Lecture#7-8: Naming & Canonical problems

S. M. Ahsan Kazmi

Outline

- Communication
  - Message-Oriented Communication
    - Transient and Persistent Communication
  - Stream Oriented Communication
- Naming
  - Flat Naming
  - Structured Naming
  - Attribute-based Naming
- Clock Synchronization
  - Physical Clocks
    - Clock Synchronization Algorithms
  - Logical clocks
    - Lamport's Logical Clocks
    - Vector Clocks
- Canonical problems and solutions
  - Synchronization
  - Mutual exclusion
  - leader election

Message-Oriented Communication

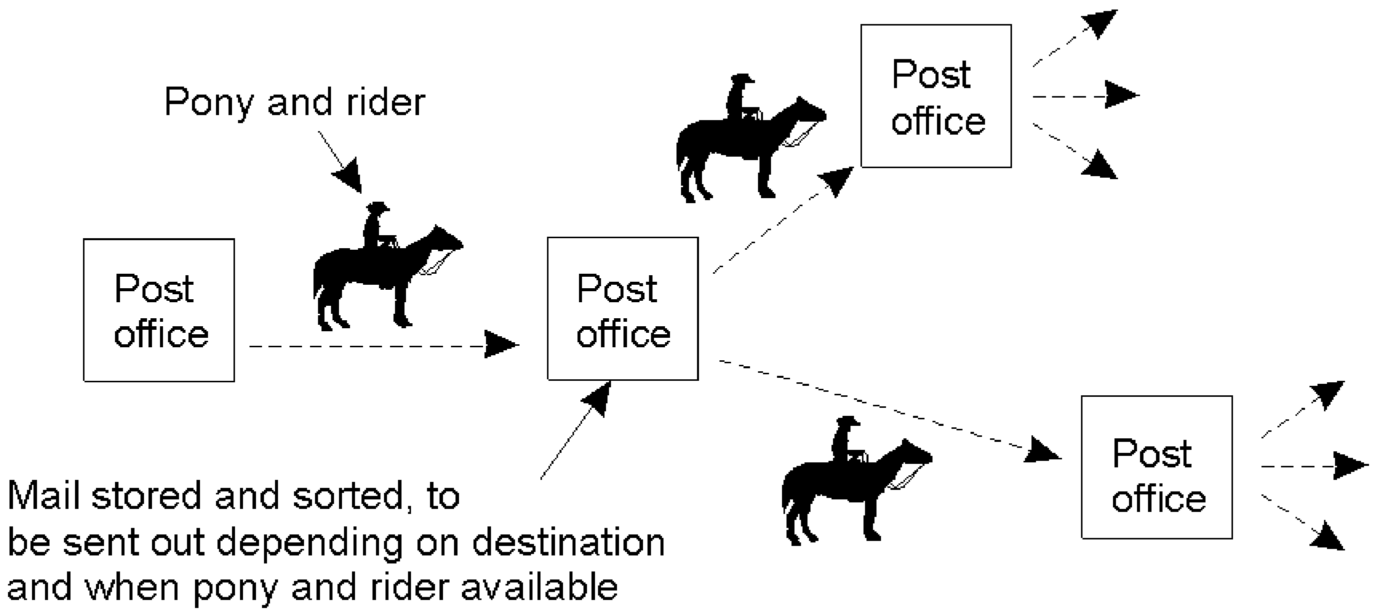
- Remote Procedure Calls –last lecture
  - RPC enhanced access transparency
  - RPC inherently synchronous in nature
    - client blocked until request processed

- Issue: Heterogeneous types of applications?

- Solution
  - Message-Oriented Communication uses messages
  - Can be viewed along two axes:
    - persistence (whether the system is persistent or transient)
    - synchronicity (whether it is synchronous or asynchronous)

Persistent Communication

- Persistent communication
  - It is also called a store-and-forward based delivery paradigm.


Pony and rider
Post office
Post office
Post office
Post office
Mail stored and sorted, to be sent out depending on destination and when pony and rider available

Transient Communication

- Message is stored only as long as sending/receiving applications are executing

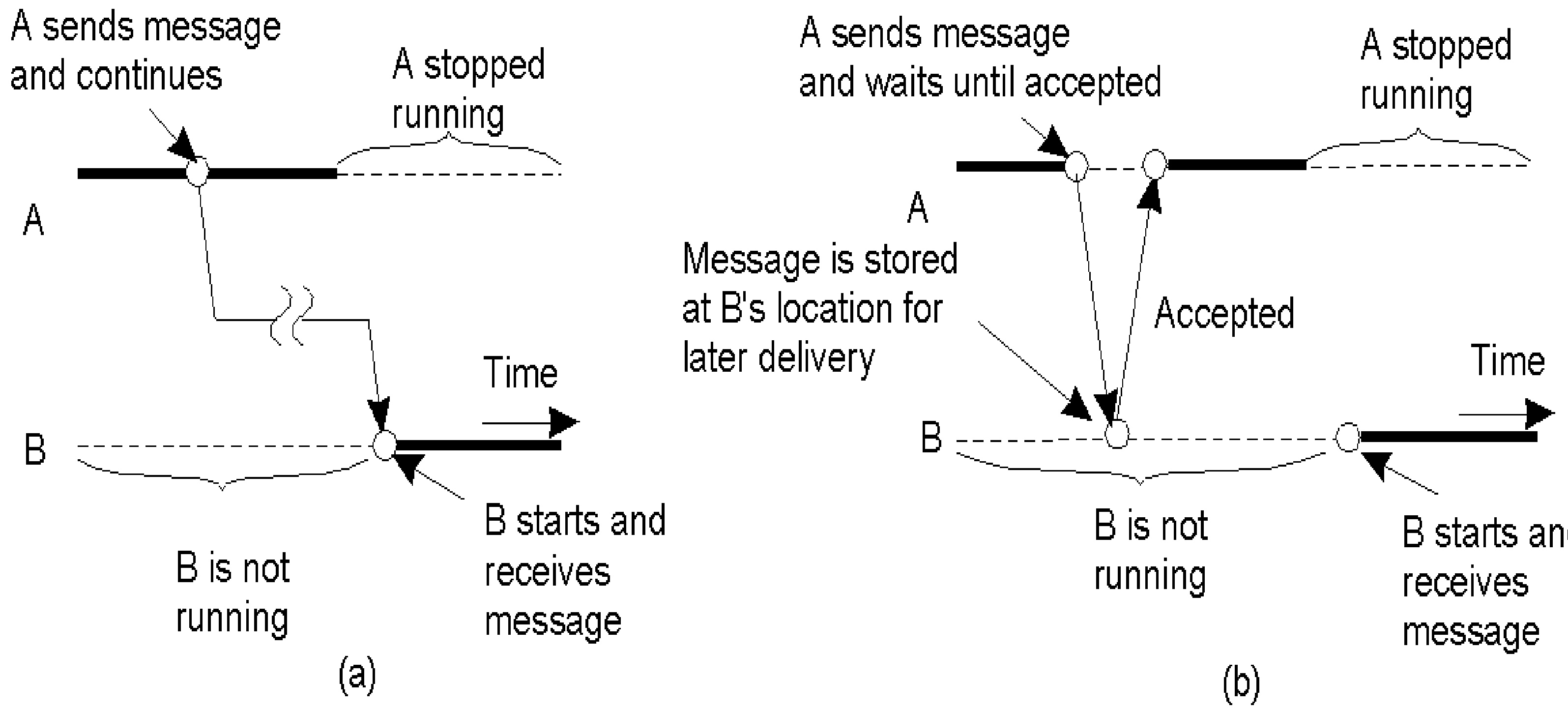
- Discard the message if it can't be delivered to the next server/receiver
  - Example: Typical network router – discard message if it can't be delivered to next router or destination

Synchronicity in Communication

- Asynchronous Communication
  - Asynchronous implies the sender proceeds as soon as it sends the message – no blocking
  - Need a local buffer at the sending host

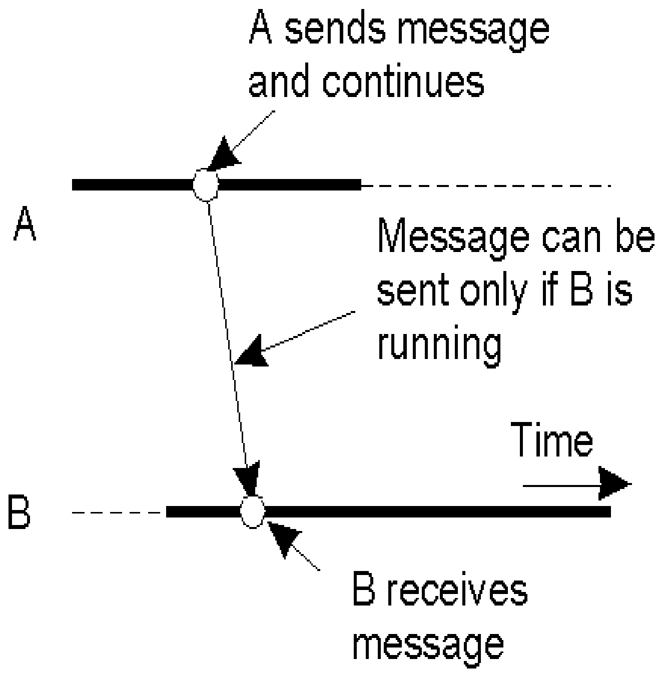
- Synchronous Communication
  - Synchronous implies the sender blocks till the receiving host buffers the message
  - Variant: block until the receiver processes the message

Persistence and Synchronicity

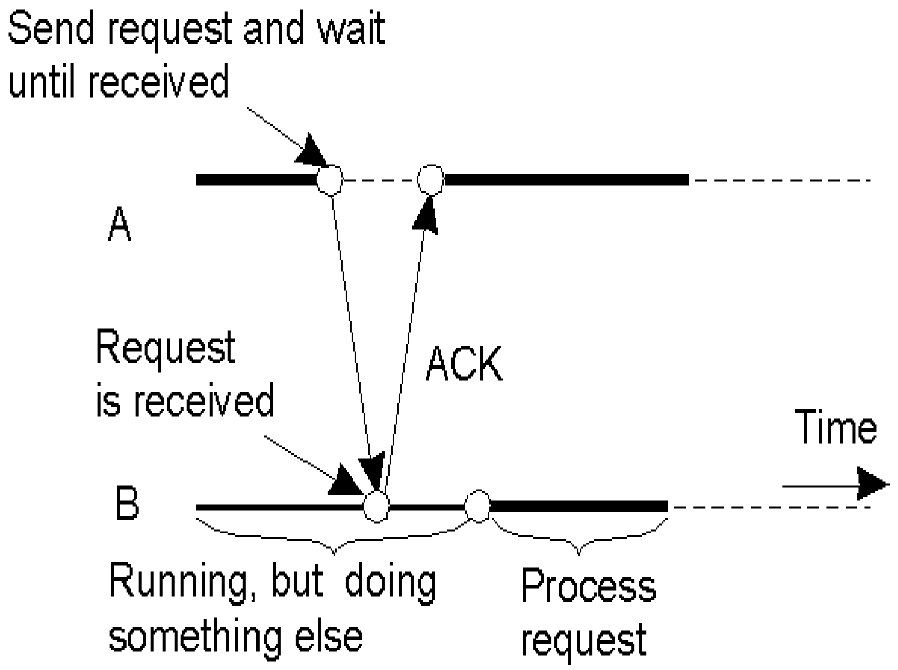

A sends message and continues
A stopped running
A
B
Time
B is not running
B starts and receives message
(a)
A sends message and waits until accepted
A stopped running
A
Message is stored at B's location for later delivery
Accepted
Time
B
B is not running
B starts and receives message
(b)

(a) *Persistent asynchronous communication (e.g. email)*
(b) *Persistent synchronous communication (e.g. Messaging and chat applications)*

Persistence and Synchronicity



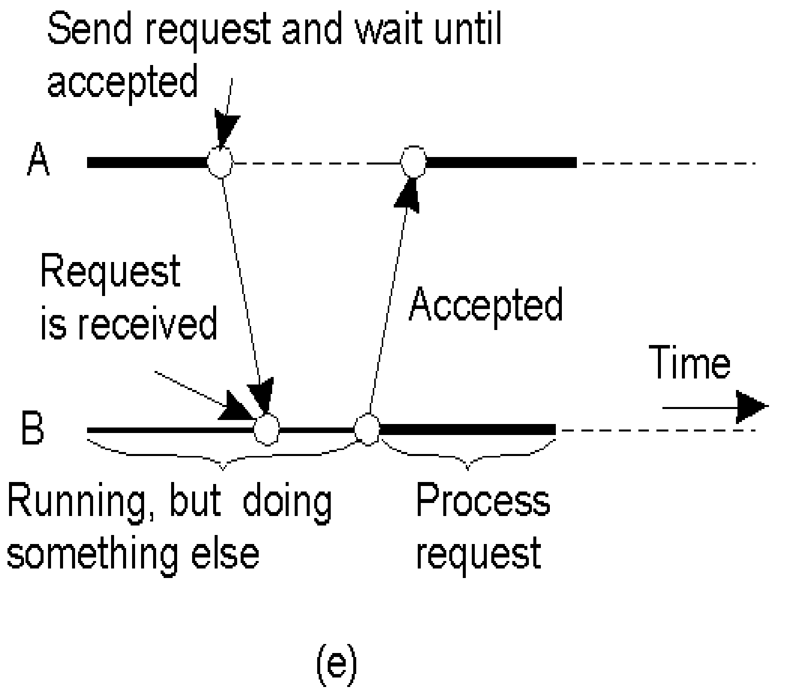
(c)

