

# Basics of Distributed Systems

## Part I

莊裕澤

Yuh-Jzer Joung

Dept. of Information Management

National Taiwan University

# Outline

- ❑ **System Models**
- ❑ **Time and Logical Clock**
- ❑ **Synchronization & Coordination**
  - **Mutual Exclusion**
  - **Leader Election**

# System Models

# Types of Models

## □ Physical models

- types of computers and devices that constitute a system and their interconnectivity, without details of specific technologies

## □ Architectural models

- describe a system in terms of the computational and communication tasks performed by its computational elements

## □ Fundamental models

- an abstract perspective in order to describe solutions to individual issues faced by most distributed systems
  - interaction model
  - failure model
  - security model

# Physical Models

- ❑ A distributed system is simply an extensible set of computer nodes interconnected by a computer network for the required passing of messages
  - From a system of a few nodes to the Internet-scale
  - From a system of tiniest embedded devices (e.g., IoT) to a complex computational Grid.
  - From a system to a distributed system of systems

# Architectural Models : communication perspectives

<i>Communicating entities (what is communicating)</i>		<i>Communication paradigms (how they communicate)</i>		
<i>System-oriented entities</i>	<i>Problem- oriented entities</i>	<i>Interprocess communication</i>	<i>Remote invocation</i>	<i>Indirect communication</i>
Nodes	Objects	Message passing	Request- reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM
				(Distributed Shared Memory)

RPC: Remote Procedure Call

RMI: Remote Method Invocation

Source: [Distributed Systems: Concepts and Design 5th ed.](#), C. Coulouris et al., 5th ed., 2011.

Higher level的協定 如:  
定期或不定期的收到訂閱的訊息、  
RPC、  
Message queue: share memory、  
Tuple: 像是信箱? 不用指名誰收

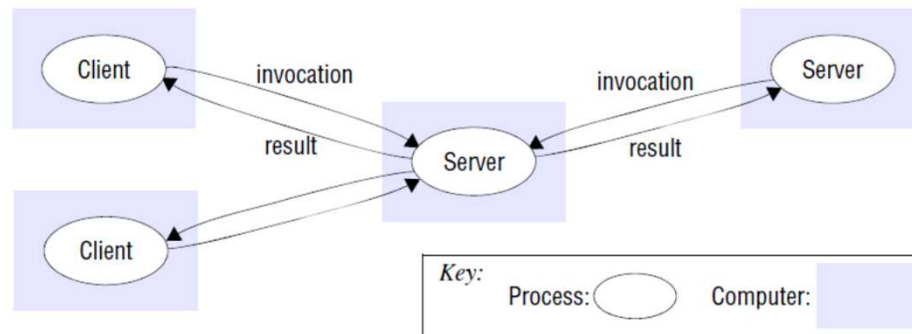
# Architectural Models: Based on Roles and Responsibilities

## ❑ Client-Server architecture

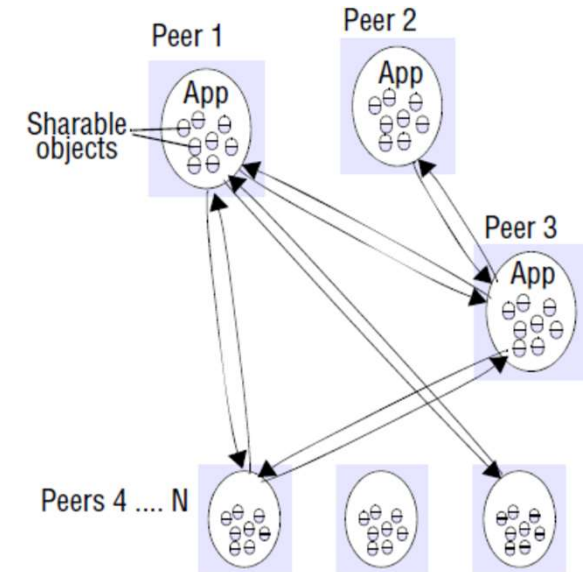
- Processes take on given roles (clients & servers) to perform a task
- E.g., WWW, most transaction systems

## ❑ Peer-to-Peer architecture

- Processes play similar roles without any distinction between clients and servers
- E.g., Napster, Gnutella, BitTorrent, BitCoin



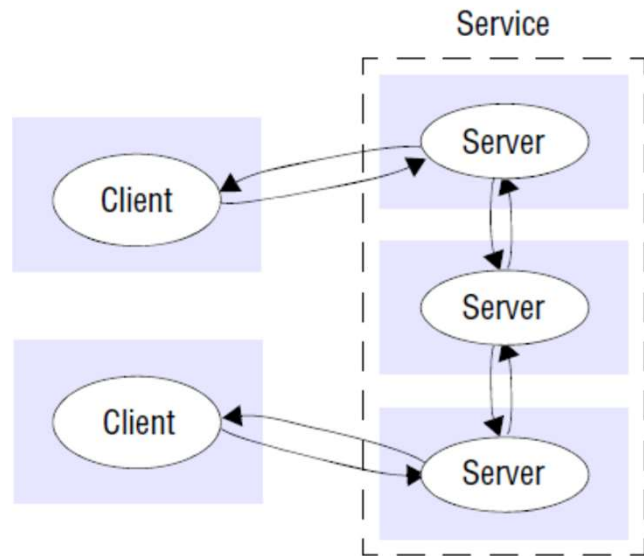
Client-Server



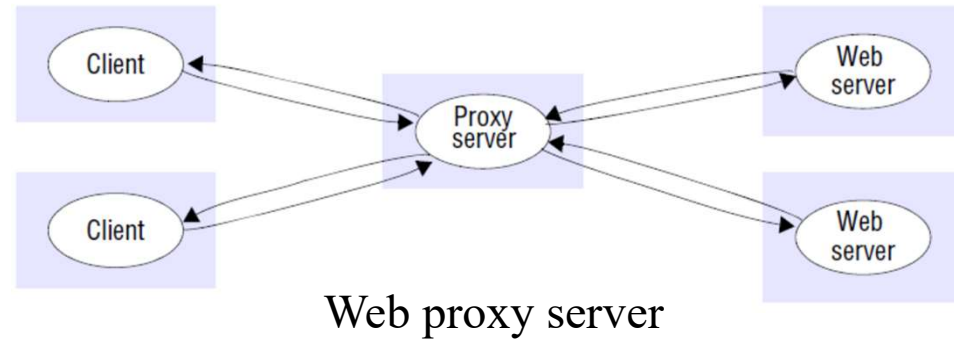
Peer-to-Peer

Source: [Distributed Systems: Concepts and Design 5th ed.](#), C. Coulouris et al., 5th ed., 2011.

# Placement of Entities in the Underlying Physical Distributed Infrastructure

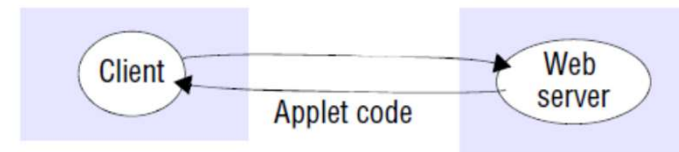


A service provided by multiple servers



Web proxy server

a) client request results in the downloading of applet code



b) client interacts with the applet



Similar concept: mobile agents

Note of potential security threats

Web applets



Source: Distributed Systems: Concepts and Design 5th ed., C. Coulouris et al., 5th ed., 2011.

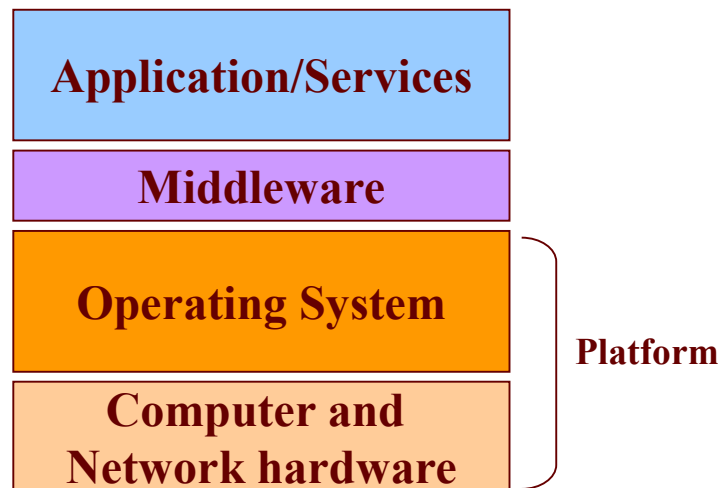
System Models, 請勿流傳



# Architectural Patterns: Layering

## □ Layering:

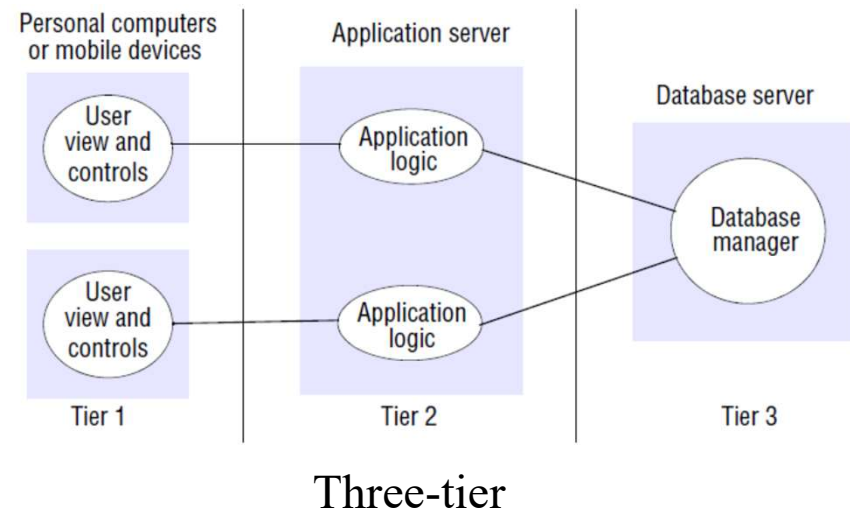
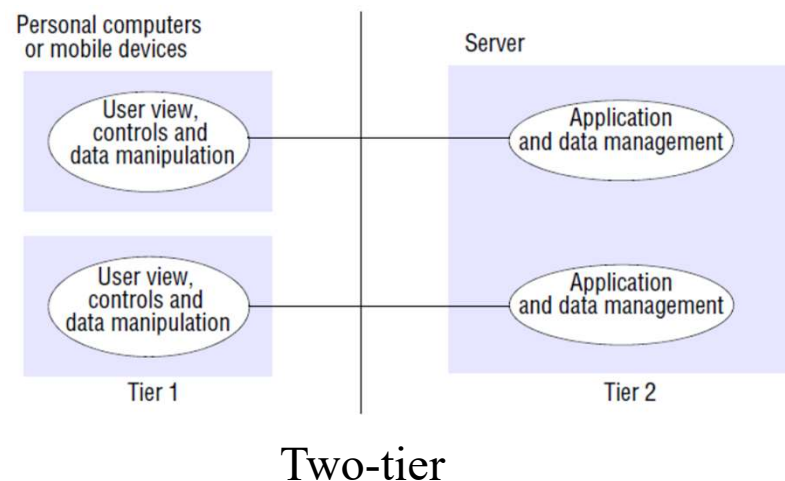
- a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below.
- A given layer offers a software **abstraction**, with higher layers being unaware of implementation details of the layer (and the layers below).



# Architectural Patterns: Tiered

## ❑ Tiered architecture:

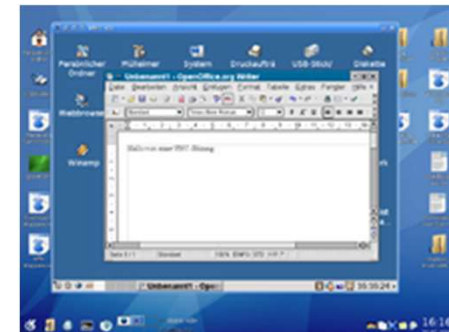
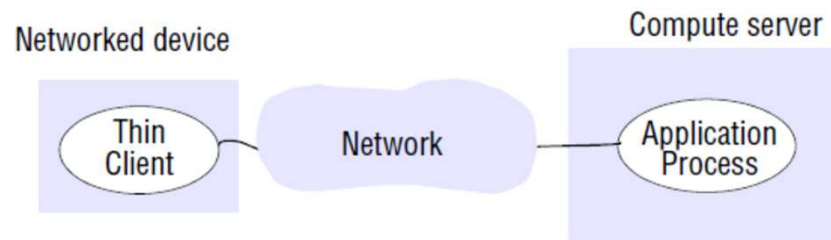
- organize functionality of a given layer and place this functionality into appropriate servers (and then on to physical nodes)
- Separation of the tiers depends on how you organize the functionality logically



Source: [Distributed Systems: Concepts and Design 5th ed.](#), C. Coulouris et al., 5th ed., 2011.

# Architectural Patterns: Thin Clients

- ❑ Thin Clients: moving complexity away from the end-user device towards services in the Internet.
  - Not suitable for applications require large data transmission, e.g., image processing.
  - Solution: Virtual Network Computing (VNC) [1998]
    - open platform independent graphical desktop sharing system designed to remotely control another computer.
    - most versions of Linux have a pre-installed VNC server; Mac OS, too.



