

DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 1
Introduction

Outline

- **Definition of distributed system**
- Goals of distributed system
- Challenges and approaches of distributed system
- Types of distributed systems

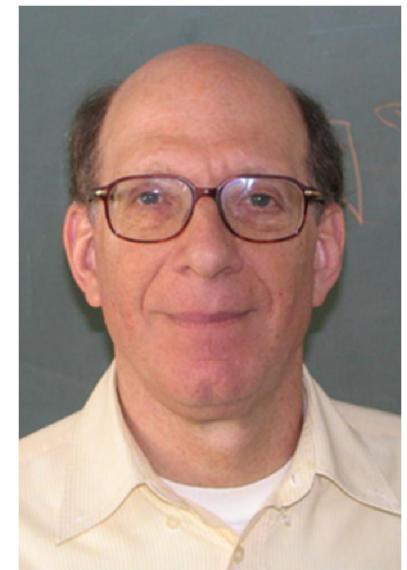
Two Tendencies

- The first was the development of powerful microprocessors.
- The second development was the invention of high-speed computer networks
- The result of these technologies is that it is now not only feasible, but easy, to put together computing systems composed of large numbers of computers connected by a high-speed network.
 - **distributed systems vs centralized systems**

Definition of a Distributed System

A distributed system is (Tannenbaum):

- A collection of independent computers that appears to its users as a single coherent system.
- Collaboration(独立计算机，但相互联系)
- Single system (单一系统)
- Autonomous (自治)



Andrew S. Tanenbaum

Distributed System Characteristics

- Differences between the various computers and the ways in which they communicate are mostly **hidden** from users.
 - The same holds for the internal organization of the distributed system.
- Users and applications can interact with a distributed system in a **consistent** and **uniform** way, regardless of where and when interaction takes place.

Distributed System Characteristics

- Distributed systems should be relatively easy to expand or scale
 - independent computers is easy to add
- Distributed systems should normally be continuously available
 - Even if some parts may be temporarily out of order

Distributed System as middleware

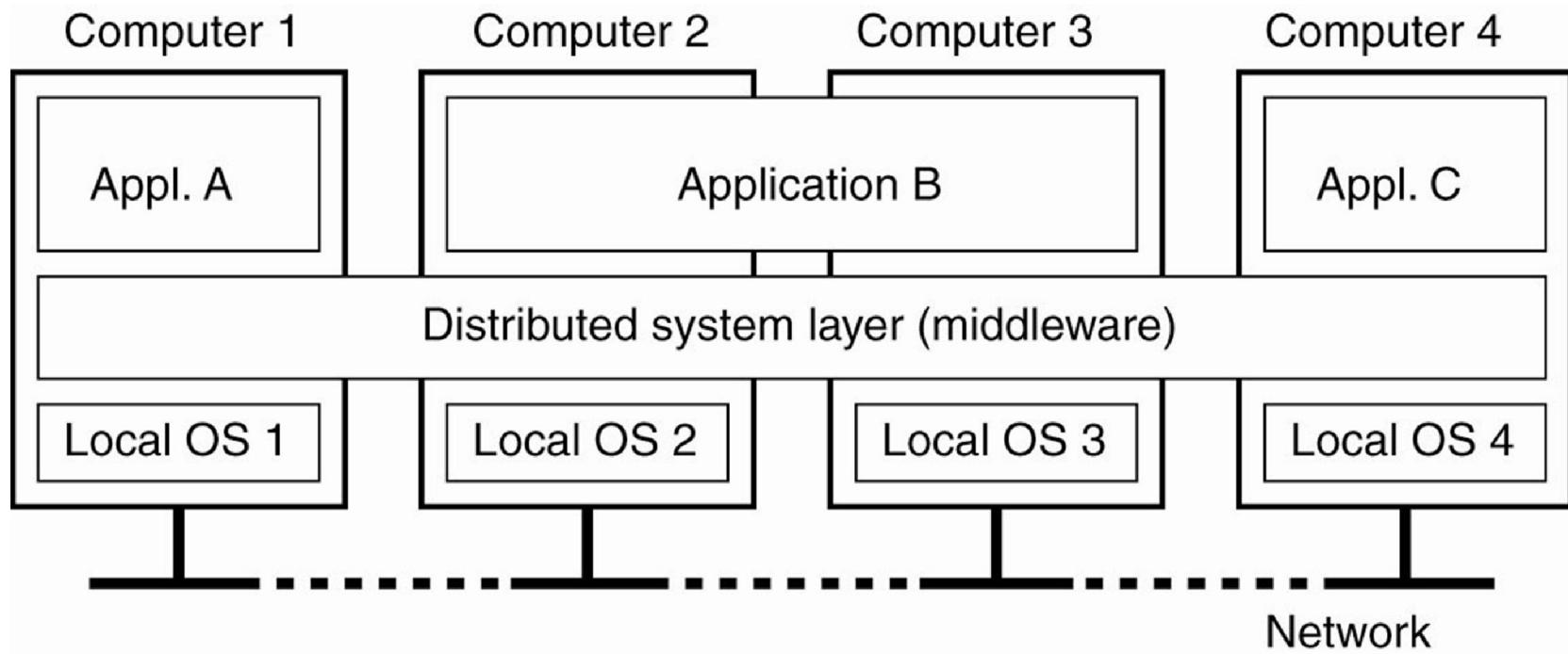


Figure 1-1. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

Parallel algorithms

- It can be **executed a piece at a time** on many different processing devices, and then **combined** together again at the end to get the correct result.
- As opposed to a traditional **serial algorithm**.

Distributed algorithms

- Sub-type of parallel algorithm, typically executed concurrently.
- Designed to run on computer hardware constructed from interconnected processors, especially in distributed system.
 - Leader election
 - Distributed search
 - Mutual exclusion
 - Resource allocation

Decentralized algorithms

- Sub-type of distributed algorithm.
- No machine has complete information about the system state.
- Machines make decisions based only on local information,
- Failure of one machine does not ruin the algorithm.

Outline

- Definition of distributed system
- **Goals of distributed system**
- Challenges and approaches of distributed system
- Types of distributed systems

Goals of Distributed System

- It should make resources easily accessible;
 - Access remote resources
 - Share resources
- It should reasonably hide the fact that resources are distributed across a network;
 - Distribution transparency
- It should be open;
- It should be scalable

1. Transparency

A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent.

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

Figure 1-2. Different forms of transparency in a distributed system (ISO, 1995).

Other forms:

Parallelism – Hide the number of nodes working on a task

Size – Hide the number of components in the system

Revision – Hide changes in software/hardware versions

Transparency in a reasonable degree

- Newspaper to appear in your mailbox before 7 A.M. local time, but where you are?
- Naturally network delay;
- Never attempting to mask a transient server failure;
- Change should be propagated to all copies before allowing
- Replication of copies may take seconds, which something that cannot be hidden from users.
- It is better to send the print job to a busy nearby printer rather than to an idle one in a quite far country

2. Openness

- An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services.
 - Protocols in computer networks
 - Interface Definition Language (IDL) in distributed system

Openness in another way

- **Interoperability** characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard.
- **Portability** characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A.

3. Scalability

- **Size scalability** : a system can be scalable with respect to its size;
- **Geographical scalability**: a geographically scalable system is one in which the users and resources may lie far apart.
- **Administrative scalability**: a system can be administratively scalable, it can still be easy to manage even if it spans many independent administrative organizations.

3.1 Problems of Size Scalability

- Unlimited processing and storage capacity, communication with that server will eventually prohibit further growth;
- Using only a single server is sometimes unavoidable
 - highly confidential information
- The larger the system, the larger the uncertainty of clock synchronization.

3.2 Problems of Geographical scalability

- Synchronous communication cost is huge in wide-area systems
- Communication in wide-area networks is inherently unreliable

3.3 Problems of Administrative scalability

- Conflicting policies with respect to resource usage and payment, management, and security in different domain
 - (Beyond the range of this course).

4. Pitfalls

- The network is reliable.
- The network is secure.
- The network is homogeneous.
- The topology does not change.
- Latency is zero.
- Bandwidth is infinite.
- Transport cost is zero.
- There is one administrator (at one time).

Outline

- Definition of distributed system
- Goals of distributed system
- Challenges and approaches of distributed system
- Types of distributed systems

Challenges

- Performance
- Concurrency
- Transparency
- Failures
- Scalability
- Heterogeneity
- Openness
- Security
- Multiplicity of ownership, authority
- Quality of service/user experience
- System updates
- Debugging

Approaches

- Virtual clocks
- Group communication
- Heartbeats/failure detection, group membership
- Distributed agreement, snapshots
- Leader election
- Transaction protocols
- Redundancy, replication, caching
- Indirection naming
- Distributed mutual exclusion
- Middleware, modularization, layering
- Cryptographic protocols

Outline

- Definition of distributed system
- Goals of distributed system
- Challenges and approaches of distributed system
- **Types of distributed systems**

Types of Distributed Systems

- Types of distributed systems
 - Distributed Computing Systems
 - Cluster Computing Systems
 - Grid Computing Systems
 - Distributed Information Systems
 - Transaction Processing Systems
 - Enterprise Application Integration
 - Distributed Pervasive Systems
 - Home Systems
 - Electronic Health Care Systems
 - Sensor Networks

1. Distributed Computing Systems

- An important class of distributed systems is the one used for high-performance computing tasks
 - Cluster Computing Systems (homogeneity)
 - Similar workstations or PCs,
 - High speed local-area network
 - Running the same operating system.
 - Grid Computing Systems (heterogeneity)
 - Federation of computer systems
 - Each system fall under a different administrative domain
 - System's hardware, software, and deployed network technology may be very different

Cloud Computing may both of them

1.1 Cluster Computing Systems

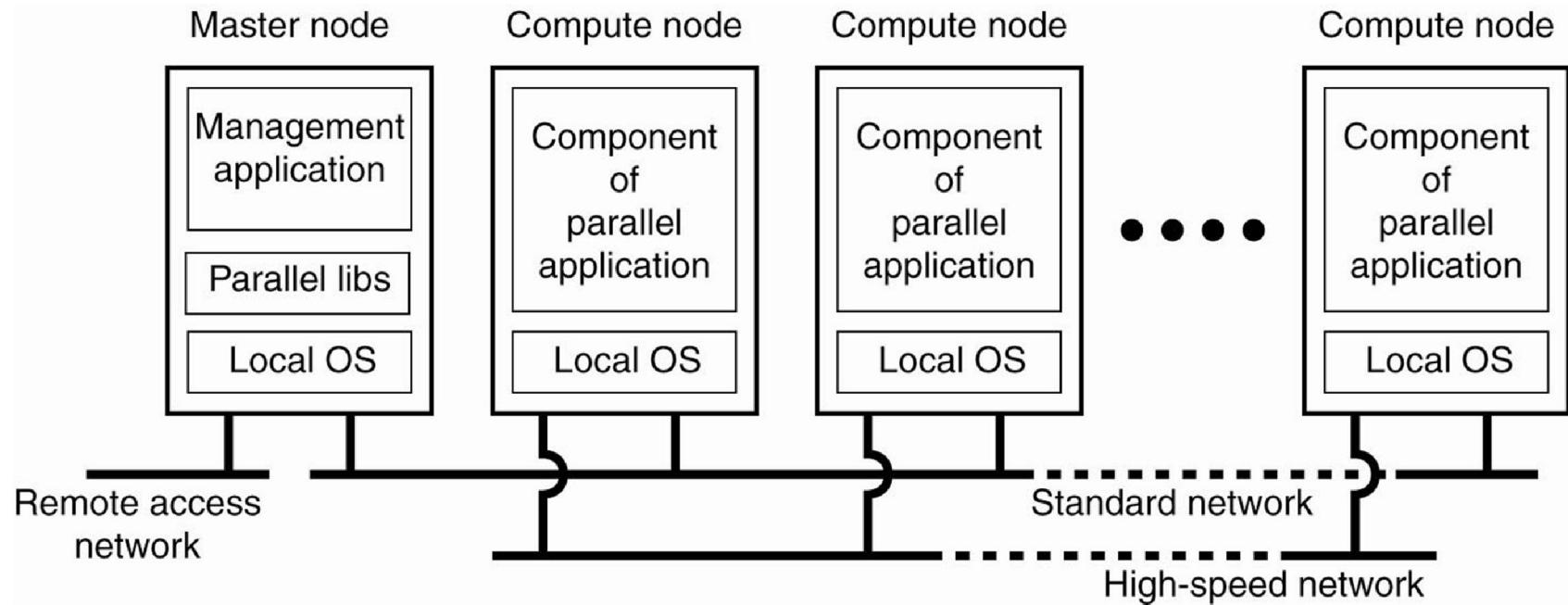


Figure 1-6. An example of a cluster computing system.

1.2 Grid Computing Systems

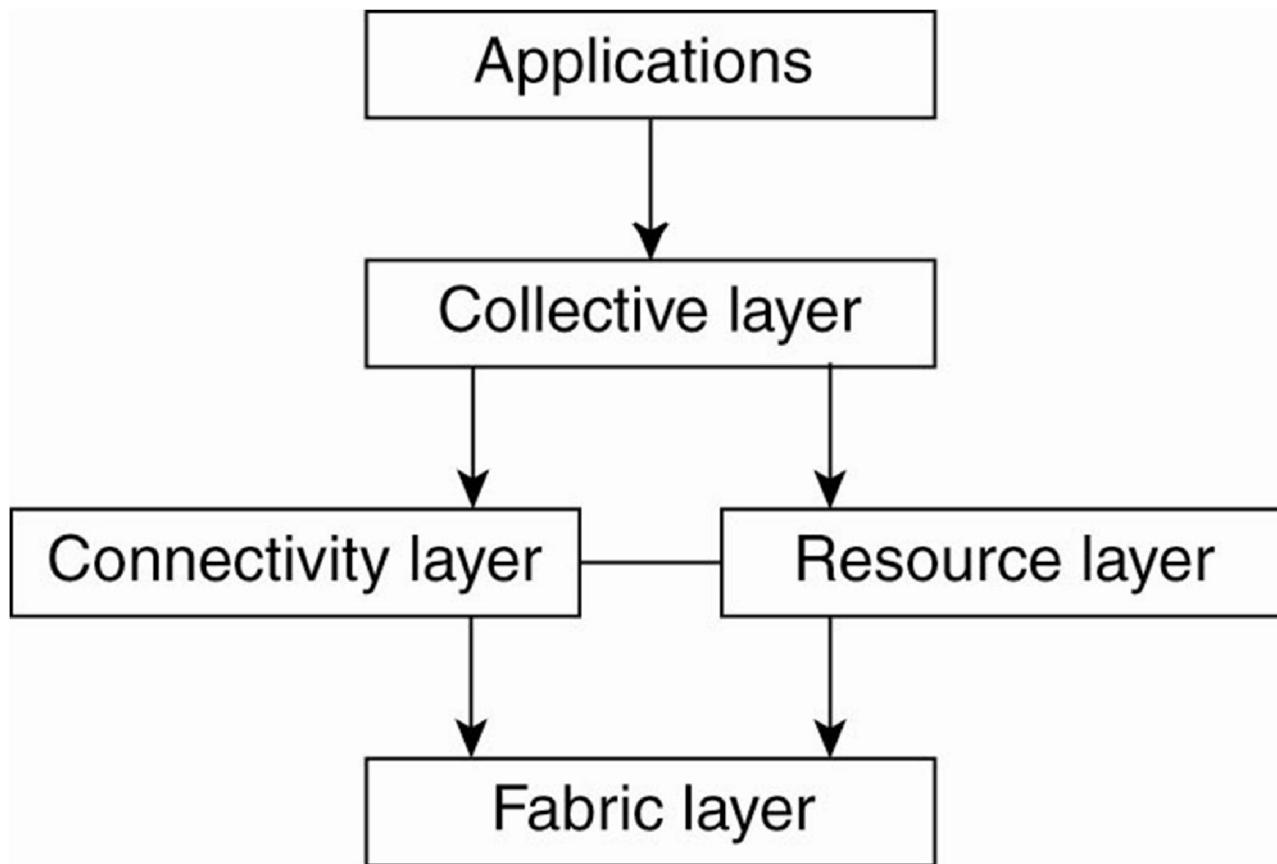
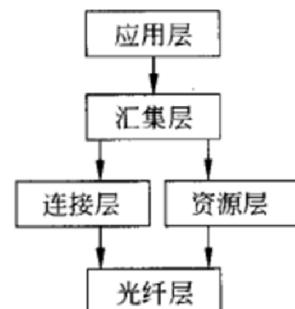


Figure 1-7. A layered architecture for grid computing systems.



2. Distributed Information Systems

- Transaction Processing Systems
 - The key idea was that all, or none of the requests would be executed
 - Data is integrated
- Enterprise Application Integration (EAI)
 - Let applications communicate directly with each other
 - Applications are integrated

2.1 Transaction Processing Systems

Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Figure 1-8. Example primitives for transactions.

Transactions

Characteristic properties of transactions:

- **Atomic**: To the outside world, the transaction is undivided and happens indivisibly.
- **Consistent**: The transaction does not violate system invariants.
- **Isolated**: Concurrent transactions do not interfere with each other.
- **Durable**: Once a transaction commits, the changes are permanent.

Known as **ACID** properties

Atomic

- Each transaction either happens completely, or not at all, and if it happens, it happens in a single indivisible, instantaneous action. (As Database Transactions)
- While a transaction is in progress, other processes (whether or not they are themselves involved in transactions) cannot see any of the intermediate states.

Consistent

- If a system has certain invariants that must always hold
 - If they held before the transaction, they will hold afterward too.
- Banking system
 - Law of conservation of money

Isolated

- If two or more transactions are running at the same time
 - The final result looks as though all transactions ran sequentially in some (system dependent) order

Durable

- A top-level transaction has a commit point.

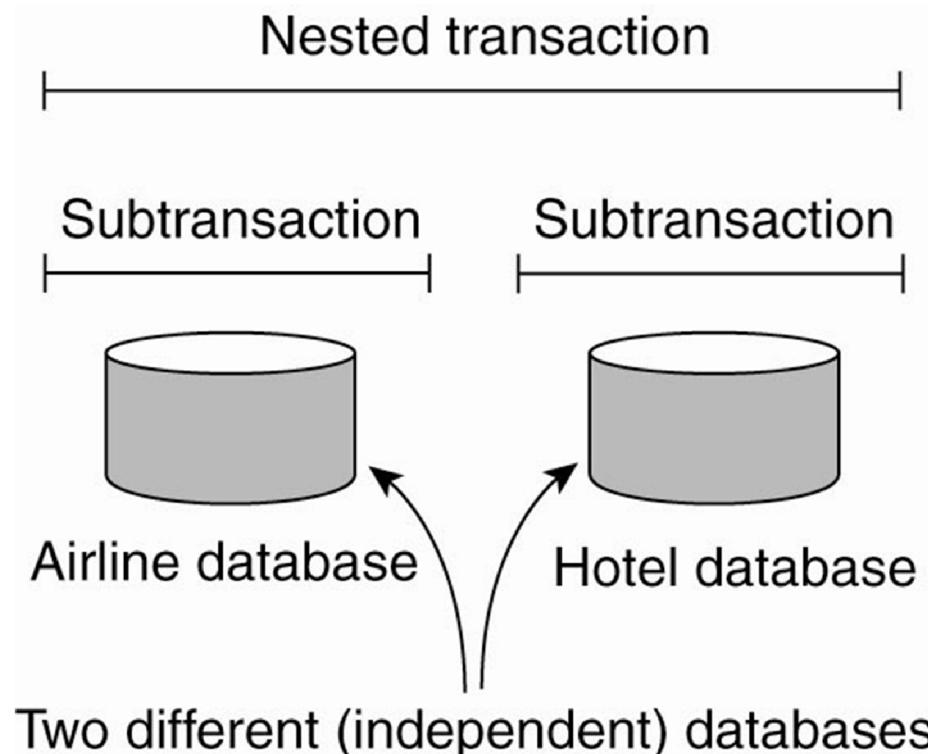


Figure 1-9. A nested transaction.

TP (Transaction Processing) Monitor

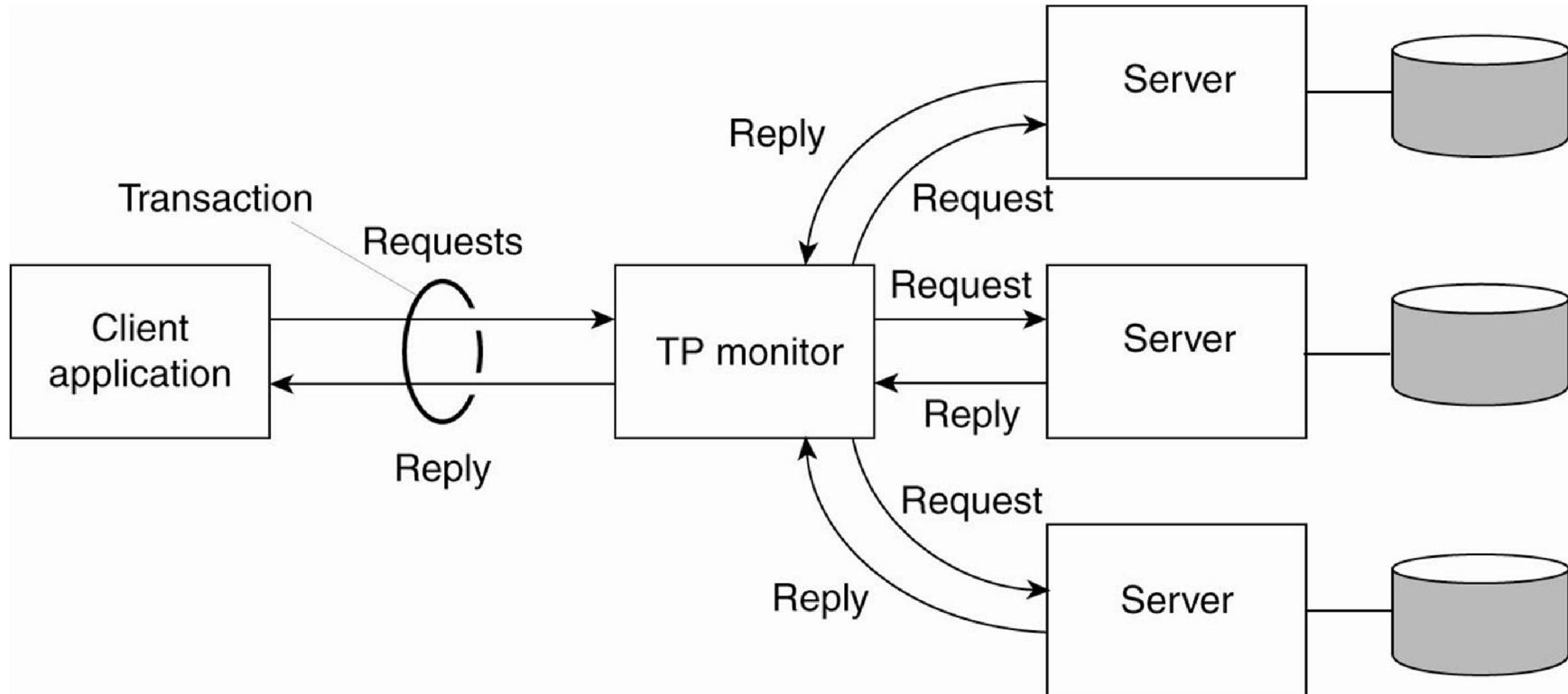


Figure 1-10. The role of a TP monitor (a.k.a. Coordinator) in distributed systems.