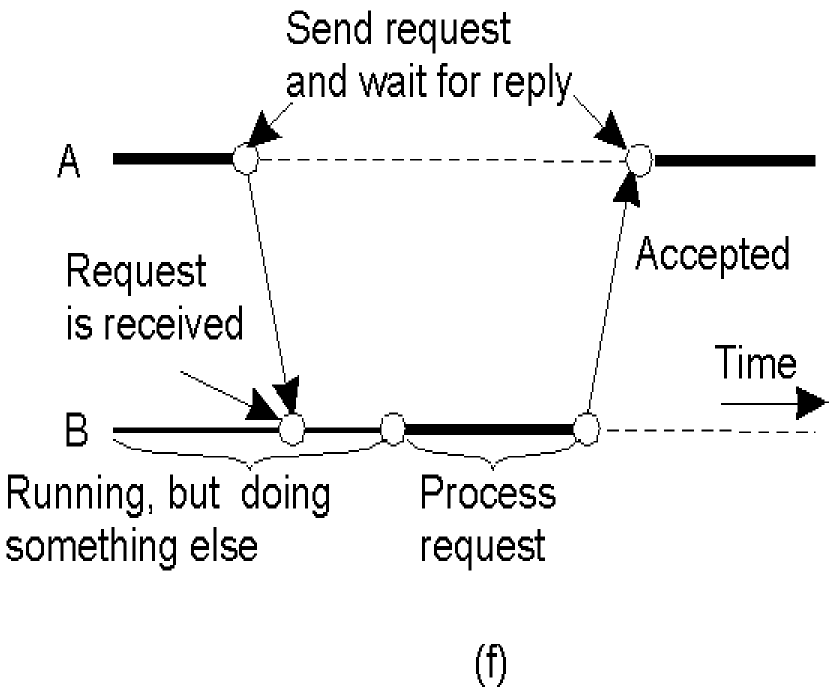


(d)

(c) Transient asynchronous communication (UDP)
(d) Receipt-based transient synchronous communication

Persistence and Synchronicity


Send request and wait until accepted
A
B
Request is received
Accepted
Time
Running, but doing something else
Process request
(e)


Send request and wait for reply
A
B
Request is received
Accepted
Time
Running, but doing something else
Process request
(f)

(e) Delivery-based transient synchronous communication at message delivery (async. RPC)
(f) Response-based transient synchronous communication (RPC)

Message-Oriented Transient Communication

- Many distributed systems and applications are built on top of the simple message-oriented model
- No on-disk queues used for message storing
- Message-oriented models
  - Berkeley Sockets: Socket interface as introduced in Berkeley UNIX
  - The Message-Passing Interface (MPI): designed for parallel applications and tailored to transient communication

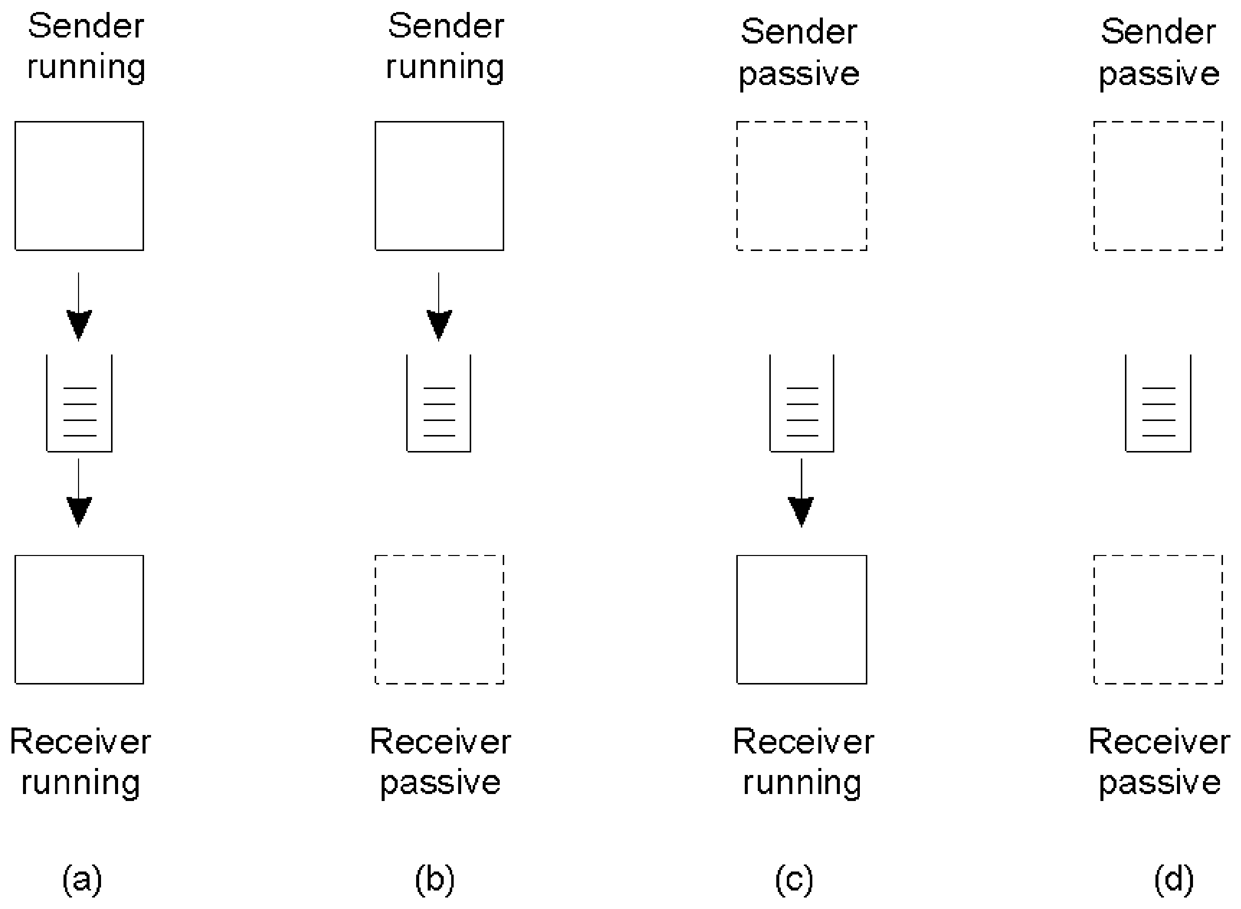
Message-Oriented Persistent Communication

- Message-Queuing Model
  - Apps communicate by inserting messages in specific queues
    - Loosely-coupled communication
  - Support for:
    - Longer message transfers (e.g., e-mail systems)
  - Basic interface to a queue in a message-queuing system:

Primitive	**Meaning**
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

Message-Oriented Persistent Communication

- Message-Queuing Model
  - Four combinations for loosely-coupled communication using queues:



Stream Oriented Communication

- Message-oriented communication: request-response
  - communication occurs and speed do not affect correctness
- Timing is crucial in certain forms of communication
  - Examples: audio and video (“continuous media”)
- Streams and Quality of Service
  - The required bit rate at which data should be transported
  - The maximum end-to-end delay
  - The maximum delay variance, or jitter
  - The maximum round-trip delay

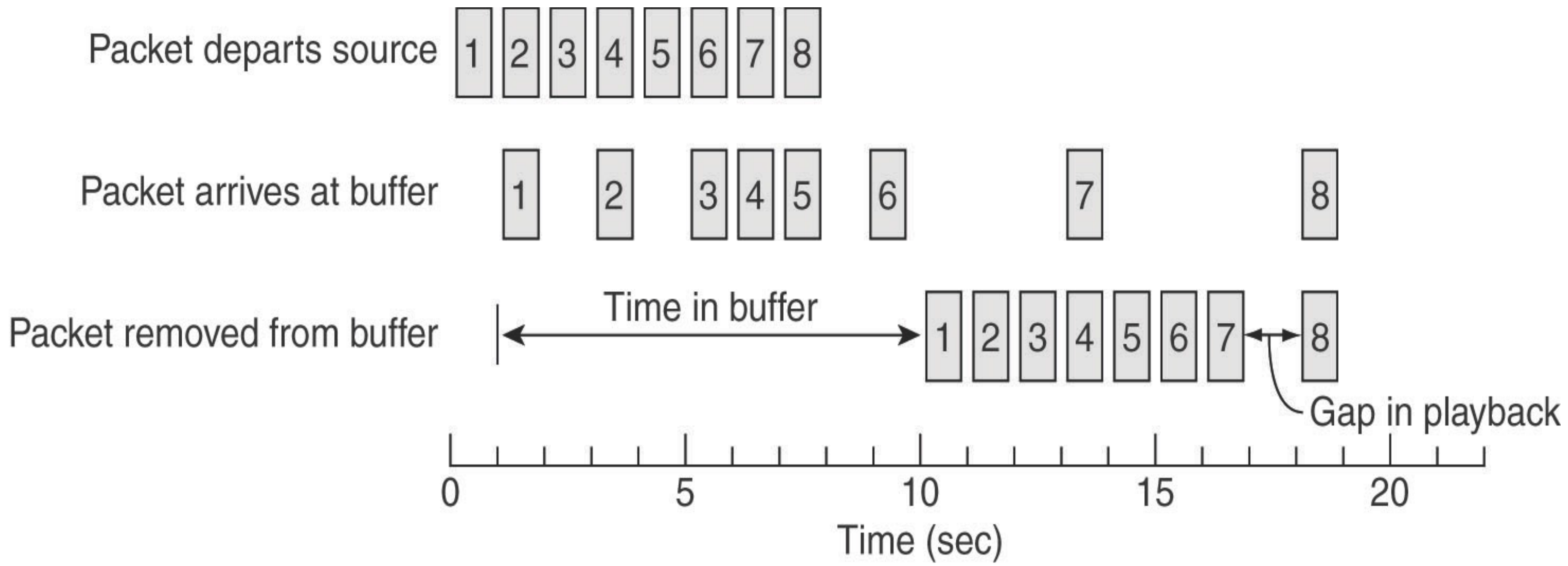
Quality of Service (QoS)

- Time-dependent and other requirements are specified as *quality of service (QoS)*
  - Requirements guarantees from the underlying systems
  - Application specifies workload and requests a certain service quality
  - Contract between the application and the system

- Enforcing QoS Techniques
  - Enforce at end-points (e.g., token bucket)
    - No network support needed
  - Mark packets and use router support
  - Differentiated services: expedited & assured forwarding

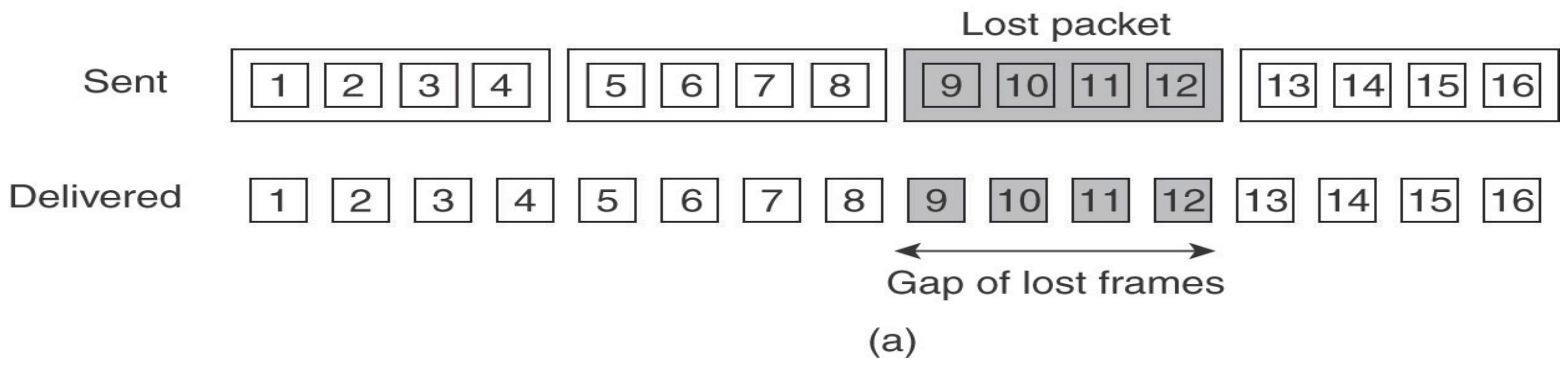
Enforcing QoS

- Use buffers at receiver to mask jitter

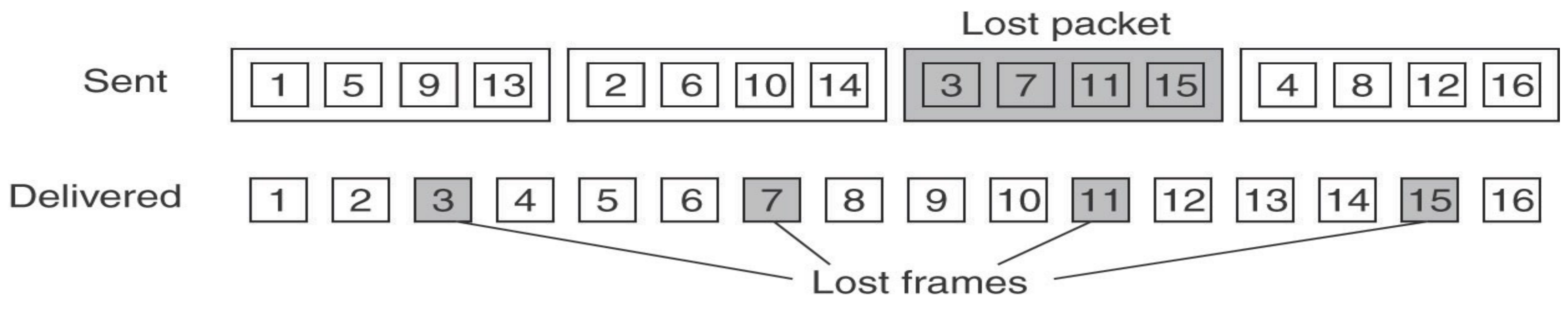


Enforcing QoS

- Handling Packet losses
  - Handle using forward error correction
  - Use interleaving to reduce impact



