Operation of NTP Protocol

- When a time server **A** contacts time server **B** for synchronization
  - If `stratum(A) <= stratum(B)`, then **A** does not synchronize with B
  - If `stratum(A) > stratum(B)`, then:
    - Time server **A** synchronizes with **B**
    - An algorithm similar to Cristian's algorithm is used to synchronize. However, larger statistical samples are taken before updating the clock
    - Time server **A** updates its stratum

```
stratum(A) = stratum(B) + 1
```

Discussion of NTP Design

Accurate synchronization to UTC time

- NTP enables clients across the Internet to be synchronized accurately to the UTC
- Large and variable message delays are tolerated through statistical filtering of timing data from different servers

Scalability

- NTP servers are hierarchically organized to speed up synchronization, and to scale to a large number of clients and servers

Reliability and Fault-tolerance

- There are redundant time servers, and redundant paths between the time servers
- The architecture provides reliable service that can tolerate lengthy losses of connectivity
- A synchronization subnet can reconfigure as servers become unreachable. For example, if Stratum 1 server fails, then it can become a Stratum 2 secondary server

Security

- NTP protocol uses authentication to check of the timing message originated from the claimed trusted sources

Summary of Clock Synchronization

- Physical clocks on computers are not accurate
- Clock synchronization algorithms provide mechanisms to synchronize clocks on networked computers in a DS
  - Computers on a local network use various algorithms for synchronization
    - Some algorithms (e.g, Cristian's algorithm) synchronize time by contacting centralized time servers
    - Some algorithms (e.g., Berkeley algorithm) synchronize in a distributed manner by exchanging the time information on various computers
  - NTP provides architecture and protocol for time synchronization over wide-area networks such as the Internet

Today

- Last Session:
  - UTC, tracking time on a computer, physical clock synchronization
- Today's Session:
  - Logical Clock Synchronization
    - Lamport's and Vector Clocks
  - Introduction to Distributed Mutual Exclusion
- Announcements:
  - PS3 is due tomorrow by midnight
  - Midterm is on March 9 during class time (open book; open notes)

Continuing Synchronization

Previous lecture

- Time Synchronization
  - Physical Clock Synchronization (or, simply, Clock Synchronization)
    - Here, actual time on the computers is synchronized
  - Logical Clock Synchronization
    - Computers are synchronized based on the relative ordering of events

Today's lecture

- Mutual Exclusion
  - How to coordinate between processes that access the same resource?
- Election Algorithms
  - Here, a group of entities elect one entity as the coordinator for solving a problem

Next lecture

Overview

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
- Election Algorithms

Why Logical Clocks?

- Lamport (in 1978) showed that:
  - Clock synchronization is not necessary in all scenarios
    - If two processes do not interact, it is not necessary that their clocks are synchronized
  - Many times, it is sufficient if processes agree on the *order* in which the events have occurred in a DS
    - For example, for a distributed *make* utility, it is sufficient to know if a source file was modified *before* or *after* its object file

Logical Clocks

- Logical clocks are used to define an order of events without measuring the physical time at which the events have occurred

- We will study two types of logical clocks
  1. Lamport’s Logical Clock (or simply, Lamport’s Clock)
  2. Vector Clock

Logical Clocks

- We will study two types of logical clocks

1. Lamport's Clock

2. Vector Clock

Lamport's Clock

- Lamport advocated maintaining *logical clocks at the processes* to keep track of the order of events

- To synchronize logical clocks, Lamport defined a relation called "**`happened-before`**"

- The expression **`a`**→**`b`** (reads as "**`a`** happened before **`b`**") means that *all* entities in a DS agree that event **`a`** occurred before event **`b`**

The Happened-before Relation

- The **`happened-before`** relation can be observed directly in two situations:
  1. If **`a`** and **`b`** are events in the same process, and **`a`** occurs before **`b`**, then **`a`**$\rightarrow$**`b`** is true
  2. If **`a`** is an event of message **`m`** being sent by a process, and **`b`** is the event of **`m`** (i.e., the same message) being received by another process, then **`a`**$\rightarrow$**`b`** is true
- The **`happened-before`** relation is *transitive*
  - If **`a`**$\rightarrow$**`b`** and **`b`**$\rightarrow$**`c`**, then **`a`**$\rightarrow$**`c`**

Time values in Logical Clocks

- For every event **`a`**, assign a logical *time value* **`C(a)`** on which all processes agree (***C*** *corresponds to the process and not to the event, but gets updated when the event happens*)

- Time value for events have the property that:
  - If **`a`**→**`b`**, then **`C(a)< C(b)`**

Properties of Logical Clock

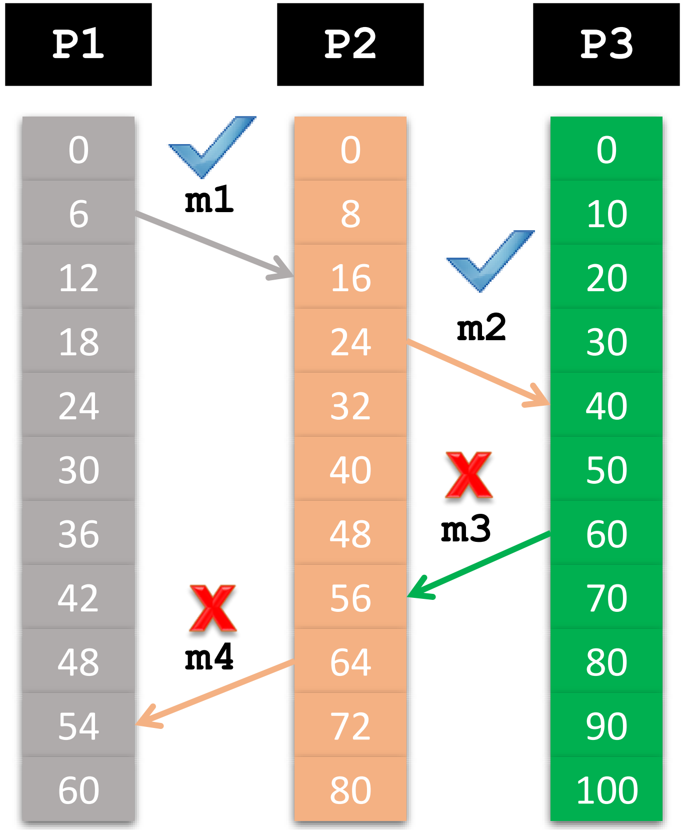
- From the **happened-before** relation, we can infer that:
  - If two events **a** and **b** occur within the same process and **a→b**, then $C(a)$ and $C(b)$ are assigned time values such that $C(a) < C(b)$
  - If **a** is the event of sending message **m** from one process (say P1), and **b** is the event of receiving **m** (i.e., the same message) at another process (say, P2), then:
    - The time values $C_1(a)$ and $C_2(b)$ are assigned in a way such that the two processes agree that $C_1(a) < C_2(b)$
  - The clock time **C** must always go forward (increasing), and never backward (decreasing)