2.2 Enterprise Application Integration

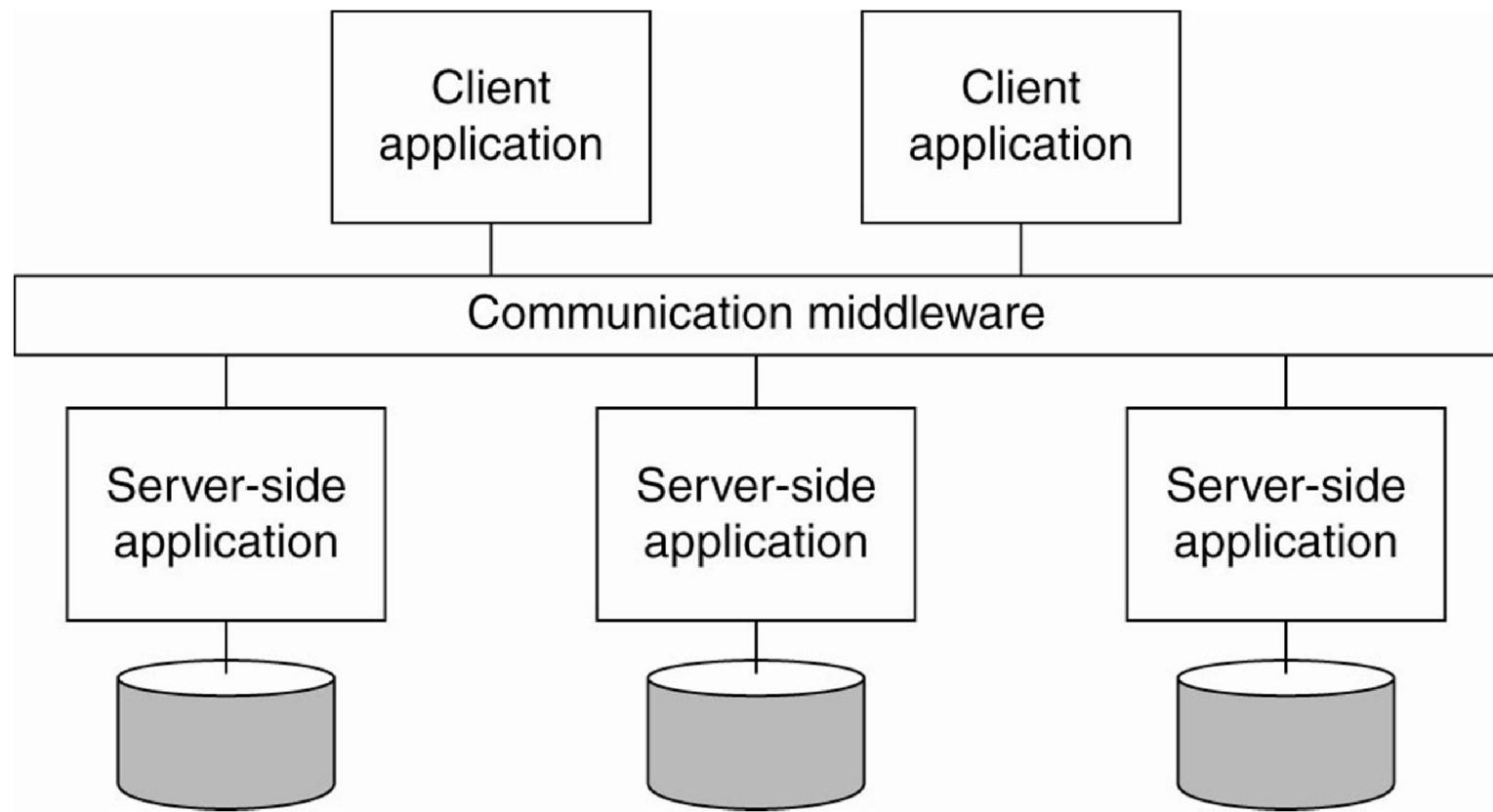


Figure 1-11. Middleware as a communication facilitator in enterprise application integration.

3. Distributed Pervasive Systems

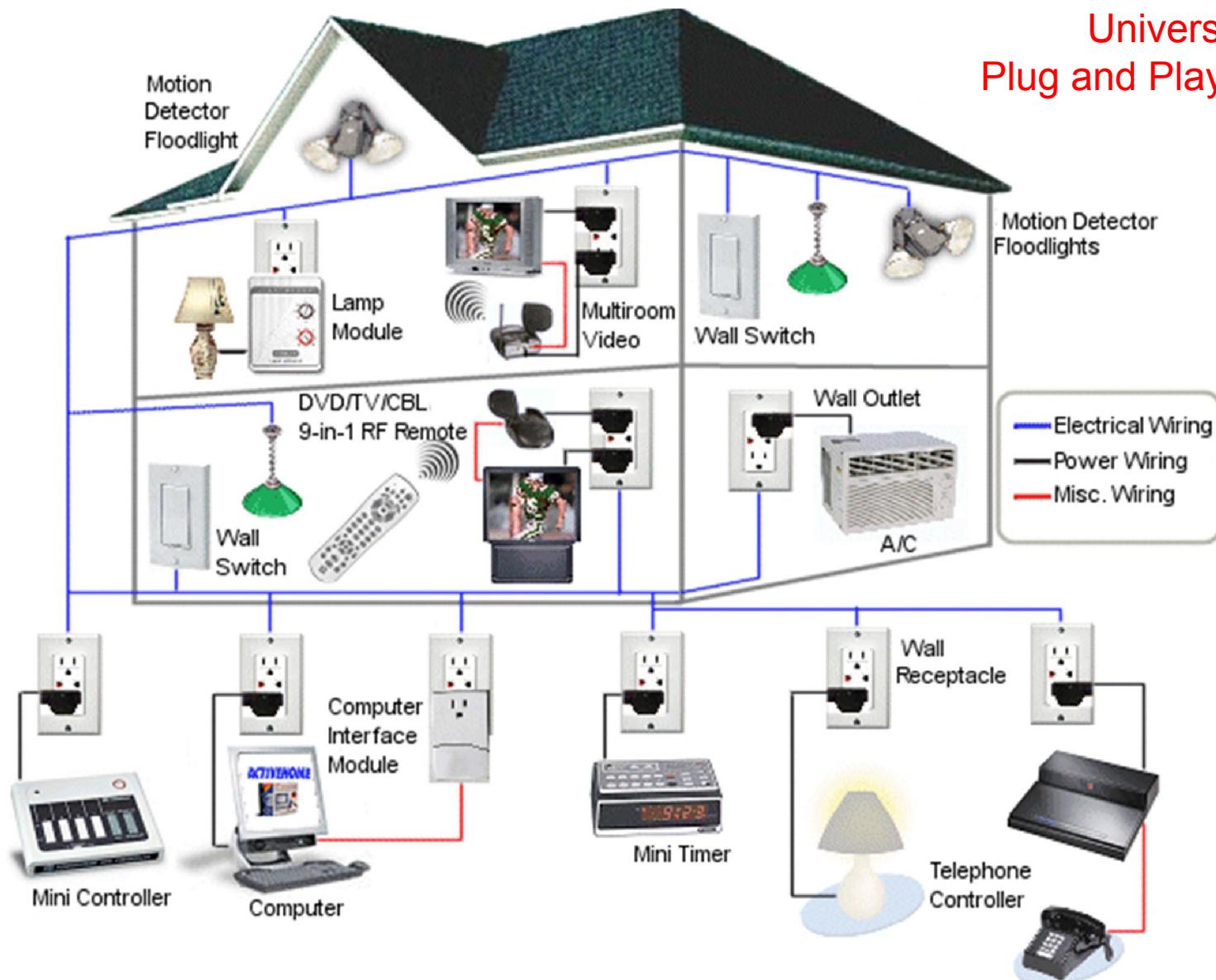
- Stability :
 - Nodes are fixed and have a more or less permanent
 - High-quality connection to a network.
- Instability
 - Small, battery-powered, mobile, and having only a wireless connection
- Home Systems
- Electronic Health Care Systems
- Sensor Networks

Three Requirements

- Embrace contextual changes.
 - Device must be continuously be aware of the fact that its environment may change all the time
- Encourage ad hoc composition.
 - One device is used in very different ways by different users
 - Suite of applications should be easy to configure
 - By the user
 - Through automated (but controlled) interposition
- Recognize sharing as the default
 - Devices join the system for shared information
 - Easily read, store, manage, and share information

3.1 Home Systems

Universal
Plug and Play (UPnP)



3.2 Electronic Health Care Systems

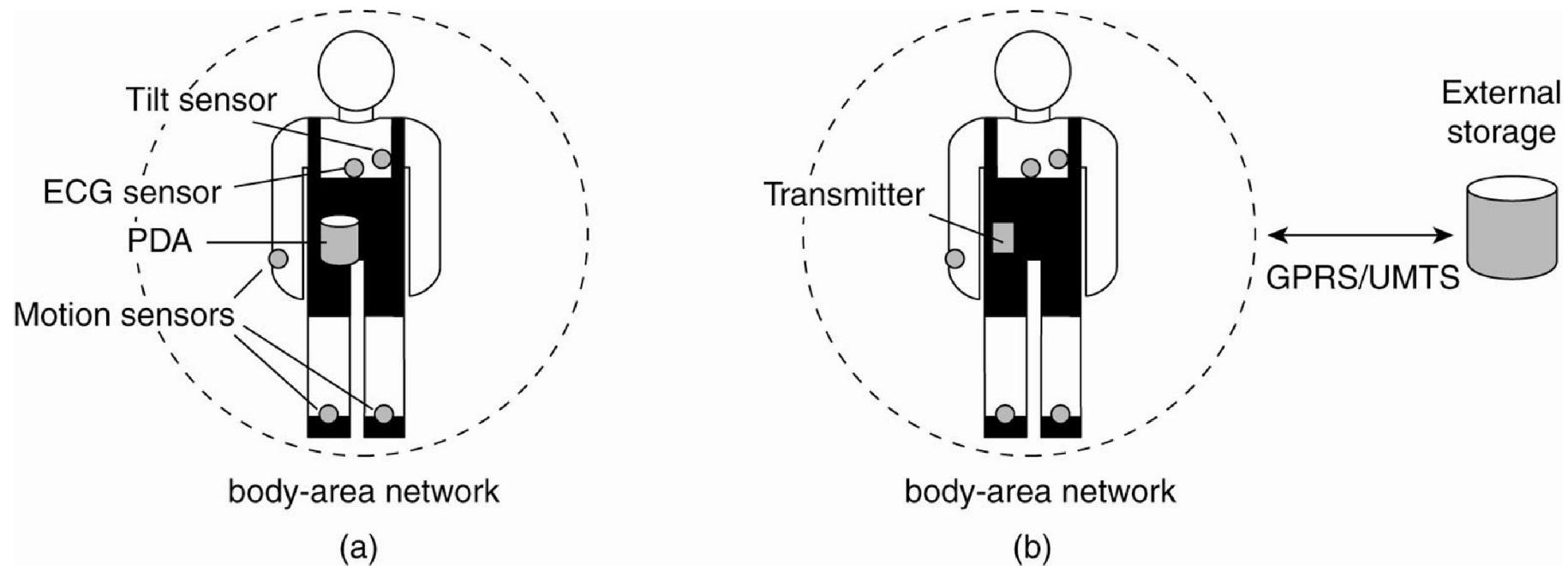
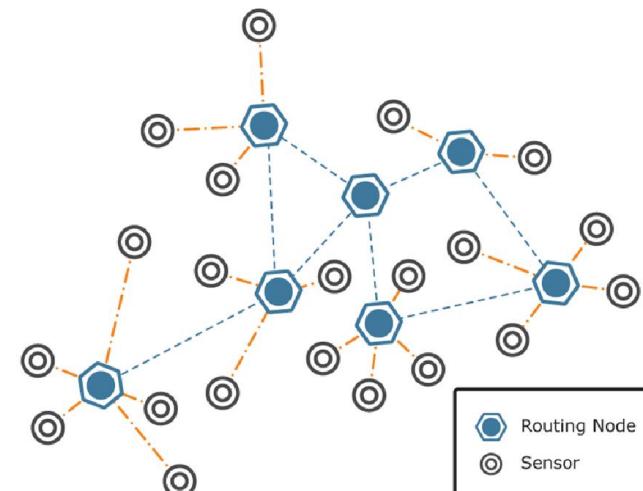


Figure 1-12. Monitoring a person in a pervasive electronic health care system, using (a) a local hub or (b) a continuous wireless connection.

3.3 Sensor Networks

- A sensor network typically consists of tens to hundreds or thousands of relatively small nodes, each equipped with a sensing device.
 - Wireless communication (restricted communication capabilities)
 - Battery powered (constrained power consumption, low efficiency)



Open questions

Questions concerning sensor networks:

- How do we (dynamically) set up an efficient tree in a sensor network?
- How does aggregation of results take place? Can it be controlled?
- What happens when network links fail?

Next Lesson...

DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 2
ARCHITECTURES

Outline

- Software Architecture and its styles
- System Architectures
 - Centralized system architecture
 - Decentralized system architecture
 - Hybrid system architecture

Software Architectures

- How the various software components are to be organized and how they should interact.
- Terminology
 - **Component** is a modular unit with well-defined required and provided interfaces that is replaceable within its environment
 - **Connector** is a mechanism that mediates communication, coordination, or cooperation among components
 - **Architectural style** is formulated in terms of components, the way that components are connected to each other, the data exchanged between components.

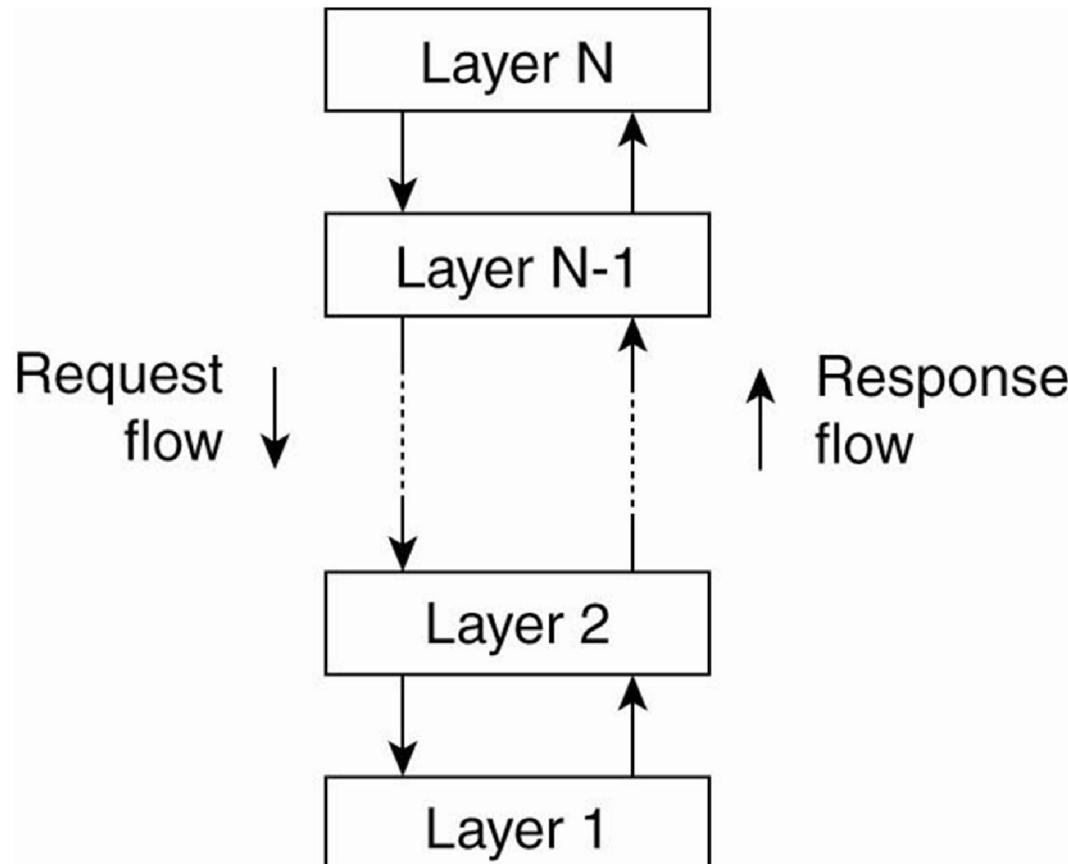
Examples of Abstractions

- Operating system
 - Devices
 - Drivers
 - Services
- Software
 - Objects
 - Layers
- Communication
 - Protocol
 - Distributed communication models

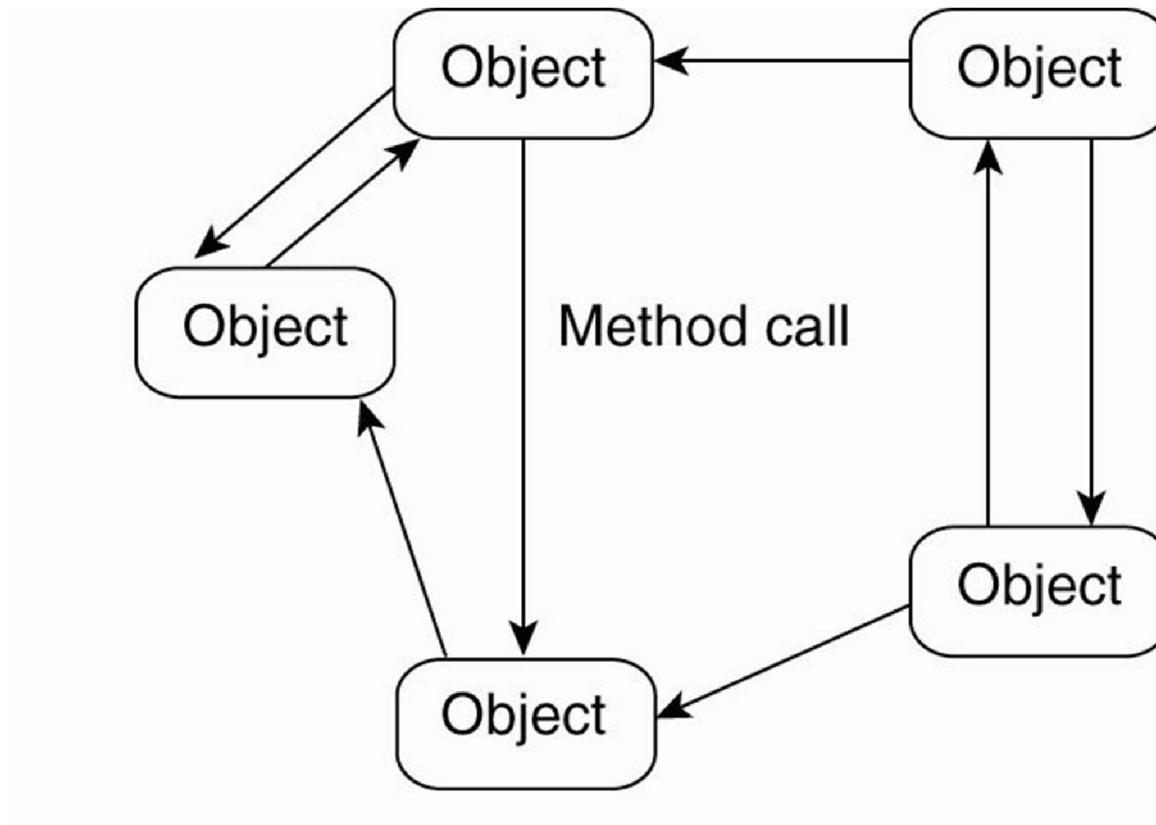
Architectural Styles for Distributed System

- Layered architecture
- Object-based architecture
- Data-centered architecture
- Event-based architecture

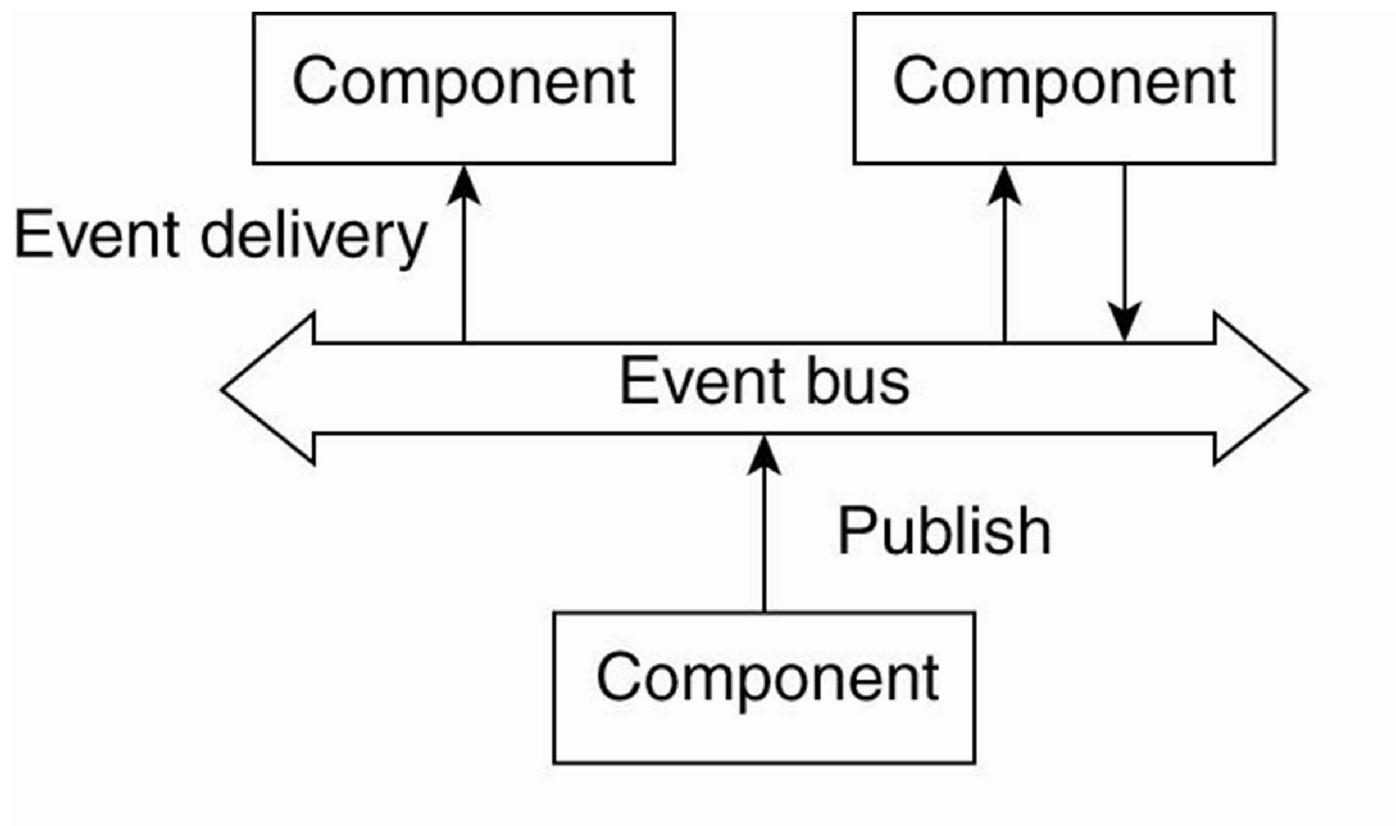
Layered Architecture



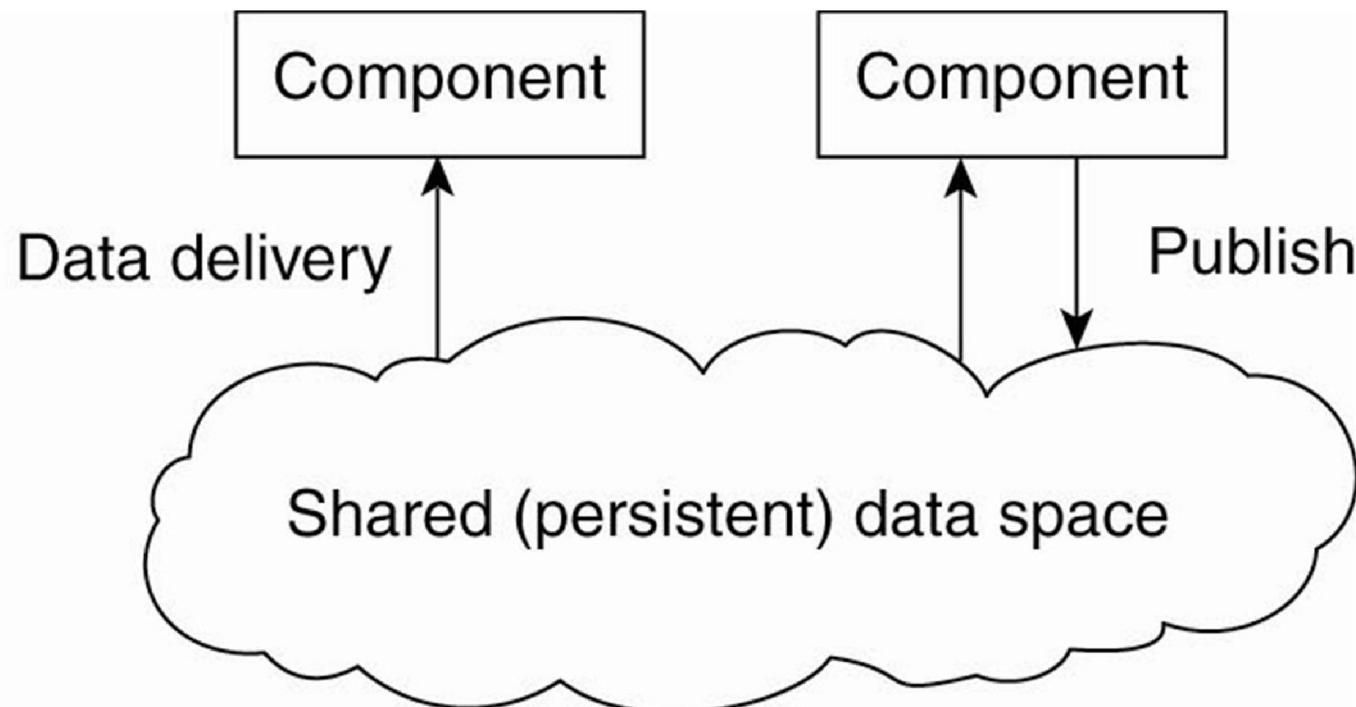
Object-based Architecture



Event-based Architecture



Shared Data-space Architecture



Outline

- Software Architecture and its styles
- **System Architectures**
 - Centralized system architecture
 - Decentralized system architecture
 - Hybrid system architecture

System Architectures

- System architecture is **an instance of a software architecture** which realizes the **(logical)** software components, their **(logical)** interaction, and their placement **physically**;
- The final **(logical+physical)** instantiation of a software architecture is a system architecture.