(a)



(b)

Naming

Naming

- Names are used to denote entities in a distributed system.
- To operate on an entity, we need to access it at an **access point**.
- Differences in naming in distributed and non-distributed systems
  - Distributed systems: naming systems is itself distributed

Types of Names in DS

- Flat
    - All names are equivalent in name space
    - Must be globally unique
- Structured
    - Names (usually) have structure
    - Unique only within immediately containing level

Flat Naming

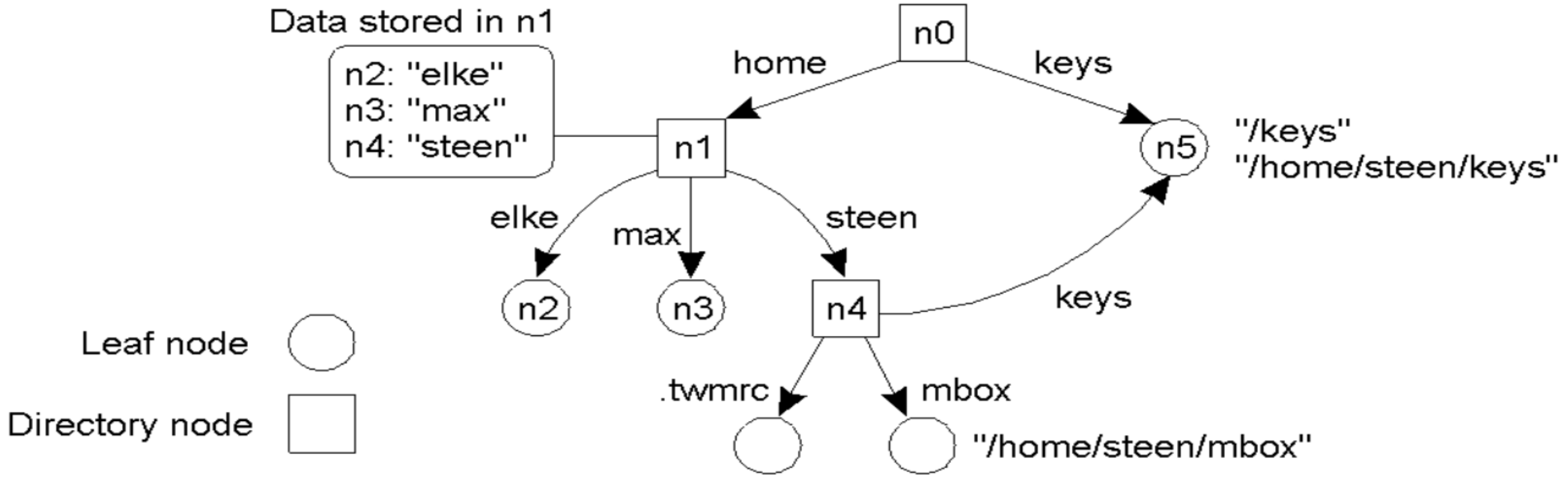
- Need a global directory
    - May be replicated
- **Problem**
    - Given an essentially unstructured name, how can we locate its associated access point?
        - Simple solution (broadcasting)
        - Distributed Hash Tables (structured P2P)

Structured Naming Systems

- Names organized into *name spaces*
- Name spaces organized into directed graph
    - Leaf nodes represent named entities
    - Interior nodes represent *directories*
- A directory node contains a (directory) table of (edge label, node identifier) pairs.
- Everyone must know the *root node*

Example: File Names

- Hierarchical directory structure (DAG)
  - Each file name is a unique path in the DAG
  - Resolution of /home/steen/mbox a traversal of the DAG
- File names are human-friendly

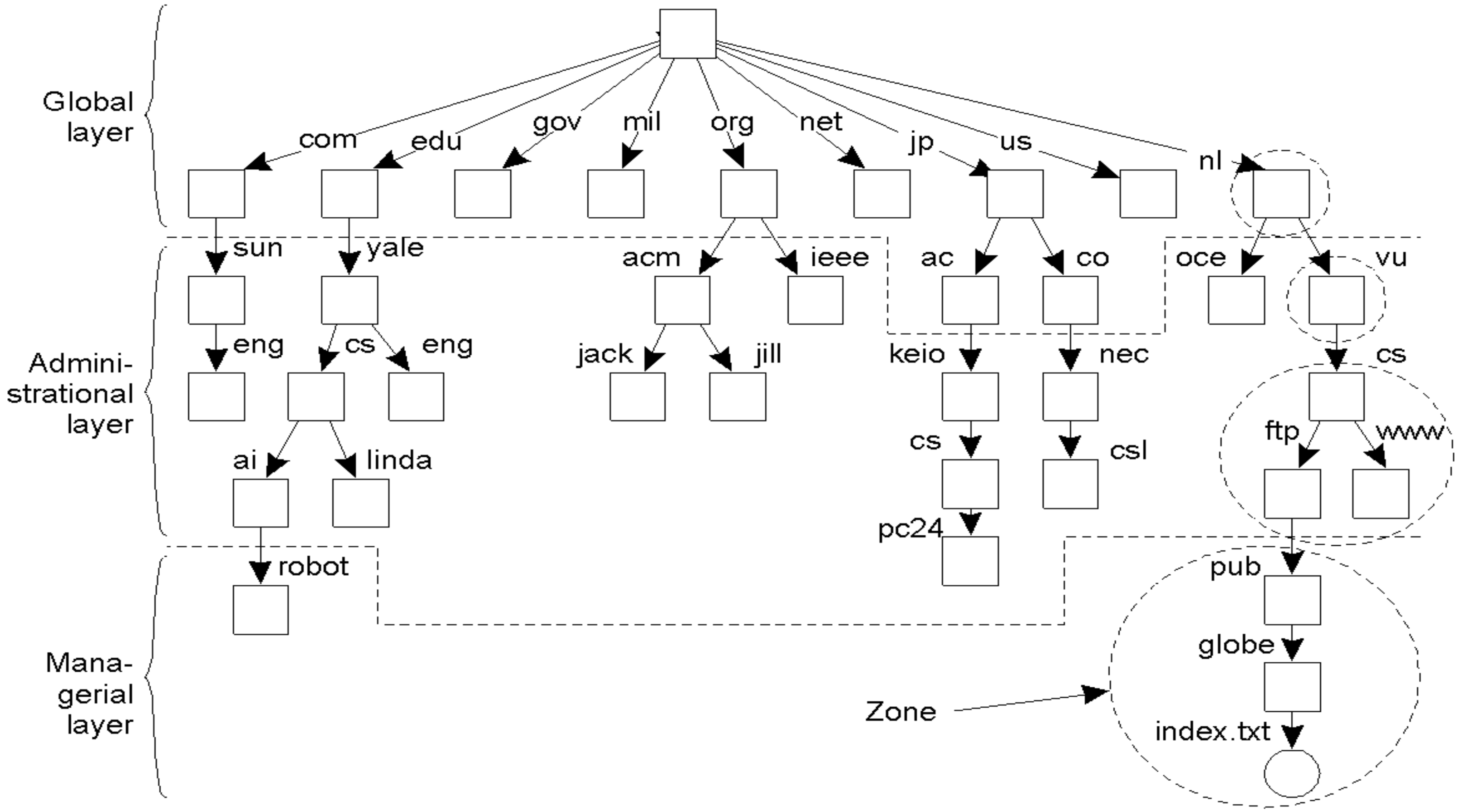


Name Space Distribution

- For large-scale DS, namespaces are organized **hierarchically**

- Name Spaces are partitioned into three logical layers
  - **Global Layer** → formed by highest-level nodes and the root of hierarchy; mostly static
  - **Administration Layer** → formed by directory nodes managed within a single organization
  - **Managerial Layer** → formed by nodes that may typically change regularly

Name Space Distribution



An example partitioning of the DNS namespace

Name Space Distribution

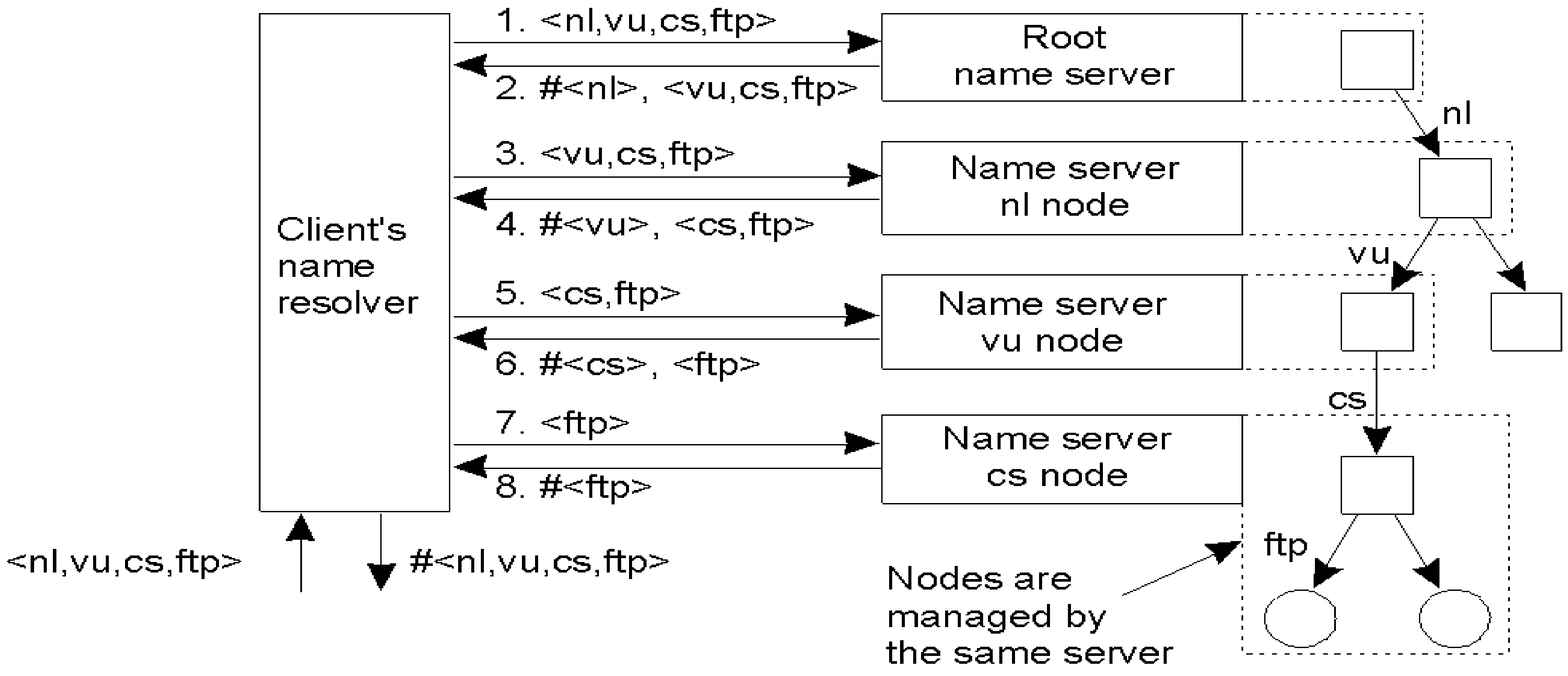
Item	**Global**	**Administrational**	**Managerial**
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