# RDP vs VNC

## ❑ VNC (Virtual Network Computing)

- open protocol for interacting with the desktop of a remote computer via the Remote Frame Buffer protocol (RFB) (default port 5900)
- connects a remote user to a computer by sharing its screen, keyboard and mouse (so one user's move on the mouse is seen by the others, useful for technical support)
- basically “dumb” in the sense that it functions by sending the actual images across the network
- Some VNCs may not encrypt image data, so better used via VPN

## ❑ RDP (Remote Desktop Protocol)

- proprietary protocol built by Microsoft
- logs in a remote user to a computer by effectively creating a real desktop session on the computer
- RDP is aware of controls, fonts, and other similar graphical primitives; so the client side may infer part of the desktop layout to avoid the need of sending the corresponding image by the server
- Default port 3389
- RDP uses RSA Security's RC4 cipher, a stream cipher designed to efficiently encrypt small amounts of data.

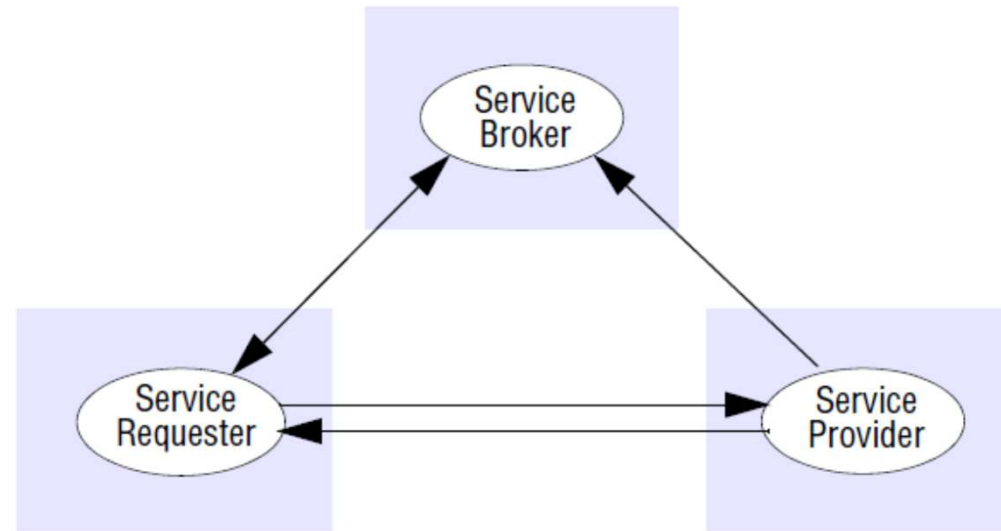
REALVNC



Ultra VNC



# Architectural Patterns: Brokerage



The web service architectural pattern

Goal: support **interoperability** in potentially complex distributed infrastructures

# Fundamental Models

## □ Three aspects of distributed systems to capture

### ■ Interaction

- synchronous vs. asynchronous
- message passing vs. shared memory

### ■ Failure

- node failures
- link failures
- Byzantine failures

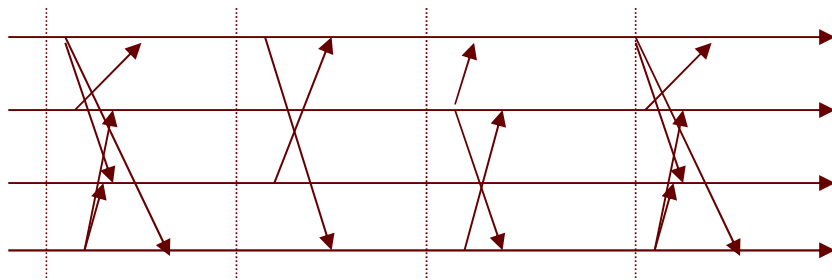
### ■ Security

# Interaction Models

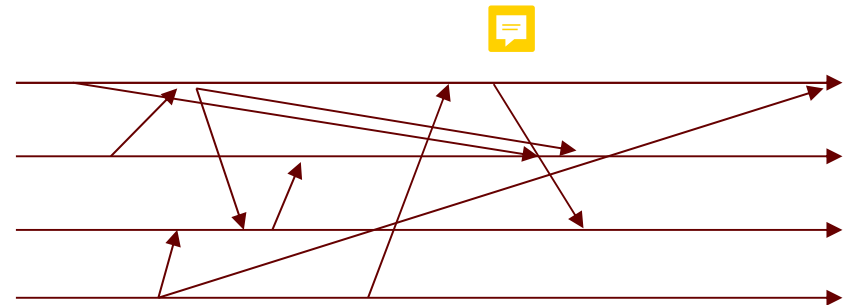
- ❑ A distributed system composed of multiple processes whose behavior and state can be described by **a distributed algorithm** --  
-- steps of taken by each process, including internal actions and send/receive messages (or shared memory access) between them.
- ❑ Whether or not to use any assumption on the time bound of execution speed and message transmission distinguishes two types of models:
  - **Synchronous**
  - **Asynchronous**

# Synchronous vs. Asynchronous

- ❑ **Asynchronous Systems:** make no assumptions about process execution speeds and/or message delivery delays.
- ❑ **Synchronous Systems:** Timeouts and other time-based protocol techniques are possible only when a system is synchronous.
  - time to execute each step of a process has known lower and upper bound.
  - message transmission time is bounded.
  - local clock whose drift rate from real time is bounded.




Synchronous



Asynchronous

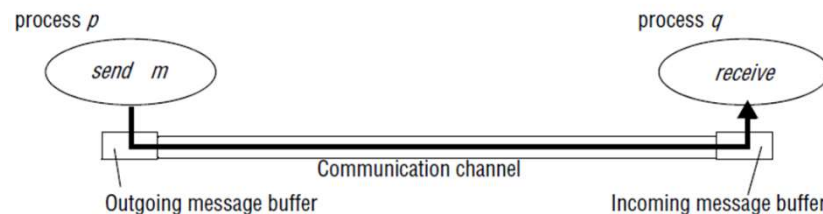


# Distributed Systems vs. Parallel Systems

- ❑ Loosely  coupled vs. tightly coupled
- ❑ Asynchrony vs. Synchrony
- ❑ Coarse-grained parallelism vs. fine-grained parallelism
- ❑ Concurrency vs. Parallelism

# Types of Failures (non-time-critical)

	<i>Class of failure</i>	<i>Affects</i>	<i>Description</i>
benign failures	Fail-stop	Process	Process halts and remains halted. Other processes may <b>detect</b> this state.
	Crash	Process	Process halts and remains halted. Other processes may <b>not be able to detect</b> this state.
	Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
	Send-omission	Process	A process completes a <i>send</i> operation but the message is not put in its outgoing message buffer.
	Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
	Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times or commit omissions; a process may stop or take an incorrect step.



Source: [Distributed Systems: Concepts and Design 5th ed.](#), C. Coulouris et al., 5th ed., 2011.

# Types of Failures (time-critical)

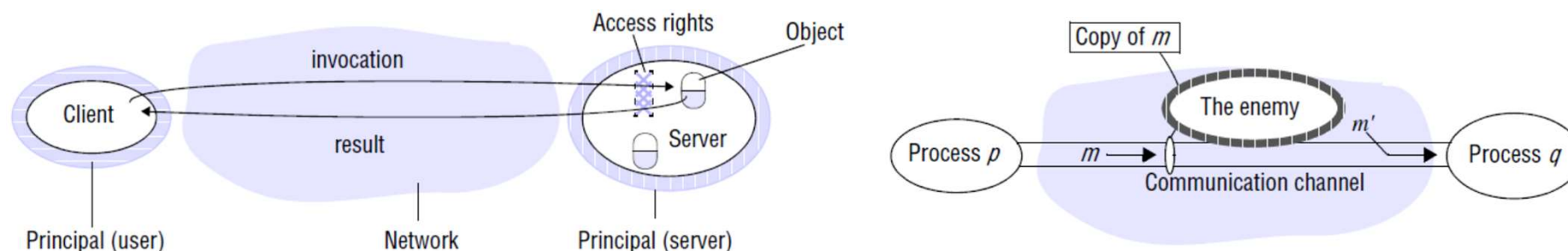
<i>Class of failure</i>	<i>Affects</i>	<i>Description</i>
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Source: [Distributed Systems: Concepts and Design 5th ed.](#), C. Coulouris et al., 5th ed., 2011.

Note:

- Timing failures are applicable to synchronous distributed systems or real-time systems.
- They are not applicable to asynchronous distributed systems as there is no assumption of any time constraint in process execution or message delivery, nor is there any time assumption in clock.

# Security Models



Source: [Distributed Systems: Concepts and Design 5th ed.](#), C. Coulouris et al., 5th ed., 2011.

The security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

Common defending techniques:

- Cryptography and shared secrets
- Authentication
- Secure channels

However, they cannot protect from “Denial of Service” or “Mobile code” attacks  
There are also non-technical **management** issues (ISO 27000 series)



# Time & Logical Clock

# Time

- ❑ Time is important in computer systems
  - Every file is stamped with the time it was created, modified, and accessed.
  - Every email, transaction, ... are also timestamped.
  - Setting timeouts and measuring latencies
- ❑ Sometimes we need precise physical time and sometimes we only need relative time.

# Synchronizing Physical Time

## Observations:

- ❑ In some systems (e.g., real-time systems) actual time are important, and we typically equip every computer host with one physical clock.
- ❑ Computer clocks are unlikely to tick at the same rate, whether or not they are of the ‘same’ physical construction.
  - E.g., a quartz crystal clock has a drift rate of  $10^{-6}$  (ordinary), or  $10^{-7}$  to  $10^{-8}$  (high precision).
  - C.f. an atomic clock has a drift rate of  $10^{-13}$ .

## Questions:

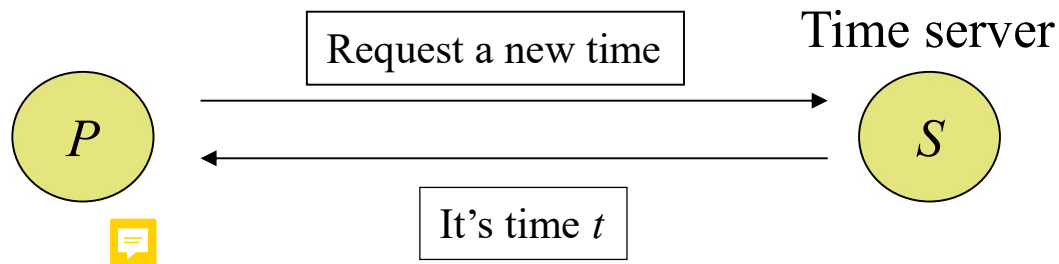
1. How do we synchronize computer clocks with real-world clocks?
2. How do we synchronize computer clocks themselves?

# Compensation for Clock Drift

- ❑ A computer clock usually can be adjusted forward but not backward.
  - Typical example: Y2K problem.
- ❑ Common terminology:
  - **Skew (offset)**: the instantaneous difference between (the readings of) two clocks.
  - **Drift rate**: the difference between the clock and a nominal perfect reference clock per unit of time.
- ❑ Linear adjustment:
  - Let  $C$  be the software reading of a hardware clock  $H$ . Then the operating system usually produces  $C$  in terms of  $H$  by the following:  $C(t) = \alpha H(t) + \beta$



# Cristian's Algorithm



When  $P$  receives the message, it should set its time to  $t + T_{trans}$ , where  $T_{trans}$  is the time to transmit the message.

$T_{trans} \approx T_{round}/2$ , where  $T_{round}$  is the round-trip time

## Accuracy.

Let  $min$  be the minimum time to transmit a message one-way.

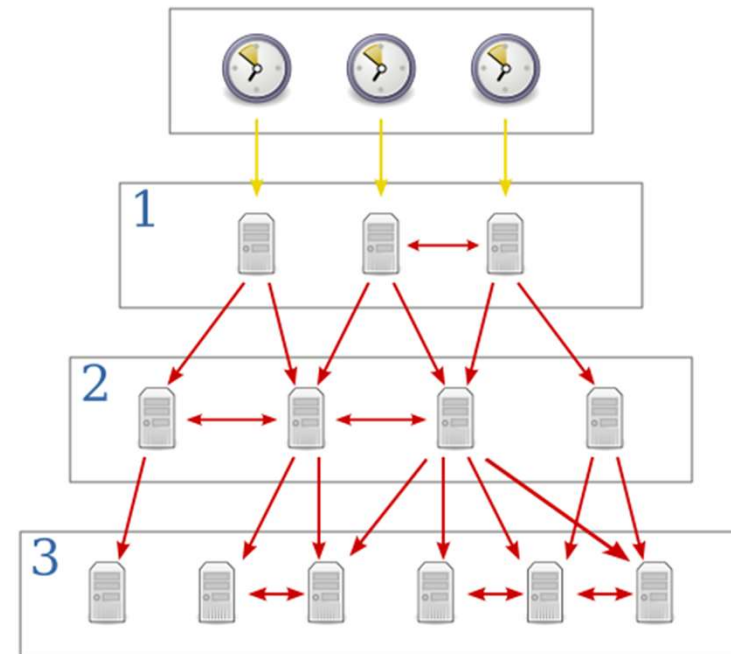
Then  $P$  could receive  $S$ 's message any time between

$$[t + min, \quad t + T_{round} - min]$$

So accuracy is  $\pm(T_{round}/2 - min)$

# The Network Time Protocol (NTP)

- ❑ Provide a service enabling clients across the Internet to be synchronized accurately to UTC, despite the large and variable message delays encountered in Internet communication.
- ❑ The NTP servers are connected in a logical hierarchy, where servers in level  $n$  are synchronized directly to those in level  $n-1$  (which have a higher accuracy). The logical hierarchy can be reconfigured as servers become unreachable or failed.