Synchronizing Logical Clocks

- Three processes **P1**, **P2** and **P3** running at different rates

- If the processes communicate between each other, there might be discrepancies in agreeing on the event ordering
  - The ordering of sending and receiving messages **m1** and **m2** is correct

  - However, **m3** and **m4** violate the happened-before relationship

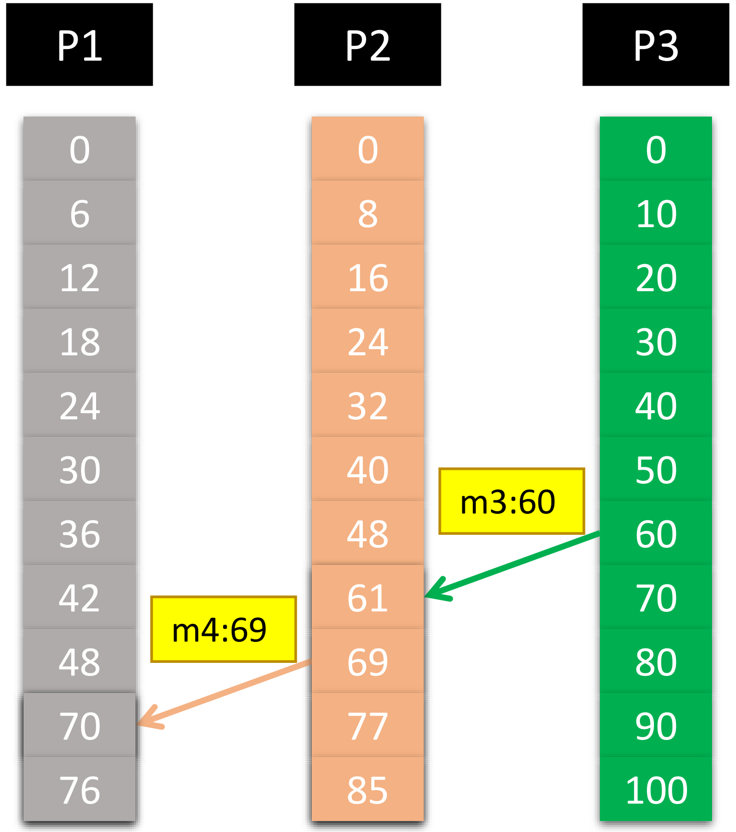


Lamport's Clock Algorithm

- When a message is being sent:
  - Each message carries a **`timestamp`** according to the sender's logical clock

- When a message is received:
  - If the receiver logical clock is less than the message sending time in the packet, then adjust the receiver's clock such that:

```
currentTime = timestamp + 1
```



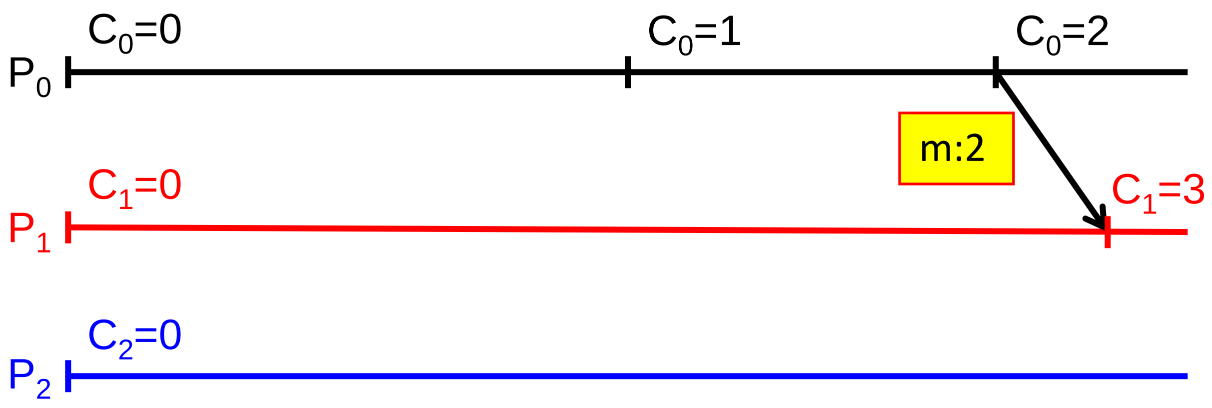
Logical Clock Without a Physical Clock

- Previous examples assumed that there is a physical clock at each computer (probably running at different rates)
- How to attach a time value to an event when there is no global clock?

Implementation of Lamport's Clock

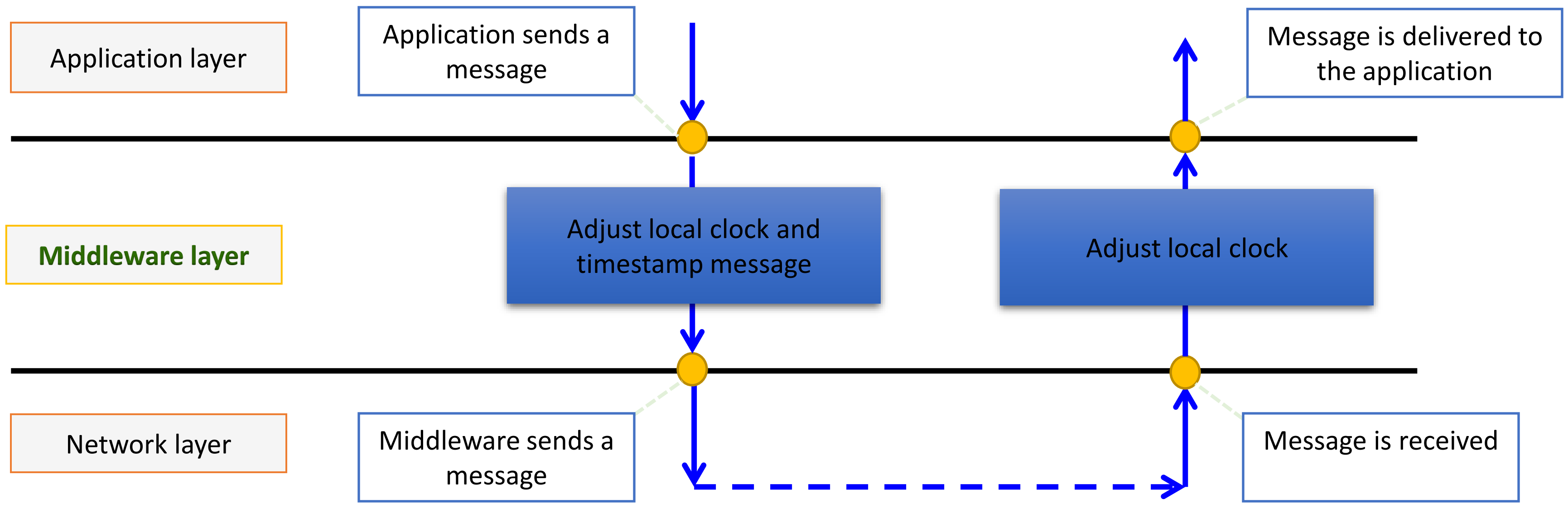
- Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:

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 , m is assigned a timestamp $ts(m) = C_i$
3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\mathbf{max(C_j,\ ts(m))\ +\ 1}$



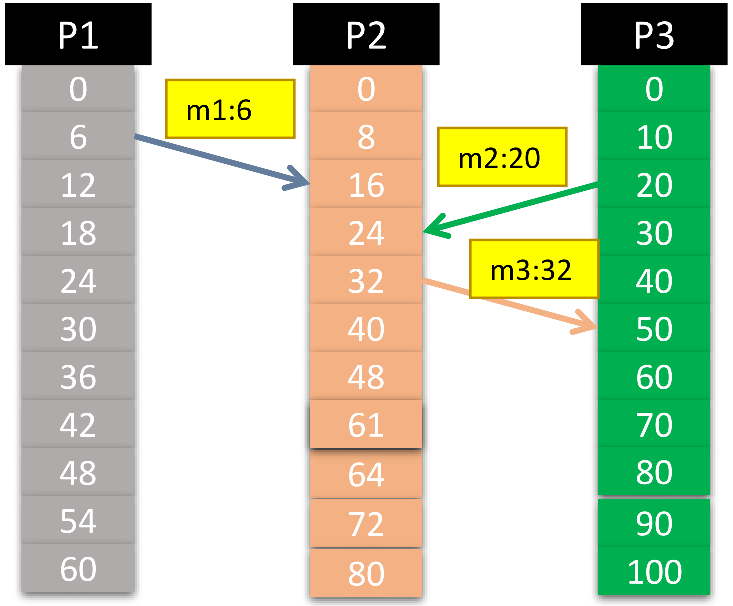
Placement of Logical Clock

- In a computer, several processes can use different logical clocks
- However, instead of each process maintaining its own logical clock, a single logical clock can be implemented in the middleware as a time service



Limitation of Lamport’s Clock

- Lamport’s clock ensures that if **a→b**, then `C(a) < C(b)`
- However, it does not say anything about any two *arbitrary* (*concurrent* or *independent*) events **a** and **b** by only comparing their time values
  - For any two arbitrary events **a** and **b**, `C(a) < C(b)` does not mean that **a→b**
- Example:



Compare m1 and m3
P2 can infer that **m1→m3**

Compare m1 and m2
P2 **cannot** infer that **m1→m2** or **m2→m1**

Summary of Lamport’s Clock

- Lamport suggested using logical clocks
  - Processes synchronize based on the time values of their logical clocks rather than the absolute time values of their physical clocks
- Which applications in DS need logical clocks?
  - Applications with provable ordering of events
    - Perfect physical clock synchronization is hard to achieve in practice
  - Applications with rare events
    - Events are rarely generated, and physical clock synchronization overhead is not justified
- However, Lamport’s Clock cannot guarantee perfect ordering of events by just observing the time values of two <u>*arbitrary*</u> events

Logical Clocks

- We will study two types of logical clocks

  1. Lamport's Clock
  2. Vector Clock

Vector Clocks

- Vector clock was proposed to overcome the limitation of Lamport's clock
  - The property of *inferring* that **a** occurred before **b** is known as the causality property
- A vector clock for a system of **N** processes is an array of **N** integers
- Every process $\mathbf{P_i}$ stores its own vector clock $\mathbf{VC_i}$
  - Lamport's time values for events are stored in $\mathbf{VC_i}$
  - $\mathbf{VC_i(a)}$ is assigned to an event **a**
- If $\mathbf{VC_i(a) < VC_i(b)}$, then we can infer that $\mathbf{a \rightarrow b}$ (or more precisely, that event **a** *causally* preceded event **b**)

Updating Vector Clocks

- Vector clocks are constructed as follows:

1. $\mathbf{VC_i[i]}$ is the number of events that have occurred at process $\mathbf{P_i}$ so far
    - $\mathbf{VC_i[i]}$ is the local logical clock at process $\mathbf{P_i}$

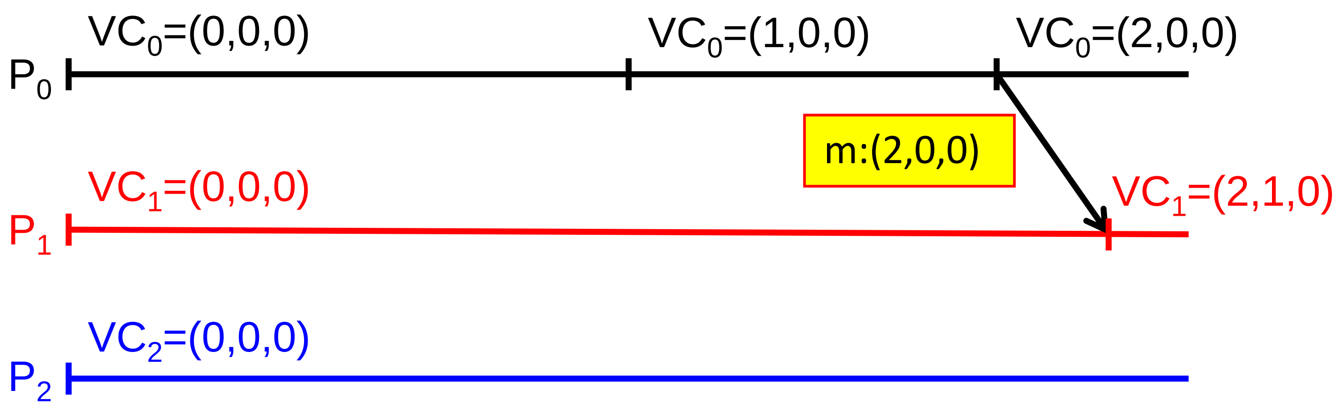
→ Increment VC_i whenever a new event occurs

2. If $\mathbf{VC_i[j]= k}$, then $\mathbf{P_i}$ knows that $\mathbf{k}$ events have occurred at $\mathbf{P_j}$
    - $\mathbf{VC_i[j]}$ is $\mathbf{P_i}$'s knowledge of the local time at $\mathbf{P_j}$