Implementation of Name Resolution

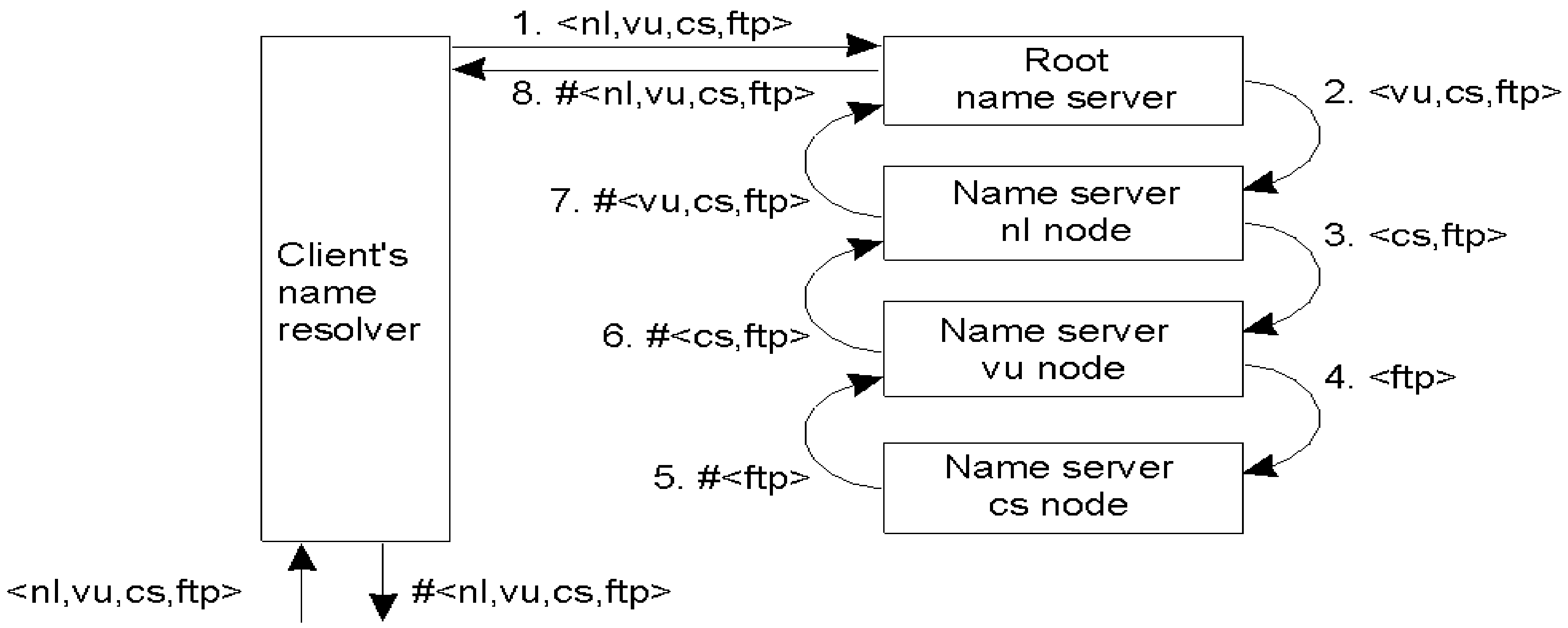
- Assumptions
  - No replication of name servers
  - No client-side caching
  - Each client has access to a local name server

- Two possible implementations
  - Iterative Name Resolution
    - Server will resolve the path name as far as it can, and return each intermediate result to the client
  - Recursive Name Resolution
    - A name server passes the result to the next name server found by it

Iterative Name Resolution



Recursive Name Resolution

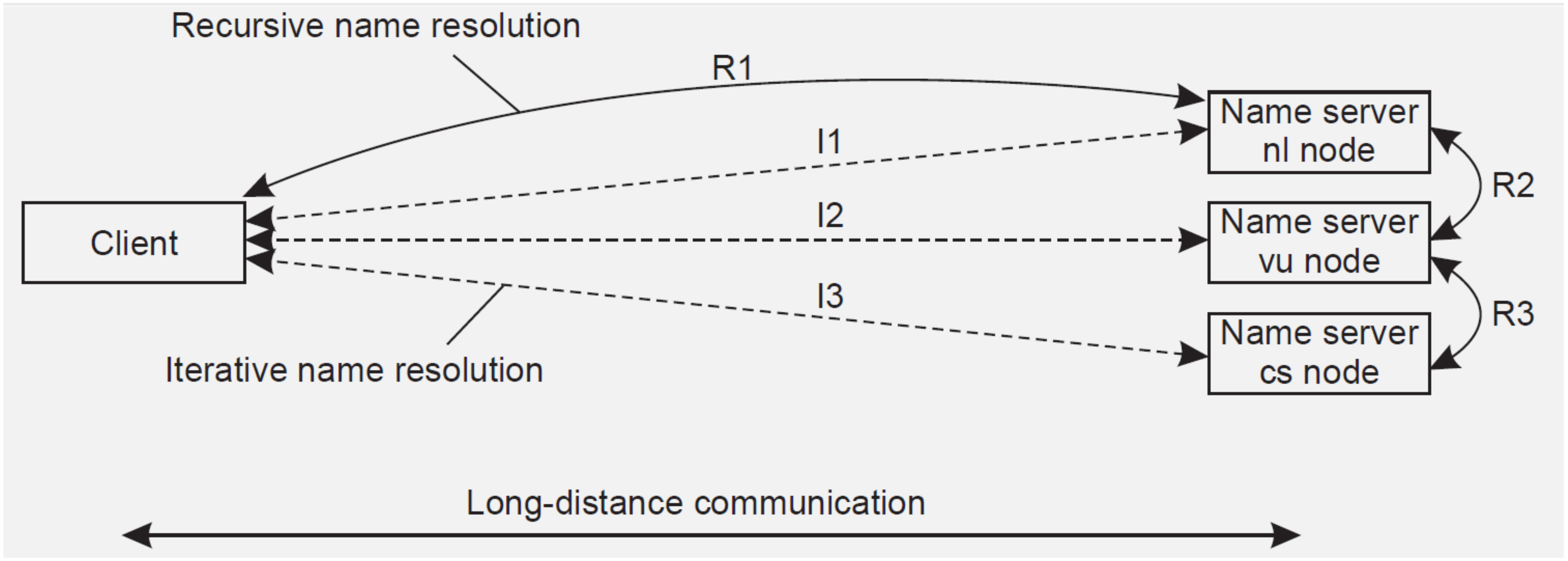


Scalability issues

- **Size scalability**
  - We need to ensure that servers can handle many requests per time unit => high-level servers are in big trouble.
- **Solution**
  - Assume (at least at the global and administrational level) that the content of nodes hardly ever changes.
  - We can then apply extensive replication by mapping nodes to multiple servers and start name resolution at the nearest server.

Scalability issues

- **Geographical scalability**
  - We need to ensure that the name resolution process scales across large geographical distances.
- **Problem**
  - By mapping nodes to servers that can be located anywhere, we introduce an implicit location dependency


Recursive name resolution
R1
I1
I2
I3
Client
Name server nl node
Name server vu node
Name server cs node
R2
R3
Iterative name resolution
Long-distance communication

Canonical problems and solutions

Introduction

- Synchronization: coordination of actions between processes.
- Processes are usually asynchronous, (operate independent of events in other processes)
- Sometimes need to cooperate/synchronize
  - For mutual exclusion
  - For event ordering (was message x from process P sent before or after message y from process Q?)

Synchronization: Challenges

- Time in unambiguous in centralized systems
  - System clock keeps time, all entities use this for time
- Distributed systems: each node has own system clock
  - Crystal-based clocks are less accurate (1 part in million)
- Problem: An event that occurred after another may be assigned an earlier time

Global Time

- **Global Time** is utilized to provide timestamps for processes and data.

- Two types of clocks:
  - **Physical clock**: concerned with “People” time
  - **Logical clock**: concerned with relative time and maintain logical consistency

Physical Clocks

- How to tell time?
  - Use astronomical metrics (solar day)

- Accurate clocks are atomic oscillators (one part in 10^^{13})

- Coordinated universal time *(UTC) – an* international standard based on atomic time
  - Add leap seconds to be consistent with astronomical time
  - UTC broadcast on radio (satellite and earth)
  - Receivers accurate to 0.1 – 10 ms

- Most clocks are less accurate (e.g., mechanical watches)
  - Computers use crystal-based clocks (one part in a million)
  - Results in *clock drift*

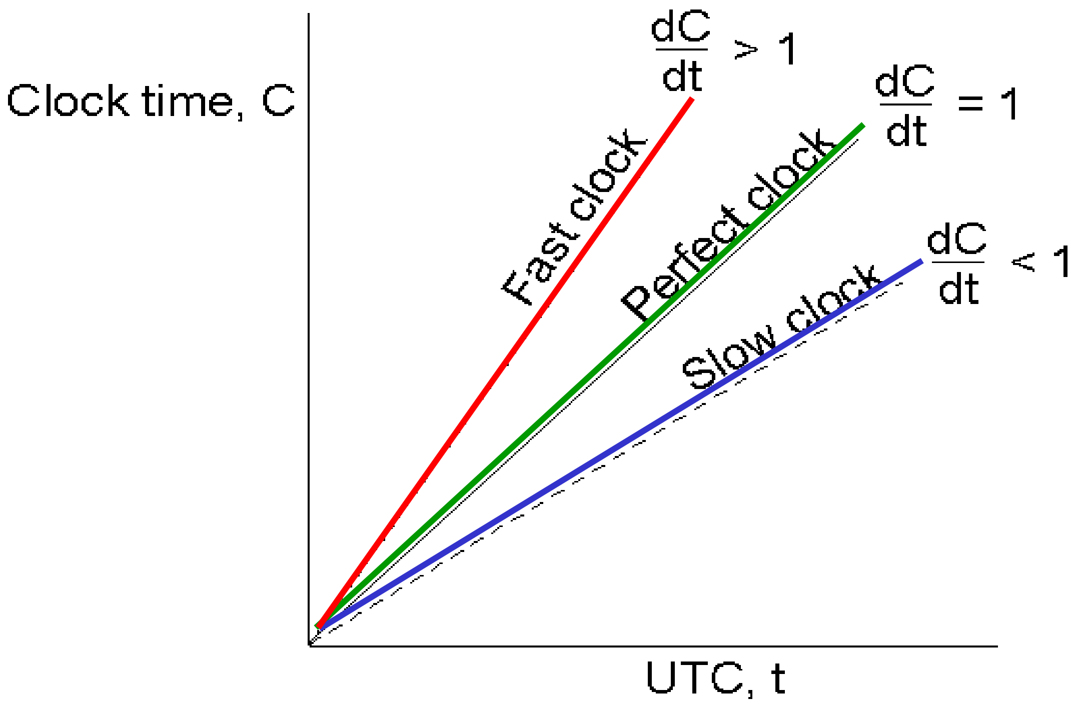
- Need to synchronize machines with a master or with one another

Clock Skew

- In a distributed system each computer has its own clock
- Each crystal will oscillate at slightly different rate
- Over time, the software clock values on the different computers are no longer the same
  - Example, ordinary quartz clocks drift by about 1 sec in 11-12 days

Clock Skew

$1 - \rho < dC/dt < 1 + \rho$ ρ *maximum drift rate*



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

Physical Clocks

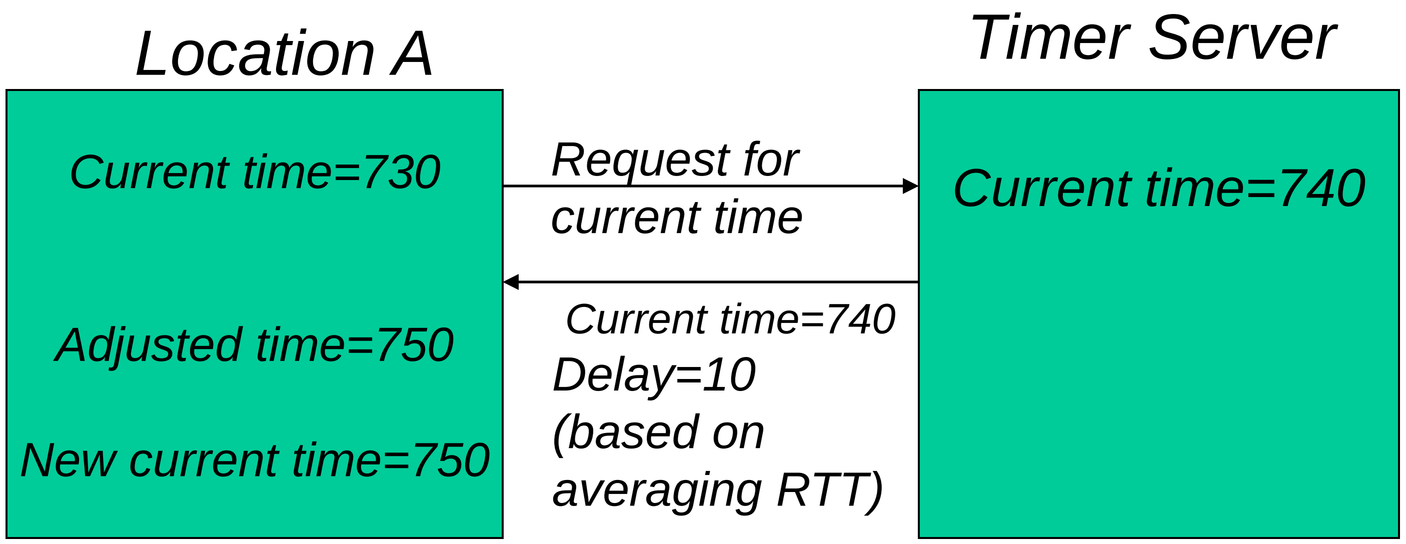
- There are two aspects:
  - Obtaining an accurate value for physical time
  - Synchronizing the concept of physical time throughout the distributed system