Distributed System Architecture

- Centralized architectures
 - a single server (**physical**) implements most of the software components (**logical**) while remote clients (**physical**) can access that server (**physical**) using simple communication.
- Decentralized architectures
 - A server (**logical**) may be **physically split up into logically equivalent parts**, each part is operating on its own share of the complete data set, thus balancing the load.
 - **Physical** machines play equal **logical** roles
- Hybrid architectures

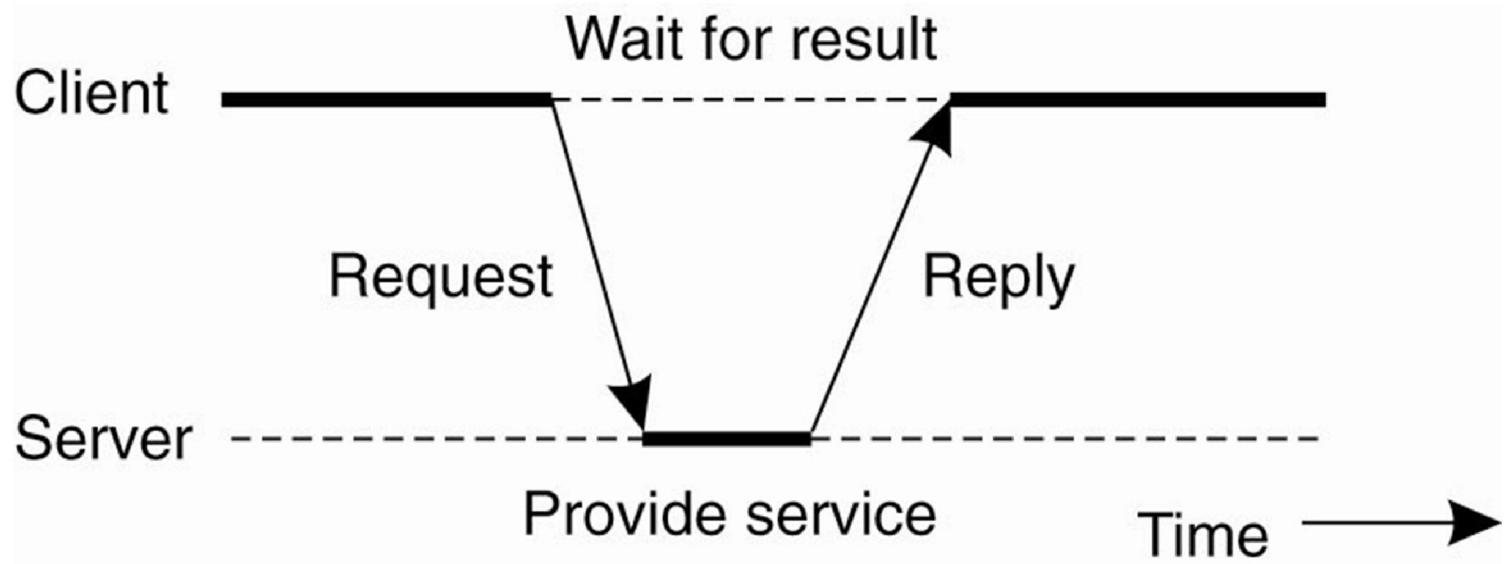
Outline

- Software Architecture and its styles
- System Architectures
 - Centralized system architecture
 - Decentralized system architecture
 - Hybrid system architecture

Client-Server Model (abstract level)

- **Processes** in a distributed system are divided into two (possibly overlapping) groups.
 - A **server** is a process implementing a specific service, for example, a file system service or a database service.
 - A **client** is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply.

Client-Server Interaction or Request-Reply Behavior

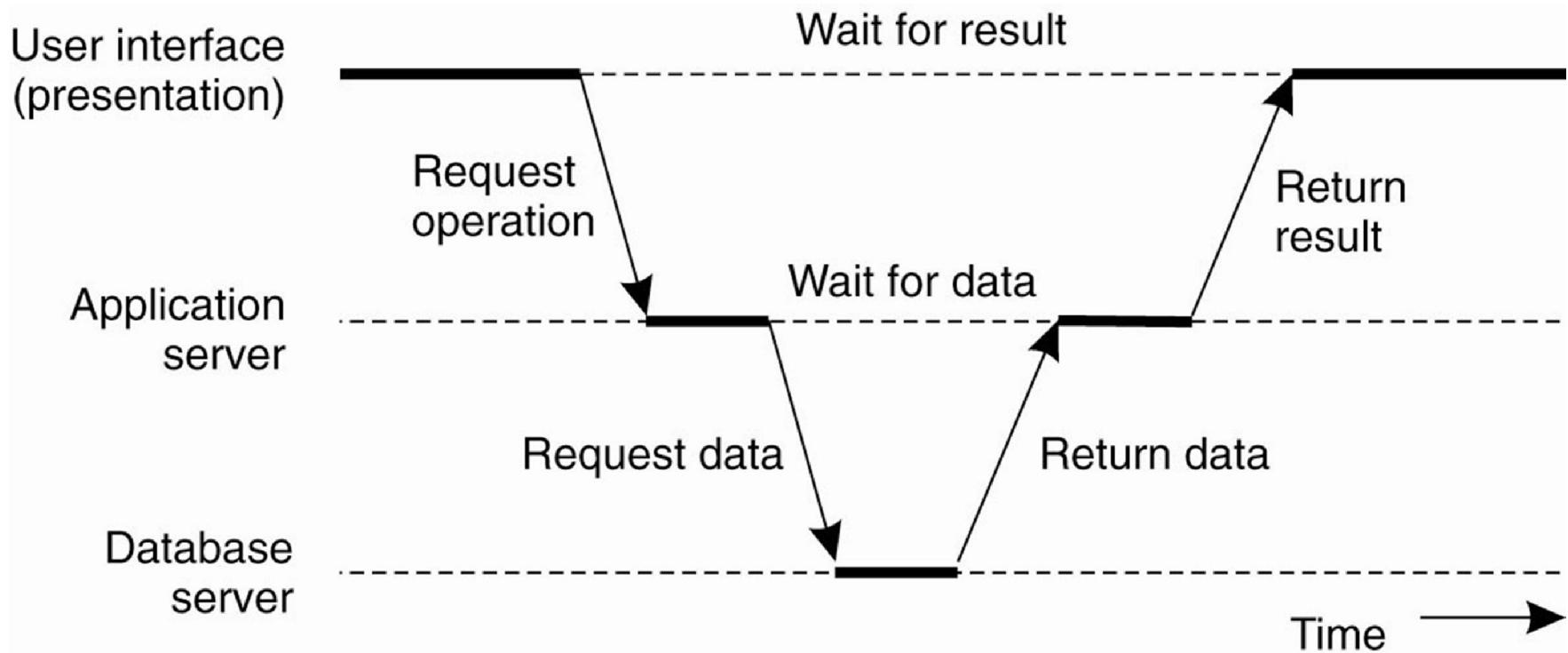


Note that client-server interaction can also be used for multi-threaded, asynchronous process models

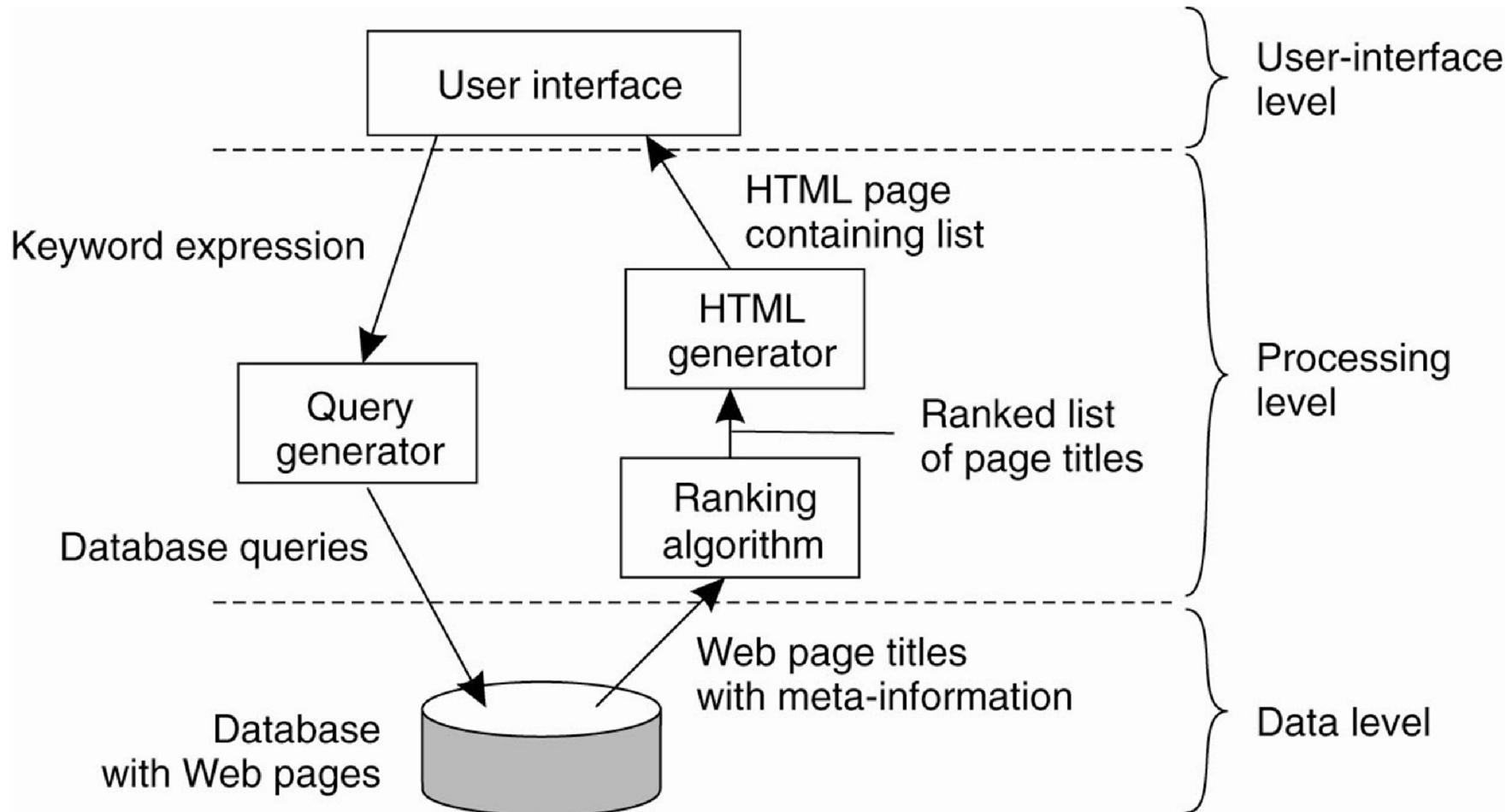
Application Layering

- How to draw a clear distinction between a client and a server.
 - No clear distinction
- The user-interface level
- The processing level
- The data level

Three Layers Application



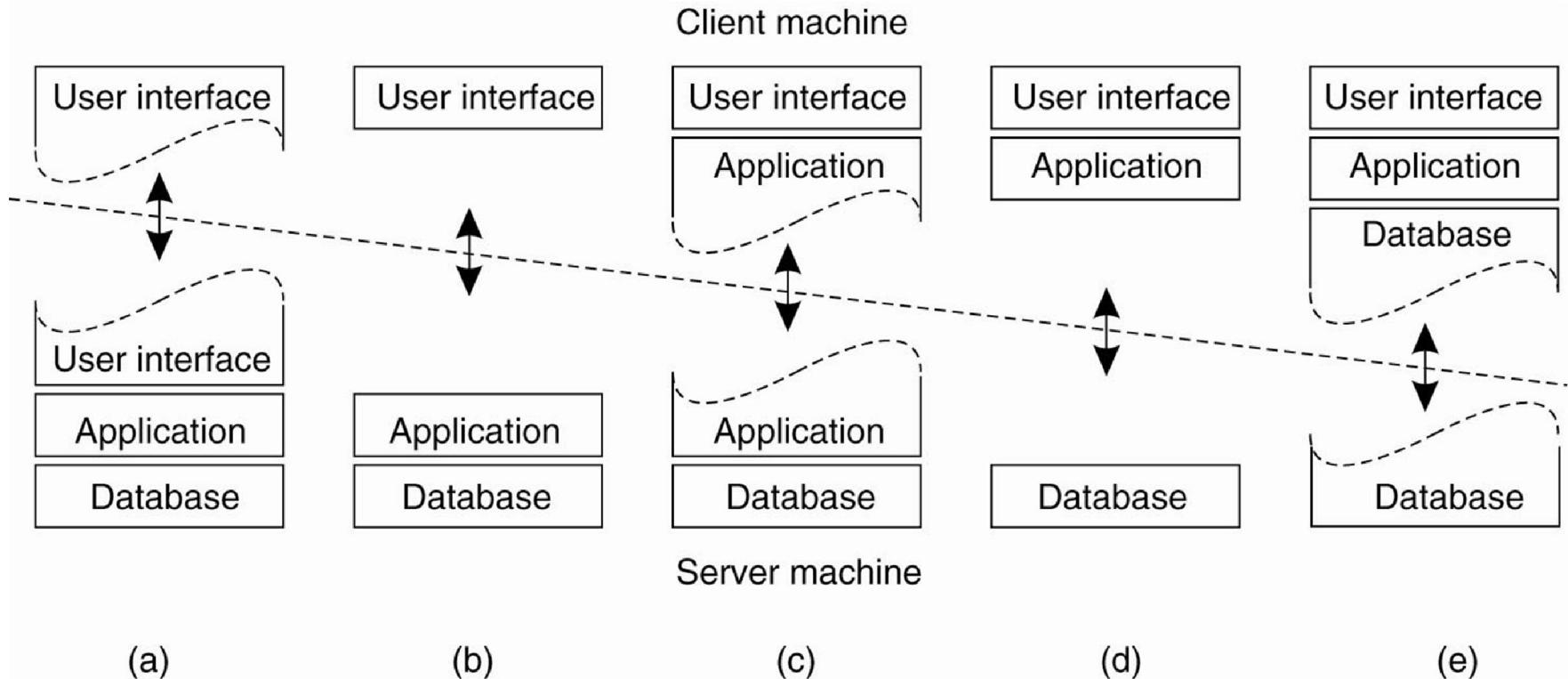
Example: Internet search engine



Two-tiered System Architectures

- The simplest organization is to have only two types of machines for client-server model (**logical**):
 - A **client machine** containing only the programs implementing (part of) the user-interface level
 - A **server machine** containing the rest, the programs implementing the processing and data level

Two-tiered System Architectures



Centralized Architectures

- Client-Server System Architecture
- Multi-tiered System Architecture

Outline

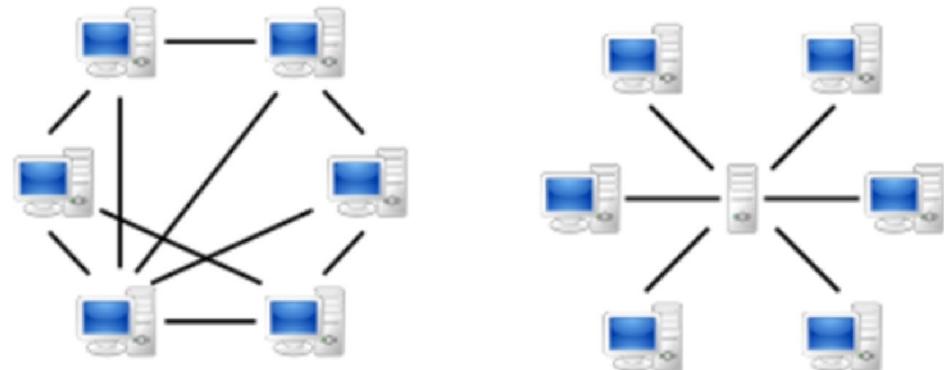
- Software Architecture and its styles
- System Architectures
 - Centralized system architecture
 - Decentralized system architecture
 - Hybrid system architecture

Decentralized Architectures

- Vertical Distribution (垂直分布)
 - Distributed processing is equivalent to organizing a client-server application as a multi-tiered architecture
- Horizontal Distribution (水平分布)
 - A client or server may be physically split up into logically equivalent parts

Peer-to-peer Architectures

- From a high-level perspective, the nodes that constitute a peer-to-peer system are all equal.
- System functions are represented by every node
- Interaction between processes is symmetric
 - **Servent**: act as a client and a server at the same time
 - **Overlay network**



Overlay Network (覆盖网络)

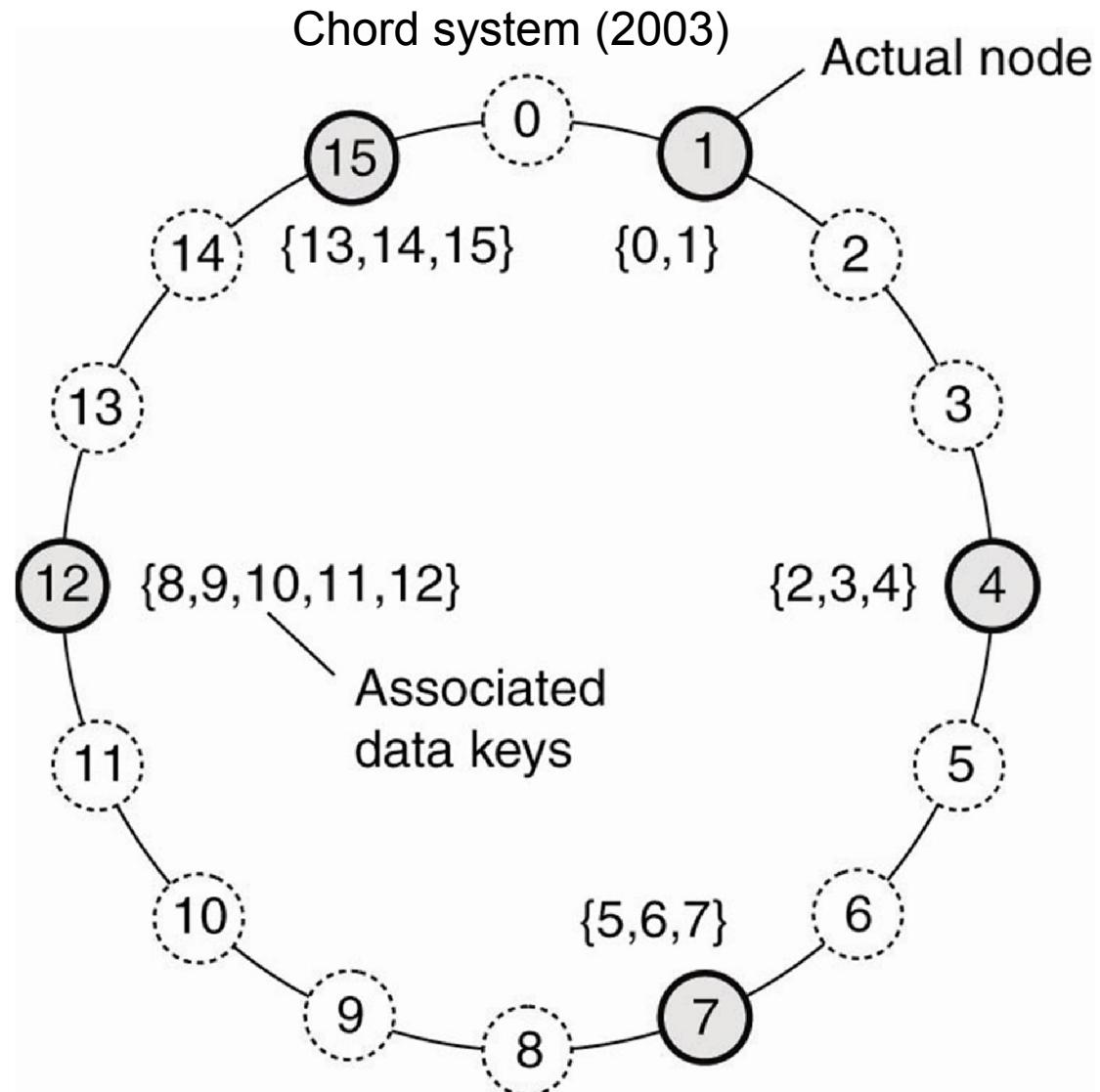
- An overlay network is a computer network, which is built on the top of another network, or it is a network of application layer.
 - Nodes in the overlay can be thought of as being **connected by virtual or logical links**, each of which corresponds to a path, perhaps through many physical links, in the underlying network.
- Networks of distributed systems are overlay networks.

P2P Architecture Types

- Structure of Overlay Network
 - **Structured** : organized into a specific topology, and the protocol ensures that any node can efficiently search the network for a file/resource;
 - **Unstructured** : do not impose a particular structure on the overlay network by design, but rather are formed by nodes that randomly form connections to each other.