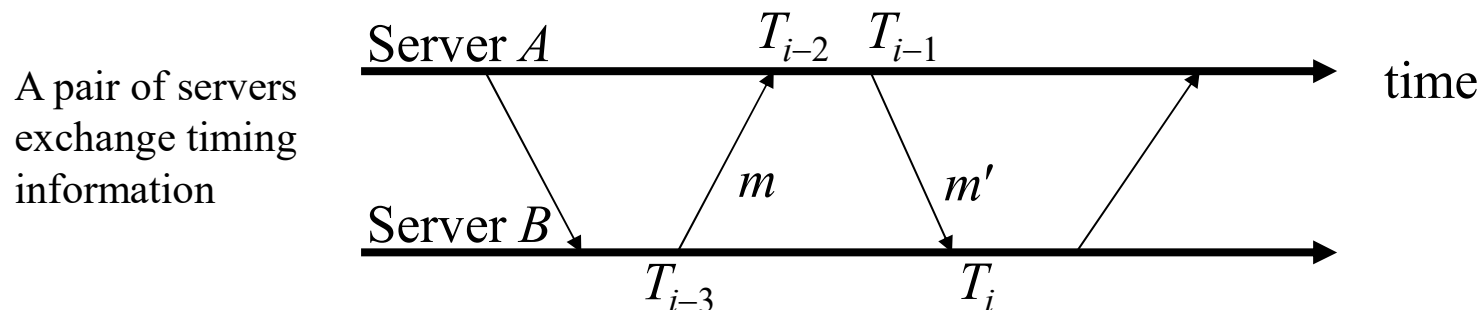
Source: wiki

# Three Modes of Synchronization in NTP

- ❑ NTP servers synchronize with one another in one of three modes (in the order of increasing accuracy):
  - Multicast on high speed local LANs
  - Procedure call mode (*a la* Cristian's algorithm)
  - Symmetric mode (for achieving highest accuracy).



# Symmetric Mode



Assume: message  $m$  takes  $t$  to transfer,  $m'$  takes  $t'$  to transfer.  
 Offset between A's clock and B's clock is  $o$ ; i.e.,  $A(t) = B(t) + o$

Then,  $T_{i-2} = T_{i-3} + t + o$  and  $T_i = T_{i-1} - o + t'$

Assuming that  $t \approx t'$ , then the offset  $o$  can be estimated as follows:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i) / 2$$

Since  $T_{i-2} - T_{i-3} + T_i - T_{i-1} = t + t'$  (let's say,  $t + t'$  equal to  $d_i$ )

Then  $o = o_i + (t' - t)/2$

Given that  $t', t \geq 0$ , the accuracy of the estimate of  $o_i$  is:

$$o_i - d_i / 2 \leq o \leq o_i + d_i / 2$$

## Symmetric Mode (contd.)

- ❑ The eight most recent pairs  $\langle o_i, d_i \rangle$  are retained; the value of  $o_i$  that corresponds to the minimum  $d_i$  (the transmission time  $t + t'$ ) is chosen to estimate  $o$ .
- ❑ Timing messages are delivered using UDP.

# Logical Time

## Motivation

- ❑ Event ordering linked with the concept of **causality**.
  - Saying that event  $a$  “*happened before*” event  $b$  is the same as saying that event  $a$  could have affected the outcome of event  $b$
  - If events  $a$  and  $b$  happen on processes that do not exchange any data, their exact ordering is not important

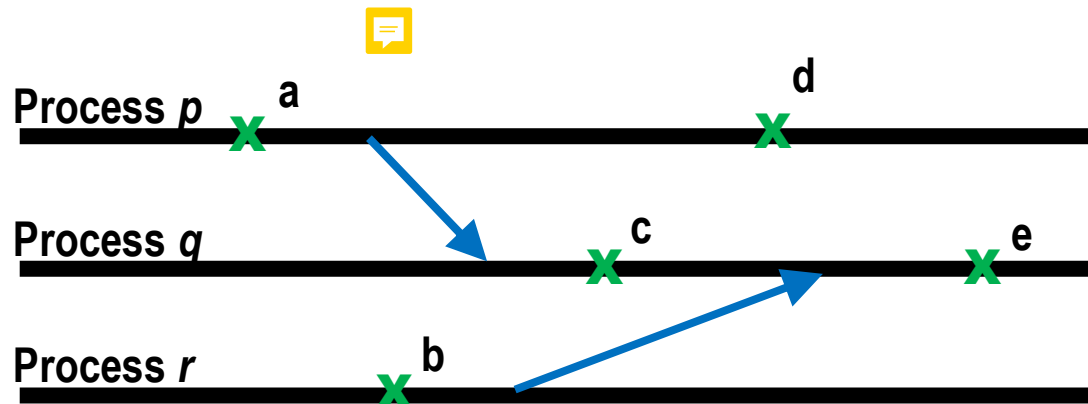
## Observation

- ❑ If two events occurred at the same process, then they occurred in the order in which the process observes them.
- ❑ Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

# Causal Ordering (*happened-before* Relation)

1. If process  $p$  executes  $x$  before  $y$ , then  $x \rightarrow y$ .
2. For any message  $m$ ,  $send(m) \rightarrow rcv(m)$ .
3. If  $x \rightarrow y$  and  $y \rightarrow z$ , then  $x \rightarrow z$ .

Two events  $a$  and  $b$  are said **concurrent** if neither  $a \rightarrow b$  nor  $b \rightarrow a$ .



# Logical Clocks (L. Lamport, 1978)

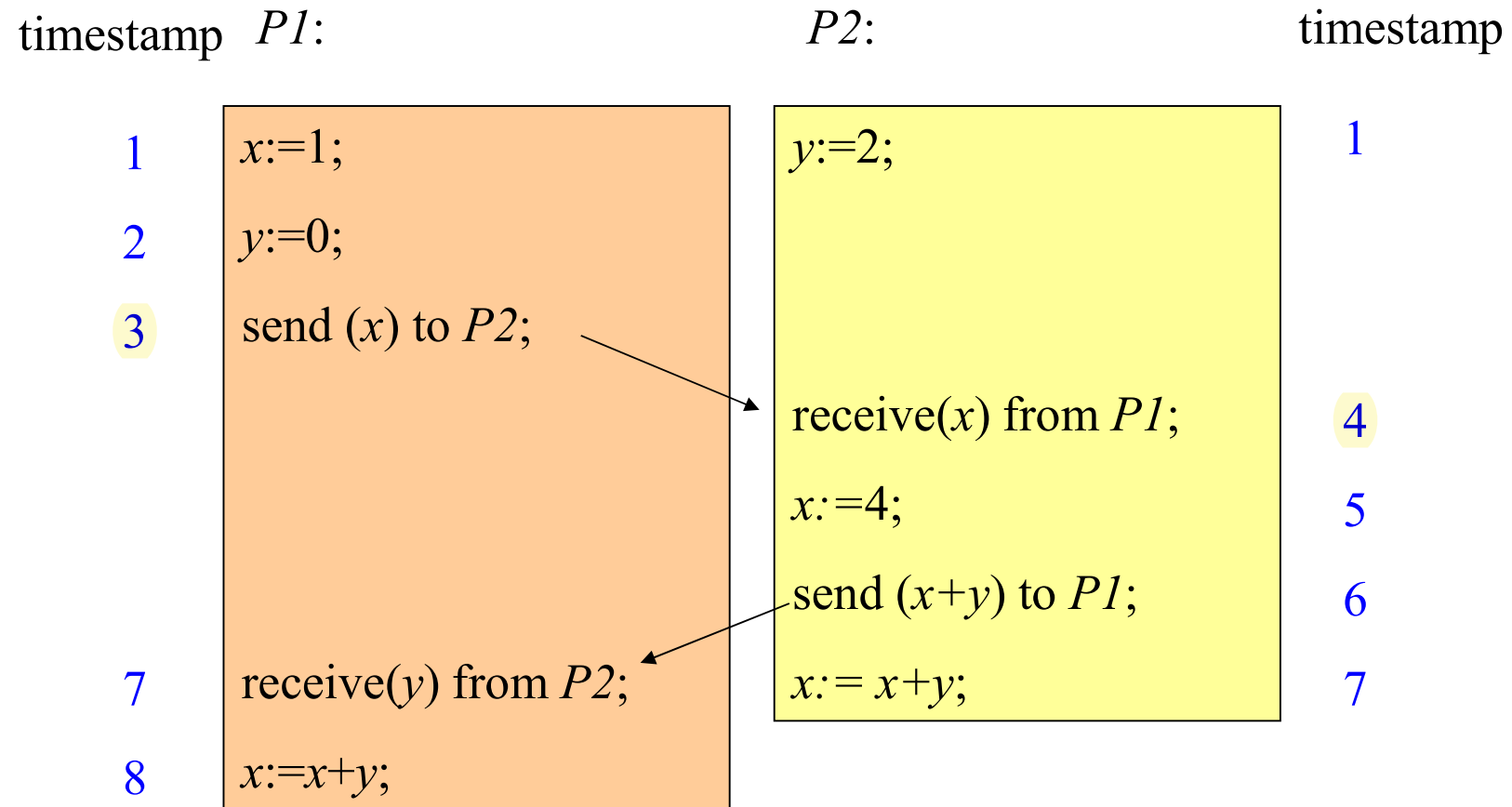
A **logical clock**  $C_p$  of a process  $p$  is a software counter that is used to timestamp events executed by  $p$  so that the *happened-before* relation is respected by the timestamps.

The rule for increasing the counter is as follows:

- LC1:  $C_p$  is incremented before each event issued at process  $p$ .
- LC2: When a process  $q$  sends a message  $m$  to  $p$ , it piggybacks on  $m$  the current value  $t$  of  $C_q$ ; on receiving  $m$ ,  $p$  advances its  $C_p$  to  $\max(t, C_p)$ .




# Illustration of Timestamps



# Reasoning about Timestamps

Consequence:

if  $a \rightarrow b$ , then  $C(a) < C(b)$

*Does  $C(a) < C(b)$  imply  $a \rightarrow b$  ?* 

# Total Ordering of Events

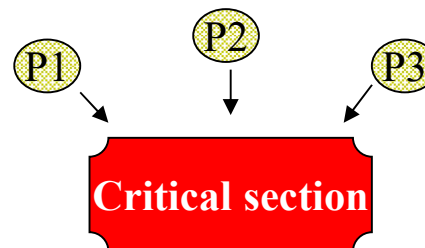
- ❑ Happened Before defines a partial ordering of events (arising from causal relationships).
- ❑ The partial ordering can be made total by additionally considering process ids.
- ❑ Suppose event  $a$  is issued by process  $p$ , and event  $b$  by process  $q$ . Then the total ordering  $\rightarrow_t$  can be defined as follows:

$$a \rightarrow_t b \text{ iff } C(a) < C(b) \text{ or } C(a) = C(b) \text{ and } ID(p) < ID(q).$$

- ❑ Using this method, we can assign a unique timestamp to each event in a distributed system to provide a total ordering of all events

# Applications of Logical Timestamps

- ❑ Logical timestamps allow events to be totally ordered that respect causality relation
  - When processes are competing for a resource in a purely decentralized manner, one cannot forever win the competition if another has consulted it.
    - mutual exclusion
  - to determine ordering of distributed updates among replicas by different processes



# Vector Timestamps

- ❑ Each process  $P_i$  maintains a vector of clocks  $VT_i$  such that  $VT_i[k]$  represents a count of events that have occurred at  $P_k$  and that are known at  $P_i$ .
- ❑ The vector is updated as follows:
  1. All processes  $P_i$  initializes its  $VT_i$  to zeros.
  2. When  $P_i$  generates a new event, it increments  $VT_i[i]$  by 1;  $VT_i$  is assigned as the timestamp of the event. Message-sending events are timestamped.
  3. When  $P_j$  receives a message with timestamp  $vt$ , it updates its vector clock as follows:

$$VT_i[k] := \max(VT_i[k], vt[k])$$

# Illustration of Vector Timestamps



timestamp  $P1$ :

$\langle 1,0 \rangle$

$\langle 2,0 \rangle$

$\langle 3,0 \rangle$

$\langle 4,4 \rangle$

$\langle 5,4 \rangle$

$x:=1;$

$y:=0;$

send ( $x$ ) to  $P2$ ;

receive( $y$ ) from  $P2$ ;

$x:=x+y;$

$P2$ :

$y:=2;$

receive( $x$ ) from  $P1$ ;

$x:=4;$

send ( $x+y$ ) to  $P2$ ;

$x:=x+y;$

timestamp

$\langle 0,1 \rangle$

$\langle 3,2 \rangle$

$\langle 3,3 \rangle$

$\langle 3,4 \rangle$

$\langle 3,5 \rangle$

# Reasoning about Vector Timestamps

Partial orders ' $\leq$ ' and ' $<$ ' on two vector timestamps  $u$ ,  $v$  are defined as follows:

$u \leq v$  iff  $u[k] \leq v[k]$  for all  $k$ 's, and  $u < v$  iff  $u \leq v$  and  $u \neq v$ .