These can be implemented using **centralized** or **distributed** algorithms

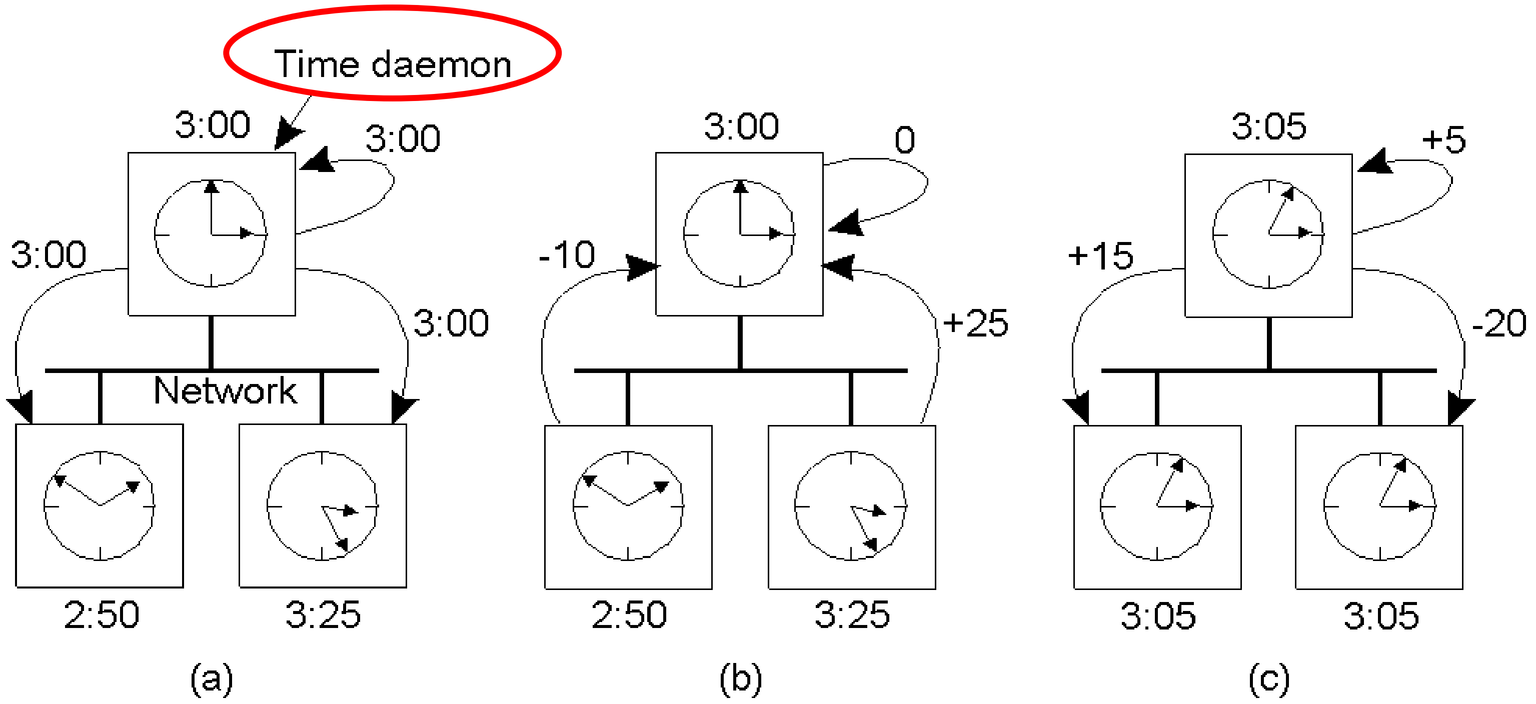
Physical Time Services

- Broadcast Based
- Request Driven

Request Driven (Cristian's Algorithm)



The Berkeley Algorithm
synchronization without time server



a) The time daemon asks all the other machines for their clock value
b) The machines answer
c) The time daemon tells everyone how to adjust their clock to the average

- no Universal Coordinated Time available

The Network Time Protocol (NTP)

- Distributed Approach
- Uses a hierarchy of time servers
  - Class 1 servers have highly-accurate clocks connected directly to atomic clocks, etc.
  - Class 2 servers get time from only Class 1 servers
  - Class 3 servers get time from any server
- Synchronization like Cristian's alg.
  - Modified to use multiple one-way messages instead of immediate round-trip

NTP Protocol

- NTP use UDP
- Widely used standard - based on Cristian’s algo
- **Clock can not go backward**
- Each message bears timestamps of recent events:
  - Local times of Send and Receive of previous message
  - Local times of Send of current message

Adjusting the Clock

- Two choices:
  - If T_A is slow, add ε to clock rate
    - To speed it up gradually
  - If T_A is fast, subtract ε from clock rate
    - To slow it down gradually

Logical Clocks

- **Why Logical Clocks?**

  It is difficult to utilize physical clocks to order events uniquely in distributed systems.

- For many problems, the internal consistency of clocks is important
  - Absolute time is less important
  - Use logical clocks

- The essence of logical clocks is based on the happened-before relationship presented by **Lamport**

Logical Clocks (Cont.)

- Synchronization based on “relative time”.
  - “relative time” may not relate to “real-time”.
- What’s important is that the processes in the Distributed System *agree on the ordering in which certain events occur*.
- Such “clocks” are referred to as *Logical Clocks*.

Two Versions

- **Lamport's logical clocks** synchronizes logical clocks
  - Can be used to determine an absolute ordering among a set of events
  - The order doesn't necessarily reflect causal relations between events

- **Vector clocks**: can capture the causal relationships between events

Lamport’s Logical Time

- Lamport defined a “happens-before” relation between events in a process
- "Events" are defined by the application
- Relation is undefined across processes that do not exchange messages

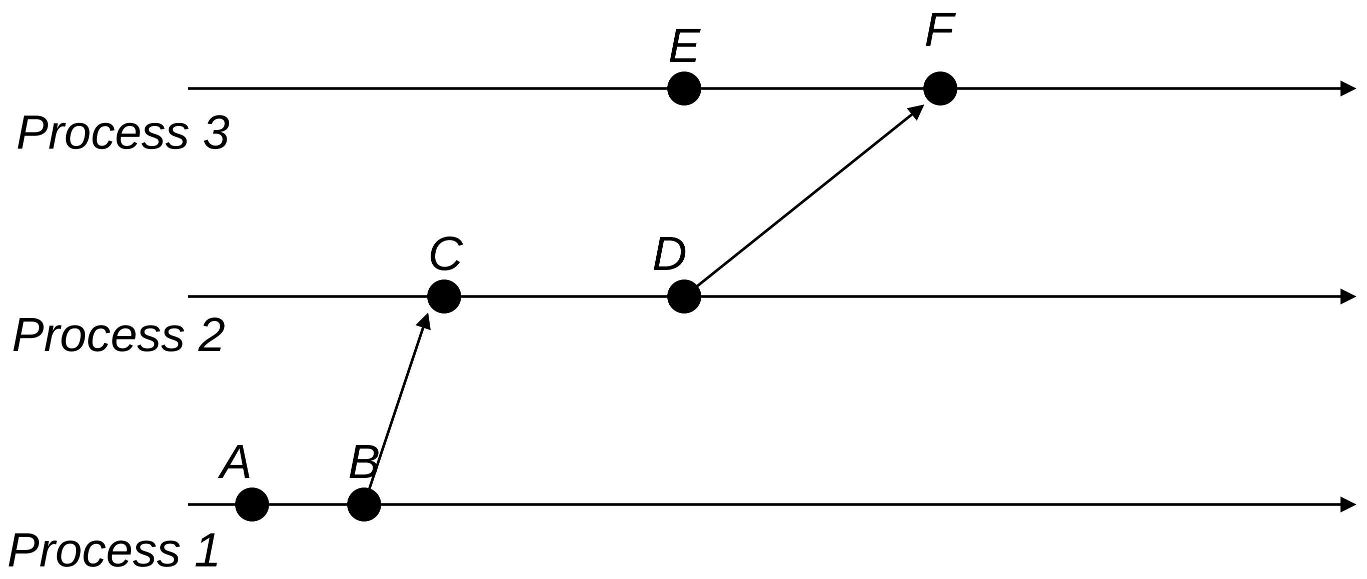
Happen-Before Relationship

- If two events, *a* and *b*, occurred at the same process, they occurred in the order of which they were observed.
  - That is, *a* -> *b*.

- If a sends a message to *b*, then *a* -> *b*.
  - That is, you cannot receive something before it is sent.
  - This relationship holds regardless of where events *a* and *b* occur.

- The happen-before relationship is transitive.
  - If a happens before b and b happens before c, then a happens before c. That is, if a -> b and b -> c, then a -> c.

For example



A>B>C>D>F E

Concurrent Events

- *Happens-before* defines a partial order of events in a distributed system.
- Some events can't be placed in the order
- *a* and *b* are concurrent (*a* || *b*) if
  !(*a* -> *b*) and !(*b* -> *a*).
- If a and b aren't connected by the happened-before relation, there's no way one could affect the other.

Lamport Logical Clocks

- **Goal:** method to assign a "timestamp" to an event *a* (call it C(a)), even in the absence of a global clock

- The method must guarantee that the clocks have certain properties, in order to reflect the definition of happens-before.

- Define a clock (event counter), C_i, at each process (processor) P_i.

- When an event a occurs, its timestamp ts(a) = C(a).

Correctness Conditions

- If a and b are in the same process, and a >b then C (a) < C (b)
- If a is the event of sending a message from Pi, and b is the event of receiving the message by Pj, then C_i (a) < C_j (b).
- The value of C must be increasing (time doesn't go backward).
  - Corollary: any clock corrections must be made by adding a positive number to a time

Implementation Rules

- Between any two successive events a & b in Pi, increment the local clock ($C_i = C_i + 1$)
  - thus $C_i(b) = C_i(a) + 1$
- When a message m is sent from P_i, set its time-stamp ts_m to C_i
- When the message is received at P_j the local time must be greater than ts_m.
  - The rule is ($C_j = \max\{C_j, ts_m\} + 1$).

Lamport's Logical Clocks

***Event a**: P1 sends m1 to P2 at t = 6,*
***Event b**: P2 receives m1 at t = 16.*
If C(a) is the time m1 was sent, and C(b) is the time m1 is received, do C(a) and C(b) satisfy the correctness conditions?

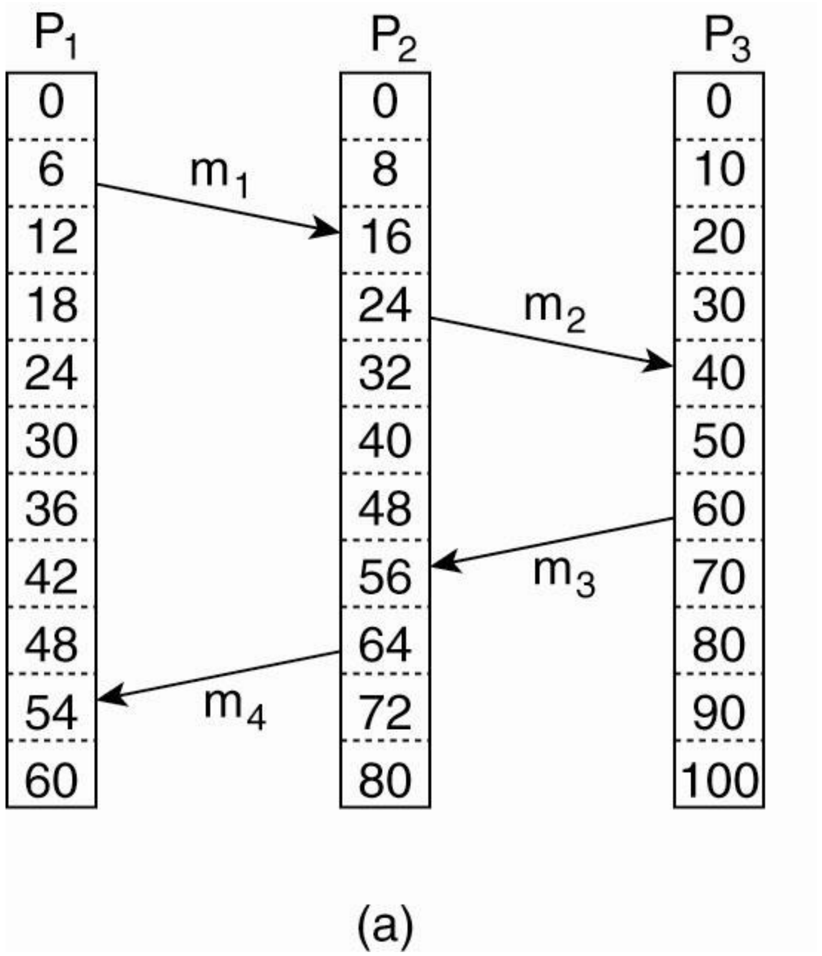
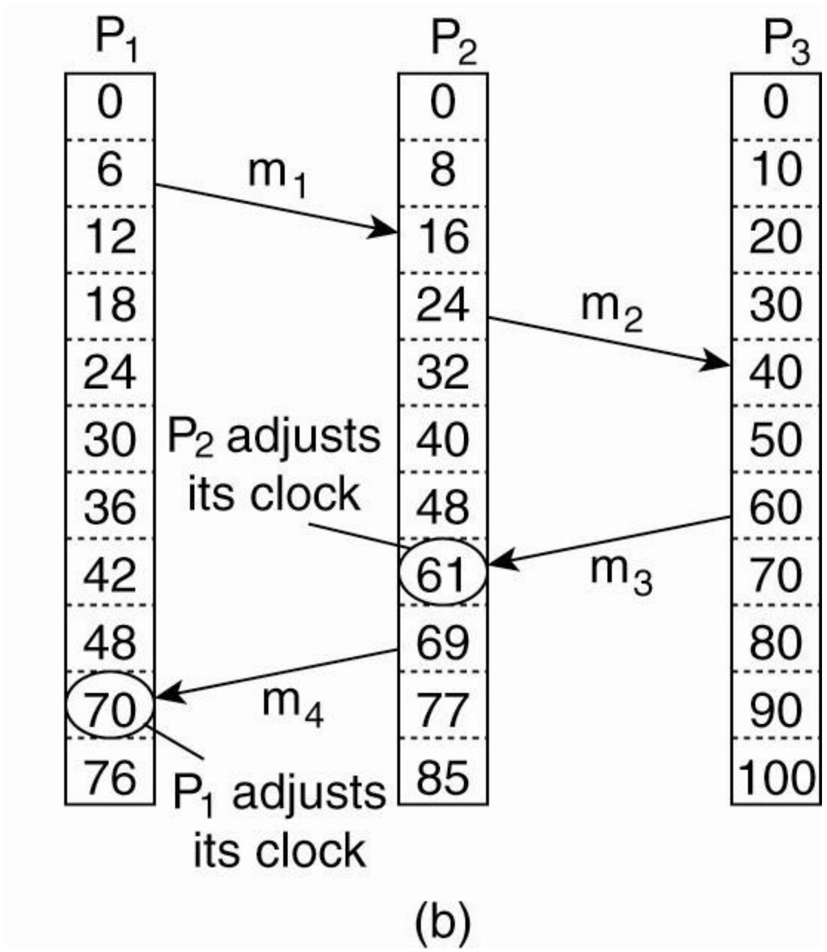


Figure 6-9. (a) Three processes, each with its own clock. The clocks "run" at different rates.