→ Pass VC_j along with the message

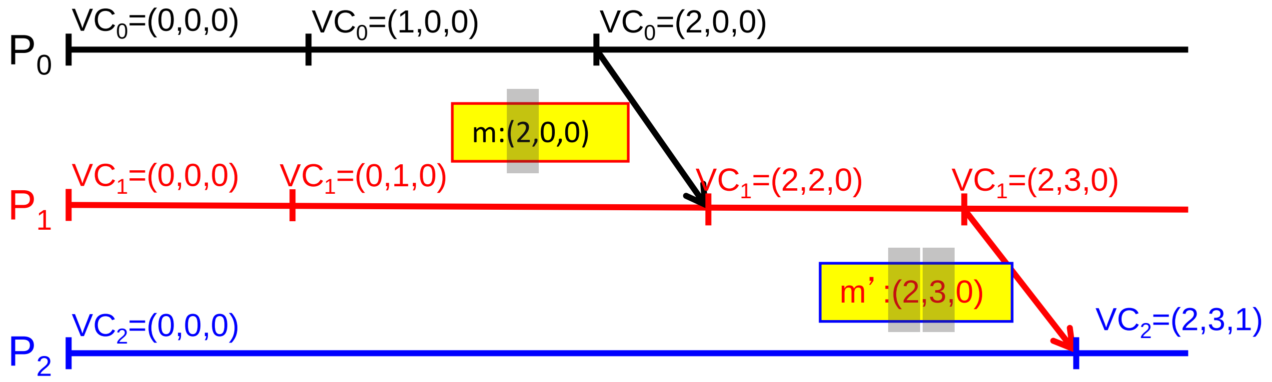
Vector Clock Update Algorithm

- Whenever there is a new event at $\mathbf{P_i}$, increment $\mathbf{VC_i[i]}$
- When a process $\mathbf{P_i}$ sends a message **m** to $\mathbf{P_j}$:
  - Increment $\mathbf{VC_i[i]}$
  - Set **m**'s timestamp **ts(m)** to the vector $\mathbf{VC_i}$
- When message m is received process $\mathbf{P_j}$:
  - $\mathbf{VC_j[k] = max(VC_j[k],\ ts(m)[k])}$; (for all **k**)
  - Increment $\mathbf{VC_j[j]}$



Inferring Events with Vector Clocks

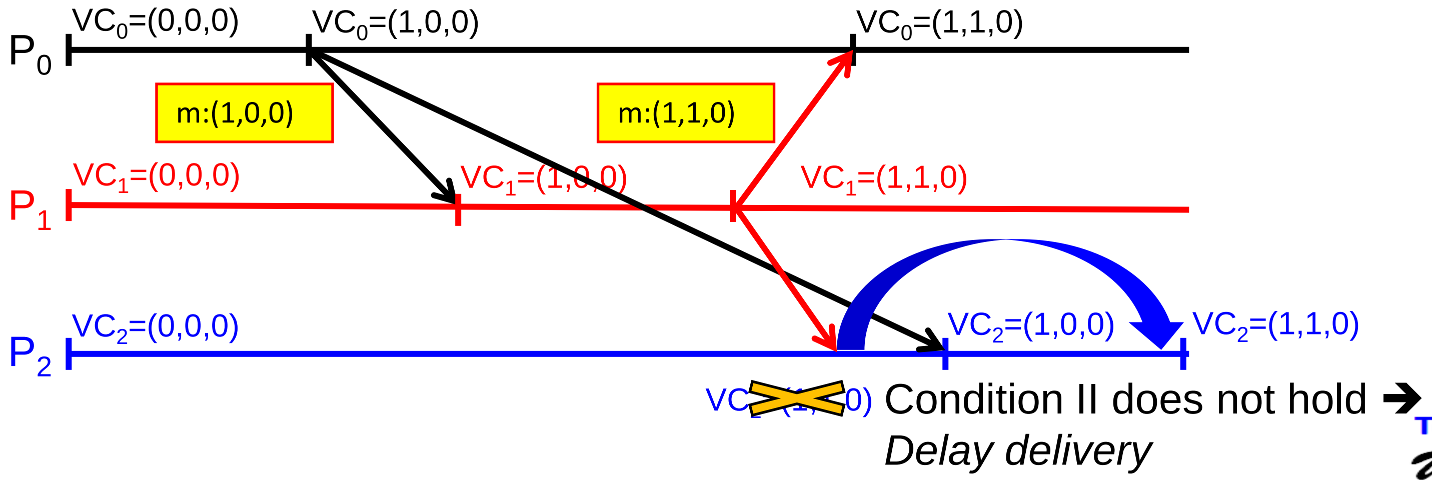
- Let a process $\mathbf{P_i}$ send a message $\mathbf{m}$ to $\mathbf{P_j}$ with timestamp $\mathbf{ts(m)}$, then:
  - $\mathbf{P_j}$ knows the number of events at the sender $\mathbf{P_i}$ that causally precede $\mathbf{m}$
    - $\mathbf{(ts(m)[i] - 1)}$ denotes the number of events at $\mathbf{P_i}$
  - $\mathbf{P_j}$ also knows the minimum number of events at other processes $\mathbf{P_k}$ that causally precede $\mathbf{m}$
    - $\mathbf{(ts(m)[k] - 1)}$ denotes the minimum number of events at $\mathbf{P_k}$



Enforcing Causal Communication

- Assume that messages are *multicast* within a group of processes, P_0, P_1 and P_2
- To enforce causally-ordered multicasting, the delivery of a message ***m*** sent from P_i to P_j can be delayed until the following two conditions are met:
  - ts(m)[i] = VC_j[i] + 1 (**Condition I**)
  - ts(m)[k] <= VC_j[k] for all k != i (**Condition II**)

Assuming that P_i only increments VC_i[i] upon sending m and adjusts VC_i[k] to max{VC_i[k], ts(m)[k]} for each k upon receiving a message m'



Teknik Informatika
Universitas Trunojoyo Madura

Summary – Logical Clocks

- Logical clocks are employed when processes have to agree on relative ordering of events, but not necessarily actual time of events

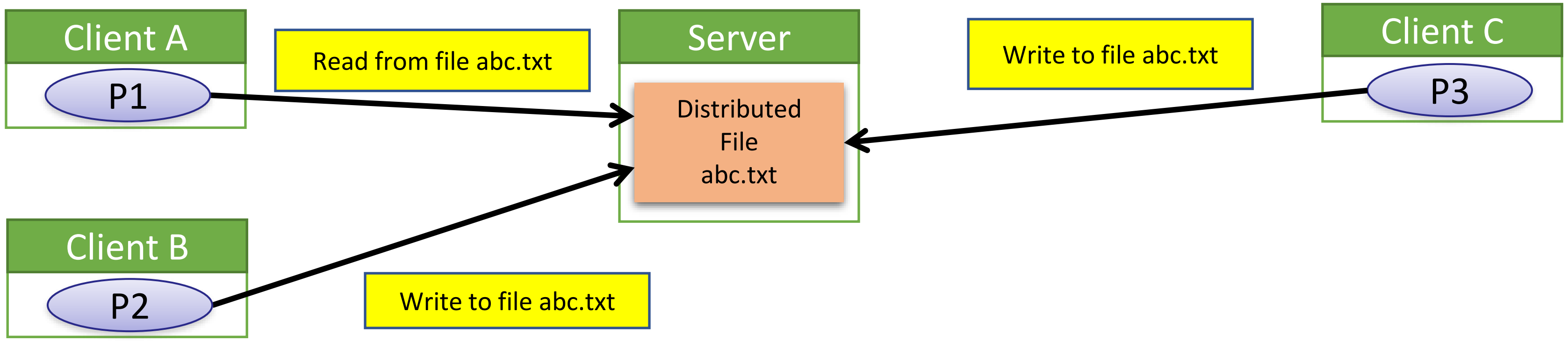
- Two types of logical clocks:
  - Lamport's Logical Clocks
    - Supports relative ordering of events across different processes by using the *happened-before* relationship
  - Vector Clocks
    - Supports *causal* ordering of events