Property:  $e$  happened-before  $f$  if, and only if,  $vt(e) < vt(f)$ .

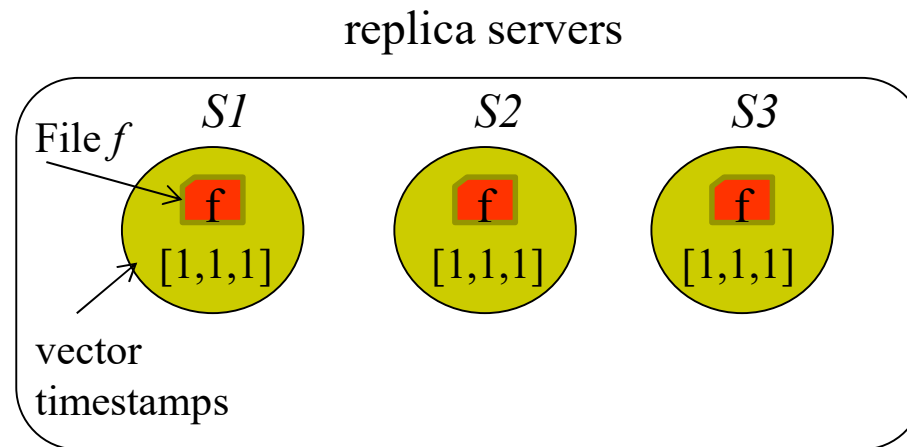
# Logical vs. Vector Timestamps

- ❑ Lamport's logical timestamp cannot infer happened-before relation from timestamps
  - if  $a \rightarrow b$ , then  $C(a) < C(b)$
  - But  $C(a) < C(b)$  does not imply  $a \rightarrow b$
- ❑ **Vector Timestamps** solve the problem
  - $a \rightarrow b$  iff  $VC(a) < VC(b)$
- ❑ Applications of vector timestamps:
  - Allow one to infer whether two events are concurrent or happened-before dependent



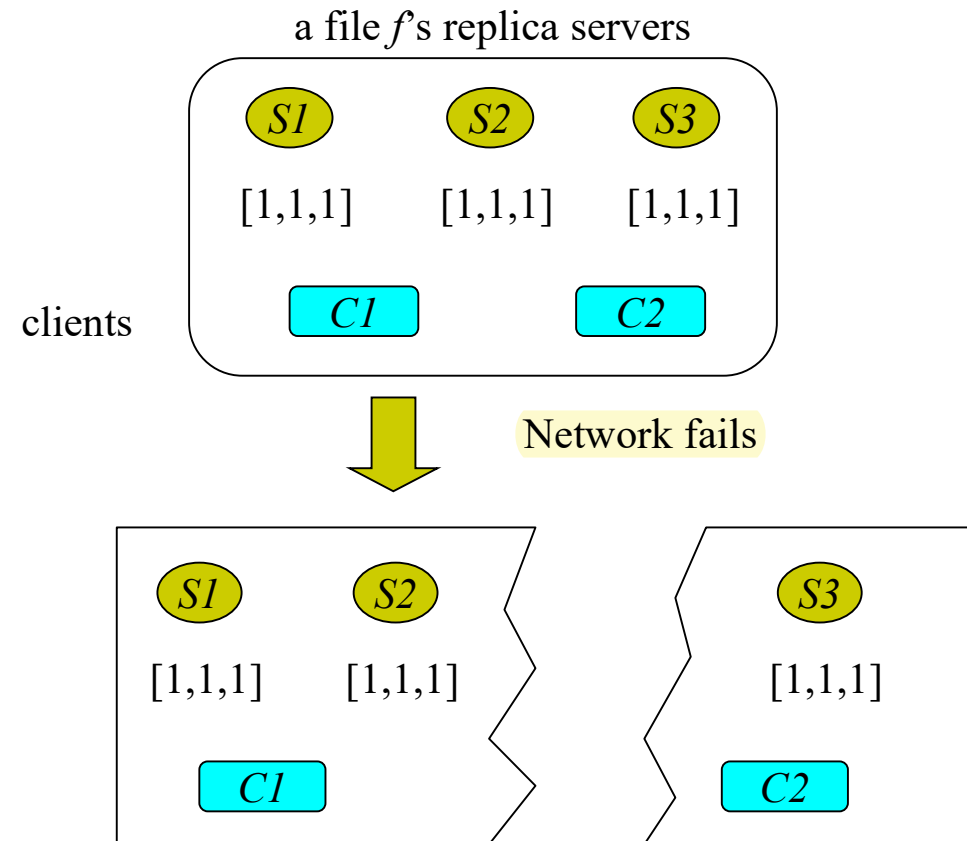
# Example: Replication in File Systems

- ❑ A file volume is stored in a set of replica servers
- ❑ A client wishing to open a file can access some subset of the replica servers (usually a “quorum”).
- ❑ To detect conflict, each file is attached a vector timestamp estimating the number of modifications performed on the version of the file that is held at each server.

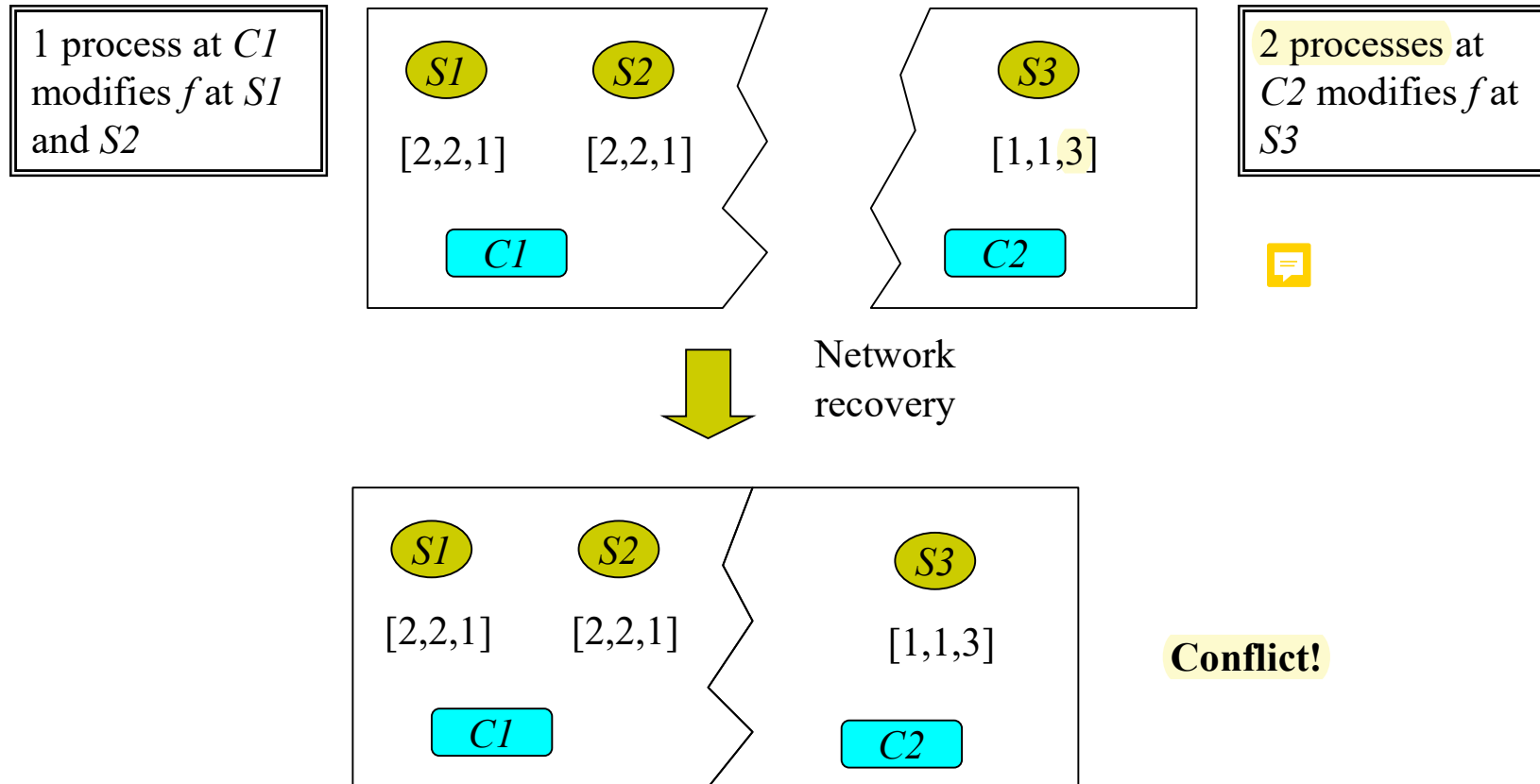


note: concept used by a file system called Coda developed at CMU in the early 90s.

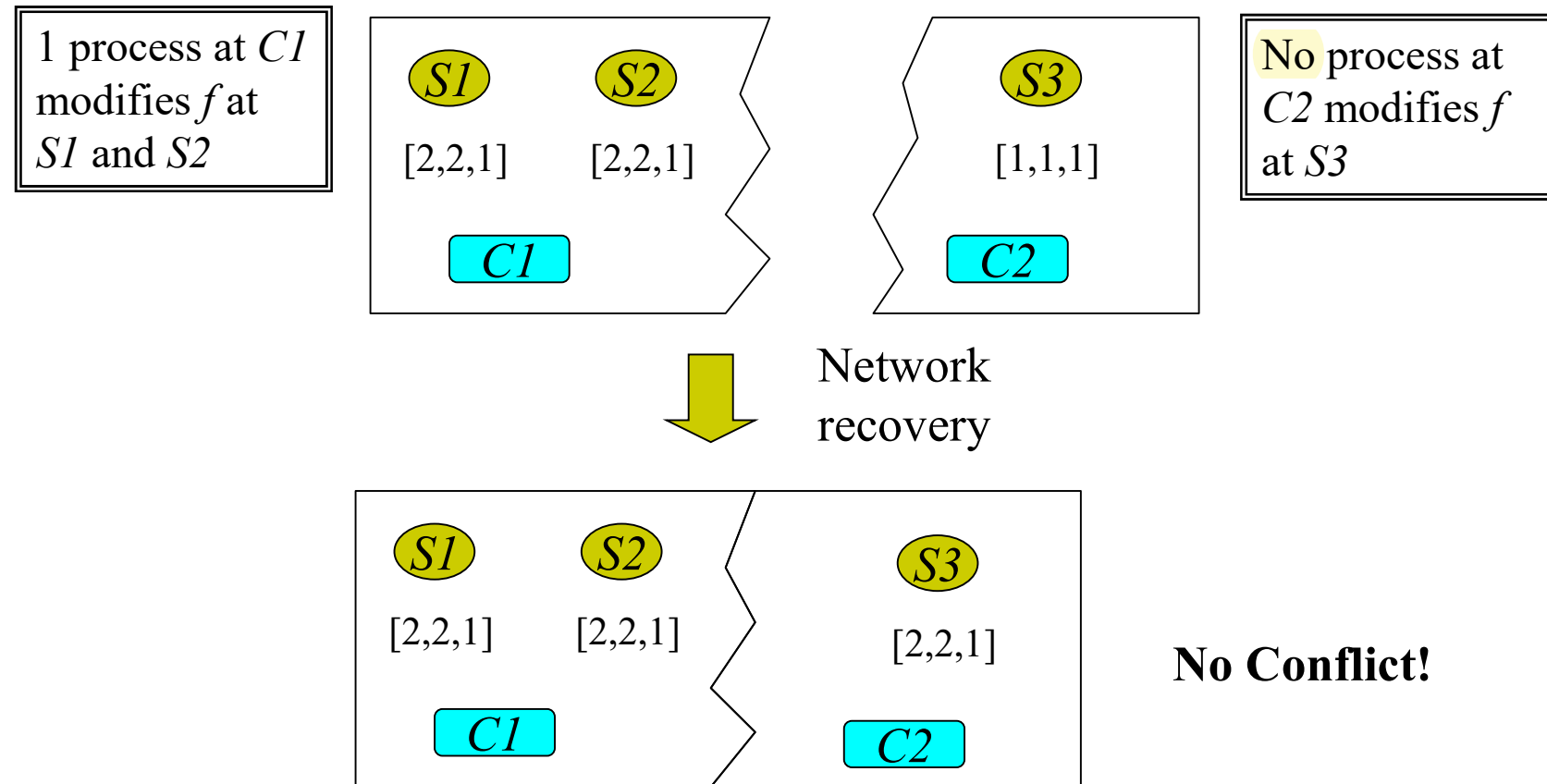
# Replication in Coda (cont.)