Lamport’s Logical Clocks



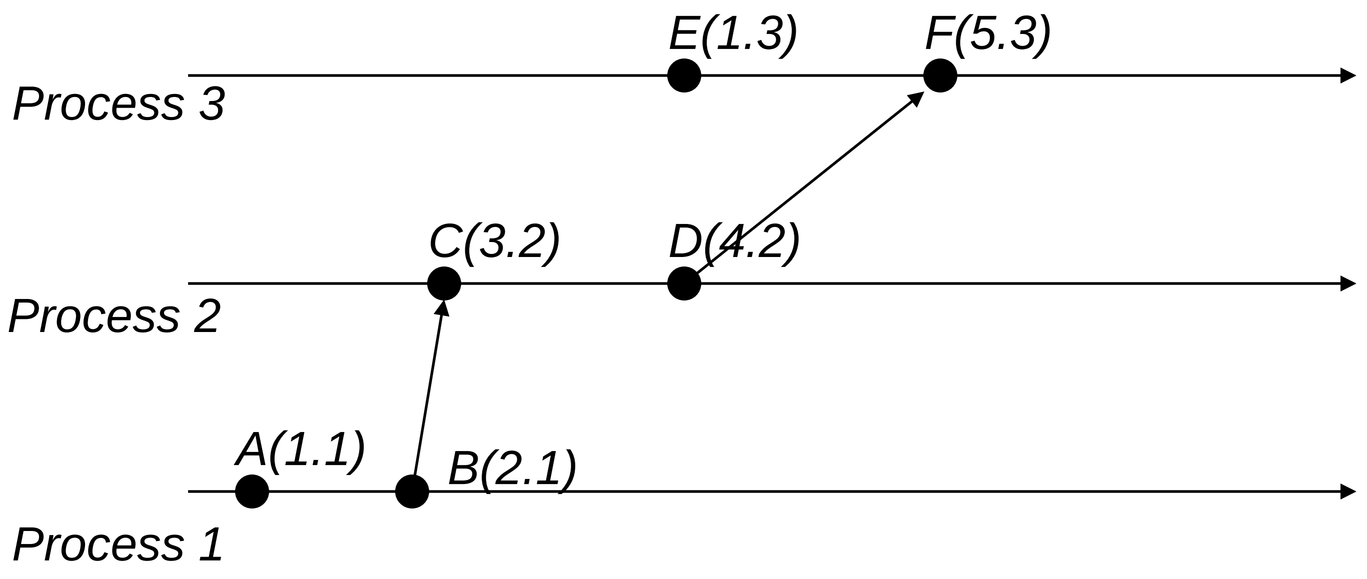
***Event c**: P3 sends m3 to P2 at t = 60*
***Event d**: P2 receives m3 at t = 56*
Do C(c) and C(d) satisfy the conditions?

Figure 6-9. (b) Lamport’s algorithm corrects the clocks.

A Total Ordering Rule
(does not guarantee causality)

- A total ordering of events can be obtained:
  - if we ensure that no two events happen at the same time (have the same timestamp).
- Why?
  - So all processors can agree on an unambiguous order.
- How?
  - Attach process number to the low-order end of time, separated by a decimal point;
    - e.g., event at time 40 at process P1 is 40.1, event at time 40 at process P2 is 40.2

Total Ordering



A>E>B>C>D>F

Causality

- *Causally related events*:
  - Event *a* may causally affect event *b*
    - if *a* -> *b* then C(A) < C(B)
  - Reverse is not true
    - Nothing can be said about events by comparing time-stamps
  - If neither of the above relations holds, then there is no causal relationship between a & b.
    - We say that a || b (a and b are concurrent)

Causality

- Need to maintain *causality*
  - If a -> b then a is casually related to b
  - *Causal delivery*: If send(m) -> send(n) => deliver(m) -> deliver(n)
  - Capture causal relationships between groups of processes
  - Need a time-stamping mechanism such that:
    - If *T(A)* < *T(B)* then *A* should have causally preceded *B*

Vector Clock Rationale

- Lamport clocks limitation:
  - If (a->b) then $C(a) < C(b)$ but
  - If $C(a) < C(b)$ then we cannot know the relation
  - In other words, you cannot look at the clock values of events on two different processors and decide which one “happens before”.

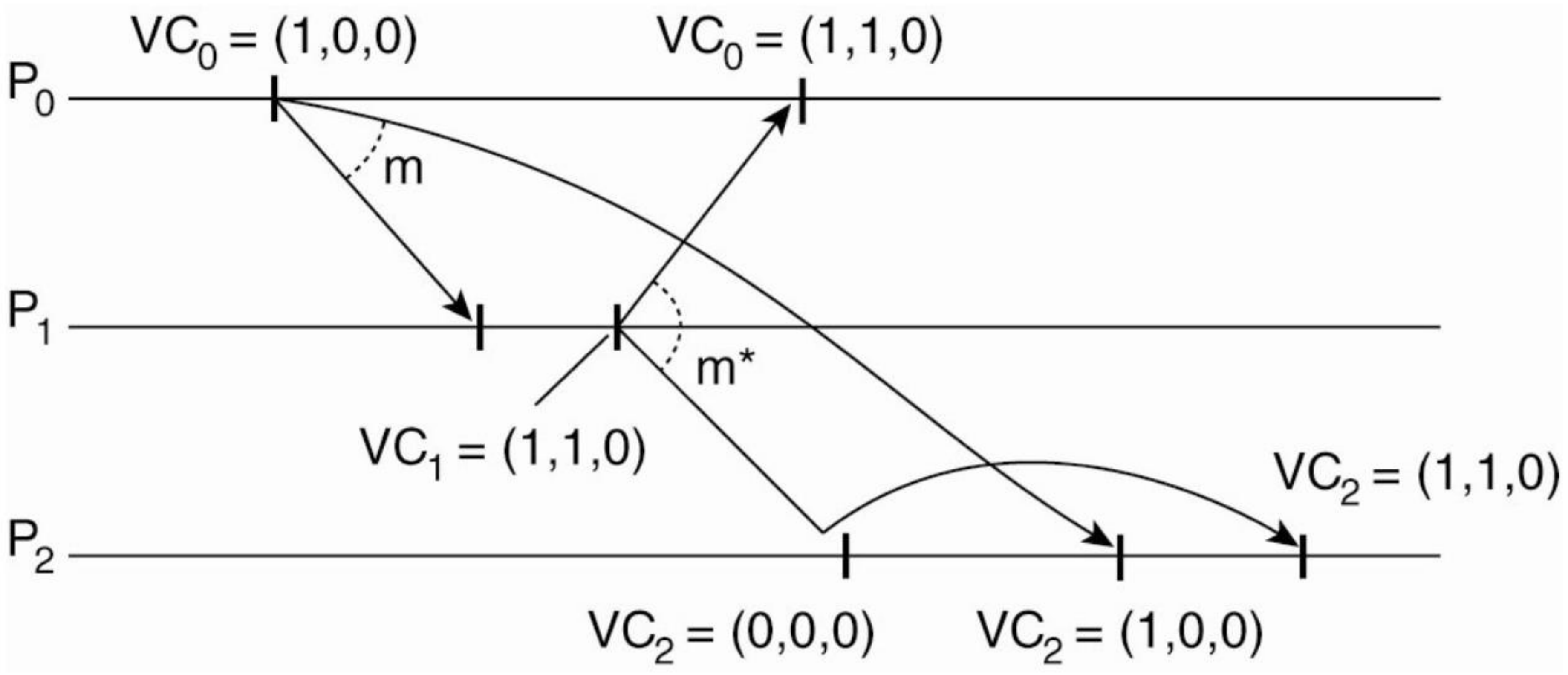
Vector Clocks – How They Work

- Each processor keeps a vector of values, instead of a single value.
- VC_i is the clock at process i; it has a component for each process in the system.
  - $VC_i[i]$ corresponds to P_i's local "time".
  - $VC_i[j]$ represents P_i's knowledge of the "time" at P_j (the # of events that P_i knows have occurred at Pj
- Each processor knows its own "time" exactly and updates the values of other processors' clocks based on timestamps received in messages

Implementation Rules

- IR1: Increment $VC_i[i]$ before each new event.
- IR2: When process i send a message *m,* it sets *m*'s (vector) timestamp to VC_i (after incrementing $VC_i[i]$)
- IR3: When a process receives a message it does a component-by-component comparison of the message timestamp to its local time and picks the maximum of the two corresponding components.
- Then deliver the message to the application.

Enforcing Causality Using Vector Timestamps



Review

- Physical clocks: hard to keep synchronized

- Logical clocks: can provide some notion of relative event occurrence

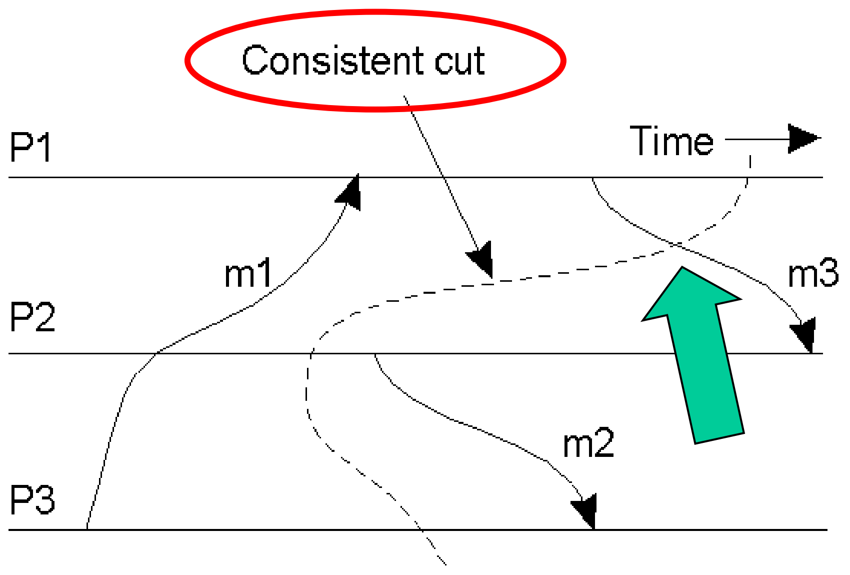
- Lamport’s logical clocks
  - happened-before relation defines relations
  - logical clocks – don’t capture causality
  - total ordering relation

- Vector clocks
  - Unlike Lamport clocks, vector clocks capture causality
  - Have a component for each process in the system