Overview

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
- Election Algorithms

Need for Mutual Exclusion

- Distributed processes need to coordinate to access shared resources
- Example: Writing a file in a Distributed File System



In uniprocessor systems, mutual exclusion to a shared resource is provided through shared variables or operating system support

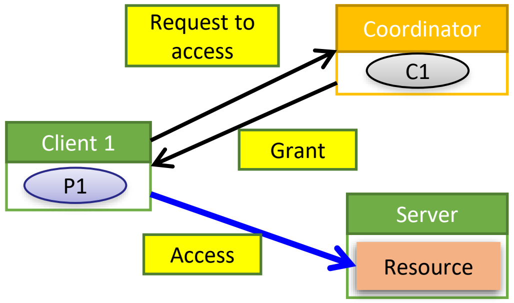
However, such support is insufficient to enable mutual exclusion of distributed entities

In distributed systems, processes coordinate accesses to a shared resource by passing messages to enforce *distributed mutual exclusion*

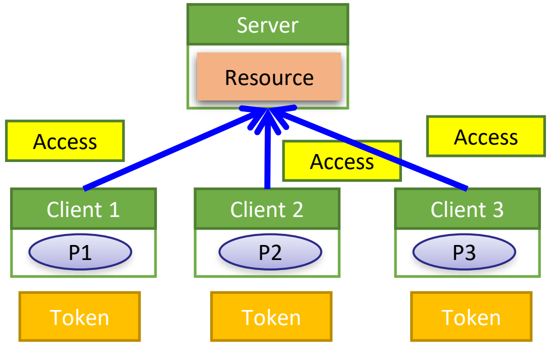
Types of Distributed Mutual Exclusion

- Mutual exclusion algorithms are classified into two categories

1. Permission-based Approaches
    - A process, which wants to access a shared resource, requests the permission from one or more coordinators



2. Token-based Approaches
    - Each shared resource has a token
    - Token is circulated among all the processes
    - A process can access the resource if it has the token



Overview

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
  - Permission-based Approaches
  - Token-based Approaches
- Election Algorithms

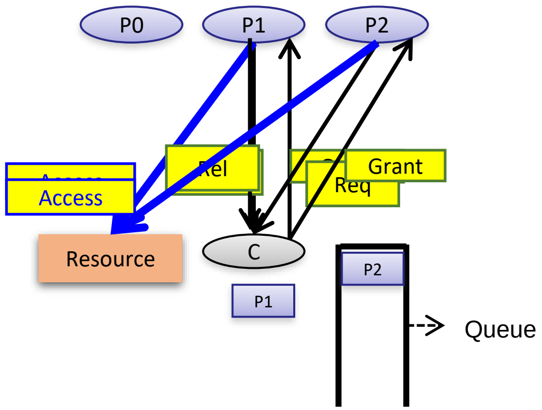
Permission-based Approaches

- There are two types of permission-based mutual exclusion algorithms
    1. Centralized Algorithms
    2. Decentralized Algorithms

- Let us study an example of each type of algorithms

A Centralized Algorithm

- One process is *elected* as a coordinator (**C**) for a shared resource
- Coordinator maintains a **`Queue`** of access requests
- Whenever a process wants to access the resource, it sends a request message to the coordinator to access the resource
- When the coordinator receives the request:
  - If no other process is currently accessing the resource, it grants the permission to the process by sending a “grant” message
  - If another process is accessing the resource, the coordinator queues the request, and does not reply to the request
- The process in action releases the exclusive access after accessing the resource
- Afterwards, the coordinator sends the “grant” message to the next process in the queue



Discussion

(+) Flexibility: Blocking versus non-blocking requests

- The coordinator can *block* the requesting process until the resource is free
- Or, the coordinator can send a "permission-denied" message back to the process
  - The process can poll the coordinator at a later time
  - Or, the coordinator queues the request (without blocking the requestor). Once the resource is released, the coordinator will send an explicit "grant" message to the process

(+) Simplicity: The algorithm guarantees mutual exclusion, and is simple to implement

(-) Fault-Tolerance Deficiency

- Centralized algorithm is vulnerable to a single-point of failure (at coordinator)
  - Processes cannot distinguish between dead coordinator and request blocking

(-) Performance Bottleneck

Next Class

- Mutual Exclusion
  - How to coordinate between processes that access the same resource?
- Election Algorithms
  - Here, a group of entities elect one entity as the coordinator for solving a problem

Today

- Last Session:
  - Logical Clocks
- Today's Session:
  - Distributed Mutual Exclusion
  - Election Algorithms
- Announcements:
  - Midterm exam is on Wednesday, March 9th during the class time
  - P2 is due on March 16th by midnight

Continuing Synchronization

Previous two lectures

- Time Synchronization
  - Physical Clock Synchronization (or, simply, Clock Synchronization)
    - Here, actual time on the computers are synchronized
  - Logical Clock Synchronization
    - Computers are synchronized based on the relative ordering of events
- Mutual Exclusion
  - How to coordinate between processes that access the same resource?
- Election Algorithms
  - Here, a group of entities elect one entity as the coordinator for solving a problem

Today's lecture

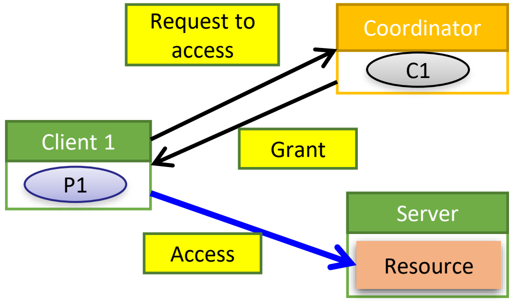
Overview

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
- Election Algorithms

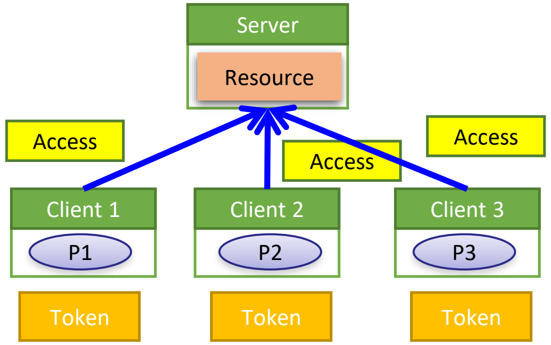
Types of Distributed Mutual Exclusion

- Mutual exclusion algorithms are classified into two categories:

1. Permission-based Approaches
    - A process, which wants to access a shared resource, requests the permission from *one or more* coordinators



2. Token-based Approaches
    - Each shared resource has a *token*
    - The token is circulated among all the processes
    - A process can access the resource if it has the token



Overview

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
  - Permission-based Approaches
  - Token-based Approaches
- Election Algorithms