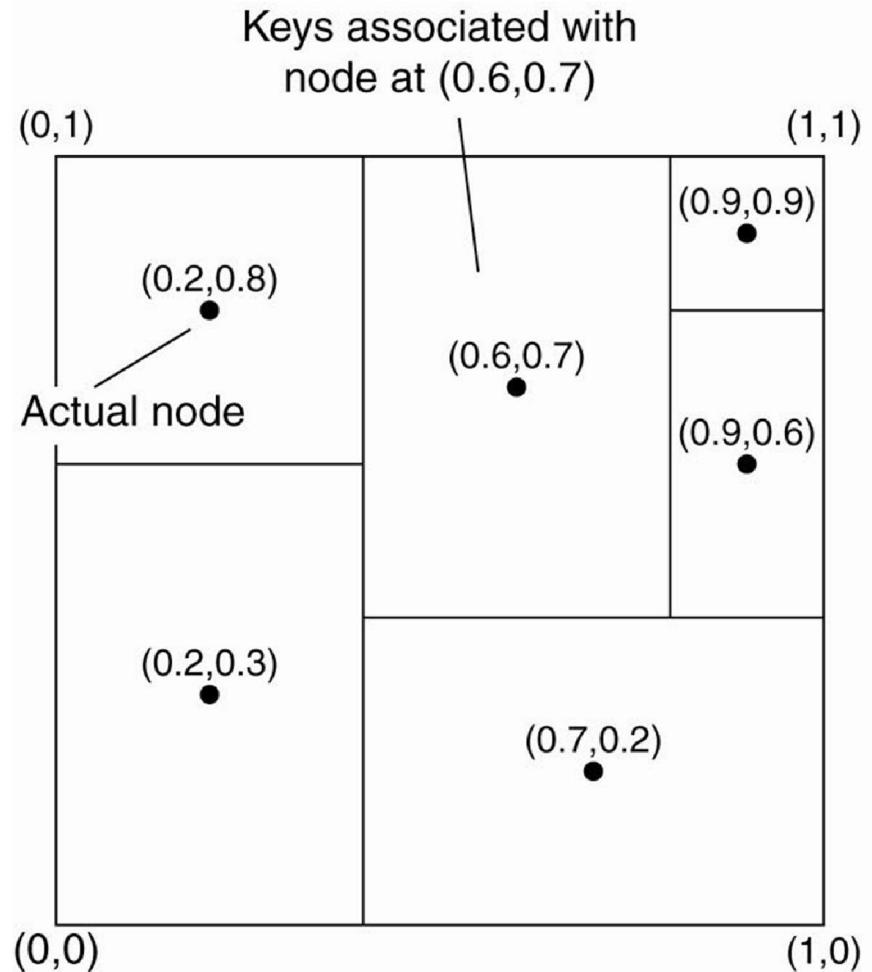
Structured Peer-to-Peer Architectures (1)

- Distributed Hash Table (DHT)
- Data items are hashed to obtain key, which is then used to place items in hash table
- Portions of the hash table assigned to each participating node



Structured Peer-to-Peer Architectures (2)

- Multi-dimensional search:
data items assigned a point in k-dimensional Cartesian space, regions assigned to nodes

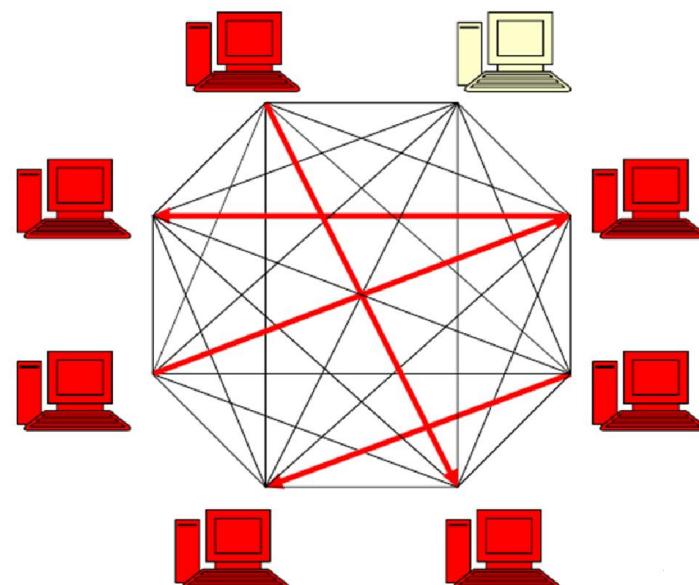
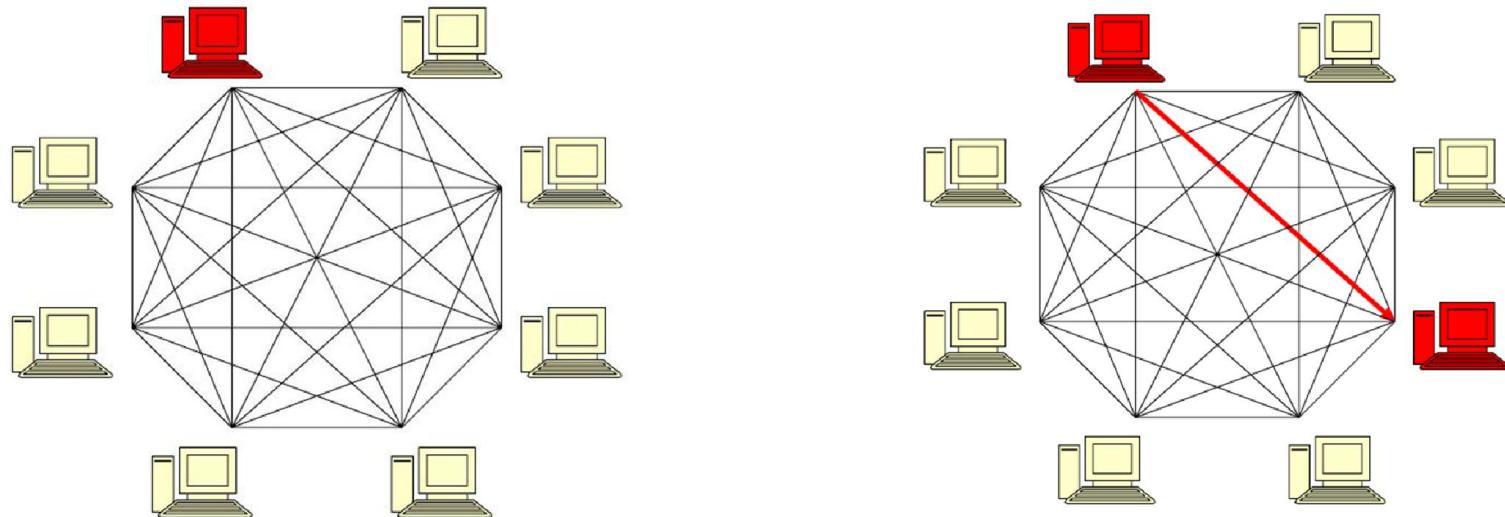


Content Addressable Network (CAN)

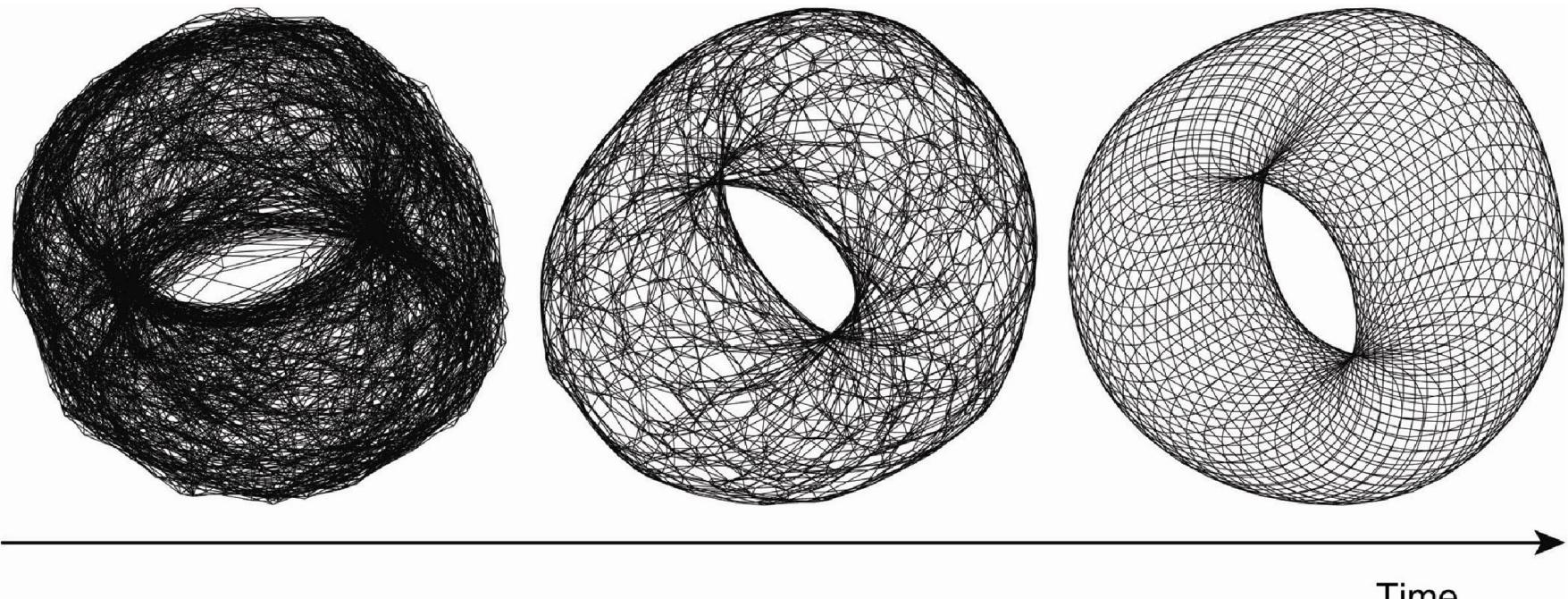
Unstructured Peer-to-Peer Architectures

- Each node maintains a list of neighbors, but that this list is constructed in a more or less random way (random graph) .
- **Partial View:** Each node maintains a list of neighbors which are randomly chosen live node from the current set of nodes
 - There are many algorithms to construct such a partial view.

Gossip



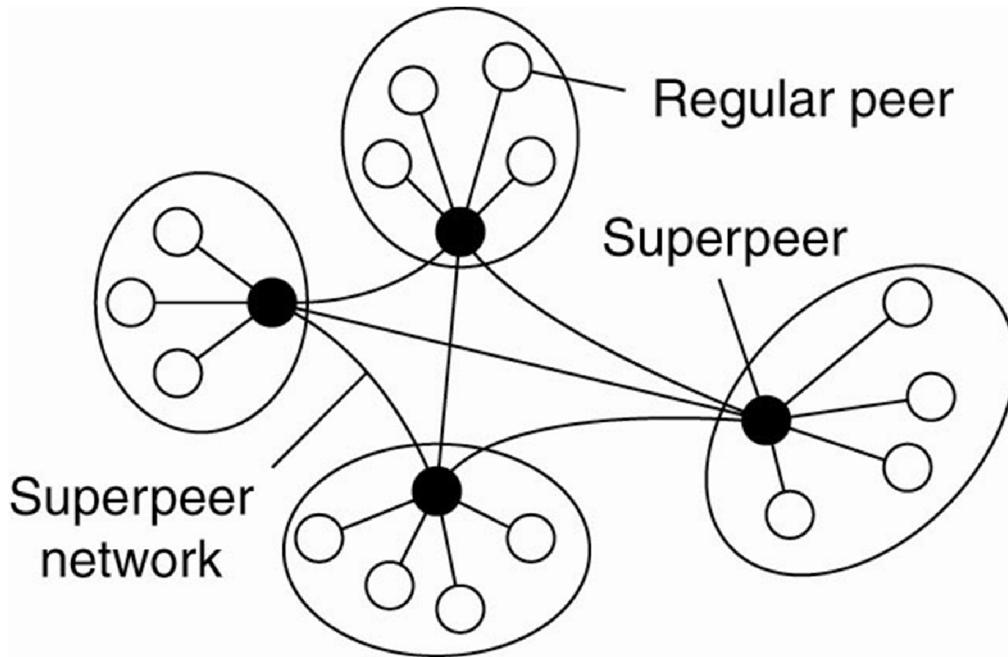
Topology Management



Generating a specific overlay network using ranking function

Superpeers

- Nodes such as those maintaining an index or acting as a broker are **superpeers**
- Client-superpeer relation can be fixed or dynamic



- Nodes such as those maintaining an index or acting as a broker
- Very regular peers connected as a client to a superpeer.
- All communication from and to a regular peer proceeds through that peer's associated superpeer.

Superpeer Selection

- Qualities a superpeer should possess:
 - Accessible
 - Responsive
 - Reliable
- These imply superpeer attributes:
 - Internet-facing (e.g., routable IP address)
 - Well-connected (high-speed internet connection)
 - Resources (CPU speed, memory)
 - "Near" clients (a relative attribute)

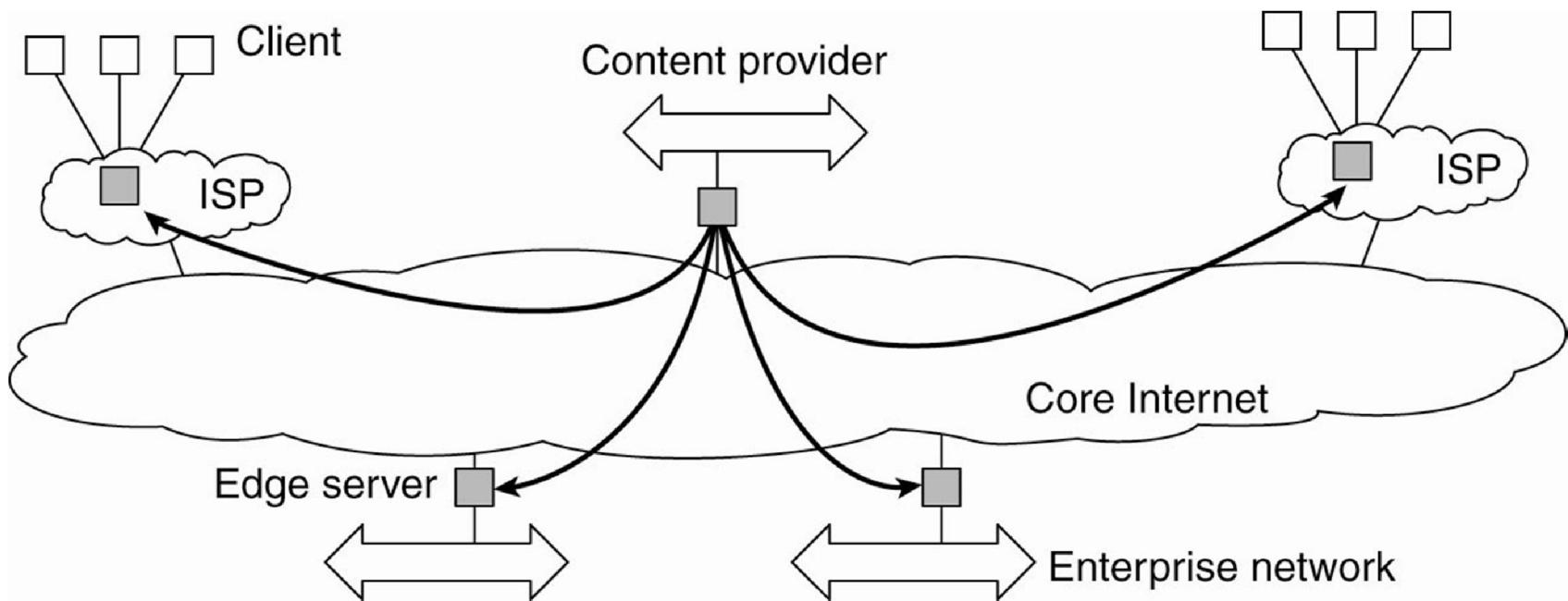
Outline

- Software Architecture and its styles
- **System Architectures**
 - Centralized system architecture
 - Decentralized system architecture
 - **Hybrid system architecture**

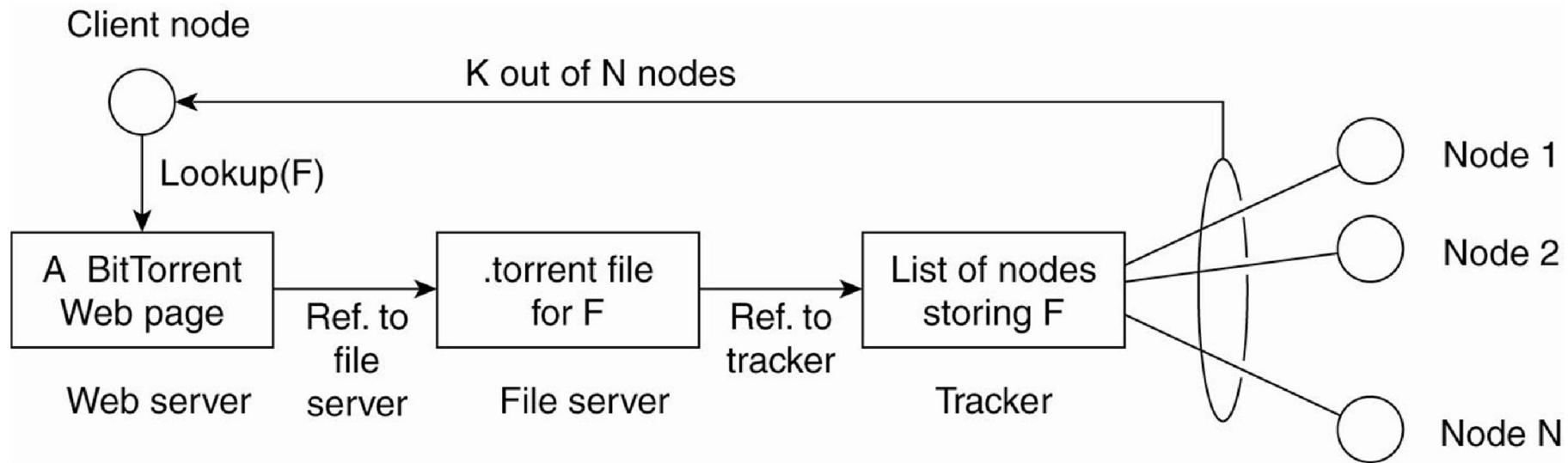
Hybrid system architecture

- Edge-Server Systems
 - Servers are placed "at the edge" of the network.
 - This edge is formed by the boundary between enterprise networks and the actual Internet.
- Collaborative Distributed Systems
 - Traditional client-server scheme is deployed when it is first get started.
 - Once a node has joined the system, it can use a fully decentralized scheme for collaboration.

Edge-Server Systems



Collaborative Distributed Systems



BitTorrent

- When an end user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file.
- A *.torrent* file contains the information that is needed to download a specific file. In particular, it refers to what is known as a **tracker**, which is a server that is keeping an accurate account of active nodes that have (chunks) of the requested file.

Next Lesson...

DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 3
Processes

Outline

- Thread
- Virtualization
- Clients
- Servers
- Code Migration

Process

- Operating systems
 - Process is generally defined as a program in execution.
 - Management and scheduling of processes
- Distributed systems
 - More issues

Thread

- Granularity of **processes** as provided by the operating systems on which distributed systems are built is not sufficient;
- Multiple threads per process makes it much easier to build distributed applications and to attain better performance.

Process & Thread

- To execute programs in **processes**, operating system manages
 - Virtual processors
 - A process table
 - CPU register values, memory maps, open files, accounting information, privileges
 - Hardware isolation for concurrency transparency
- To execute programs in threads. Threads system manages
 - Maintains only the minimum information to allow a CPU to be shared by several threads.
 - Thread context is CPU context and other information for thread management

Process & Thread

- Processes ensure concurrency transparency costly;
- Threads minimum concurrency transparency and let the developers deal with the mutex (mutual exclusion) variable;
- One CPU execute only one process in time;
- Process contains and manages many threads
 - Java threads and JVM;

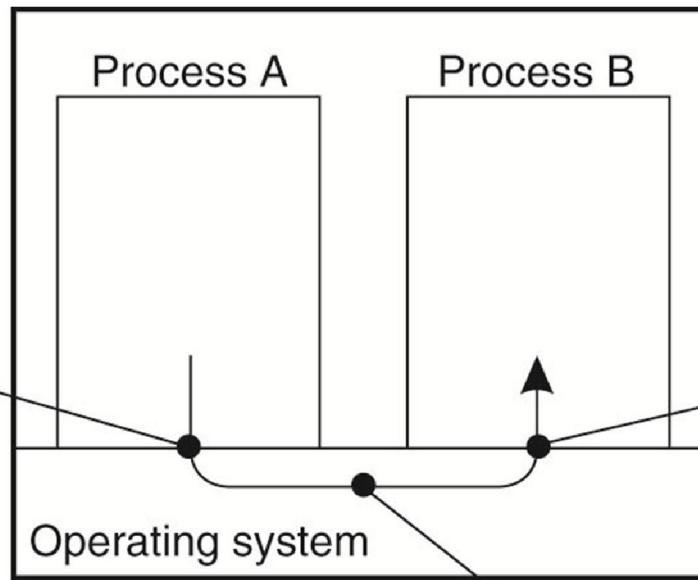
Thread

- Performance of a multi-threaded application **need hardly ever be worse** than that of its single-threaded counterpart.
- With limited concurrency transparency, development of multi-threaded applications requires additional intellectual effort.

Context switching as the result of IPC

IPC (Inter-Process Communication) 进程间通信

S1: Switch from user space
to kernel space



S2: Switch context from
process A to process B

S3: Switch from kernel
space to user space

- Multi-threads is better than multi-processes because IPC is costly in a **Non-distributed System**

Threads in Distributed Systems

- Thread can provide a convenient means of allowing **blocking system calls** without **blocking the entire process** in which the thread is running.
 - Multi-threaded Clients
 - Multi-threaded Servers

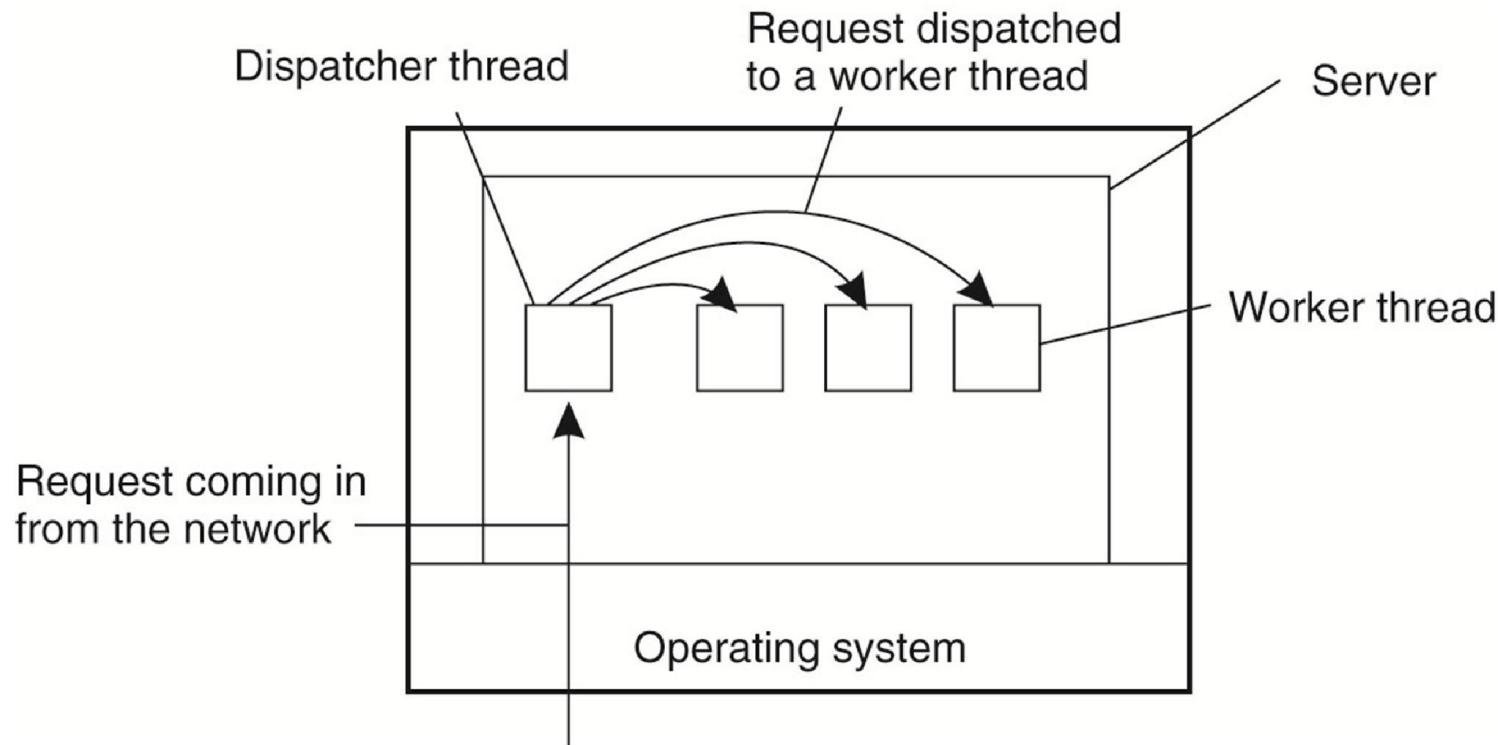
Multi-threaded Clients

- To establish a high degree of distribution transparency, distributed systems that operate in wide-area networks may need to **hide** communication latencies.
 - Initiate communication and immediately proceed with something else

Web Brower

- Browsers start displaying data while it is still coming in
 - Threads for process text, images, scrolling, scripts, etc.
- Web servers have been replicated across multiple machines, where each server provides exactly the same set of Web documents.
 - **Single thread client** : the request is forwarded to one of the servers, load-balancing such as round-robin;
 - **Multithreaded client**: connections may be set up to different replicas, allowing data to be transferred in parallel

Multi-threaded Servers



- **Dispatcher:** reads incoming requests
- **Worker thread:** handles its request

Multithreaded Servers

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

- Blocking system calls make programming easier and parallelism improves performance.
 - **Multi-threaded**: blocking system calls and still achieve parallelism.
 - **Single-threaded**: simplicity of blocking system calls, but gives up performance.
 - **The finite-state machine**: achieves high performance through parallelism, but uses non-blocking calls, thus is hard to program.