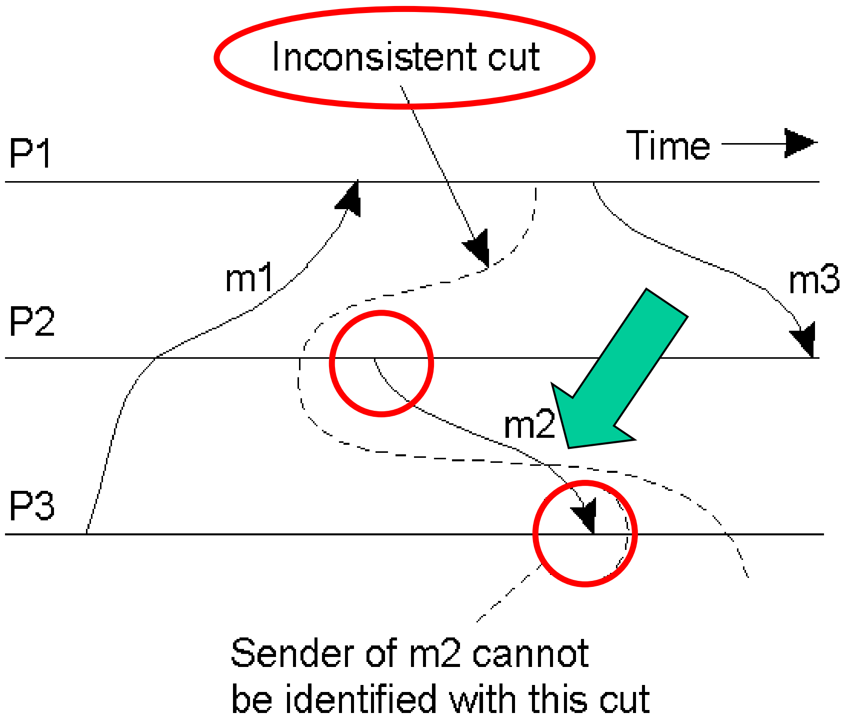
Global State

It is the local state of each process, together with the messages currently in transit in a distributed system

A distributed snapshot reflects a **consistent** global state



(a)

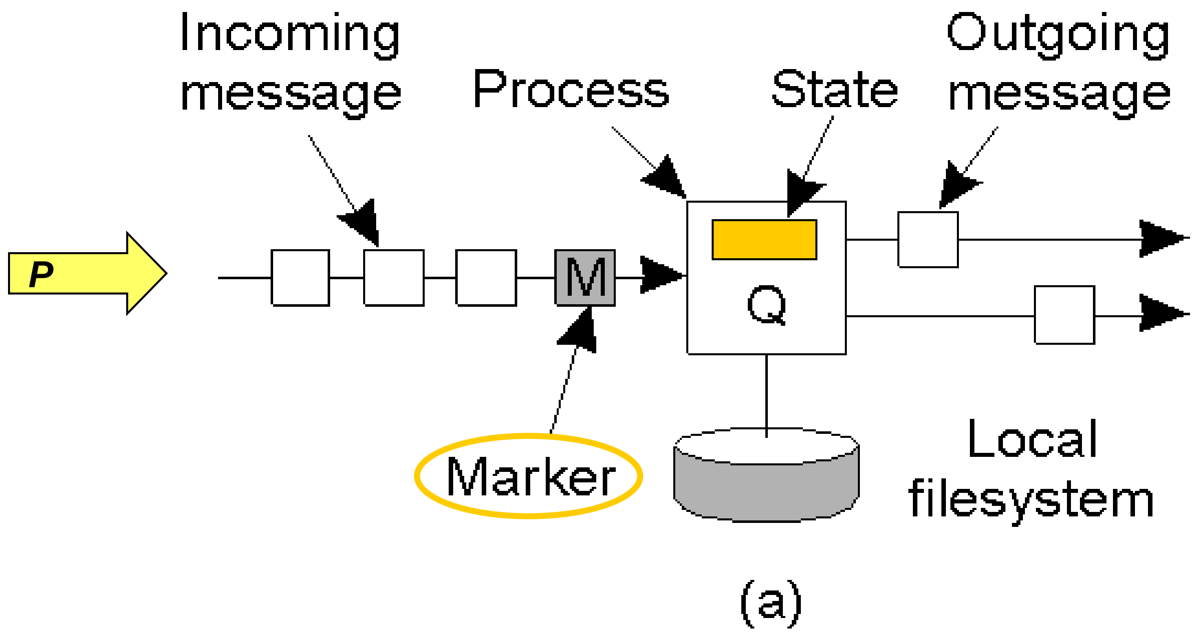


(b)

Global State

Knowledge of the state in which a distributed system currently is, is useful

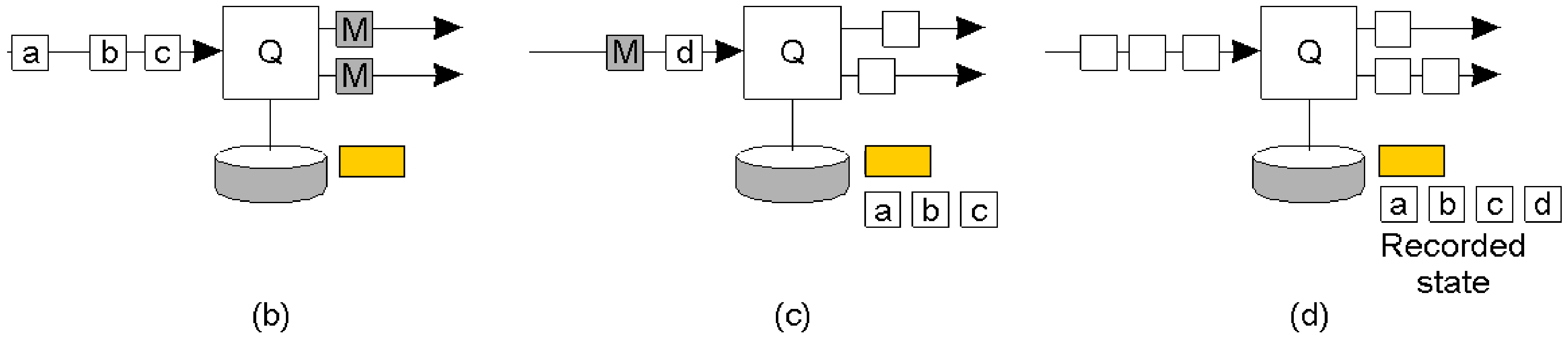
Any process P may initiate the algorithm recording its local state.



- Organization of a process and channels for a distributed snapshot

a) A process P sends a marker along *each* of its outgoing channels

Global State



b) Process Q receives a marker for the first time, records its local state, and sends a marker along each outgoing channel

c) Q records all incoming message

d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel

- A process has finished its part of the algorithm when it has received a marker along each of its incoming channels and processed each one.
- Many snapshots may be in progress at the same time.

Election Algorithms

- Many distributed algorithms require one process to act as coordinator, initiator or otherwise perform some special role.

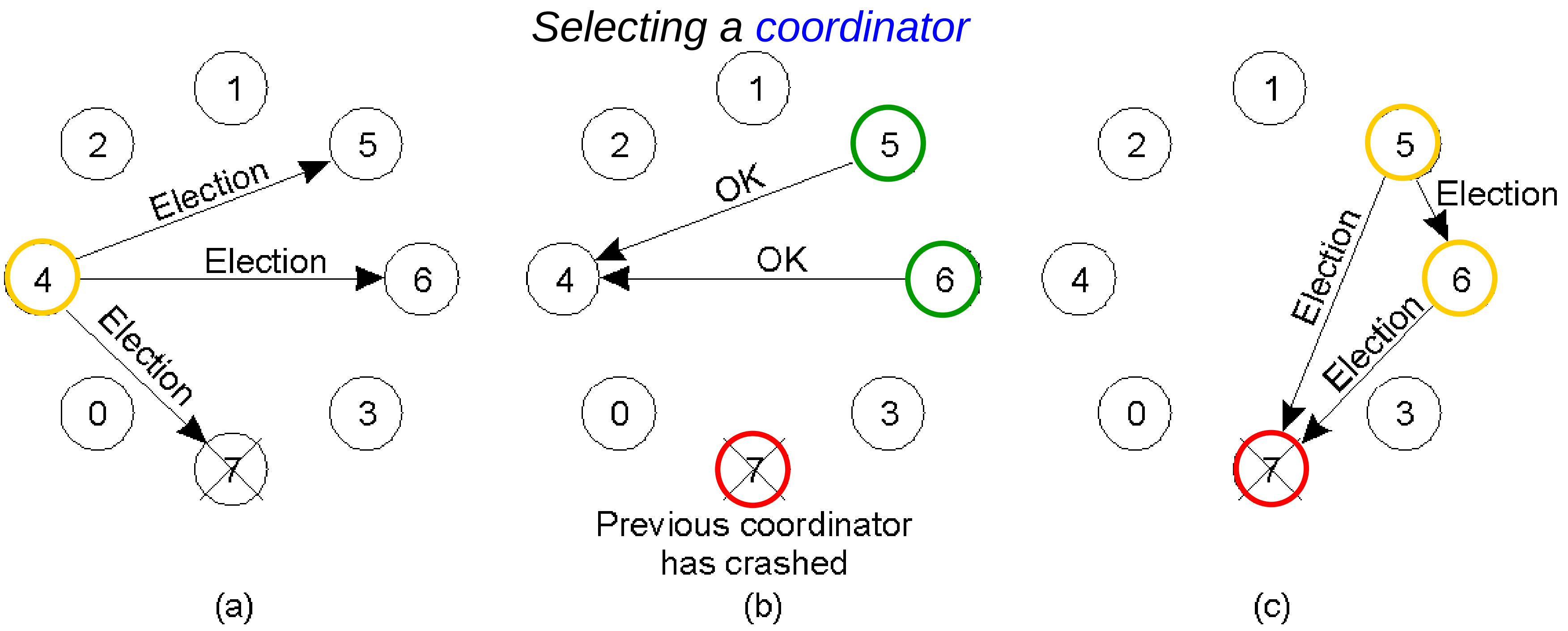
- The question is how to select this special process **dynamically.**
  - In many systems the coordinator is chosen by hand (e.g. file servers).
    - This leads to centralized solutions -single point of failure

Bully Algorithm Details

- Each process has a unique numerical ID
- Processes know the Ids and address of every other process
- Communication is assumed reliable
- *Key Idea*: select process with highest ID
- Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *election*, *OK*, *I won*
- Several processes can initiate an election simultaneously
- – Need consistent result

Election Algorithms

The Bully Algorithm (1/2)



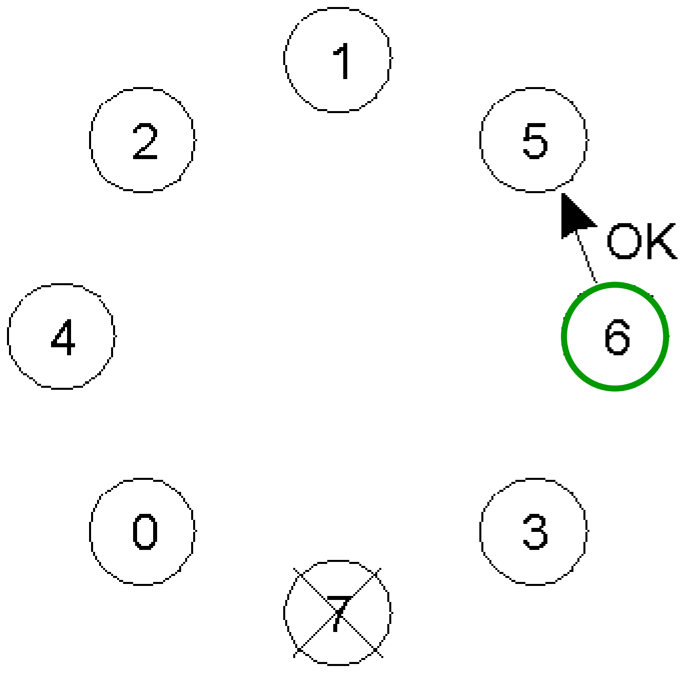
- The bully election algorithm

a) Process 4 holds an election

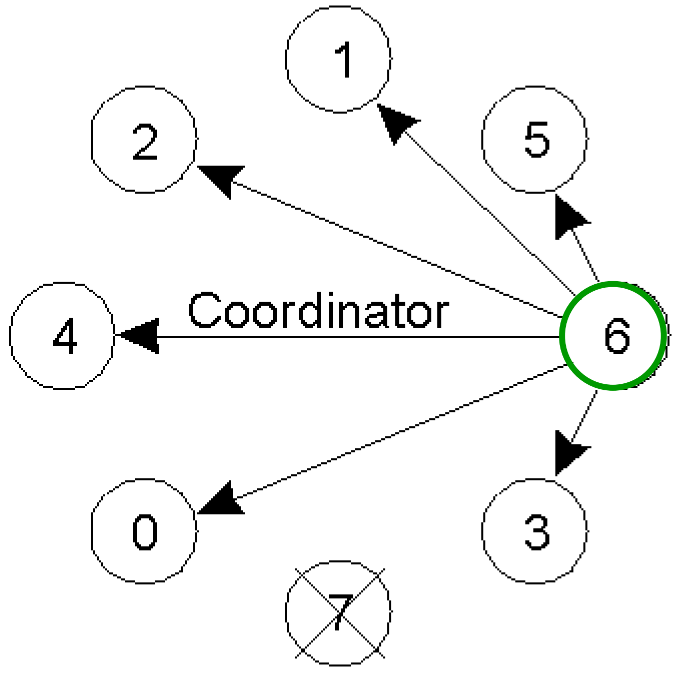
b) Process 5 and 6 respond, telling 4 to stop

c) Now 5 and 6 each hold an election

The Bully Algorithm (2/2)



(d)