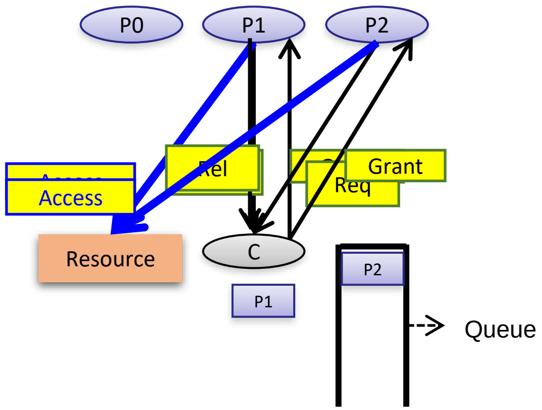
Permission-based Approaches

- There are two types of permission-based mutual exclusion algorithms
    1. Centralized Algorithms
    2. Decentralized Algorithms

- Let us study an example of each type of permission-based algorithms

A Centralized Algorithm

- One process is <u>*elected*</u> as a coordinator (`C`) for a shared resource
- Coordinator maintains a **`Queue`** of access requests
- Whenever a process wants to access the resource, it sends a request message to the coordinator to access the resource
- When the coordinator receives the request:
  - If no other process is currently accessing the resource, it grants the permission to the process by sending a “grant” message
  - If another process is accessing the resource, the coordinator queues the request, and does not reply to the request
- The process in action releases the exclusive access after accessing the resource
- Afterwards, the coordinator sends the “grant” message to the next process in the queue



Discussion

(+) Flexibility: Blocking versus non-blocking requests

- The coordinator can *block* the requesting process until the resource is free
- Or, the coordinator can send a “permission-denied” message back to the process
  - The process can poll the coordinator at a later time
  - Or, the coordinator queues the request (without blocking the requestor). Once the resource is released, the coordinator will send an explicit “grant” message to the process

(+) Simplicity: The algorithm guarantees mutual exclusion, and is simple to implement

(-) Fault-Tolerance Deficiency

- Centralized algorithm is vulnerable to a single-point of failure (at coordinator)
  - Processes cannot distinguish between dead coordinator and request blocking

(-) Performance Bottleneck

- In a large-scale system, single coordinator can be overwhelmed with requests

A Decentralized Algorithm

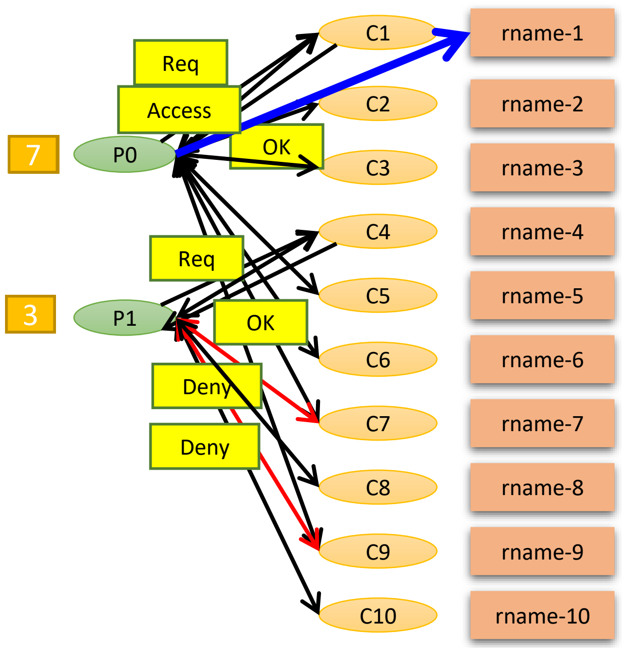
- To avoid the drawbacks of the centralized algorithm, Lin *et al.* (2004) advocated a decentralized mutual exclusion algorithm

- Assumptions:
  - Distributed processes are organized as a Distributed Hash Table (DHT) based system
  - Each resource is *replicated* **`n`** times
    - The **`i`**th replica of a resource **`rname`** is named as **`rname-i`**
  - Every replica has its *own* coordinator for controlling access
    - The coordinator for **`rname-i`** is determined by using a hash function

- Approach:
  - Whenever a process wants to access the resource, it will have to get *a majority vote* from **`m > n/2`** coordinators
  - If a coordinator does not want to vote for a process (because it has already voted for another process), it will send a "permission-denied" message to the process

A Decentralized Algorithm – An Example

- If **n=10** and **m=7**, then a process needs at-least **7** votes to access the resource



rname-x = xth replica of a resource rname | Cj = Coordinator j | Pi = Process i | n = Number of votes gained

Fault-tolerance in the Decentralized Algorithm

- This decentralized algorithm assumes that the coordinator recovers quickly from a failure

- However, the coordinator would have reset its state after recovery
  - Coordinator could have forgotten any vote it had given earlier

- Hence, the coordinator may incorrectly grant permission to a process
  - Mutual exclusion cannot be deterministically guaranteed
  - But, the algorithm still *probabilistically* guarantees mutual exclusion

Probabilistic Guarantees in the Decentralized Algorithm

- What is the minimum number of coordinators that should fail to violate mutual exclusion?
  - If f coordinators reset, correctness will be violated when a minority of non-faulty coordinators is left (i.e., $f \geq m - n/2$)

- Let the probability of violating mutual exclusion be P_v
  - Derivation of P_v
    - Let T be the lifetime of the coordinator
    - Let $p = \Delta t / T$ be the probability that a coordinator crashes during time-interval Δt
    - Let $P[k]$ be the probability that k out of m coordinators crash during the same interval

$$P[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

    - The mutual exclusion violation probability P_v can be computed as:

$$P_v = \sum_{k=m-n/2}^{n} P[k]$$

- In practice, this probability is typically very small
  - For T=3 hours, Δt=10 s, n=32, and m=0.75n : $P_v = 10^{-40}$

Teknik Informatika

Universitas Trunojoyo Madura

Quorum-Based Protocol

- This algorithm is an implementation of a more general protocol known as *quorum-based protocol*

- The quorum-based protocol can be implemented using a *voting scheme*, originally proposed by Thomas (1979) then generalized by Gifford (1979)

- Basic Idea:
    - Clients are required to *request and acquire* the permission of multiple servers before either *reading* or *writing* from or to a replicated data item
        - Rules on reads and writes should be established
        - Each replica is assigned a *version number*, which is incremented on each write

Quorum-Based Protocol

- Working Example:
  - Consider a distributed file system and suppose that a file is replicated on ***N*** servers

  - Write Rule:
    - A client must first contact ***N***/2 + 1 servers (a *majority*) before updating a file
    - Once majority votes are attained, the file is updated and its version number is incremented
      - This is pursued at the ***N***/2 + 1 replica sites

Quorum-Based Protocol

- Working Example:
  - Consider a distributed file system and suppose that a file is replicated on ***N*** servers

  - Read Rule:
    - A client must contact ***N***/2 + 1 servers, asking them to send their version numbers of its requested file
    - If all the version numbers are equal, this must be the most recent version of the file
      - This is because an attempt to update the remaining servers would fail since there are not enough of them
      - E.g., if ***N*** = 5 and a client receives 3 version numbers that are all equal to 8, it is impossible that the remaining 2 servers will have version 9
        - *Any successful update from version 8 to version 9 requires getting 3 servers to agree on it, not just 2*