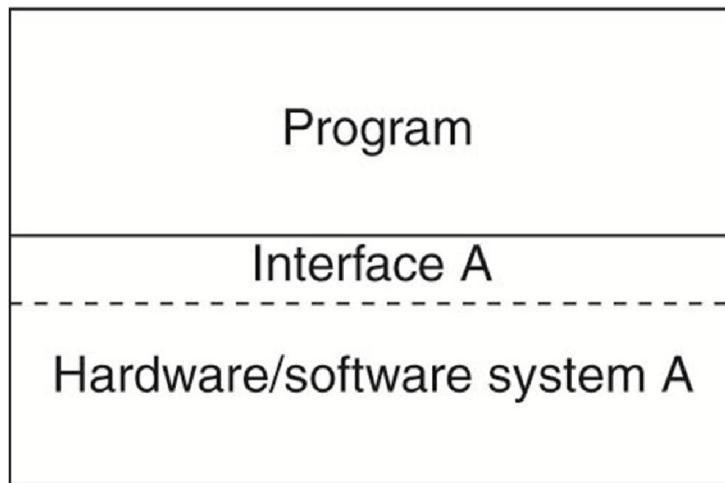
Outline

- Thread
- Virtualization
- Clients
- Servers
- Code Migration

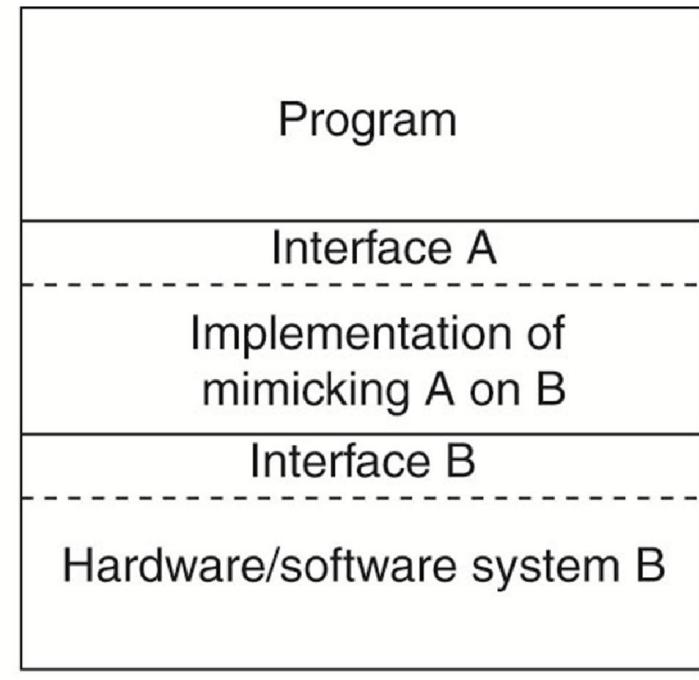
Virtualization

- Virtualization, refers to the act of creating a virtual (rather than actual) version of something, **including but not limited to** a virtual computer hardware platform, operating system (OS), storage device, or computer network resources.
 - **Hardware virtualization**
 - Desktop virtualization
 - Operating system-level virtualization
 - Application virtualization
-

Virtualization in Distributed System



(a)



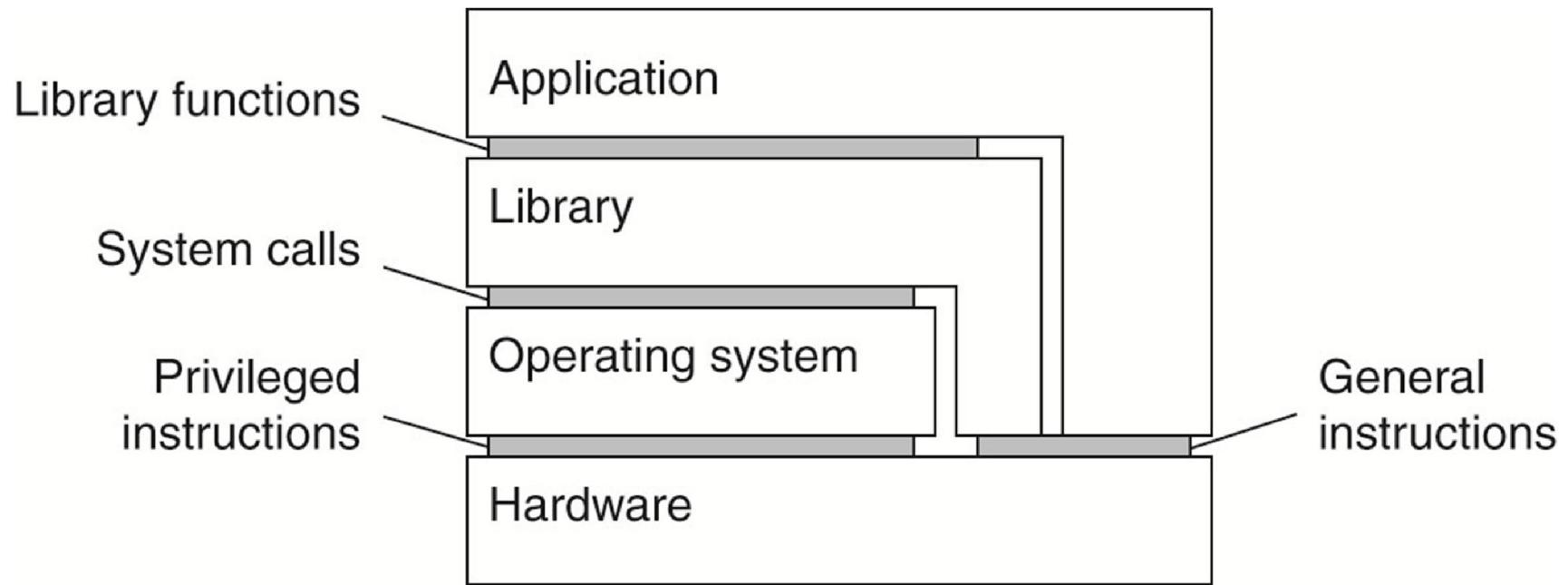
(b)

Compatibility + Homogeneity + Portability

Virtualization in Different Level

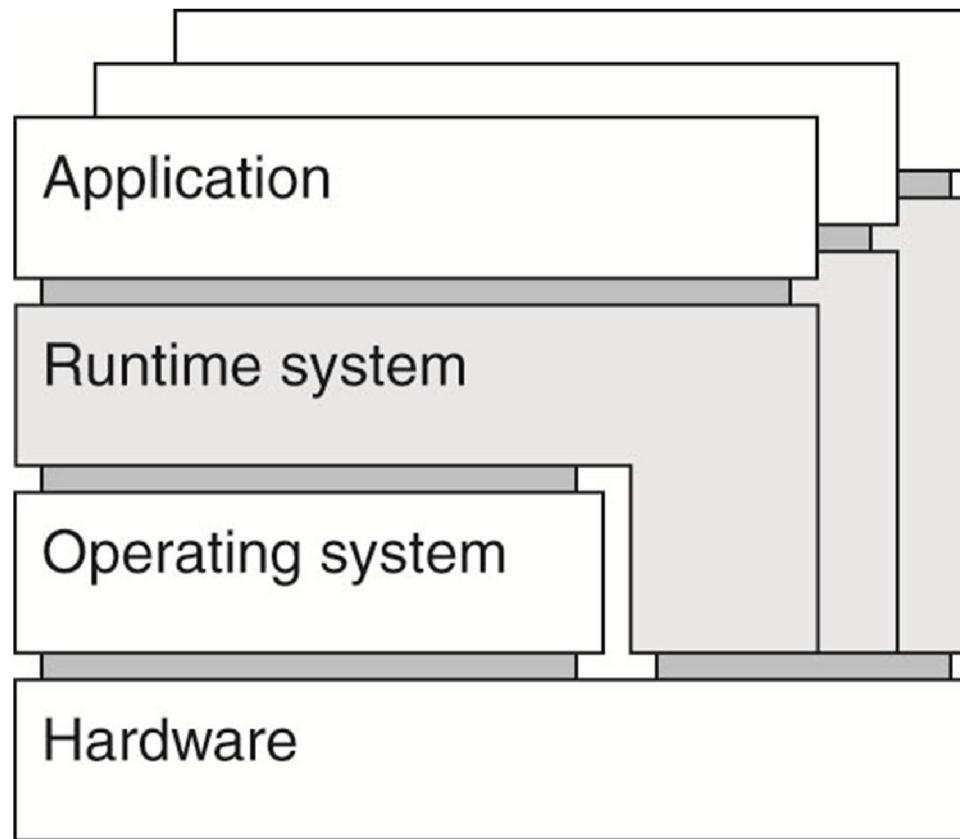
- An interface between the **hardware and software**, consisting of **machine instructions** that can be invoked by any program.
- An interface between the **hardware and software**, consisting of machine instructions that can be invoked only by **privileged programs**, such as an operating system.
- An interface consisting of **system calls** as offered by an **operating system**.
- An interface consisting of **library calls**, generally forming what is known as an application programming interface (**API**).

Virtualization in Different Level



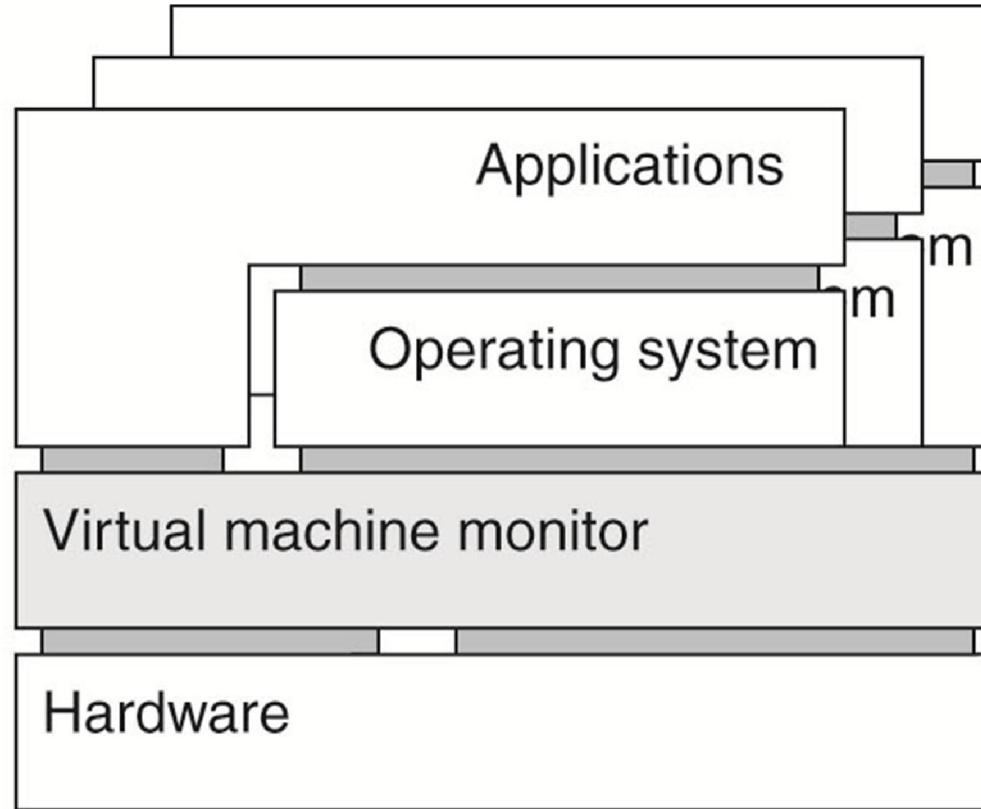
Architectures of Virtual Machines (1)

A process virtual machine, with multiple instances of (application, runtime) combinations.



Architectures of Virtual Machines (2)

A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

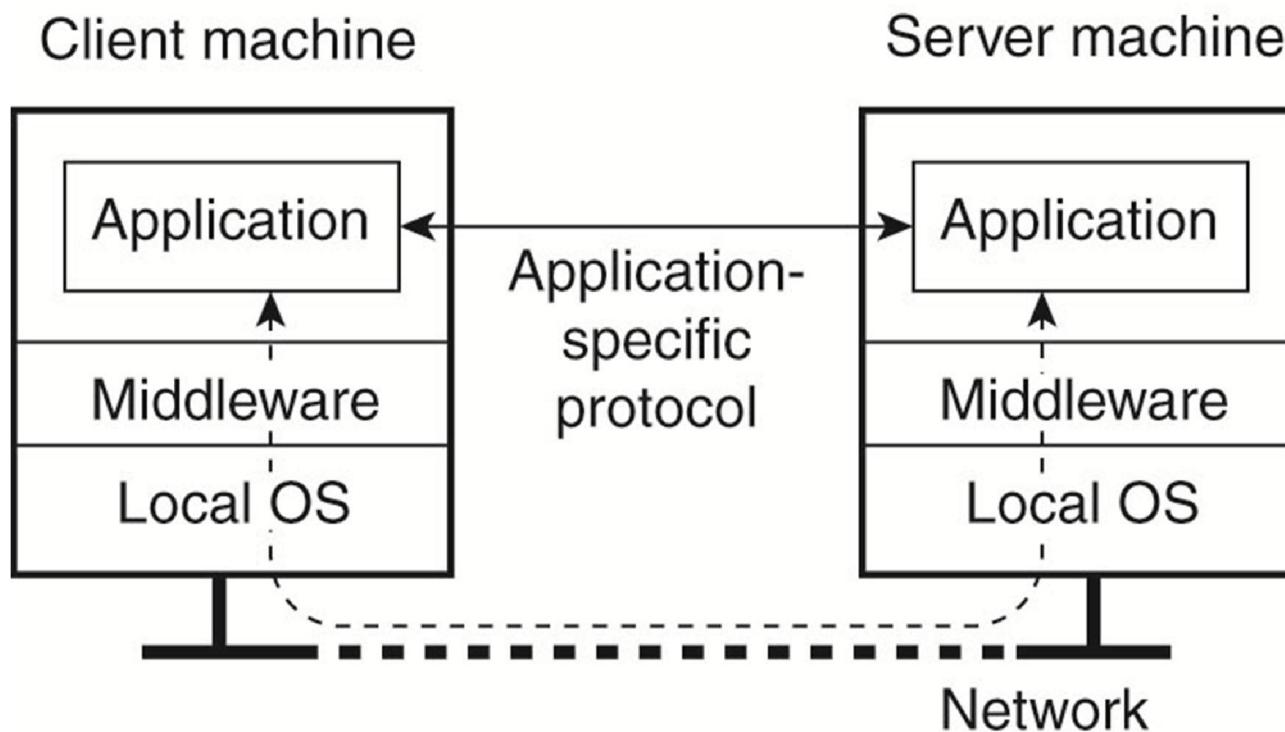


Outline

- Thread
- Virtualization
- Clients
- Servers
- Code Migration

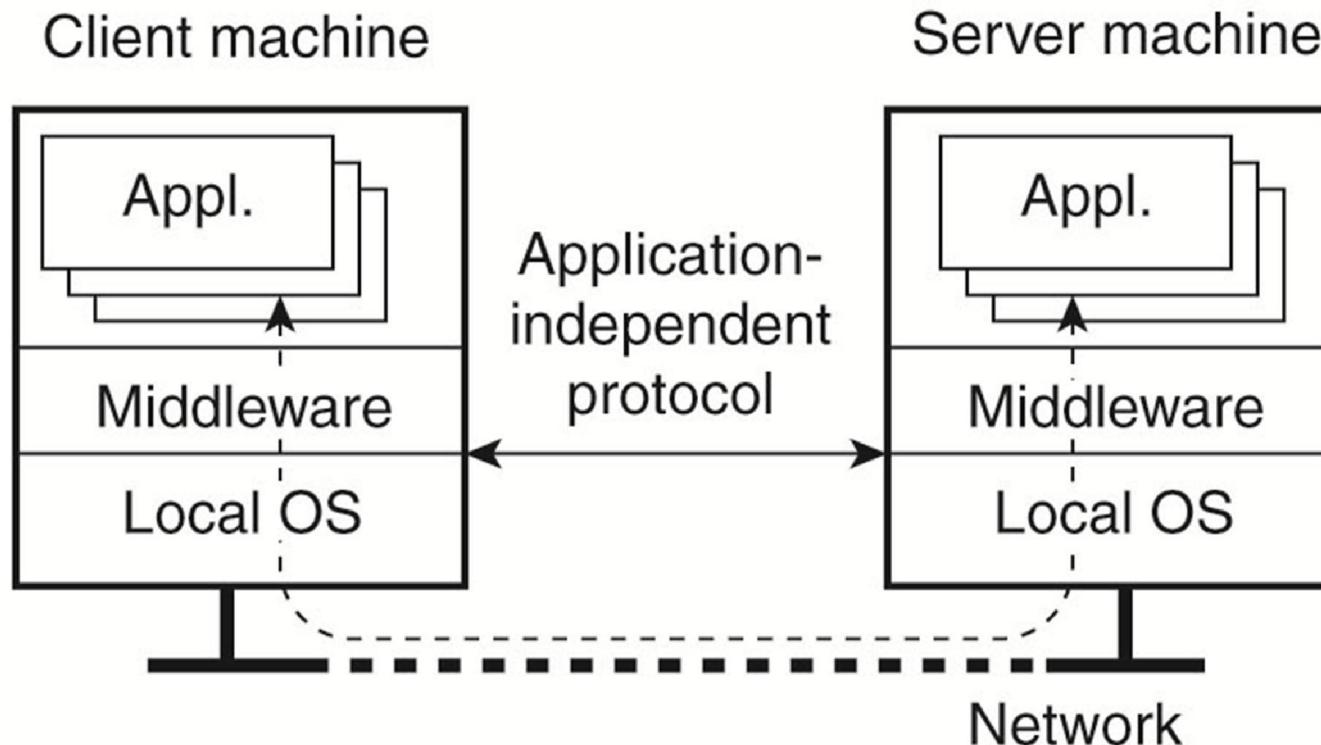
Fat client (machine)

- A fat client is a computer (machine) in client–server (system) architecture that typically provides rich functionality independent of the central server.

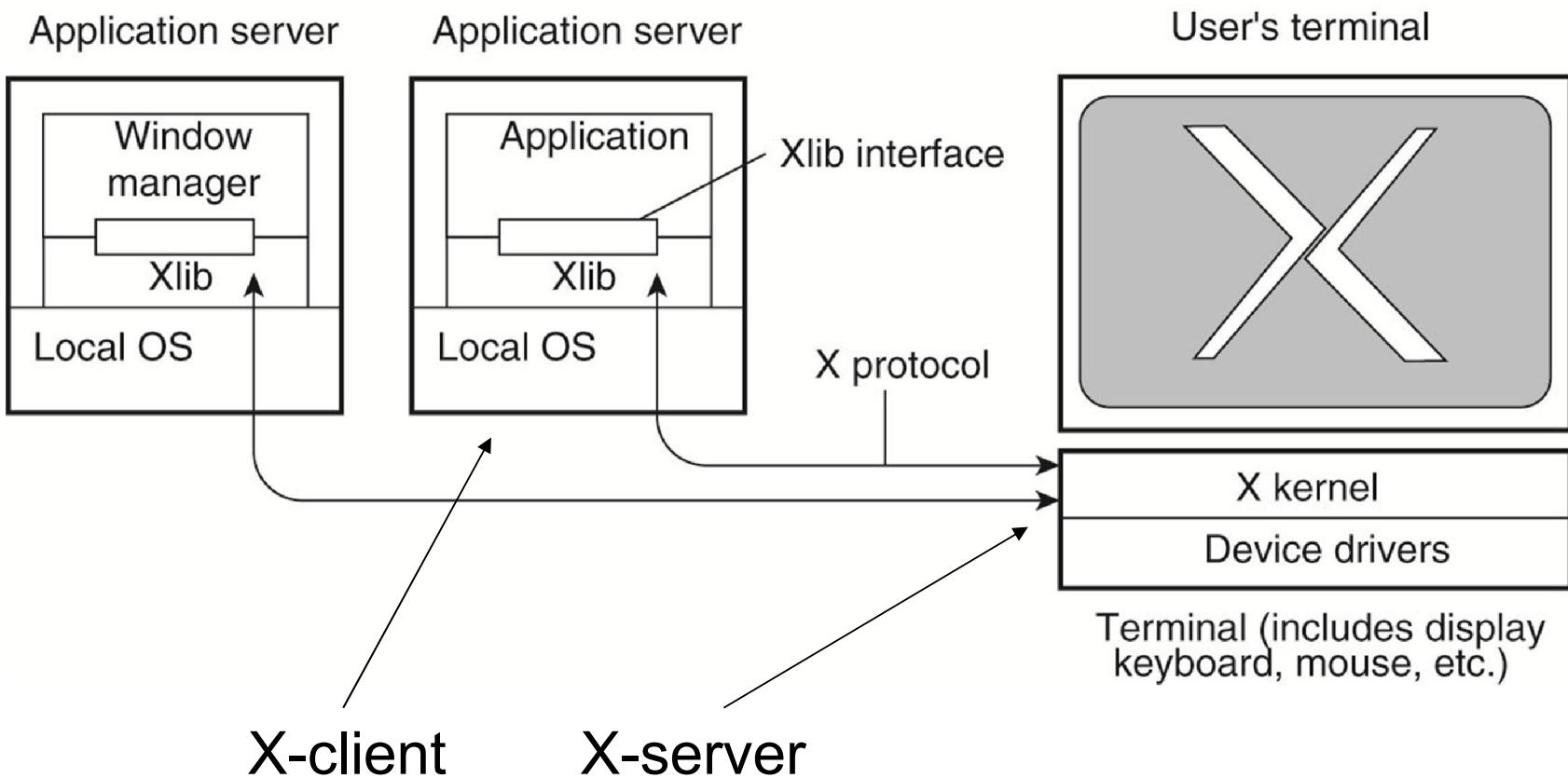


Thin client (machine)

- A thin client is a computer (machine) that depends heavily on another computer (its server) to fulfill its computational roles.



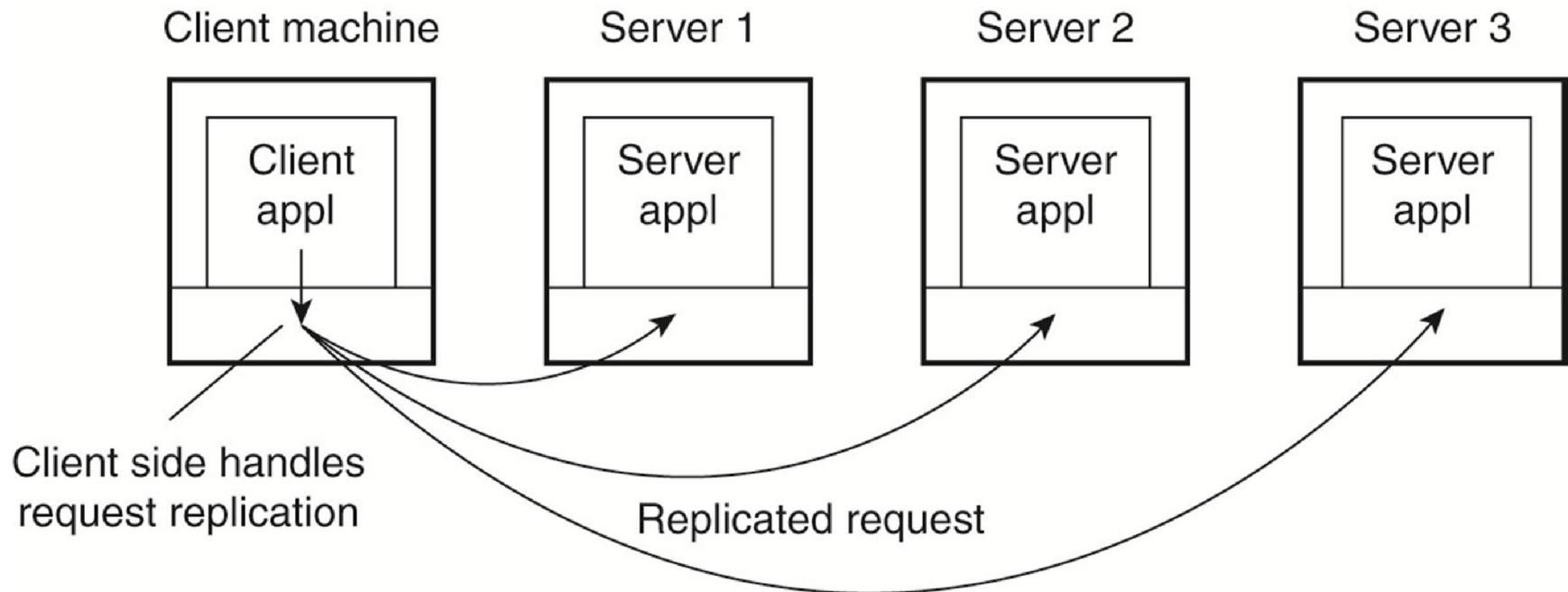
Example: The XWindow System



Client-Side Software for Distribution Transparency

- User interface is a relatively small part of the client software, in contrast to the local processing and communication facilities.
- Client software comprises components for achieving distribution transparency.
 - Access transparency: Client stub ([Chapter 4](#))
 - Location, migration, and relocation transparency: Naming ([Chapter 5](#))
 - Failure transparency : Client-side middleware. ([Chapter 8](#))
 - Replication transparency: Client-side proxy
 - Concurrency transparency and persistence transparency: Often completely handled at the server

Replication transparency



Outline

- Thread
- Virtualization
- Clients
- Servers
- Code Migration

Server (abstract)

- A server is a process implementing a specific service on behalf of a collection of clients.
 - It waits for an incoming request from a client and subsequently ensures that the request is taken care of;
 - After which it waits for the next incoming request.