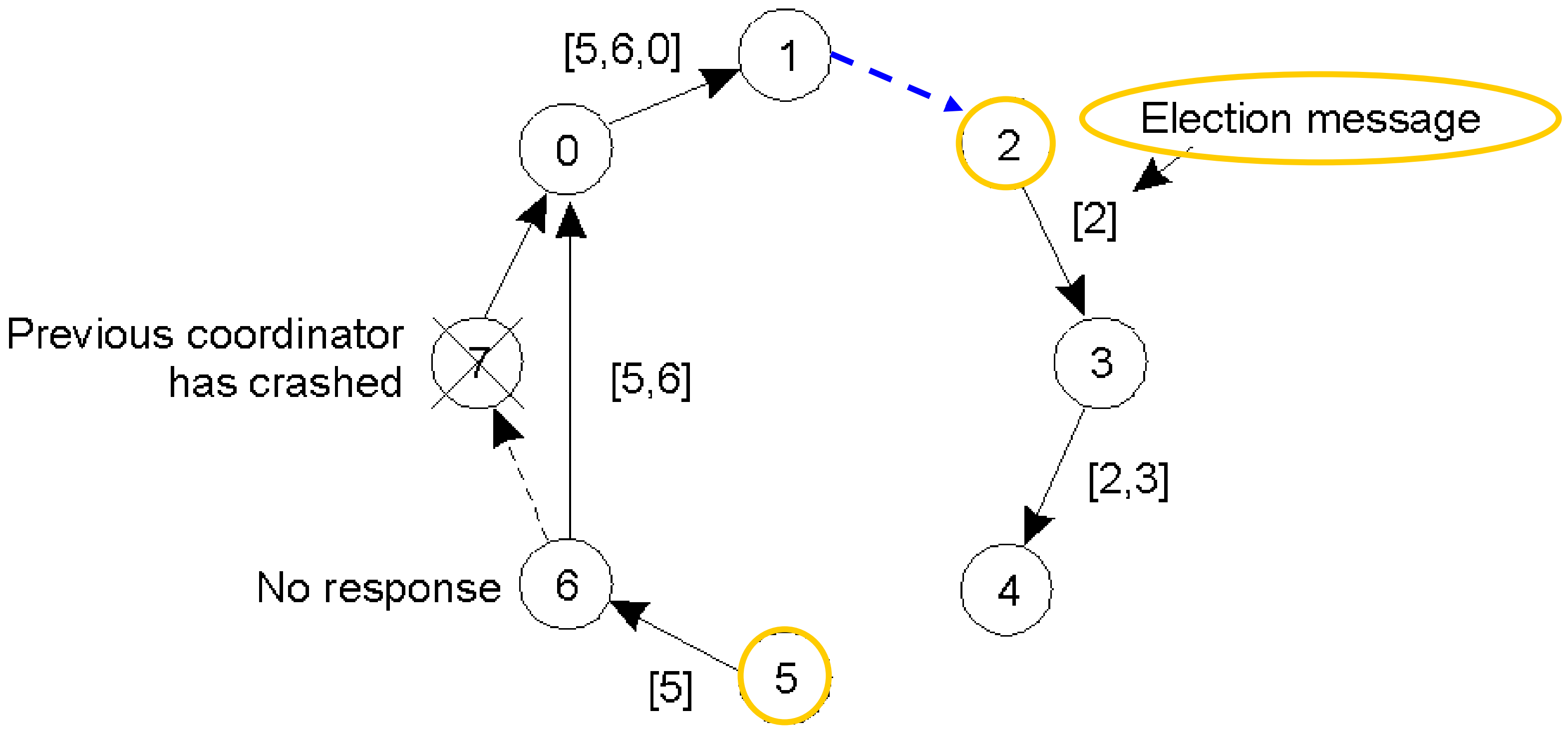


(e)

d) Process 6 tells 5 to stop
e) Process 6 wins and tells everyone

A Ring Algorithm

Each process knows who its successor is.



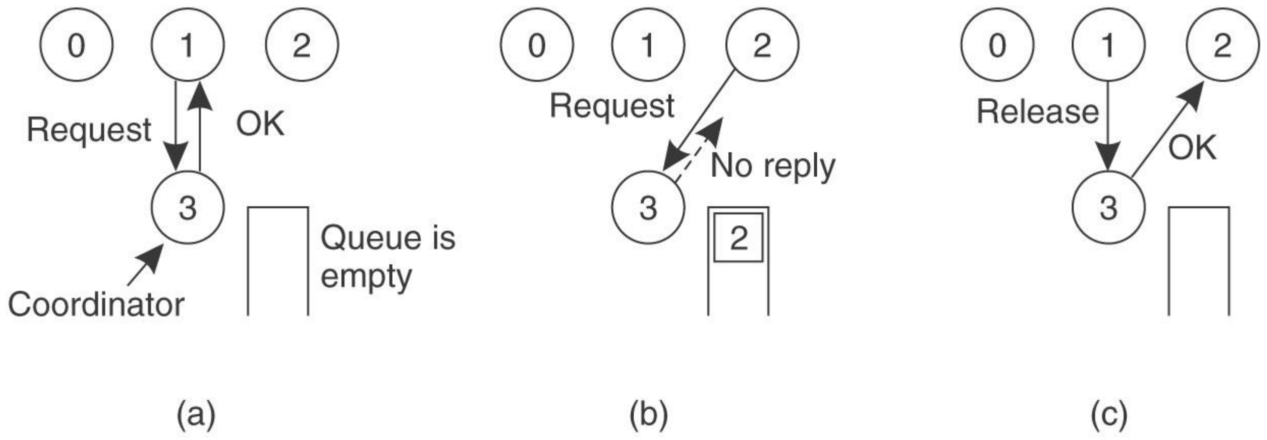
- Election algorithm using a ring (without token).

Mutual Exclusion

- Prevent inconsistent usage or updates to shared data

- Following approaches:
    - *Centralized Algorithm*
    - *Decentralized Algorithm*
    - *Distributed Algorithm*
    - *Token Ring Algorithm*

Centralized Approach



- One process is elected *coordinator* for a resource
- All others ask *permission from coordinator*.
- To obtain exclusive access: send request, await reply
- To release: send release message
- Possible responses
  - Okay; denied (ask again later); none (caller waits)

Centralized Approach

- Advantages
  - Mutual exclusion guaranteed by coordinator
  - “Fair” sharing possible without starvation
  - Simple to implement
- Disadvantages
  - Single point of failure (coordinator crashes)
  - Performance bottleneck
  - How do you detect a dead coordinator?
    - A process can not distinguish between “lock in use” from a dead coordinator
    - No response from coordinator in either case

Decentralized Approach

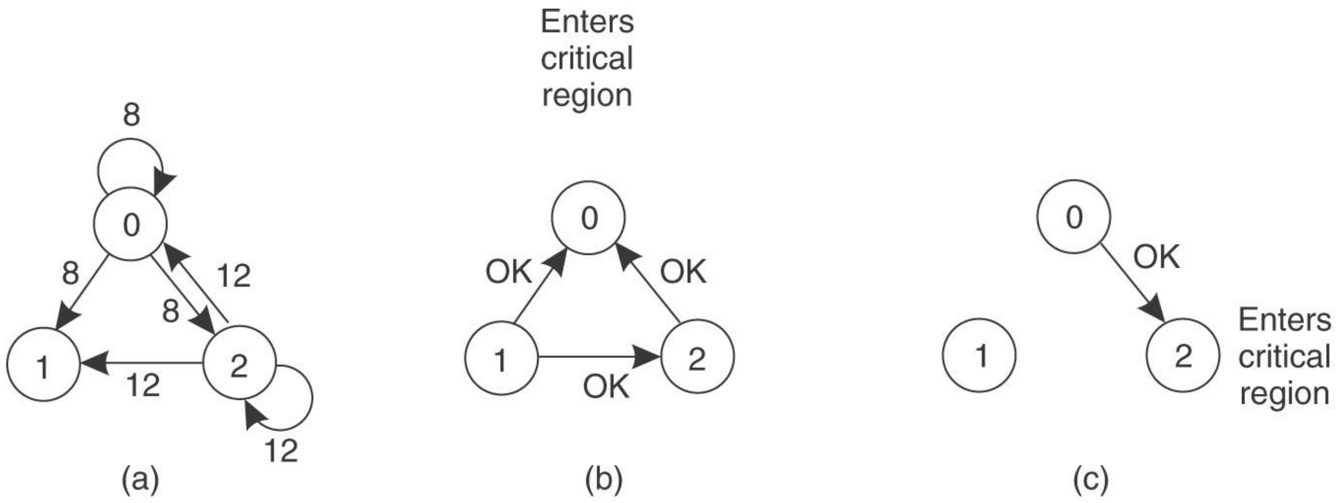
- *Use voting*
- *n* coordinators; ask all
    - E.g., *n* replicas
- Must have agreement of $m > n/2$
- Advantage
    - No single point of failure
- Disadvantage
    - *Lots* of messages
    - Really messy

Distributed Approach

- Requestor sends reliable messages to *all* other processes (including self)
    - Waits for *OK* replies from all other processes

- Replying process
    - If not interested in resource, reply *OK*
    - If currently using resource, queue request, don’t reply
    - If interested, then reply *OK* if requestor is earlier; Queue request if requestor is later

Distributed Approach



- Process *0* and Process *2* want resource
- Process *1* replies *OK* because not interested
- Process *0* has lower time-stamp, thereby goes first
- When process 0 is done, it sends an OK also, so 2 can now enter the critical region

Distributed Approach

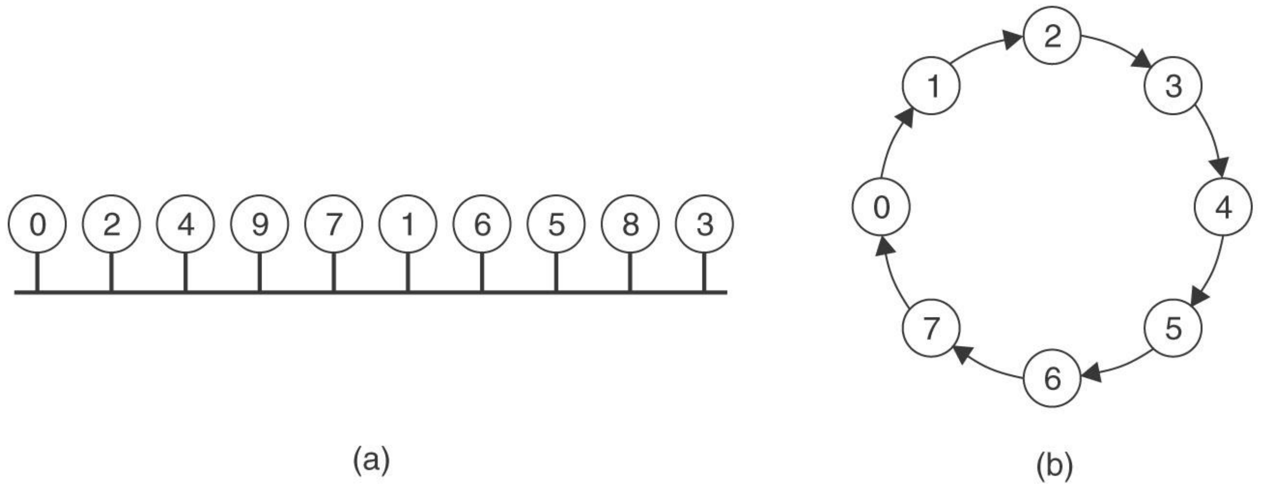
- Advantage
  - No central bottleneck
  - Fewer messages than Decentralized approach

- Disadvantage
  - *n* points of failure
    - i.e., failure of one node to respond locks up system
  - All processes are involved in all decisions
    - Any overloaded process can become a bottleneck

Token system

- Organize processes in logical *ring*
- Each process knows successor
- *Token* is passed around ring
    - If process is interested in resource, it waits for token
    - Releases token when done
- If node is dead, process skips over it
    - Passes token to successor of dead process

Token system (continued)



a) An unordered group of processes on a network.

b) A logical ring constructed in software.

- Advantages
    - Fairness, no starvation
    - Recovery from crashes if token is not lost
- Disadvantage
    - Crash of process holding token
    - Difficult to detect; difficult to regenerate *exactly* one token

Data Replication

- Data replication: common technique in distributed systems
- Why?
  - Enhances reliability.
    - If one replica is unavailable or crashes, use another
    - Protect against corrupted data
  - Improves performance.
- Replicas allows remote sites to continue working in the event of local failures.
- Replicas allow data to reside close to where it is used.
  - This directly supports the distributed systems goal of enhancing *scalability*.

Replication (cont.)

- If there are many replicas of the same thing, how do we keep all of them up-to-date or consistent ?
- Consistency can be achieved in a number of ways.
- Challenge:
  - It is not easy to keep all those replicas *consistent*.

CAP Theorem

- Conjecture by Eric Brewer at PODC 2000 conference
  - It is impossible for a web service to provide all three guarantees:
- Consistency (nodes see the same data at the same time)
- Availability (node failures do not affect the rest of the system)
- Partition-tolerance (system can tolerate message loss)

- A distributed system can satisfy any two, but not all three, at the same time
- Conjecture was established as a theorem in 2002 (by Lynch and Gilbert)

References

- Distributed Systems: Principles and Paradigms by Tanenbaum and van Steen, Chapter 4.3~4.5.

- Distributed Systems: Principles and Paradigms by Tanenbaum and van Steen, Chapter 5.
- Distributed systems: principles and paradigms, Andrew S. Tanenbaum, Maarten Van Steen, Ch 6.1-6.2, 6.4-6.6.

- Distributed Systems, Gregory Kesden, Carnegie Mellon University