Quorum-Based Protocol

- Gifford's scheme generalizes Thomas’s one

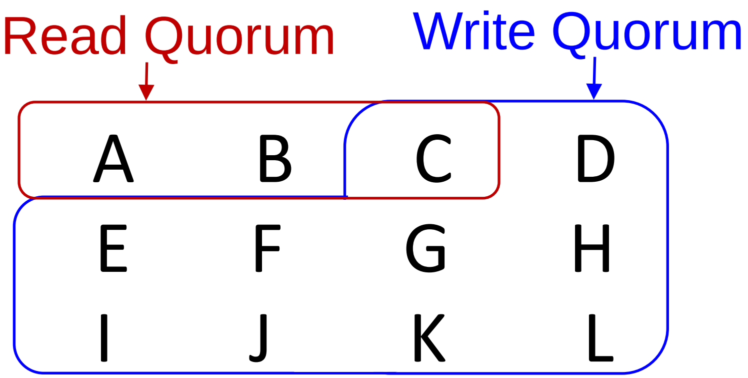
- **Gifford’s Scheme**:
  - Read Rule:
    - A client needs to assemble a *read quorum*, which is an arbitrary collection of any $\boldsymbol{N_R}$ servers, or more
  - Write Rule:
    - To modify a file, a *write quorum* of at least $\boldsymbol{N_W}$ servers is required

Quorum-Based Protocols

- The values of N_R and N_W are subject to the following two constraints:
  - Constraint 1 (or C1): $N_R + N_W > N$
  - Constraint 2 (or C2): $N_W > N/2$

- Claim:
  - C1 prevents read-write (RW) conflicts
  - C2 prevents write-write (WW) conflicts

Example 1



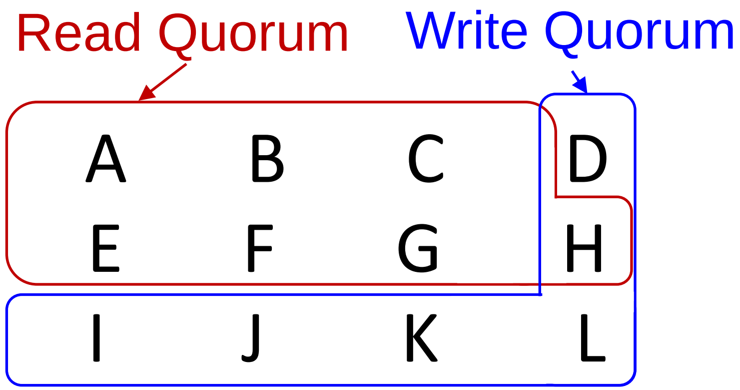
$N_R = 3$ and $N_W = 10$

C1: $N_R + N_W = 13 > N = 12$
➔ No RW conflicts

C2: $N_W > 12/2 = 6$
➔ No WW conflicts

- The most recent write quorum consisted of servers {C, D, …, L}
  - These servers got the new value and version number
- Any subsequent read quorum should contain at least 1 member in the write quorum {C, D, …, L}
  - When a client looks at this member's version, it will notice that it has the highest version number, hence, it will take it

Example 2



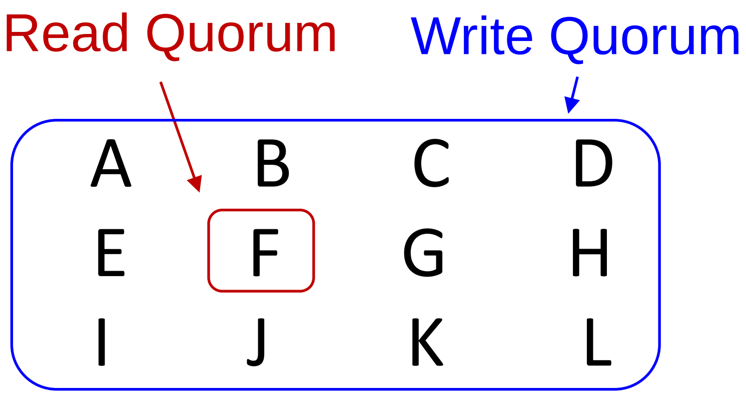
$N_R = 7$ and $N_W = 6$

C1: $N_R + N_W = 13 > N = 12$
➔ No RW conflicts

C2: $N_W \ngtr 12/2 = 6$
➔ WW conflicts may arise

- Why violating C2 causes WW conflicts?
  - If one client chooses {A, B, C, E, F, G} as its write set
  - And another client chooses {D, H, I, J, K, L} as its write set
  - The two updates will be accepted without detecting that they actually conflict, thus leading to an inconsistent view!

Example 3



$N_R = 1$ and $N_W = 12$

C1: $N_R + N_W = 13 > N = 12$
➔ No RW conflicts

C2: $N_W > 12/2 = 6$
➔ No WW conflicts

- A client can read a replicated file by finding any copy
  - Good read performance!
- A client needs to attain a write quorum on *all* copies
  - Slow write performance!
- This example demonstrates a scheme that is generally referred to as ROWA (or Read-Once, Write-All)

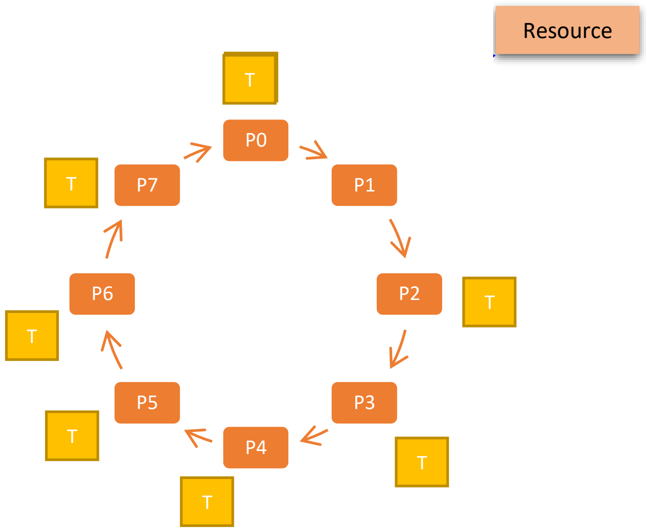
Overview

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
  - Permission-based Approaches
  - Token-based Approaches
- Election Algorithms

A Token Ring Algorithm

- With a token ring algorithm:
  - Each resource is associated with a *token*
  - The token is circulated among the processes
  - The process with the token can access the resource

- How is the token circulated among processes?
  - All processes form a logical ring where each process knows its next process
  - One process is given the *token* to access the resource
  - The process with the token has the right to access the resource
  - If the process has finished accessing the resource OR does not want to access the resource:
    - It passes the token to the next process in the ring



Discussion about Token Ring

- ✓ Token ring approach provides *deterministic* mutual exclusion
  - There is one token, and the resource cannot be accessed without a token

- ✓ Token ring approach avoids starvation
  - Each process will receive the token

- ✗ Token ring has a high-message overhead
  - When no processes need the resource, the token circulates at a high-speed