# Replication in Coda (cont.)



# Example: Replication in Coda (cont.)



# Problems about Vector Clocks

- ❑ Drawback: scalability
  - Size depends on the number of processes  $N$
- ❑ However, if we are to be able to tell whether or not two events are concurrent by inspecting their timestamps, then the size of a timestamp in proportion to  $N$  is unavoidable [Charron-Bost, 1991].
- ❑ Some techniques can be used to store and transmitting smaller amount of data, at the expense of the processing required to reconstruct complete vectors [Raynal and Singhal, 1996].
- ❑ Vector timestamps require an agreed notion of system membership

# Synchronization & Coordination

- ☐ Mutual Exclusion
- ☐ Leader Election

# Coordination in Distributed Systems

## □ Why needed to study?

- for resource sharing: concurrent updates of
  - records in a database (record locking)
  - files (file locks in stateless file servers)
  - a shared bulletin board
- to dynamically re-assign the role of master
  - choose a primary server after crash
  - choose a coordinator after network reconfiguration
- to agree on actions: whether to
  - commit/abort database transaction
  - agree on readings from a group of sensors

# Why Difficult?

- ❑ Centralized solutions not appropriate
  - communications bottleneck
- ❑ Fixed master-slave arrangements not appropriate
  - process crashes
- ❑ Varying network topologies
  - ring, tree, arbitrary; connectivity problems
- ❑ Failures must be tolerated if possible
  - link failures
  - process crashes
- ❑ Impossibility results
  - Some problems are even impossible to solve



# Typical Coordination Problems

## ❑ Mutual exclusion

- distributed form of the **critical section** problem

## ❑ Leader elections

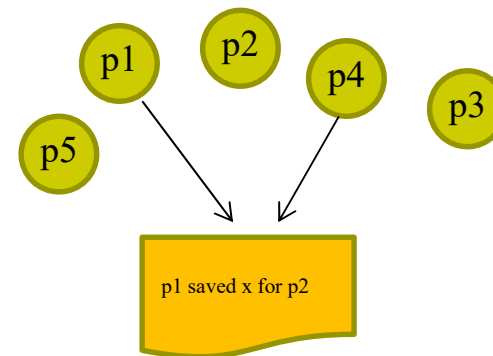
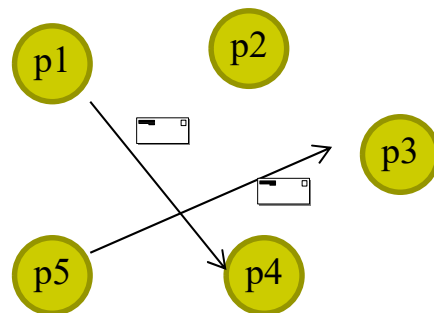
- after crash failure has occurred
- after network reconfiguration

## ❑ Consensus (to be discussed in a separate lecture)

- similar to coordinated attack
- variants depending on type of failure, network, etc

# Two Computing Paradigms

- ❑ **Message Passing** (default model of the class)
  - assumed by most distributed systems where system components may be physically apart
  - communication primitives
    - point-to-point (default)
    - multicast
    - broadcast
- ❑ **Shared Memory**
  - used when processes have a shared memory

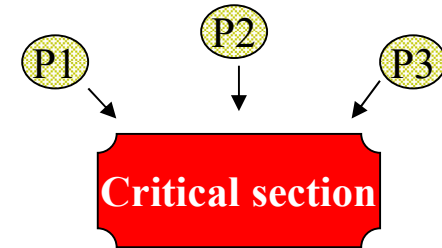


# Mutual Exclusion

# Mutual Exclusion (E. Dijkstra, 1965)

## □ The problem

- $N$  asynchronous processes (assuming no failure unless specified)
- guaranteed message delivery (reliable links)
- to execute **critical section (CS)**, each process calls:
  - request\_to\_access\_CS()
  - access\_CS()
  - exit\_CS()



## □ Requirements

- **Safety:** At most one process is in CS at any time.
- **Liveness:** Requests to enter and exit are eventually granted.
- (Optional, stronger) Requests to enter granted according to some order (e.g., causality).

# Safety vs. Liveness Properties

- ❑ A **safety** property describes a property that **always** holds; sometimes we put it in this way “*nothing ‘bad’ will happen*”.
- ❑ A **liveness** property describes a property that will **eventually** hold; sometimes we put it in this way “*something ‘good’ will eventually happen*”.



What are the following properties belong?

deadlock free, mutual exclusion, bounded delay

# Edsger W. Dijkstra (1930-2002)

- ❑ A pioneer in distributed computing
  - Many other contributions in algorithm design, formal specification and verification, program design, and (concurrent) programming languages
  - Some influential concepts/solutions
    - Dijkstra's algorithm for single-source shortest path problem
    - Mutual exclusion
    - Dining philosophers
    - self-stabilization
    - ...
- ❑ 1972 Turing Award
- ❑ Paper lists: see <http://www.cs.utexas.edu/~EWD/>



Source: wiki

# Mutual Exclusion in Shared Memory

- ❑ Dekker's algorithm
- ❑ Peterson's algorithm
- ❑ Lamport's bakery algorithm
- ❑ ...
- ❑ See *Algorithms for mutual exclusion*, M. Raynal, 1986.

# Lamport's Bakery Algorithm

## SHARED VAR

```
integer array choosing[1..N], num[1..N];  
/* initialized to 0 */
```

## PROTOCOL (for Process i):

```
/* choose a number */
```

```
choosing[i] := 1;
```

```
num[i] := max(num[1], ..., num[N]) + 1;
```

```
choosing[i] := 0;
```

```
for (j:=1; j ≤ N; j++) {
```

```
    /* wait if the process is currently choosing */
```

```
    while (choosing[j] ≠ 0) {}
```

```
    /* wait if a process has a number and comes ahead of me */
```

```
    while (num[j] ≠ 0 && ((num[j] < num[i]) || (num[j] == num[i] && (j < i)))) {}
```

```
}
```

```
ENTER CS
```

```
...
```

```
EXIT CS
```

```
num[i] := 0;
```

How do you prove its correctness?

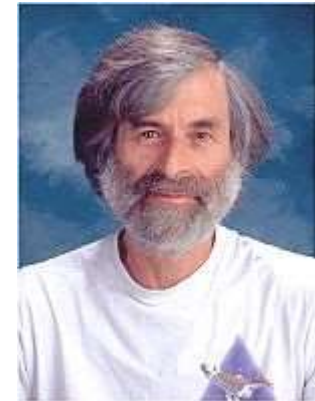
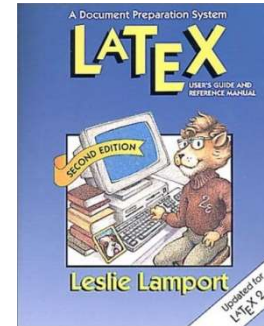
Does the algorithm depend on what value is returned by a read that overlaps a write?



# Leslie Lamport, 1941-

## □ Many influential papers in distributed computing

- Logical clocks
- Mutual exclusion
- The Byzantine Generals' Problem
- Paxos algorithm
- Temporal logic
- ...



source: wiki

## □ LaTeX

2013 Turing Award

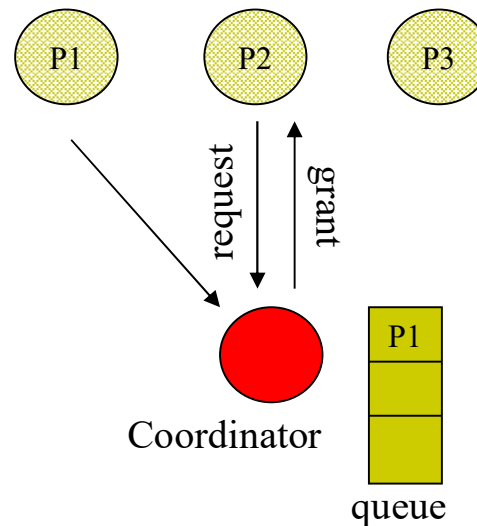
## □ List of papers (<http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html>)

# Some Message-Passing Solutions for Mutex

- ❑ Centralized
- ❑ Ricart and Agrawala's Distributed Algorithm
- ❑ Tree
- ❑ Quorum
- ❑ Token Ring
- ❑ ...

# A Centralized Solution

- ❑ Use a centralized coordinator to main a queue of requests.
- ❑ A process wishing to enter CS sends a request to the coordinator, and enters the CS when the coordinator grants its request.



理論 vs. 實務

## Drawback:

The coordinator becomes a bottleneck and single failure point.

# Classification of Mutex Algorithms

## ❑ Token-Based

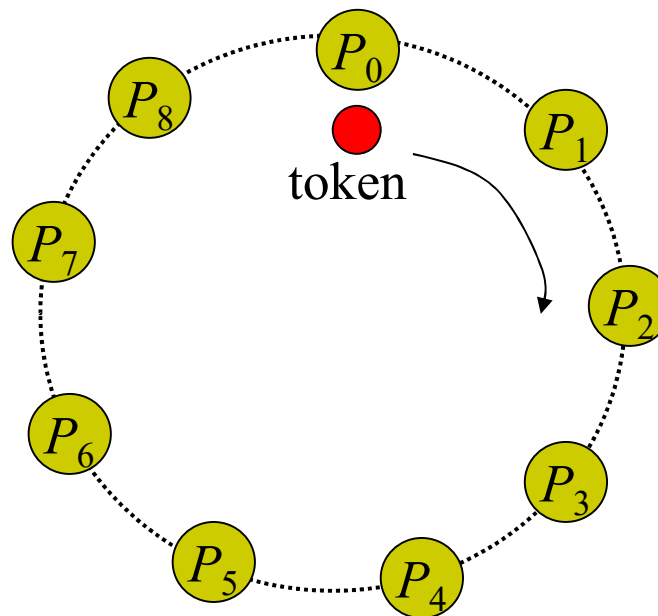
- Circulation of a unique token to control the access of CS
  - Require some mechanism to control the circulation of token
  - Require a logical structure over a physical network
    - Ring, tree, ...

## ❑ Permission-Based

- Processes request permissions from other processes to access CS
  - Should a process request permission from ALL other processes, or just some of them?
    - Quorums/Voting

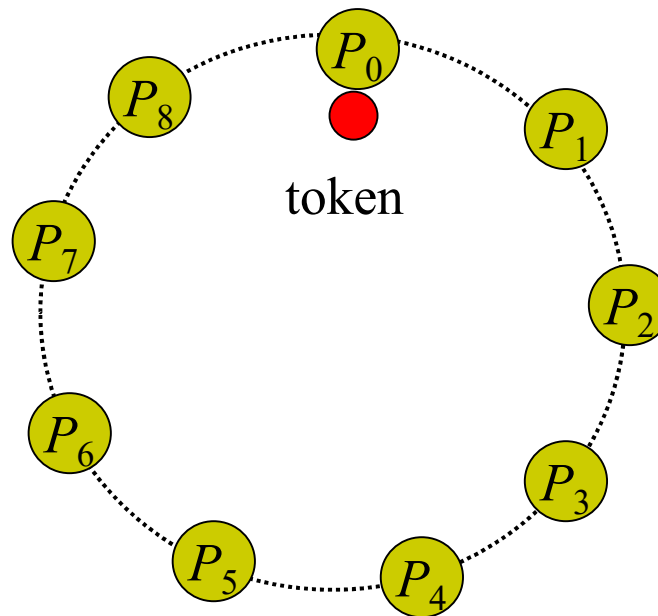
# Token Rings

- ❑ It is trivial to solve mutual exclusion over a ring---by using a unique token.
- ❑ For ordinary network, a logical ring can be constructed to apply the algorithm

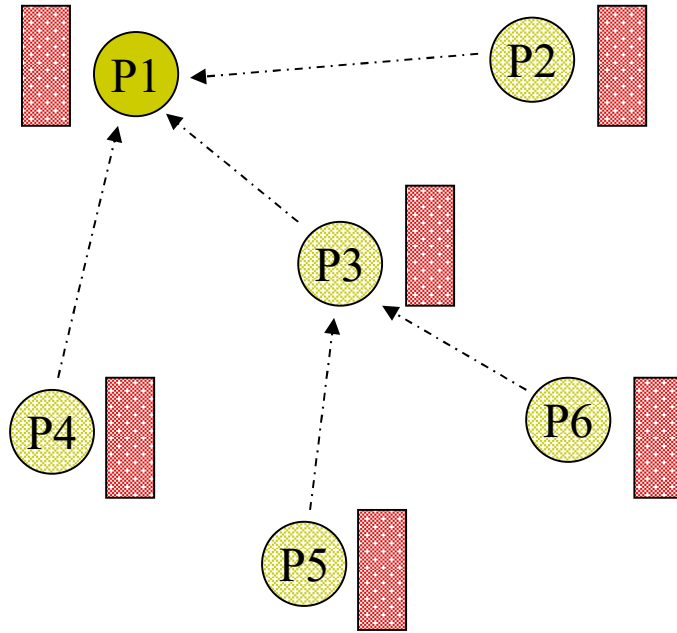


# Problems with Token Rings

- ❑ continuous use of network bandwidth
- ❑ delay to enter CS depends on the size of ring
  - Not practical for large networks
- ❑ order of requests may not be respected