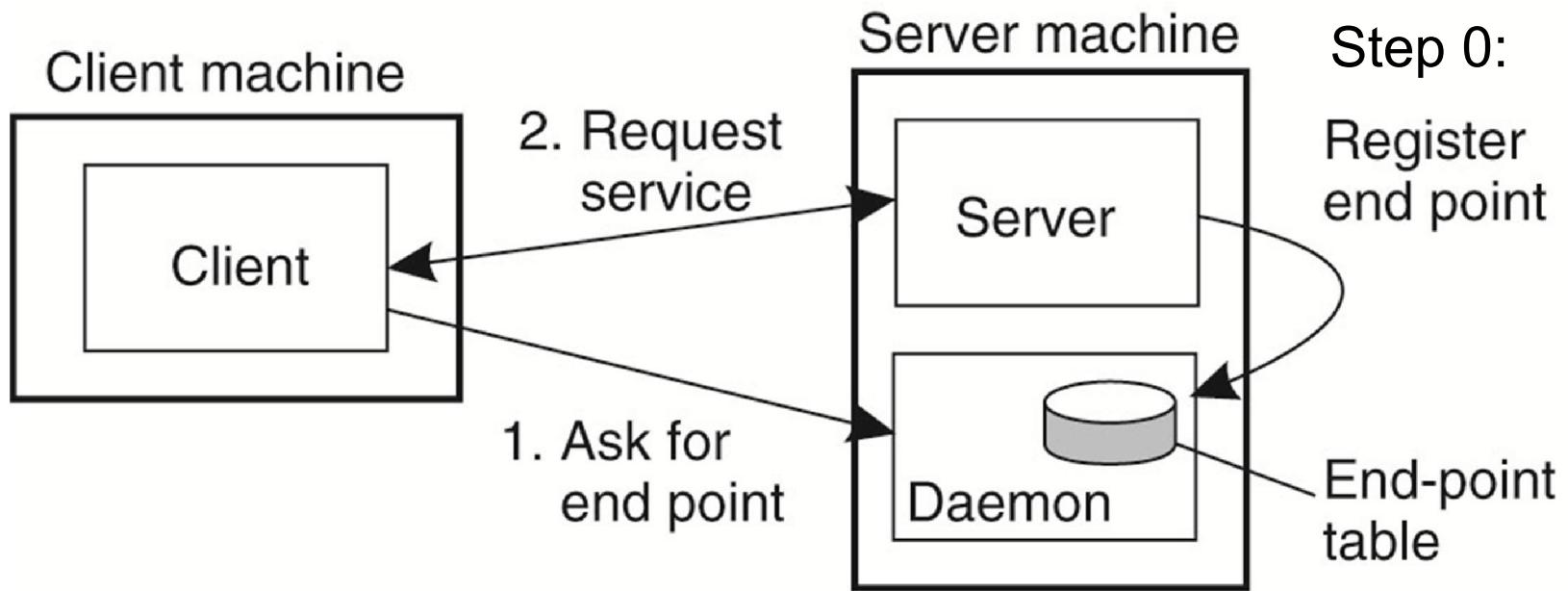
Terminology

- **Iterative server** handles the request and, if necessary, returns a response to the requesting client.
- **Concurrent server** does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request.

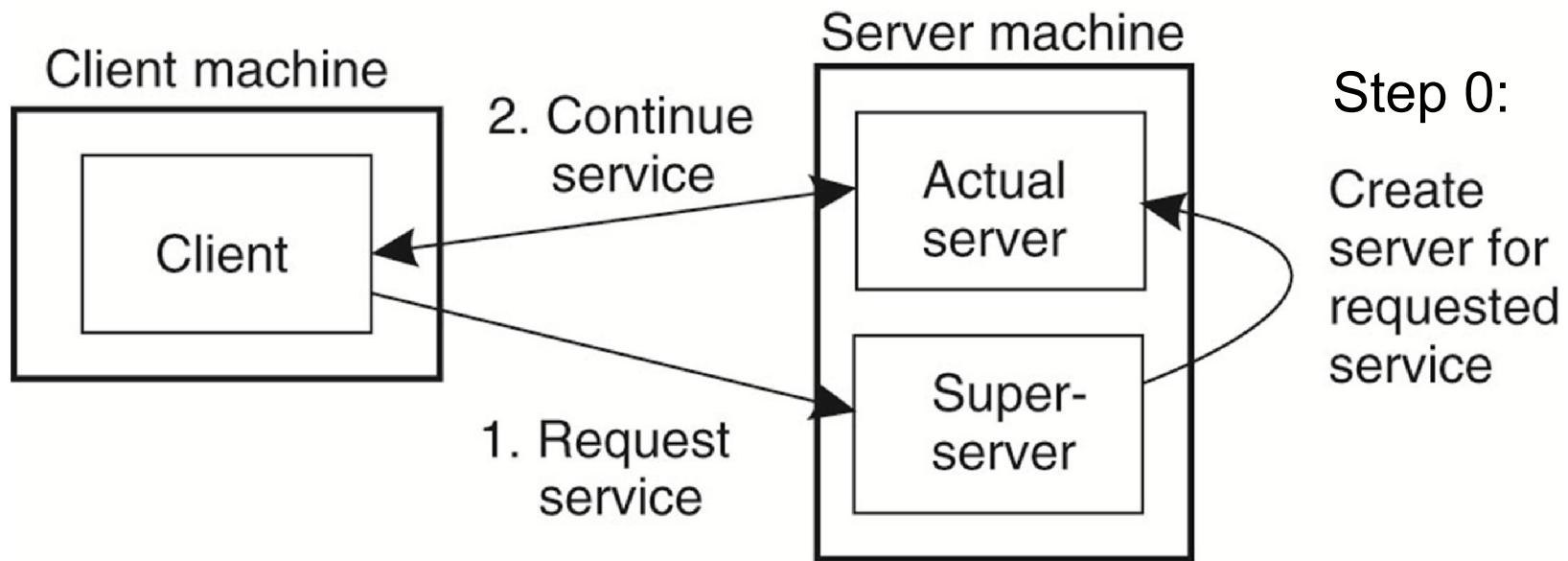
Client-to-server binding

- Port is an end point which the clients send requests to.
 - Globally assign port for well-known services, like 80;
 - Dynamic port banding by daemon running on each machine;
 - Dynamic port banding by super-server.

Client-to-server binding using a daemon



Client-to-server binding using a super-server



Stateless

- A stateless server does not keep information on the state of its clients, and can change its own state without having to inform any client.
 - Stateless server may actually maintain information on its clients, such as logs, but crucial is the fact that if this information is lost, it will not lead to a disruption of the service offered by the server.

Soft State

- The server promises to maintain state on behalf of the client, but only for a limited time.
- After that time has expired, the server falls back to stateless behavior.
 - An example: A server promising to keep a client informed about updates, but only for a limited time. After that, the client is required to poll the server for updates.

Stateful

- A stateful server generally maintains persistent information on its clients.
 - File server would maintain a table containing {client, file} entries. Such a table allows the server to keep track of which client currently has the update permissions on which file, and thus possibly also the most recent version of that file.

Session

- Session (temporary) state and permanent state
 - Session state are normally maintained in memory. If it is lost, client can simply reissue the original request.
 - Permanent state is typically client-specific information maintained in databases.

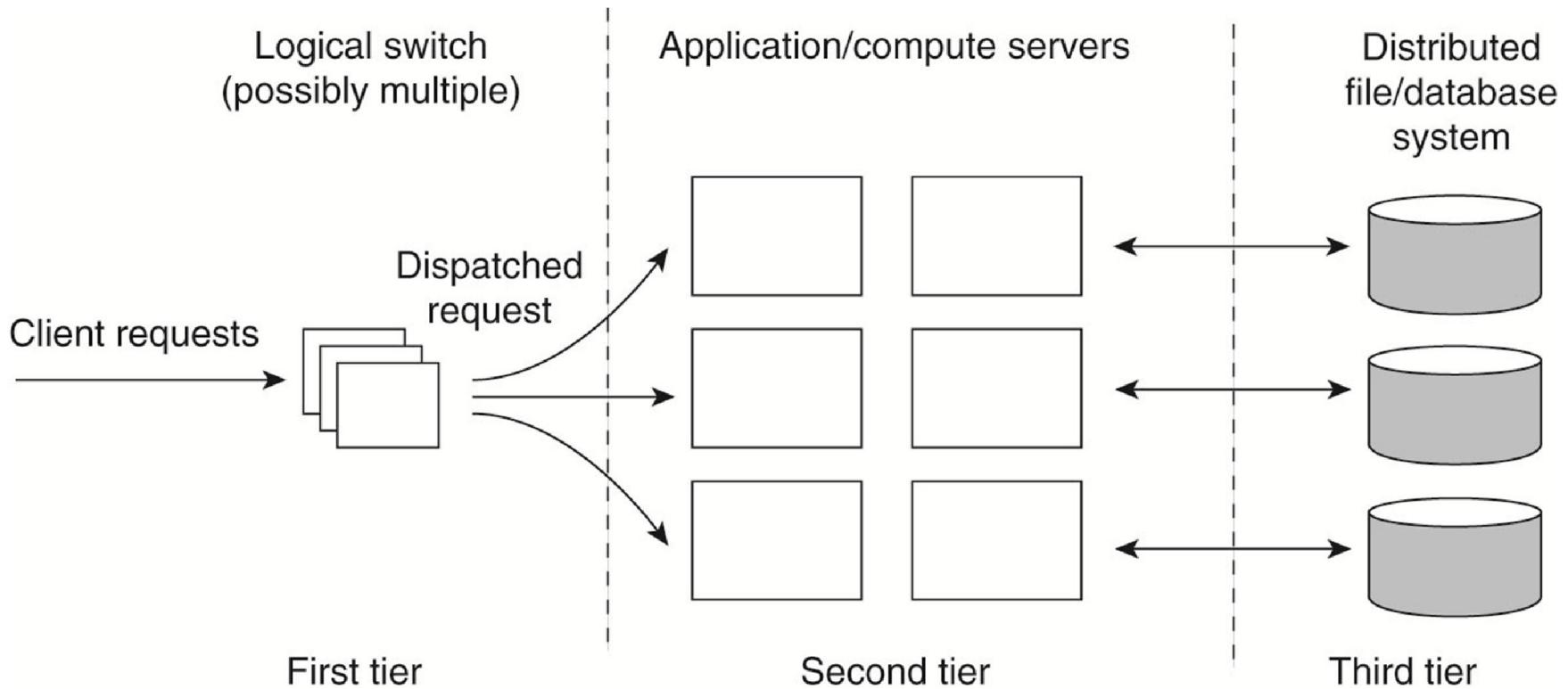
Stateless & Stateful

- Stateful servers improve performance over stateless servers;
- A stateful server needs to recover its entire state as it was just before the crash
 - Fault tolerance (Chapter 8)

For example

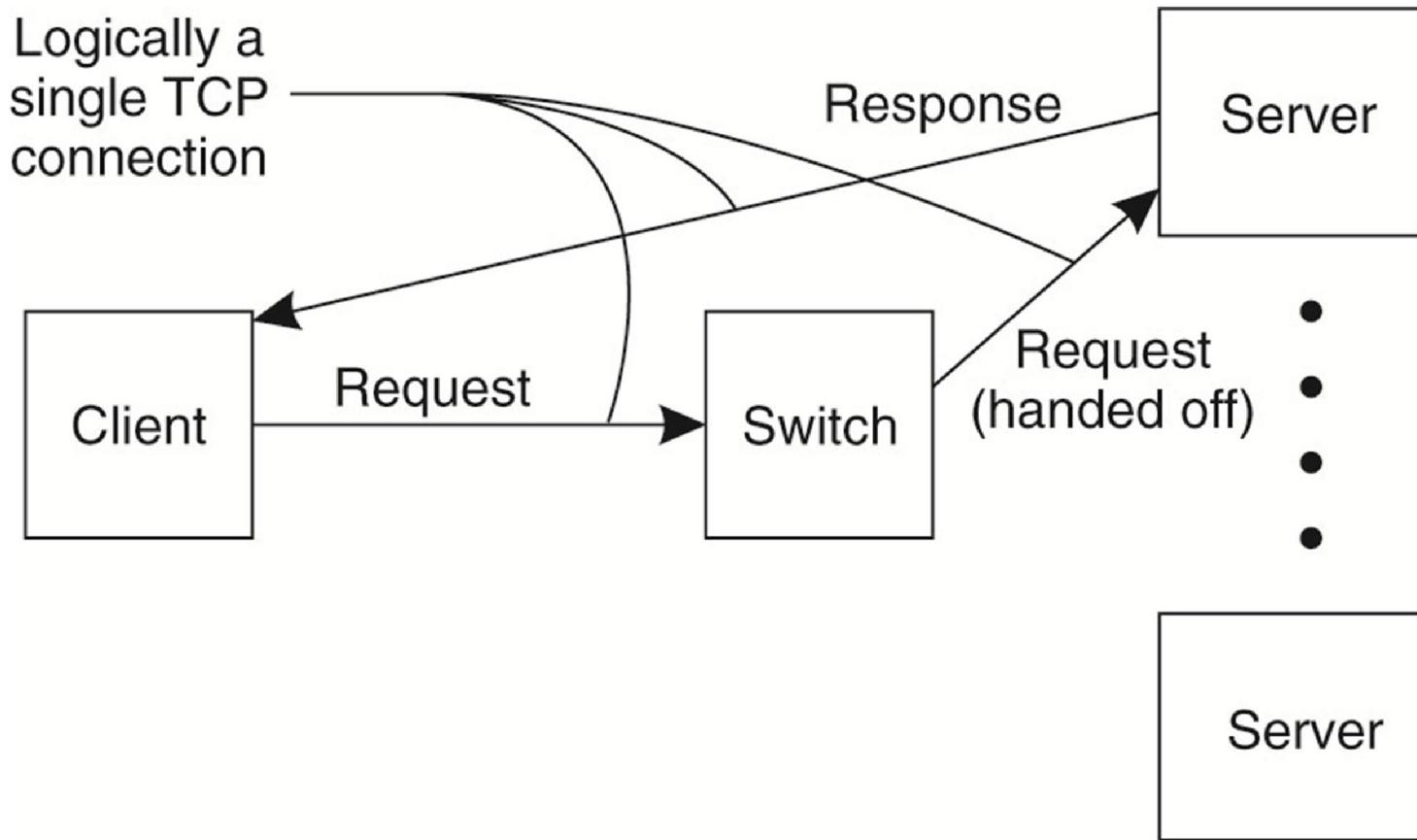
- Web server maintain session state;
- Client cookie maintain client-specific information that is of interest to the server;
- Each subsequent time the client accesses the server, its cookie for that server is sent along with the request.

Server Clusters



Bottleneck is not compute servers, but access to storage

TCP Handoff (转发)



Design goal for server clusters is to hide the fact that there are multiple servers

Managing Server Clusters

- p is the probability that a server is currently faulty, and we assume that faults are independent, then for a cluster of N servers to operate without a single server being faulty is $(1-p)^N$.
 - When $p=0.001$ and $N=1000$, there is only a 36 percent chance that all the servers are correctly functioning.

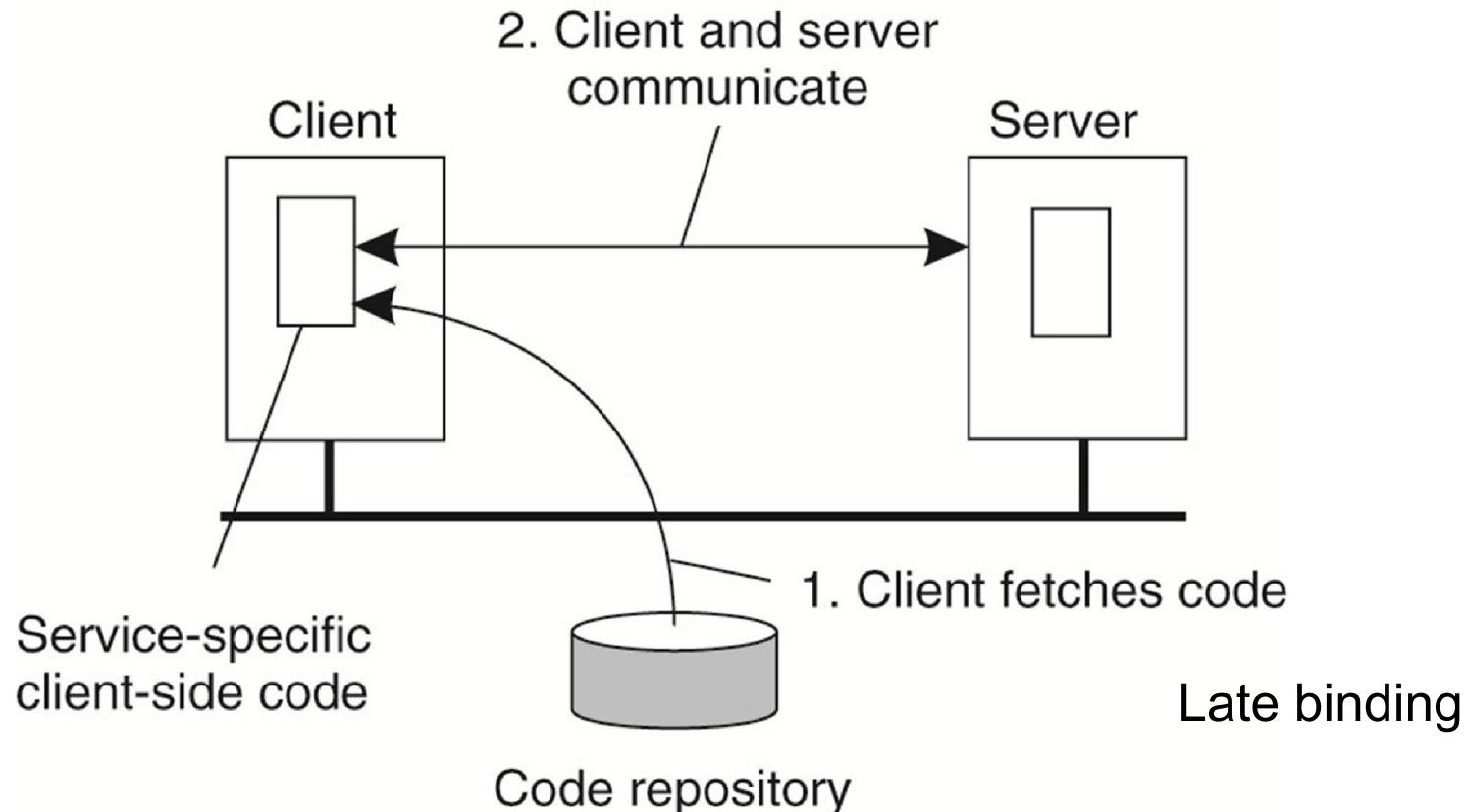
Outline

- Thread
- Virtualization
- Clients
- Servers
- **Code Migration**

Code Migration

- Communication to passing data or passing programs.
- Migration for performance reasons and flexibility
 - Move a running process from loaded host to another
 - Reduce communication load, delay
 - Exploit concurrency (e.g., mobile agents)
 - Late binding: after dynamic placement

Dynamically Configuring



Code Migration Models

- Running process has
 - Code segment
 - Resource segment: external resources needed
 - Execution segment: current execution state such as private data, stack, registers, pending signals, etc.
- Weak mobility & Strong mobility
- Sender-initiated (push) & Receiver-initiated (Pull)

Weak Mobility & Strong Mobility

- What is moved?
- Weak mobility: code with predefined starting points
 - Simple (e.g., applets)
- Strong mobility: execute segment
 - Tighter coupling, homogeneity requirements

Sender-initiated & Receiver-initiated

- Who initiates move?
- Host where code currently resides (has the job)
 - Sender-initiated (push when load is high)
- Target machine (wants the job)
 - Receiver-initiated (pull to lightly loaded host)

Sender-initiated (Push)

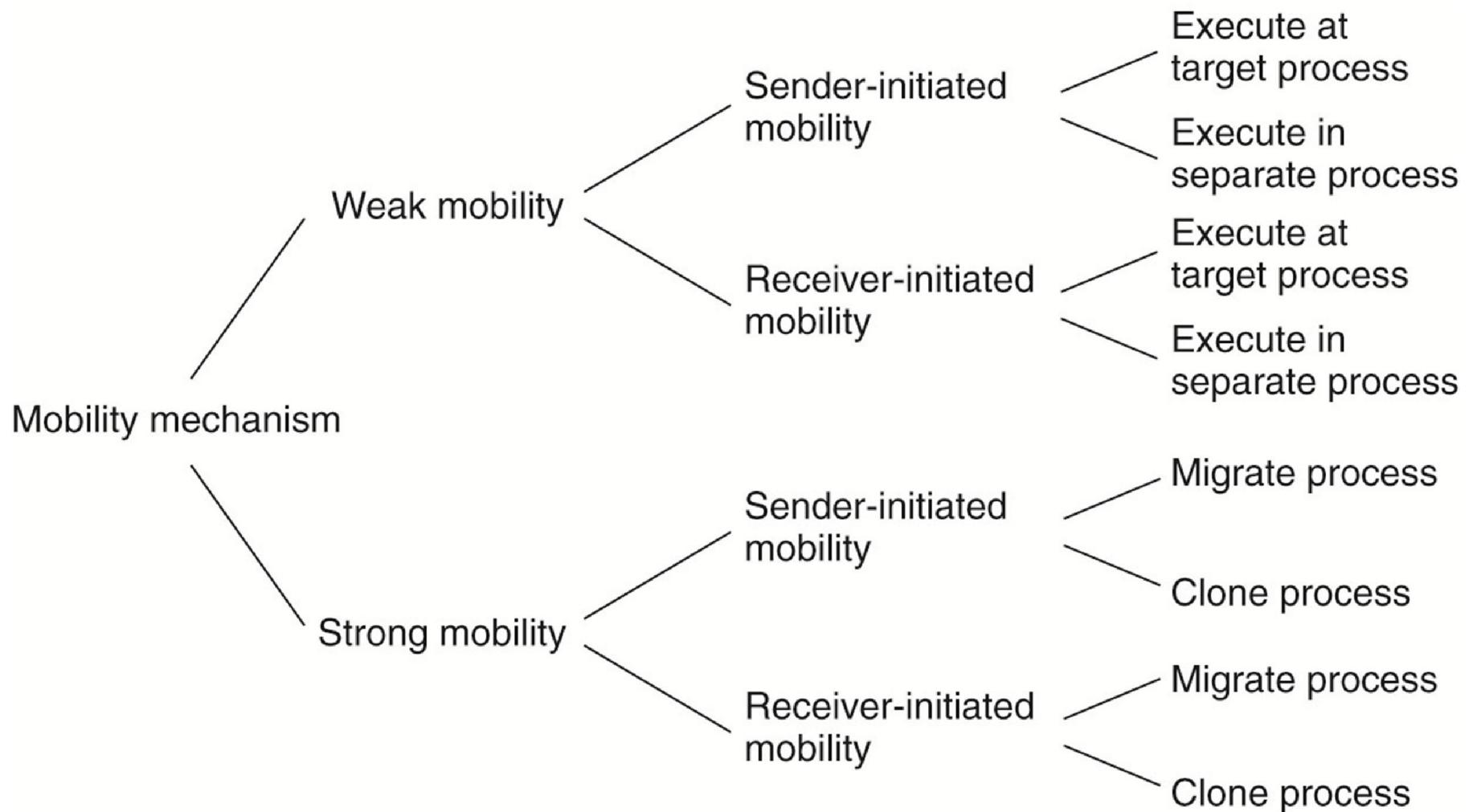
- Probe for current load levels
 - Send to least loaded target if load is lower
 - Can short-circuit if find unloaded host
- How many probes to send?
 - Too few, insufficient information
 - Too many, increased communication cost
 - Increased delay, an information gets stale
 - All senders pick same target
- Performance
 - Very low load, no overhead
 - Medium load, works pretty well
 - High load, disaster

Receiver-initiated (Pull)

- Probe for current load levels
 - Pull from heaviest loaded source if heavier load
- How many probes to send?
 - Similar issues as push
- Performance
 - Low load, lots of pull probes, but who cares?
 - Medium load, works OK, not as good as Push
 - High load, no overhead, works very well!

Resources Migration

- Code migration so difficult is that the resource segment cannot always be simply transferred along with the other segments without being changed.
 - For example: TCP port
- Process-to-resource bindings
 - Binding by identifier: URL as a example
 - Binding by value: Lib for program
 - Binding by type: printer as a example



Resources Migration

- Unattached resources
 - Data files associated only with the program.
- Fastened resources
 - May be possible with high costs.
 - Local databases and complete Web sites.
- Fixed resources
 - Bound to a specific machine or environment and cannot be moved.
 - Local devices.
 - Local communication end point.

Resources Migration

		Resource-to-machine binding		
Process-to-resource binding		Unattached	Fastened	Fixed
	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV,GR)	GR (or CP)	GR
	By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR Establish a global systemwide reference
MV Move the resource
CP Copy the value of the resource
RB Rebind process to locally-available resource

Next Lesson...

DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 4
Communication

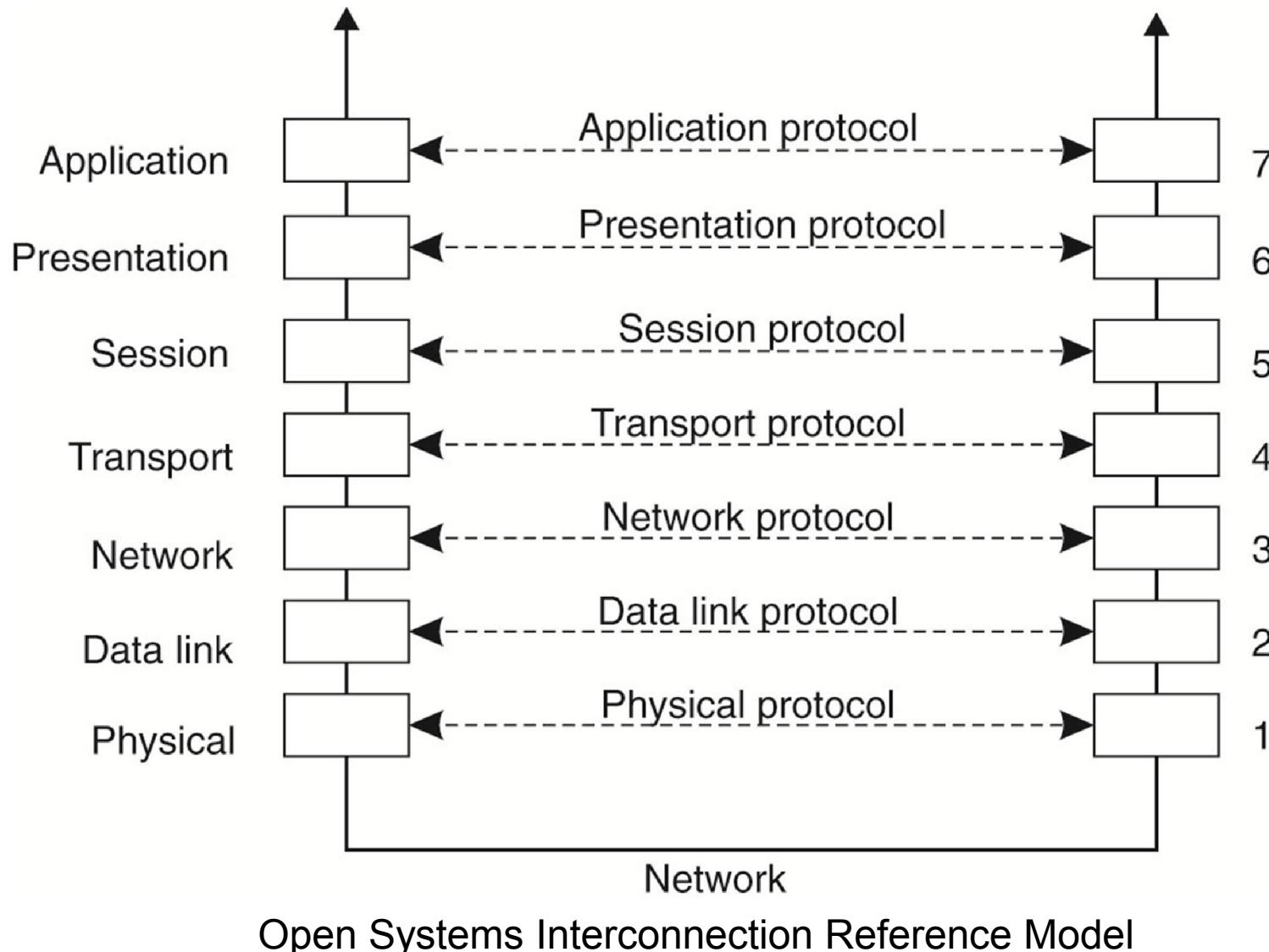
Outline

- Fundamentals
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication
- Multicast Communication

Terminology

- Layered Protocols
 - **Protocol**: the rules that communicating processes must adhere to
 - **Layers**: structuring of protocols
- **Connection oriented protocols**: before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol they will use.
- **Connectionless protocols**: no connection is setup, the sender just transmits the first message when it is ready.

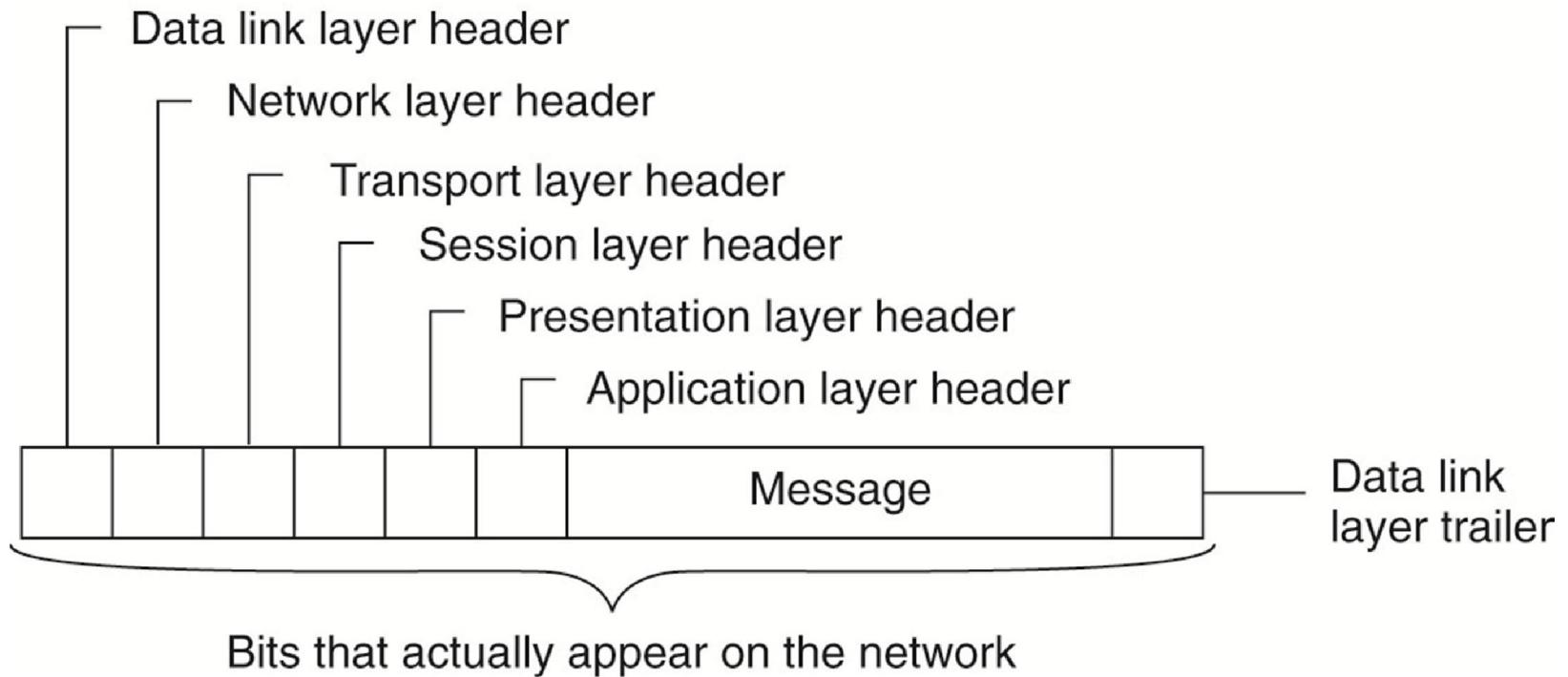
Layered Protocols



Layered Protocols

- Each layer gets services from layer immediately below it.
- Each layer provides services to the layer immediately above it.
- The interface definition specifies **Service Data Unit** (SDU) from the layer above and the **Protocol Data Unit** (PDU) given to the layer below
 - PDU of one layer is the SDU of the layer below

Encapsulated message



Note: Encapsulated message may be aggregated or fragmented by lower layer

Lower-Level Protocols

- The **physical layer** is concerned with transmitting the 0s and 1s, How many volts? how many bits per second ? whether both directions? size and shape of the network connector, number of pins and meaning of each are of concern ?
- The **data link layer** detect and correct errors by putting a special bit pattern on the start and end of each **frame**, as well as computing a checksum by adding up all the bytes in the frame.
- The **network layer** is for routing, which choose the best path between sender and receiver.

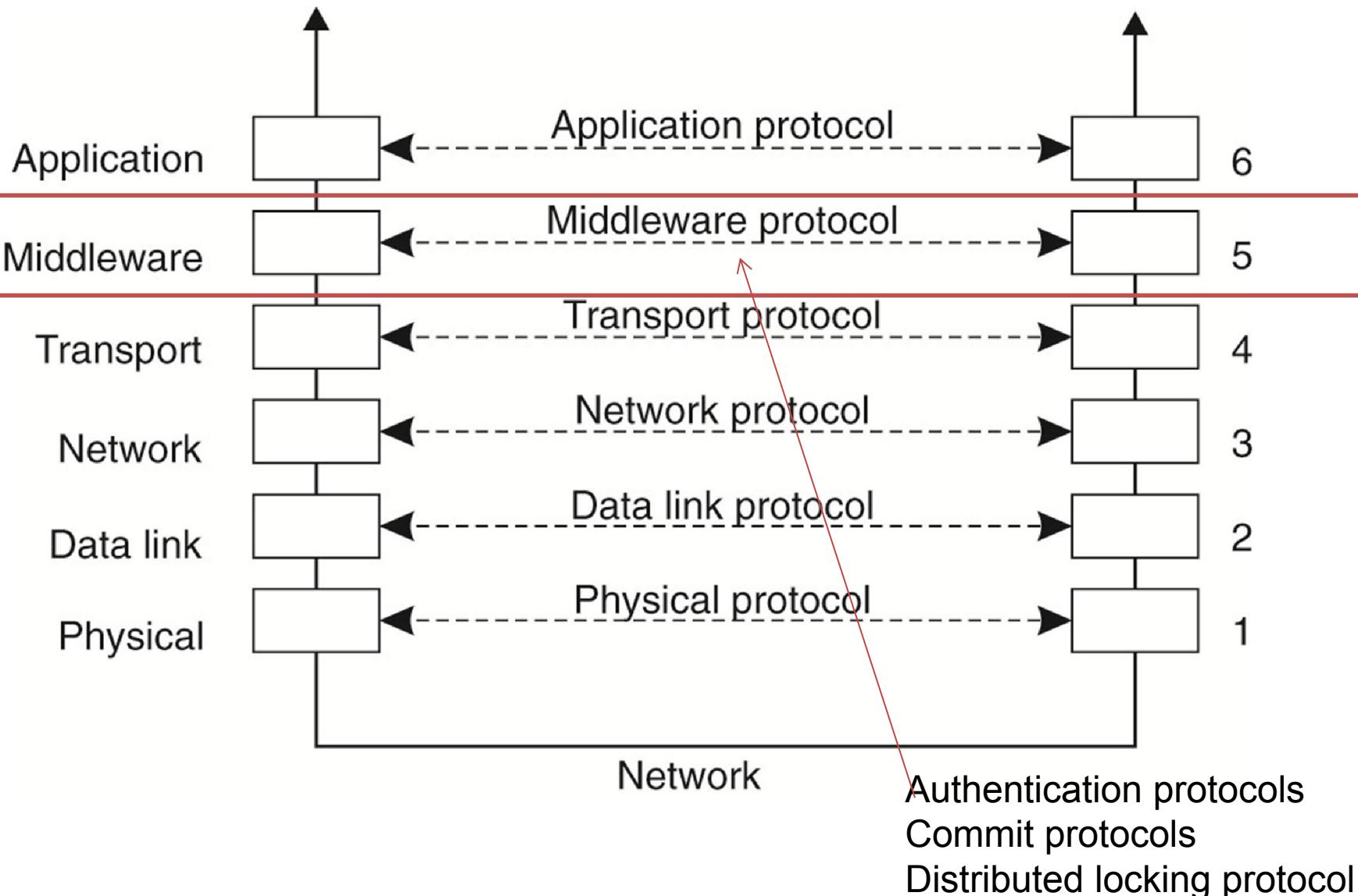
Transport Protocols

- **Transport layer** turns the underlying network into something that an application developer can use.
- **Transport layer** breaks message into packages small enough for transmission, assigns each one a sequence number, and then sends them all.

Higher-Level Protocols

- The **session layer** is essentially an enhanced version of the transport layer. It provides dialog control, to keep track of which party is currently talking, and it provides synchronization facilities.
- The **presentation layer** define structure (format) of records and then have the sender notify the receiver that structure.
- The **application layer** was originally intended to contain a collection of standard network applications such as those for electronic mail, file transfer, and terminal emulation.

Middleware Protocols



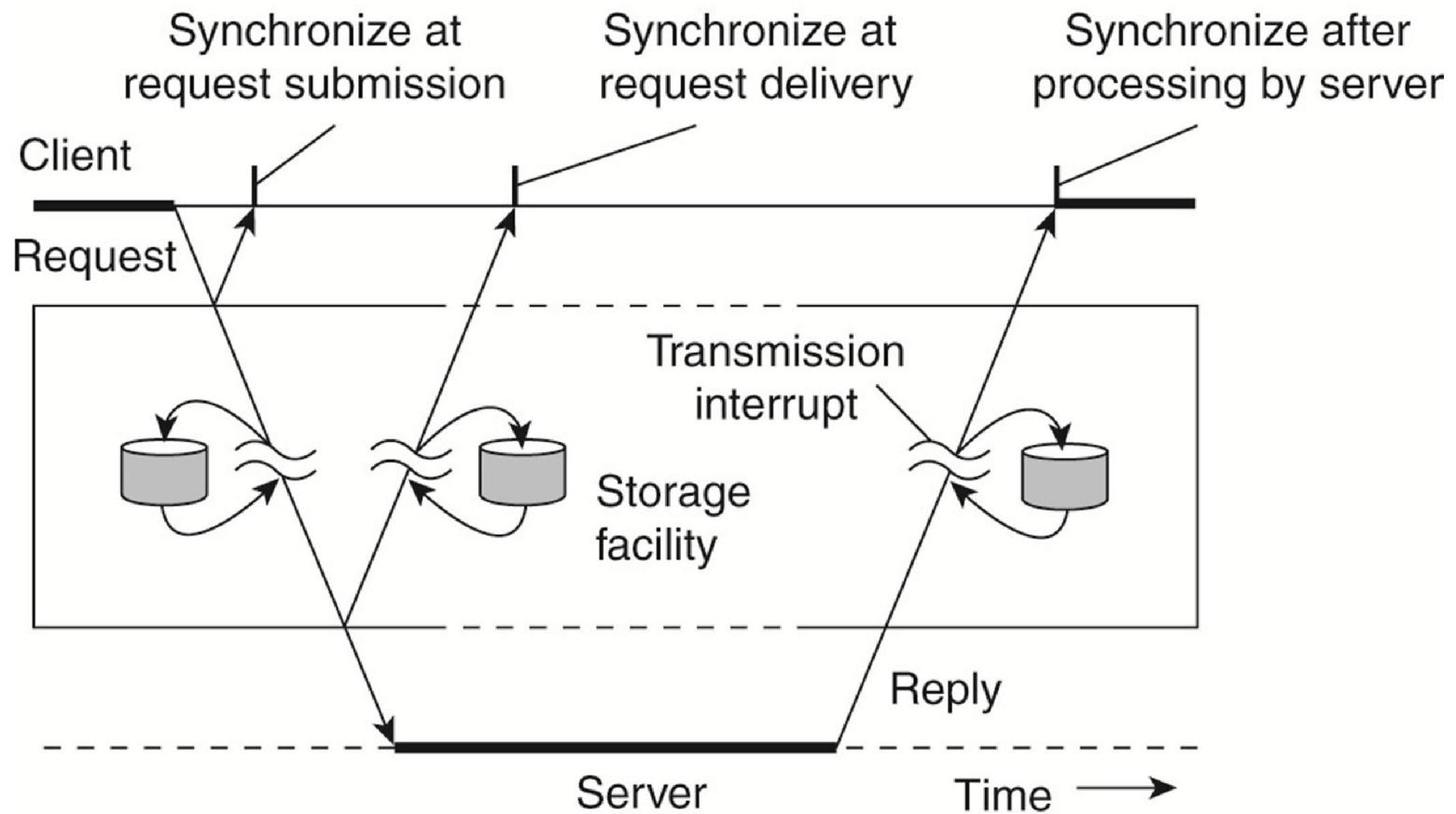
Types of Communication in Middleware

- **Persistent Communication**, a message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver.
- **Transient Communication**, a message is stored by the communication system only as long as the sending and receiving application are executing.

Types of Communication in Middleware

- **Asynchronous communication:** sender continues immediately after it has submitted its message for transmission.
 - Message is (temporarily) stored immediately by the middleware upon submission.
- **Synchronous communication:** sender is blocked until its request is known to be accepted.
 - Synchronized at request submission;
 - Synchronized at request delivery;
 - Synchronized after processing by server.

Types of Communication



Outline

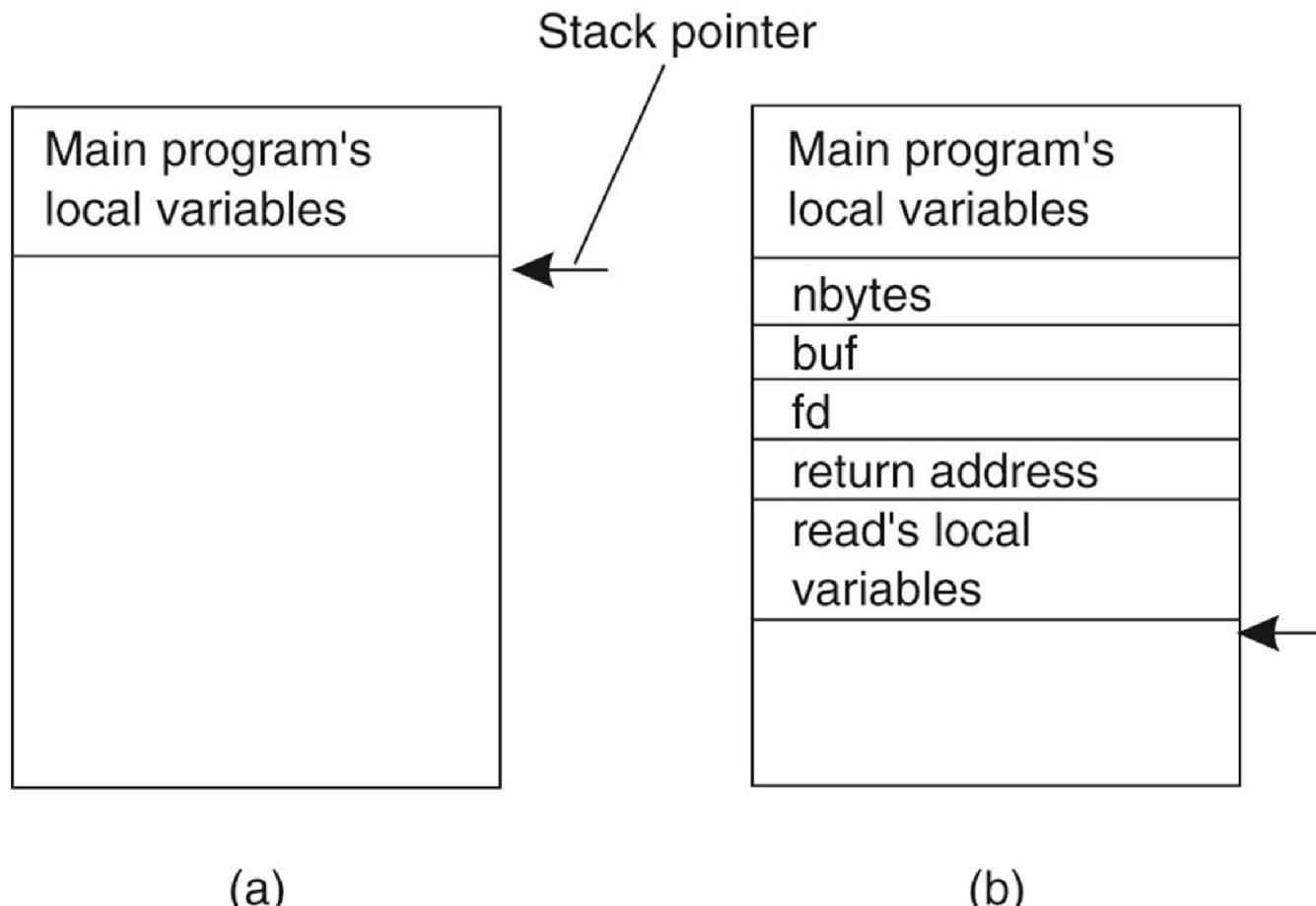
- Fundamentals
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication
- Multicast Communication

Remote Procedure Call (1984)

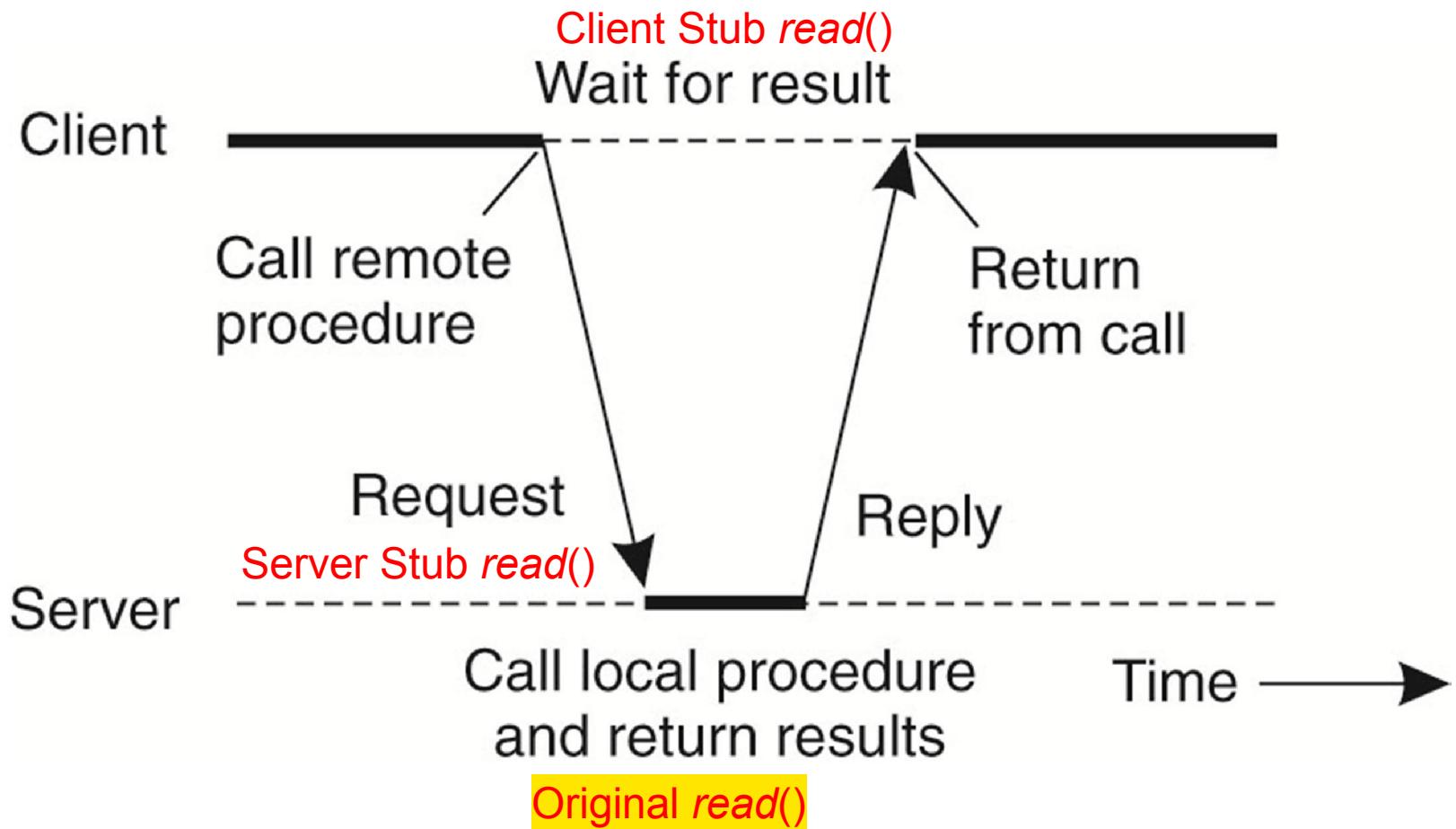
- The idea behind RPC is to make a remote procedure call look as much as possible like a local one.
- When a process on machine *A* calls' a procedure on machine *B*, the calling process on *A* is suspended, and execution of the called procedure takes place on *B*.
 - Information can be transported as the parameters and can come back in the procedure result.
 - No message passing at all is visible to the programmer.