- ✗ If the token is lost, it must be re-generated
  - Detecting the loss of the token is difficult (especially if the amount of time between successive appearances of the token is unbounded)

- ✗ Dead processes must be purged from the ring
  - ACK based token delivery can assist in purging dead processes

Comparison of Mutual Exclusion Algorithms

Algorithm	Delay before a process can access the resource (in message times)	Number of messages required for a process to access and release the shared resource	Problems
Centralized	2	3	• Coordinator crashes
Decentralized	2mk	2mk + m; k=1,2,…	• Large number of messages
Token Ring	0 to (n-1)	1 to n	• Token may be lost • Ring can cease to exist since processes crash

- Assume that:

  n = Number of processes in the distributed system

  For the Decentralized algorithm:

    m = minimum number of coordinators who have to agree for a process to access a resource

    k = average number of requests made by the process to a coordinator to request for a vote

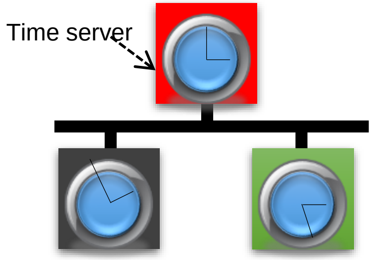
Overview

- Time Synchronization
  - Clock Synchronization
  - Logical Clock Synchronization
- Mutual Exclusion
  - Permission-based Approaches
  - Token-based Approaches
- Election Algorithms

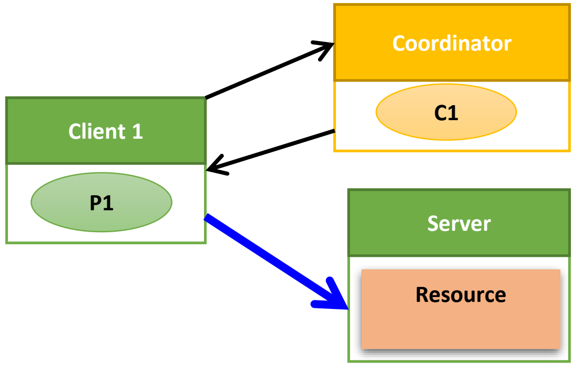
Election in Distributed Systems

- Many distributed algorithms require one process to act as a coordinator
    - Typically, it does not matter which process is elected as the coordinator

Home Node Selection in Naming



Berkeley Clock Synchronization Algorithm



A Centralized Mutual Exclusion Algorithm

The Election Process In a Nutshell

- We assume that *any* process $\mathbf{P_i}$ can initiate the election algorithm to elect a new coordinator
- At the end of the election algorithm, the elected coordinator should be unique
- Every process *may* know the process ID of every other process, *but it does not know which processes have crashed*
- Generally, we require that the coordinator is the process with the largest process ID
  - The idea can be extended to elect the *best* coordinator
    - Example: Election of a coordinator with the least computational load
      - If the computational load of process $\mathbf{P_i}$ denoted by $\mathbf{load_i}$, then the coordinator will be the process with the highest $\mathbf{1/load_i}$. Ties are broken by sorting process ID.

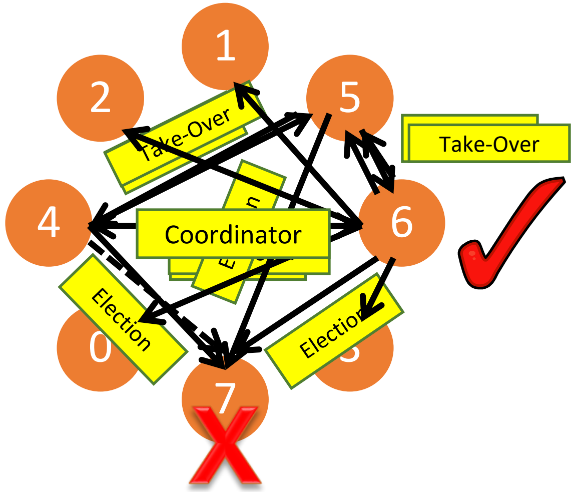
Election Algorithms

- Let us study two election algorithms:
  1. Bully Algorithm
  2. Ring Algorithm

1. Bully Algorithm

- A process (say, P_i) initiates the election algorithm when it notices that the existing coordinator is not responding

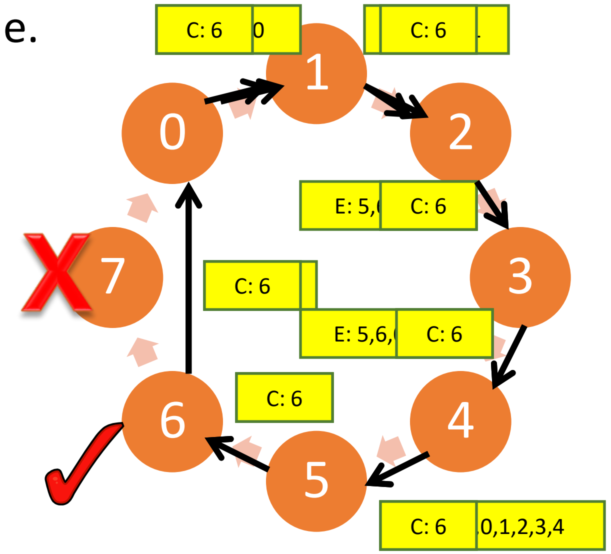
- Process P_i calls for an election as follows:
  1. P_i sends an "Election" message to all processes with higher process IDs
  2. When process P_j with $j>i$ receives the message, it responds with a "Take-over" message. P_i no more contests in the election
     i. Process P_j re-initiates another call for election. Steps 1 and 2 continue
  3. If no one responds, P_i wins the election. P_i sends "Coordinator" message to every process



2. Ring Algorithm

- This algorithm is generally used in a ring topology

- When a process P_i detects that the coordinator has crashed, it initiates the election algorithm
  1. P_i builds an "Election" message (E), and sends it to its next node. It inserts its ID into the Election message
  2. When process P_j receives the message, it appends its ID and forwards the message
     i. If the next node has crashed, P_j finds the next alive node
  3. When the message gets back to P_i:
     i. P_i elects the process with the highest ID as coordinator
     ii. P_i changes the message type to a "Coordination" message (C) and triggers its circulation in the ring



Comparison of Election Algorithms

Algorithm	Number of Messages for Electing a Coordinator	Problems
Bully Algorithm	$O(n^2)$	• Large message overhead
Ring Algorithm	2n	• An overlay ring topology is necessary

- Assume that:

  n = Number of processes in the distributed system

Summary of Election Algorithms

- Election algorithms are used for choosing a *unique* process that will coordinate certain activities

- At the end of an election algorithm, all nodes should uniquely identify the coordinator

- We studied two algorithms for performing elections:
  - Bully algorithm
    - Processes communicate in a distributed manner to elect a coordinator
  - Ring algorithm
    - Processes in a ring topology circulate election messages to choose a coordinator

Kuliah Berikutnya

- *Message Passing Interface* (MPI)

Pertanyaan?