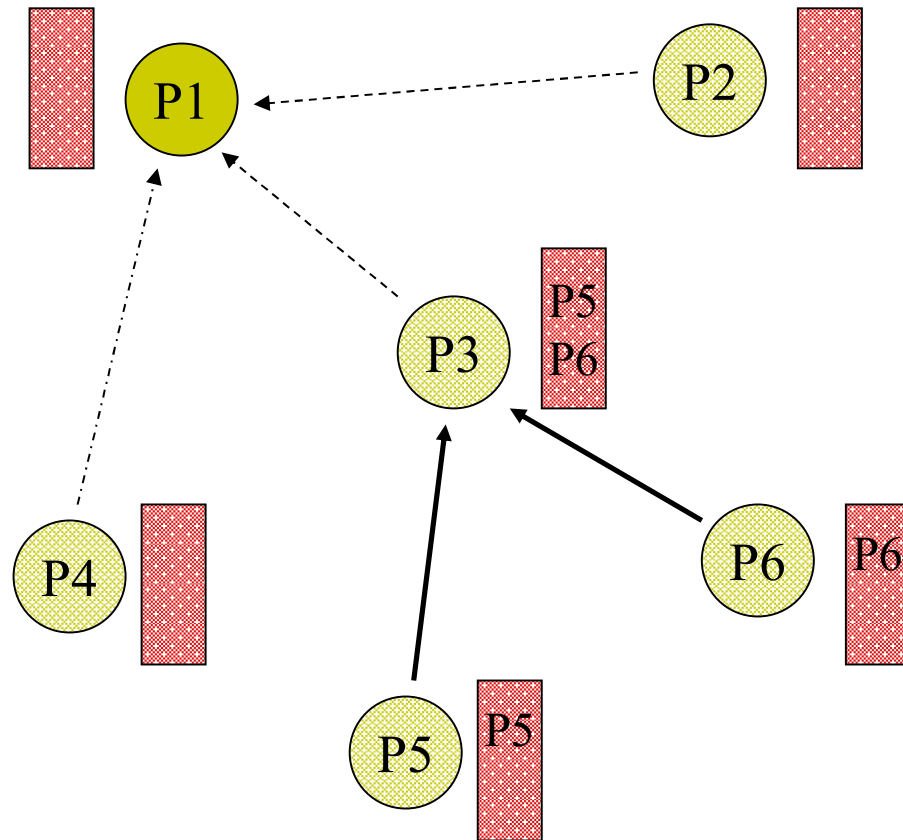
# Token-Based Solutions on Logical Trees



[Raymond 1989]

1. The tree is dynamically structured so that the root always holds the token.
2. Each process maintains a FIFO queue of requests for the Token from its successors, and a pointer to its immediate predecessor.
3. A process requesting the token or receiving a request from its successor appends the request to its queue and then request its predecessor for the token if it does not hold the token.

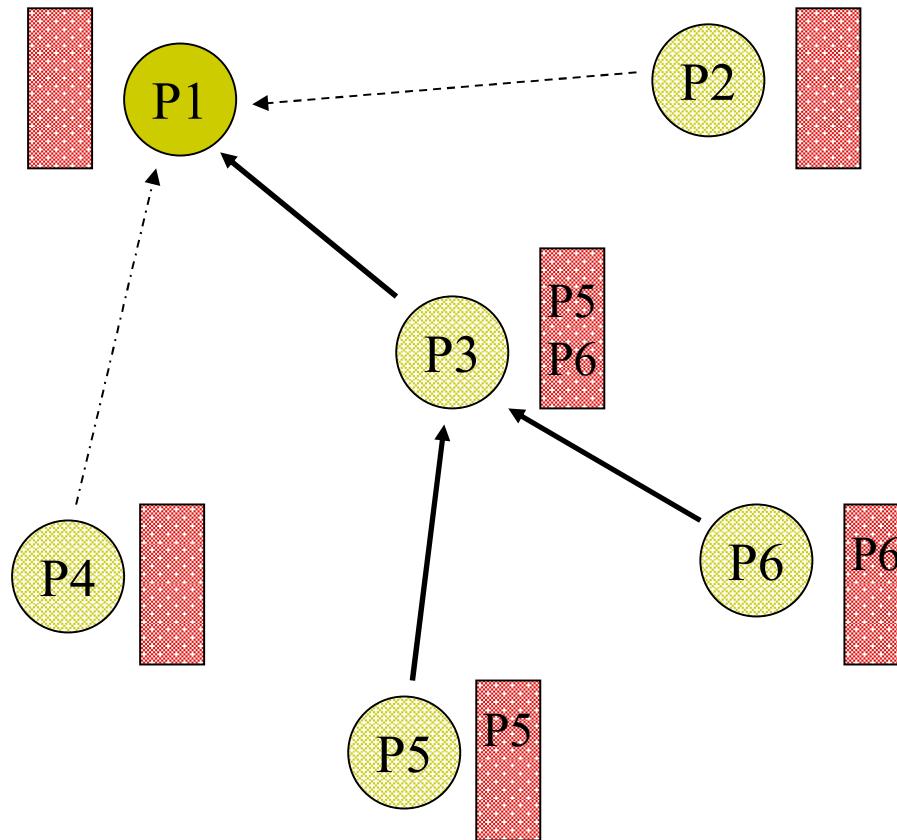
# Token- and Tree-Based (contd.)



1. P5 and P6 request the token from P3, and suppose P5's request arrives first.

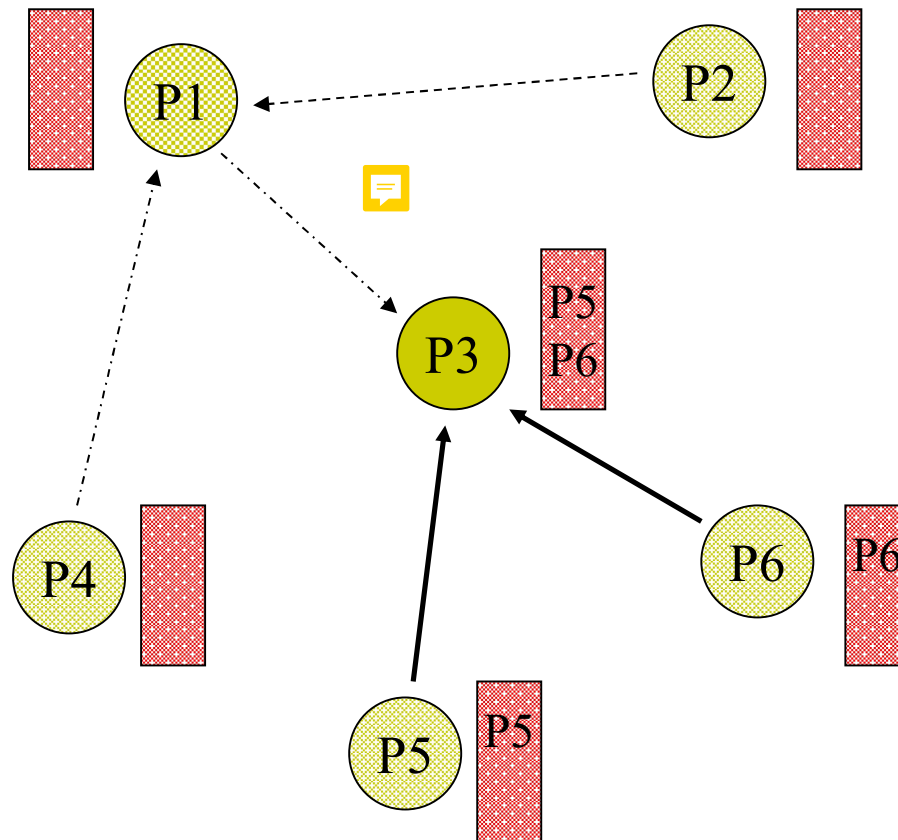


# Token- and Tree-Based (contd.)



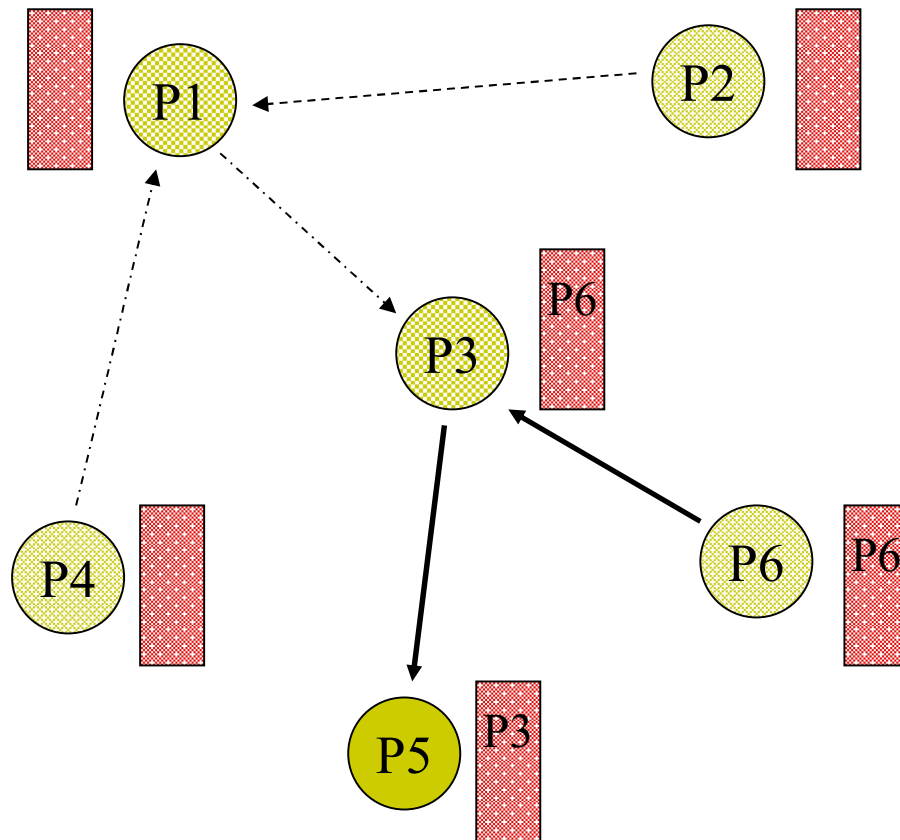
2. Since P3 does not hold the token, it requests the token from its predecessor P1.

# Token- and Tree-Based (contd.)



3. P3 receives the token. It then removes P5 from its queue, and sends the token to P5, which is the new predecessor of P3.

# Token- and Tree-Based (contd.)

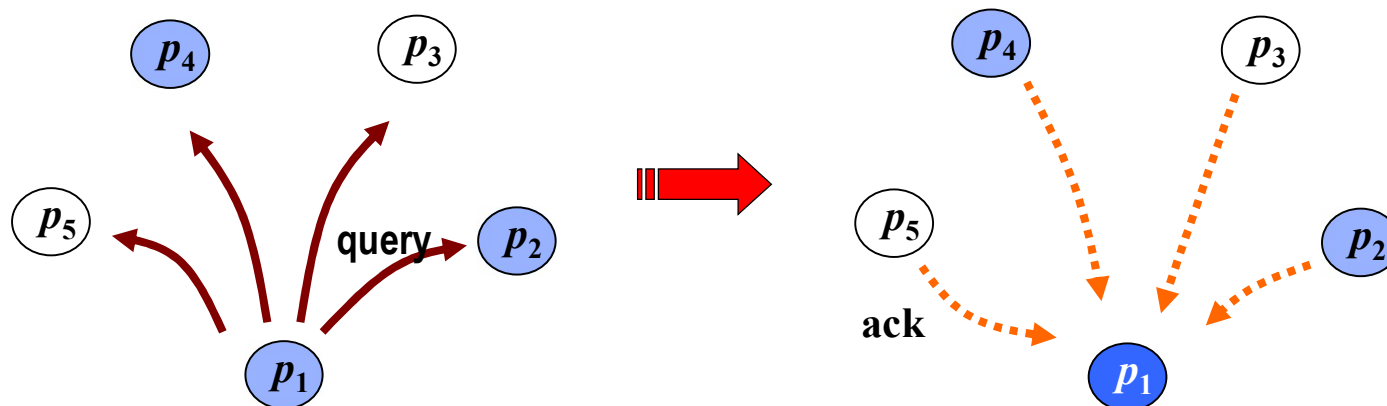


4. P3 receives the token. It then removes P5 from its queue, and sends the token to P5, which is the new predecessor of P3.

Since P3's queue is still not empty, it also sends a request to the new predecessor.

# Ricart and Agrawala's Distributed Algorithm

1. A process requesting entry to the CS sends a request to every other process in the system; and enters the CS when it obtains permissions from every other process.
2. When does a process grant another process's request?
  - Conflict resolved by **logical timestamps** of requests.
  - Does it work using physical timestamps?



# Quorum Systems

A **quorum system** is a collection of sets of processes called **quorums**.  
-- [Garcia-Molina & Barbara, 1985]

In resource allocation, a process must **acquire a quorum** (i.e., **lock all the quorum members**) in order to access a resource. Resource allocation algorithms that use quorums typically have the following advantages:

## Less message complexity

- ◆ Lower than Richart & Agrawala's  $O(N)$  algorithm, but in general higher than token-based

## Fault tolerant

# Formal Definition of Quorums

Let  $\mathbf{P} = \{p_0, p_1, p_2, \dots, p_{n-1}\}$  be a set of processes.

A **coterie**  $\mathbf{C}$  is a subset of  $2^{\mathbf{P}}$  such that<sup>-- [Garcia-Molina & Barbara, 1985]</sup>

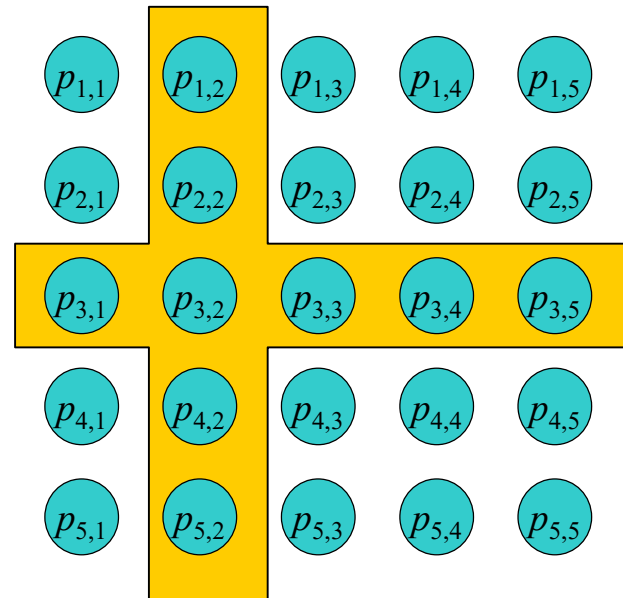
- **Intersection:**  $\forall Q_i, Q_j \in \mathbf{C}, Q_i \cap Q_j \neq \emptyset$
- **Minimality:**  $\forall Q_i, Q_j \in \mathbf{C}, Q_i \neq Q_j \Rightarrow Q_i \not\subset Q_j$

Each set in  $\mathbf{C}$  is call a **quorum**.

# Some Quorum Systems

- ❑ Majority
- ❑ Tree quorums
- ❑ Grid
- ❑ Finite Projective Plane

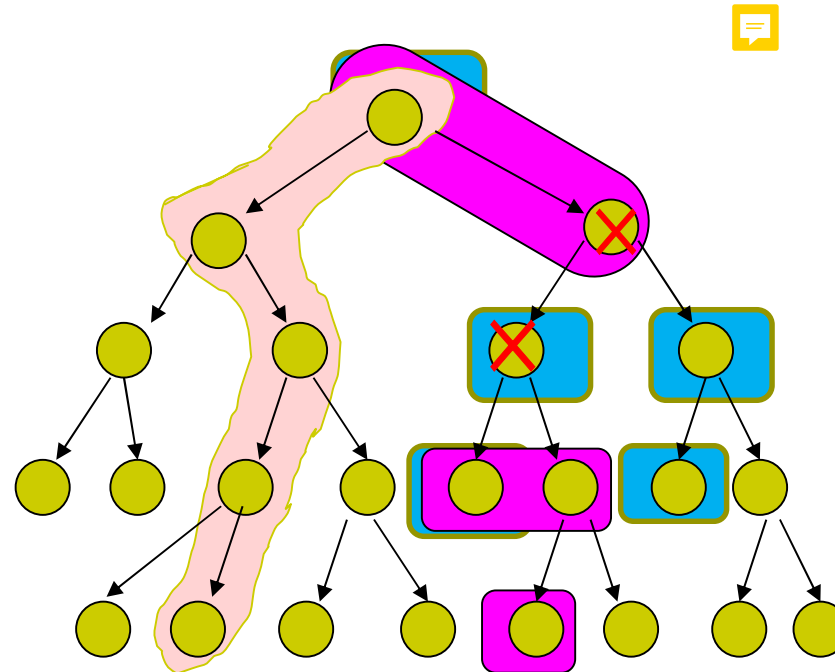
Grid Quorum Systems



Any collection of a row and a column of nodes constitute a quorum.

# Tree Quorums

- ❑ Any path of nodes from the root to a leaf constitute a quorum.
- ❑ In addition, if a node is not selected, then for each of its children nodes, a nonempty path of nodes from the child to a leaf must be included (the definition recursively applies to the sub-paths).



Are all the leaf nodes a quorum?



# Fully Distributed Quorum Systems

A quorum system  $\mathcal{C} = \{Q_1, Q_2, \dots, Q_m\}$  over  $\mathbf{P}$  that additionally satisfies the following conditions:

- **Uniform:**  $\forall 1 \leq i, j \leq m: |Q_i| = |Q_j|$
- **Regular:**  $\forall p, q \in \mathbf{P}: |n_p| = |n_q|$ , where  $n_p$  is the set  $\{Q_i \mid \exists 1 \leq i \leq m: p \in Q_i\}$ , and similarly for  $n_q$ .

E.g., Finite Projective Planes of order  $p^k$ , where  $p$  is a prime.

$$Q_1 = \{1, 2\}$$

$$Q_2 = \{1, 3\}$$

$$Q_3 = \{2, 3\}$$

$$Q_4 = \{1, 2, 3\} \quad Q_5 = \{2, 5, 7\}$$

$$Q_2 = \{1, 4, 5\} \quad Q_6 = \{3, 4, 7\}$$

$$Q_3 = \{1, 6, 7\} \quad Q_7 = \{3, 5, 6\}$$

$$Q_4 = \{2, 4, 6\}$$

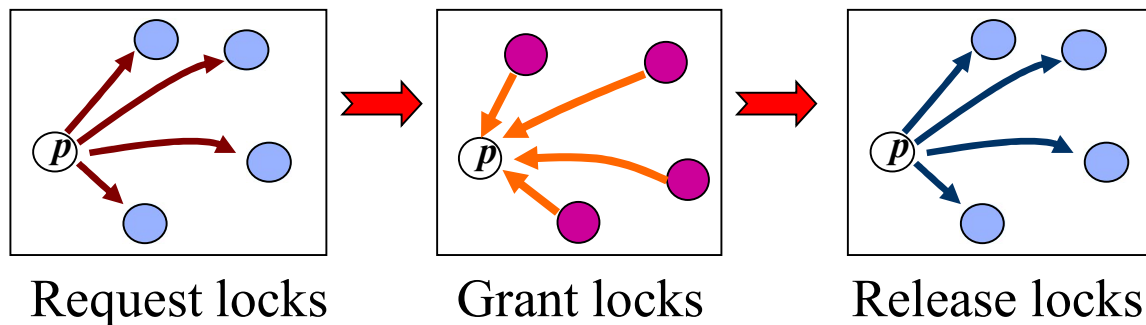
# Maekawa's Algorithm

- ❑ A process  $p$  wishing to enter CS chooses a quorum  $Q$ , and sends *lock requests* to all nodes of the quorum.
- ❑ It enters CS only when it has locked all nodes of the quorum (i.e., has acquired the quorum).
- ❑ Upon exiting CS,  $p$  unlocks the nodes.
- ❑ A node can be locked by one process at a time.
- ❑ Conflicting lock requests to a node are resolved by *priorities* (e.g., timestamps). The loser must yield the lock to the high priority one if it cannot successfully obtain all locks it needs.

# Message Complexity

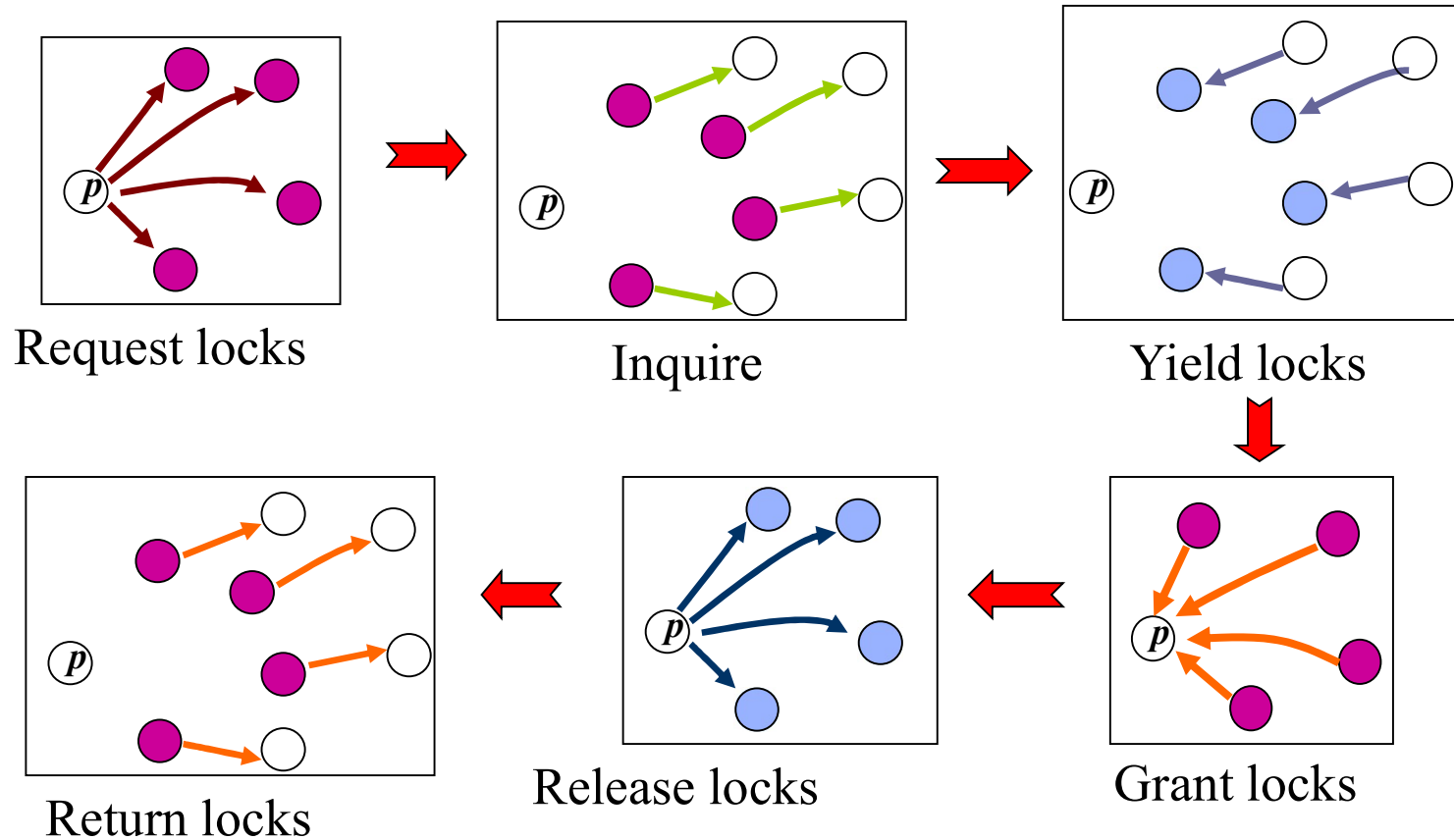
Maekawa's algorithm needs  $3c$  to  $6c$  messages per entry to CS, where  $c$  is the size of the quorum a process chooses.

Best case:  $3c$



# Worst Case

Worst case:  $6c$



# Note on Maekawa's Algorithm

- ❑ Message complexity depends on the size of quorums.
- ❑ Reduce the problem of mutex to the design of quorums.
- ❑ What the min. size of a quorum can be, given that the load of processes are to be fully distributed?



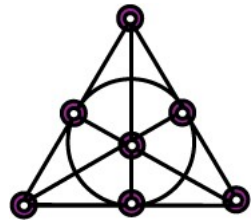
A  $\sqrt{N}$  Algorithm for Mutual Exclusion in  
Decentralized Systems, Mamoru Maekawa, ACM  
TOCS, 1985.

# Projective Planes

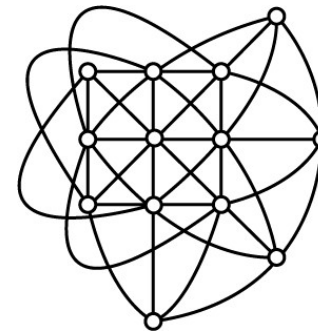
- A **Projective Plane** is a plane satisfies the following:
  - Any line has at least two points.
  - Two points are on precisely one line.
  - Any two lines meet.
  - There exists a set of four points, no three of which are collinear.
- A Projective Plane is said to be **order  $n$**  if a line contains exactly  $n+1$  points

# Projective Planes (contd.)

- A projective plane of order  $n$  has the following properties:
- Every line contains exactly  $(n+1)$  points
  - Every point is on exactly  $(n+1)$  lines
  - There are exactly  $(n^2+n+1)$  points
  - There are exactly  $(n^2+n+1)$  lines



Fano plane (the projective plane of order 2)



the projective plane of order 3

# Read/Write Quorums

For database concurrency control,

- every **read quorum** must intersect with every **write quorum**,
- every two write quorum must intersect.



# Comparison

Algorithm	Message per entry/exit	Delay before entry (in msg time)	Drawbacks
<b>Centralized</b>	3	2	Single failure point
(Ricart and Agrawala's) <b>Distributed</b>	$2(n-1)$	2 if multicast is supported; $2(n-1)$ otherwise	Crash of any process
<b>Tree</b>	$O(\log n)$	$O(\log n)$	Token loss, process crash
<b>Token Ring</b>	1 to $\infty$	1 to $n-1$	Token loss, process crash
<b>Voting</b>	quorum size	2	Need to determine a suitable <i>quorum system</i>

# Leader Election

# The Leader Election Problem

Many distributed algorithms require one process to act as coordinator, initiator, sequencer, or otherwise perform some special role, and therefore one process must be elected to take the job, even if processes may fail.

## Requirements:

- Safety: at most one process can be elected at any time.
- Liveness: some process is eventually elected.

## Assumptions

Each process has a unique id.



# The Bully Algorithm

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

1.  $P$  sends an *ELECTION* message to every process with a larger id.
2. If after some timeout period no one responds,  $P$  wins the election and becomes the coordinator.
3. If one of the higher-ups answers, it takes over the election; and  $P$ 's job is done. (A process must answer the *ELECTION* message if it is alive.)
4. When a process is ready to take over the coordinator's role, it sends a *COORDINATOR* message to every process to announce this.
5. When a previously-crashed coordinator is recovered, it assumes the job by sending a *COORDINATOR* message to every process.

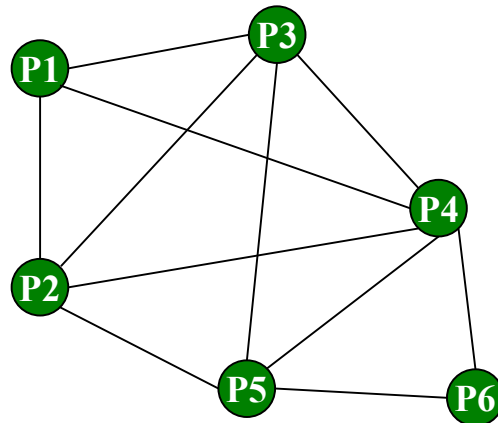
Message Complexity:

Best case:  $n-2$

Worst case:  $O(n^2)$ .

# Leader Election in Arbitrary Network

- ❑ Construct a minimum spanning tree and let the root be the leader.
- ❑ Yo-Yo algorithm



# Yo-Yo Distributed Leader Election

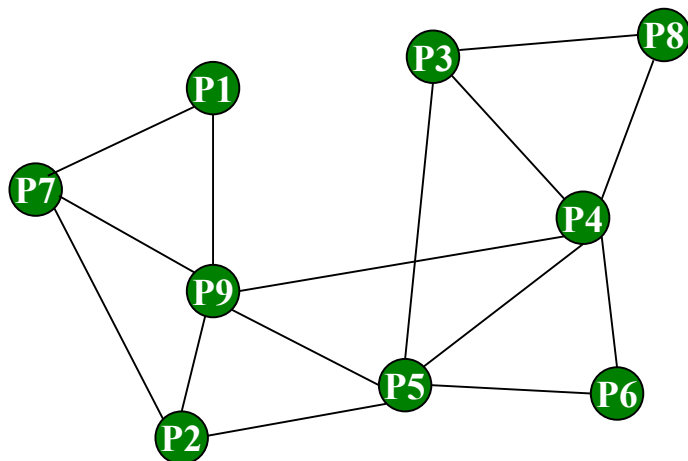
## □ Two phases:

### ■ pre-processing:

- Create a directed acyclic graph by orienting edge direction based on node ids

### ■ a sequence of iterations of YO- & -YO

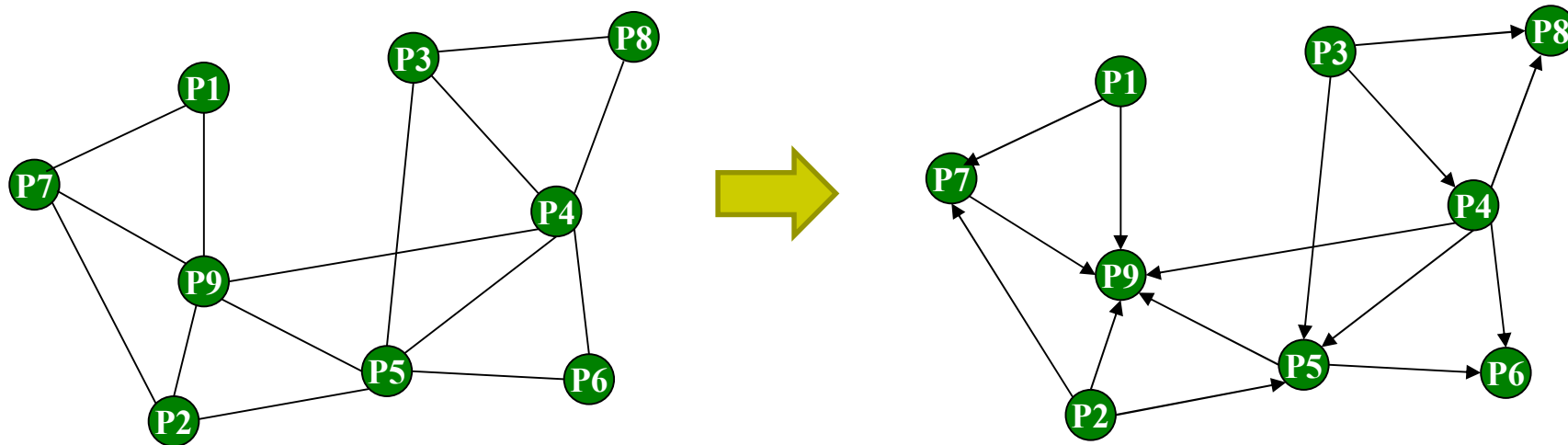
- To eliminate candidate source nodes until only one survives



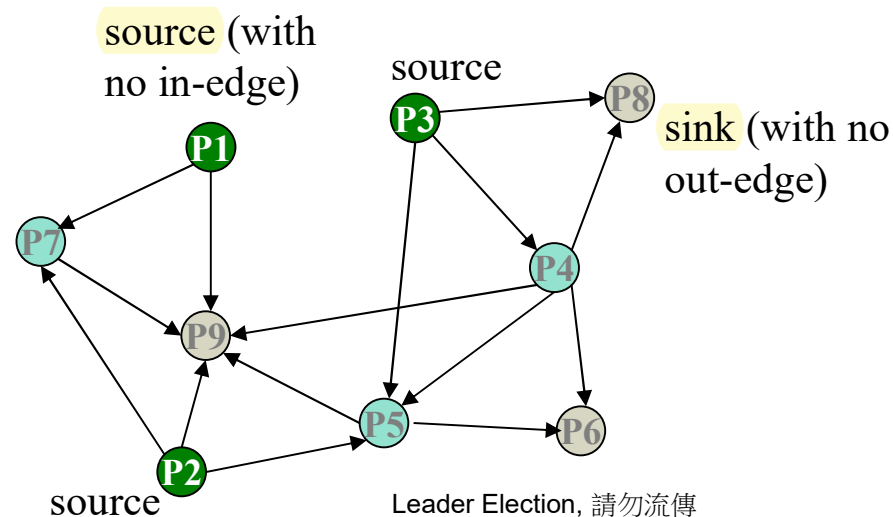
Reading: Design and Analysis of Distributed Algorithms, Chap. 3, Election, Nicola Santoro, 2006.

# Pre-processing

Every node exchanges its id with its neighbors, and orient their links based on their ids, say,  $x \rightarrow y$  if  $\text{id}(x) < \text{id}(y)$



Must be acyclic.

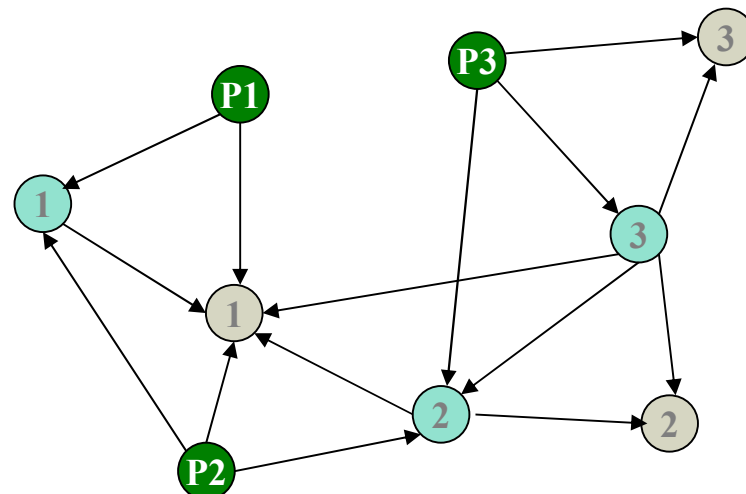
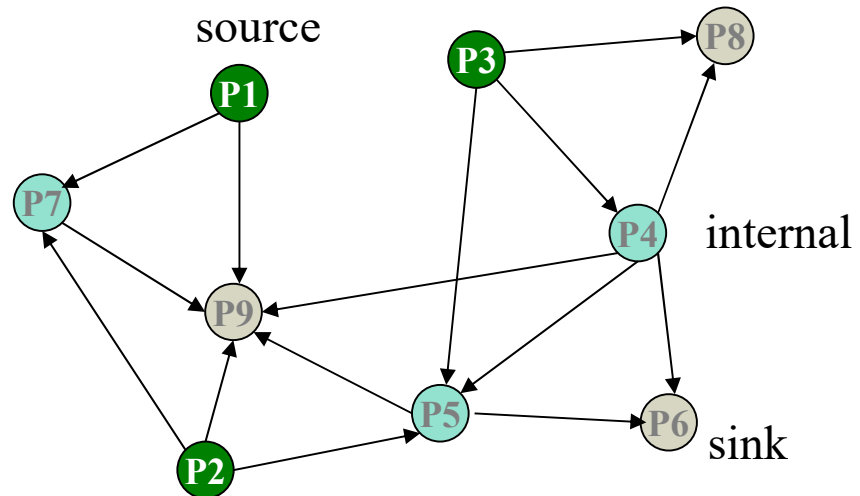


# Iteration: YO- & -YO

YO-

started by the sources to propagate to each sink the smallest among the values of the sources connected to that sink

- A source sends its value down to all its out-neighbours.
- An internal node waits until it receives a value from all its in-neighbours, and sends the minimum out-neighbours.
- A sink waits until it receives a value from all its in-neighbours, and starts the second part of the iteration.



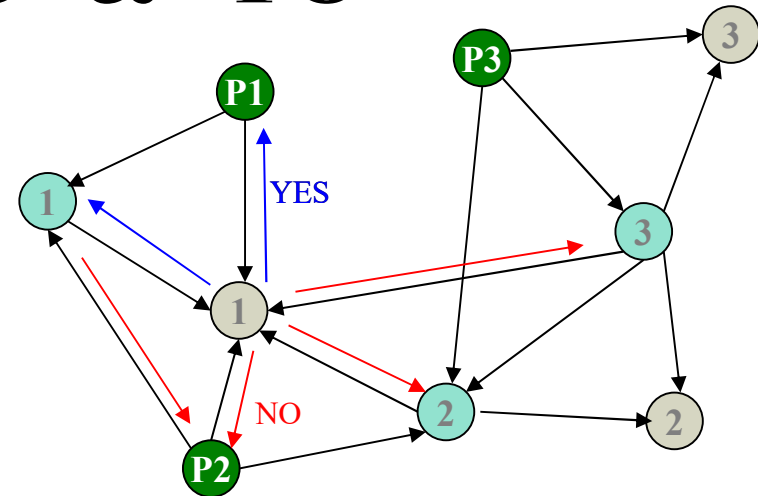


# Iteration: YO- & -YO

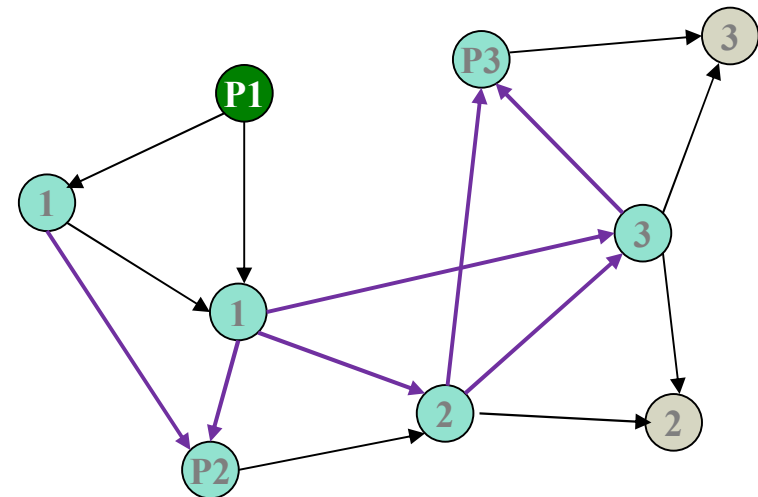
## -YO

started by the sinks to eliminate some candidates by informing their connected sources of whether their id is the smallest seen so far.

- A sink sends YES to all in-neighbours from which the smallest value has been received, and NO to all the others.
- An internal node waits until it receives a vote from all its out-neighbours. If all votes are YES, it sends YES to all in-neighbours from which the smallest value has been received, and NO to all the others. If at least a vote was NO, it sends NO to all its in-neighbours.
- A source waits until it receives a vote from all its out-neighbours. If all votes are YES, it survives this iteration and starts the next one. If at least a vote was NO, it is no longer a candidate.



“flip” the direction of each link where a NO vote is sent



only the surviving candidates are sources.

# Message Complexity

- ❑ Each phase: 2 messages each link
- ❑ Number of phases:  $\log(\text{number of sources})$
- ❑ Total:  $2 E \log S \Rightarrow O(E \log N)$ 
  - $E$  number of edges
  - $N$ : number of nodes
- ❑ Message complexity can be reduced by “pruning”
  - remove nodes and links that are “useless” during the YO-YO iterations

Time complexity?

# Pruning of YO- & -YO

- ❑ to remove nodes and links that are “useless” --- no impact on the result of the iteration
  - For details, see Design and Analysis of Distributed Algorithms, Chap. 3, Election, Nicola Santoro, 2006.
- ❑ How does this affect time & message complexity?