Conventional Procedure Call

- $count = \text{read}(fd, buf, nbytes);$
 - fd is a file, buf is an array of characters, and $nbytes$ is integer



Client and Server Stubs



Client Stub

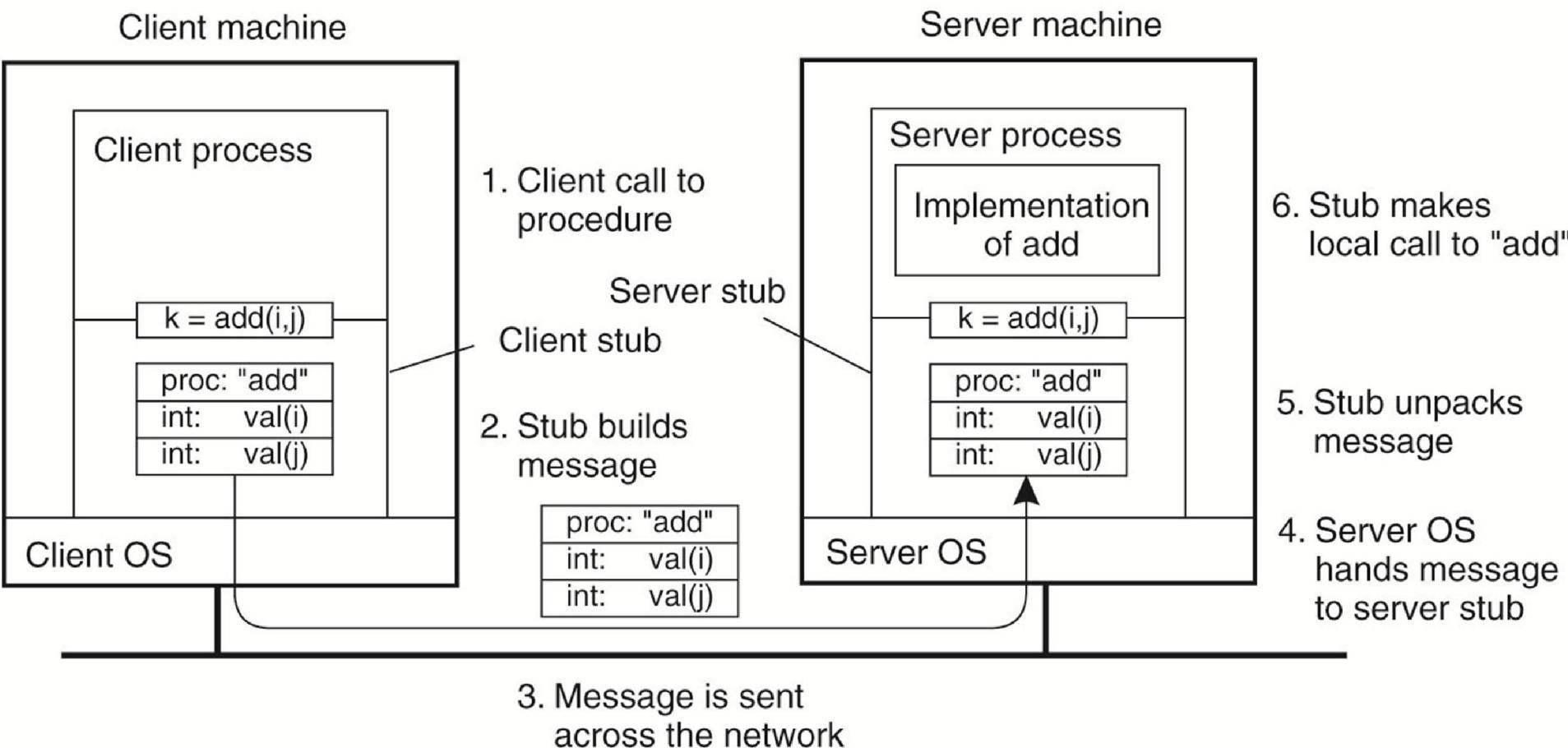
- When *read()* is actually a remote procedure, a different version of *read()* , called a client stub, is put into the library.
- When *read()* is invoked, actually a client stub *read()* is invoked.
- Client stub *read()* call to the local operating system.
 - Unlike the original one, it does not ask the operating system to give it data;
 - It packs the parameters into a message and requests that message to be sent to the server.

Server Stubs

- A server stub is a piece of code that transforms requests coming in over the network into local procedure calls.
- Typically the server stub **will have called** *receive* and be blocked waiting for incoming messages.
- The server stub unpacks the parameters from the message and then calls the server procedure in the usual way.

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local OS.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the server stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The client stub unpacks the result and returns to the client.

Passing Value Parameters

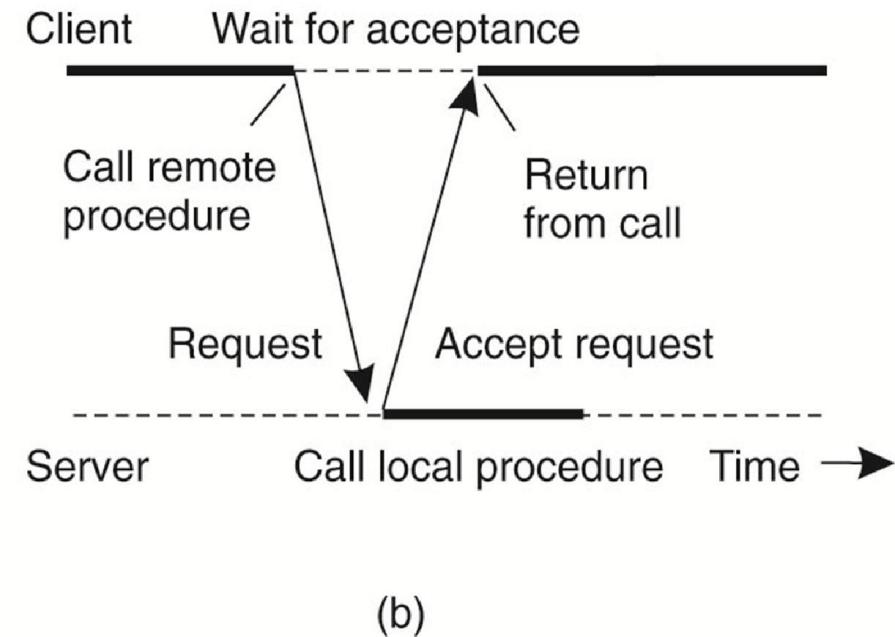
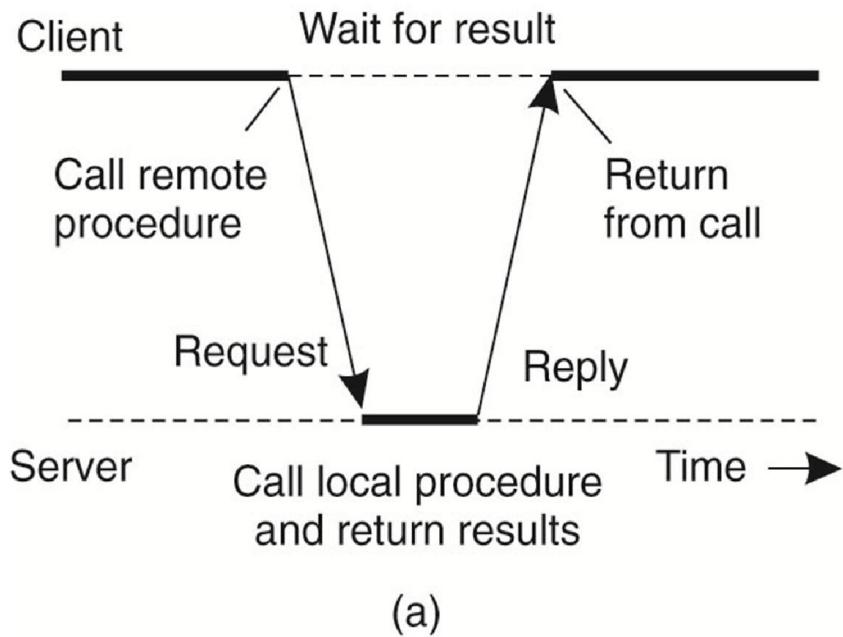


As long as the client and server machines are identical and all the parameters and results are scalar types, such as integers, characters, and booleans, this model works fine

Passing Reference Parameters

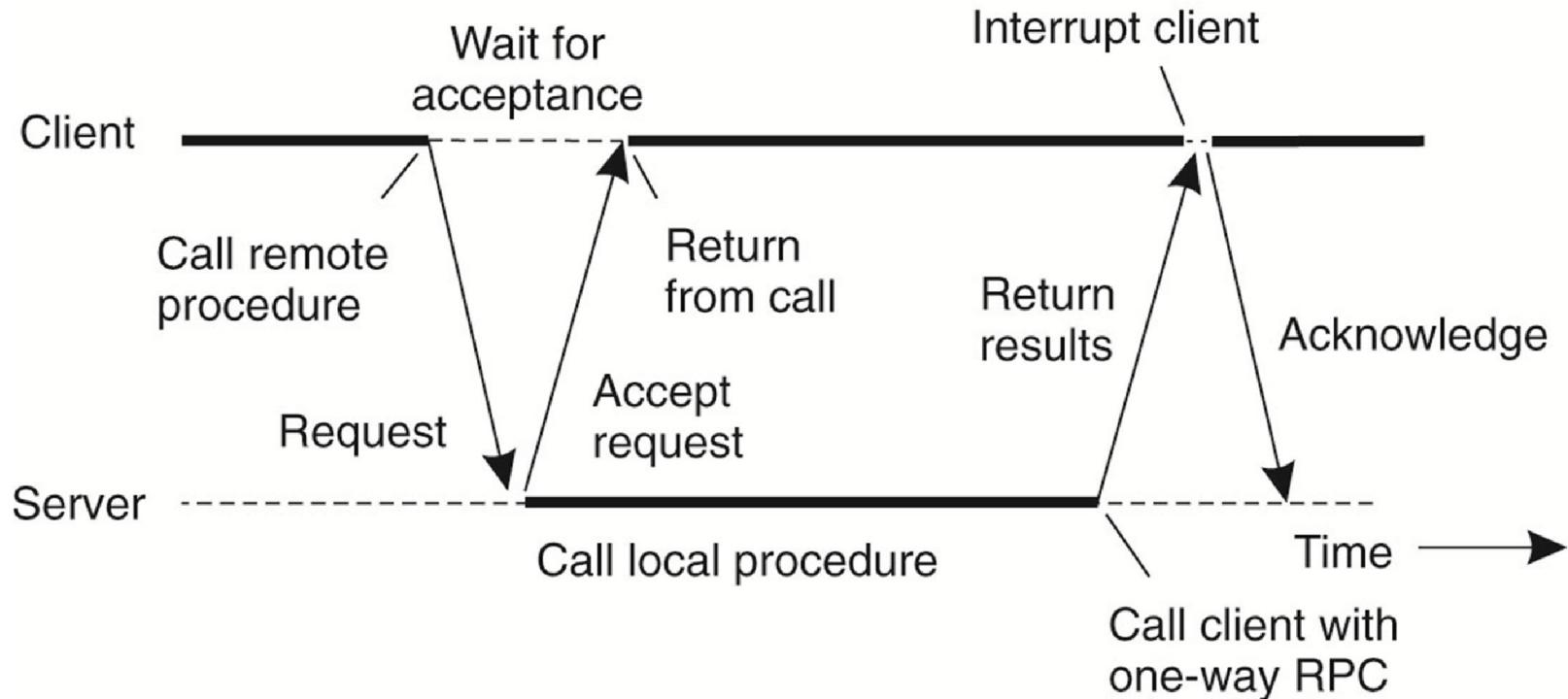
- Generally, forbid pointers and reference parameters in general.
- Serialization and deserialization

Asynchronous RPC



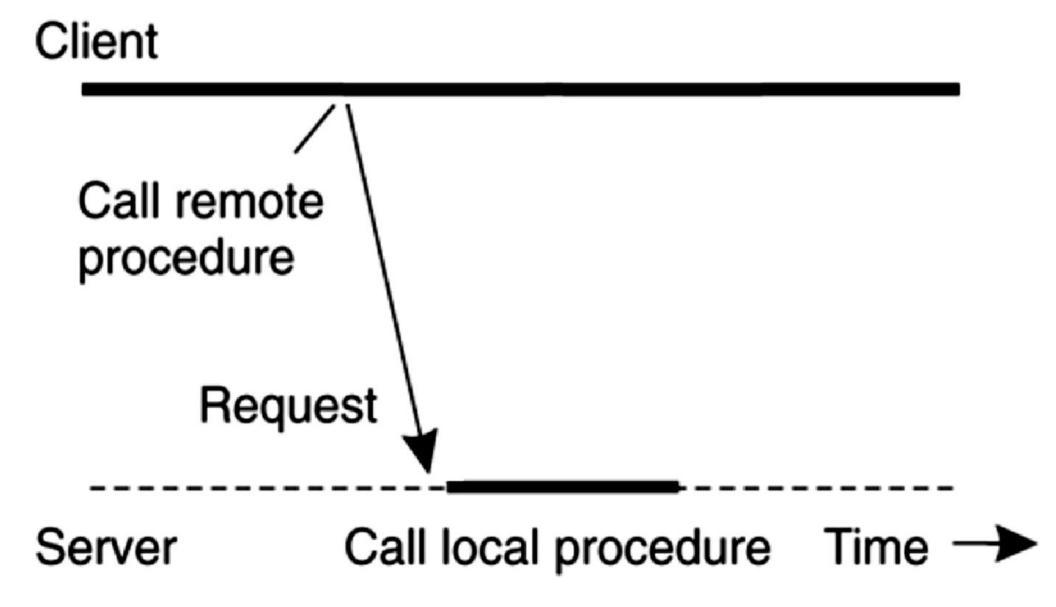
- Adding entries into a database, starting remote services, batch processing

Deferred Synchronous(延迟同步) RPC



One-way RPC

- Client does not wait for an acknowledgment of the server's acceptance of the request.
- Reliability is not guaranteed



Outline

- Fundamentals
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication
- Multicast Communication

Message Oriented Communication

- Remote Procedure Calls and Remote Method Invocations contribute to hiding communication in distributed systems
- Alternative communication services are needed if cannot be assumed that the receiving side is executing at the time a request is issued
 - Message-Oriented **Transient** Communication
 - Socket
 - MPI
 - Message-Oriented **Persistent** Communication
 - MQ

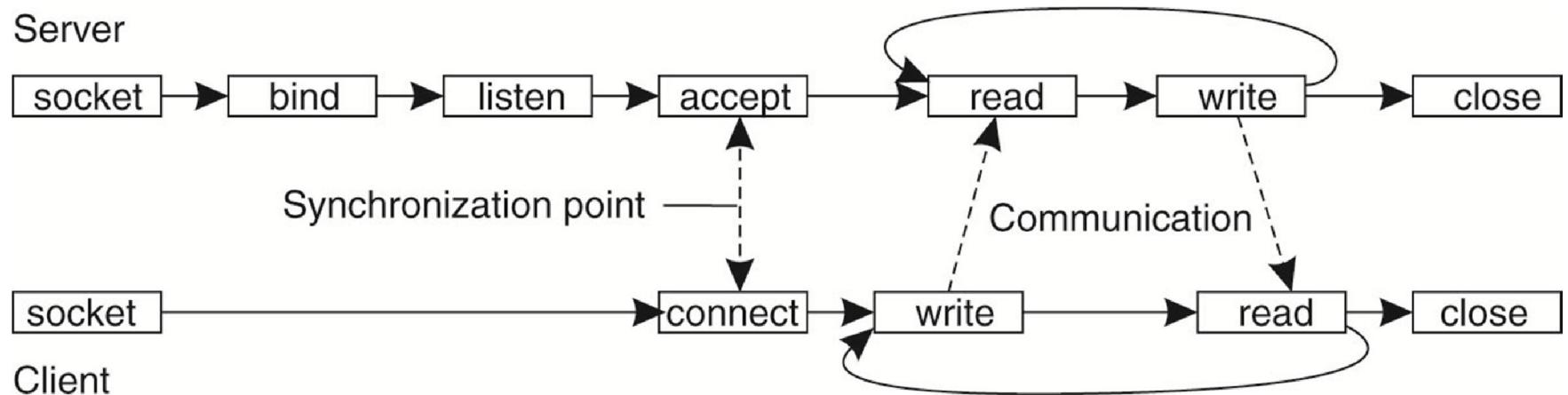
Sockets

- A socket is a **communication end point**
 - Data, which are to be sent out over the underlying network, is written to the socket
 - Data can be read from socket
- A socket is an abstraction over the actual communication end point that is used by the local operating system for incoming a specific transport protocol.

The socket primitives (TCP/IP)

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Connection-oriented communication pattern using sockets



Message-Passing Interface (MPI)

- Socket is insufficient for highly efficient applications
 - Supporting only simple send and receive primitives.
 - Designed to communicate across networks using general-purpose protocol stacks such as TCP/IP.
- Protocols of high-performance server clusters required an interface that could handle more advanced features, such as different forms of buffering and synchronization.

Message Passing Primitives of MPI (core part)

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

- MPI has been designed for high-performance parallel applications
- MPI diversity in different communication primitives

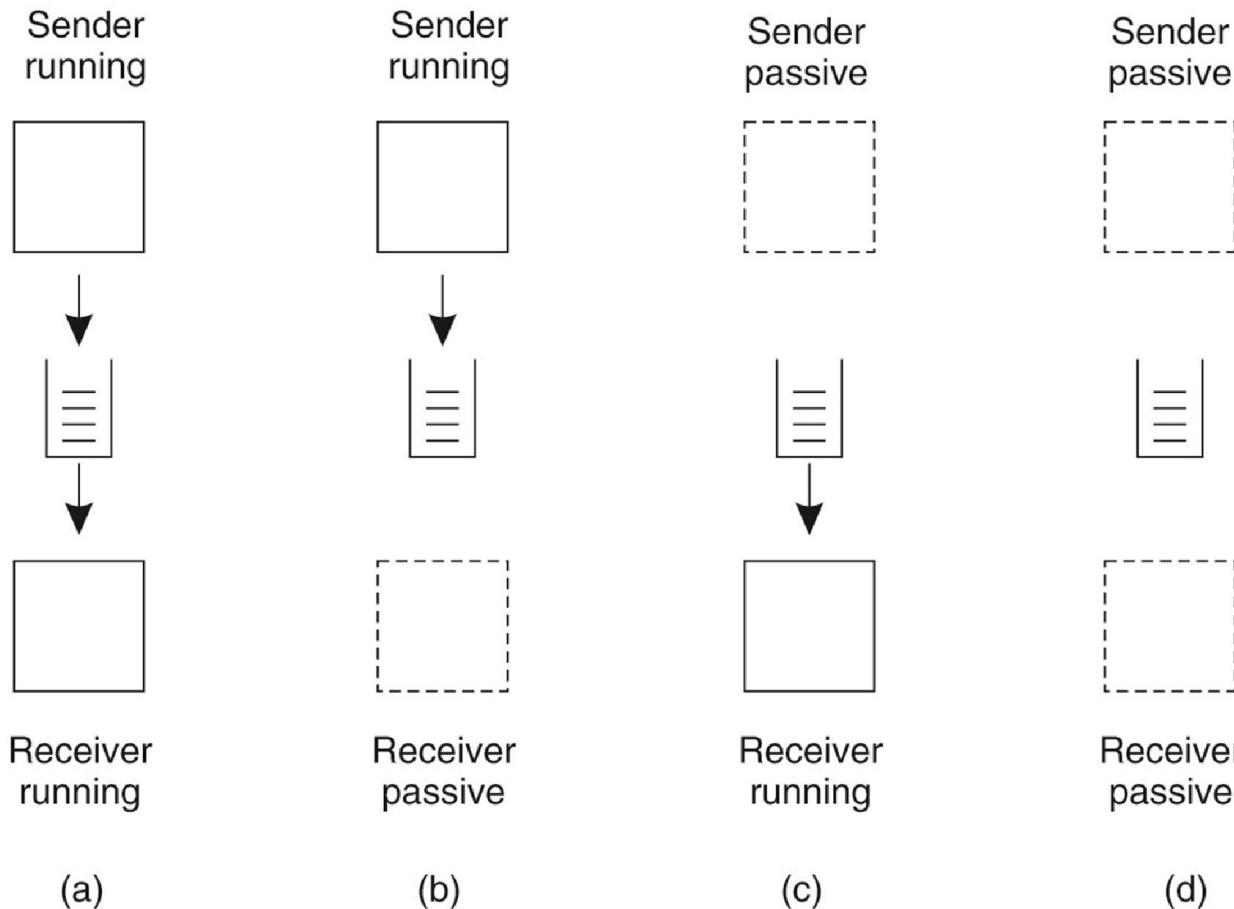
Message-Oriented Persistent Communication

- Message Queuing Systems (MQS)
- Message Oriented Middleware (MOM)
 - They offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission.

Message-Queuing Model

- Applications communicate by inserting messages in specific queues.
 - Eventually delivered to the destination, even if it was down when the message was sent.
- Each application has its own private queue
 - A queue can be read only by its associated application,
- Multiple applications to share a single queue.

Message-Queuing Model



- System is storing (and possibly transmitting) messages even while sender and receiver are passive

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

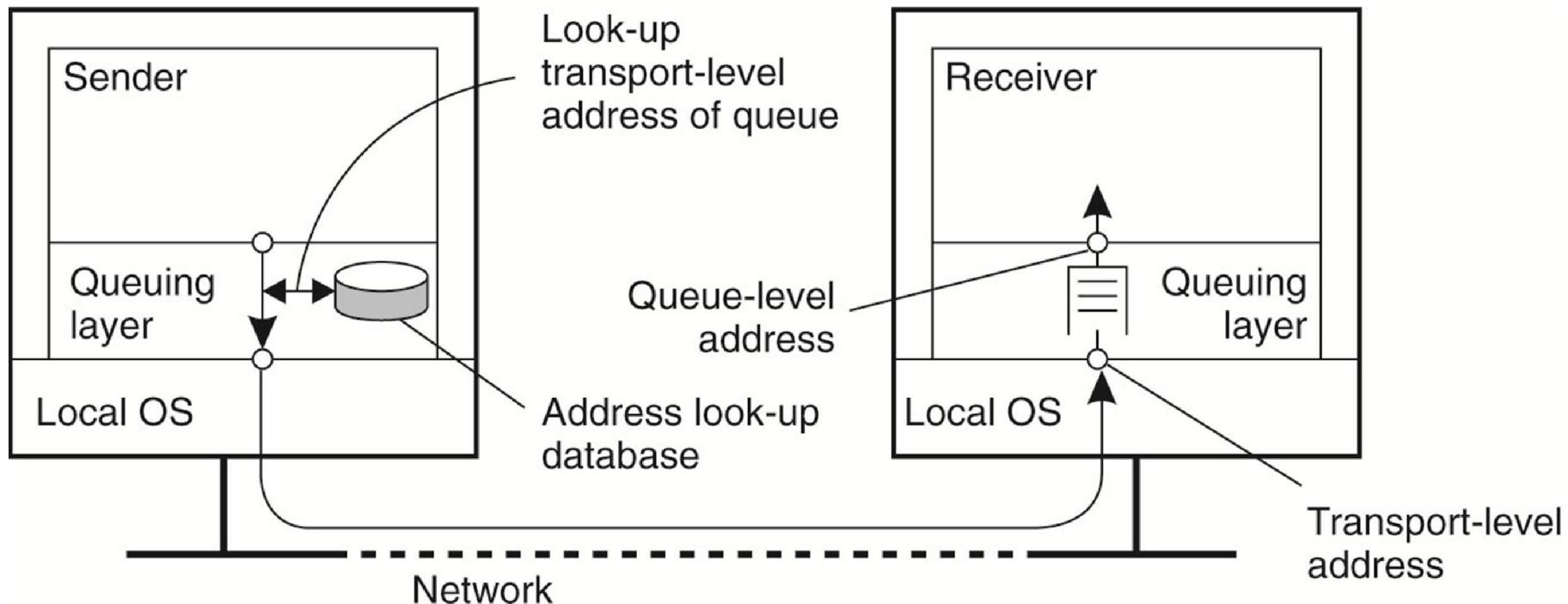
Callback function

- **Callback function** is automatically invoked whenever a message is put into the queue.
 - Callbacks can also be used to automatically start a process that will fetch messages from the queue.
- **Daemon** on the receiver's side that continuously monitors the queue for incoming messages and handles accordingly.

Architecture of a MQ System

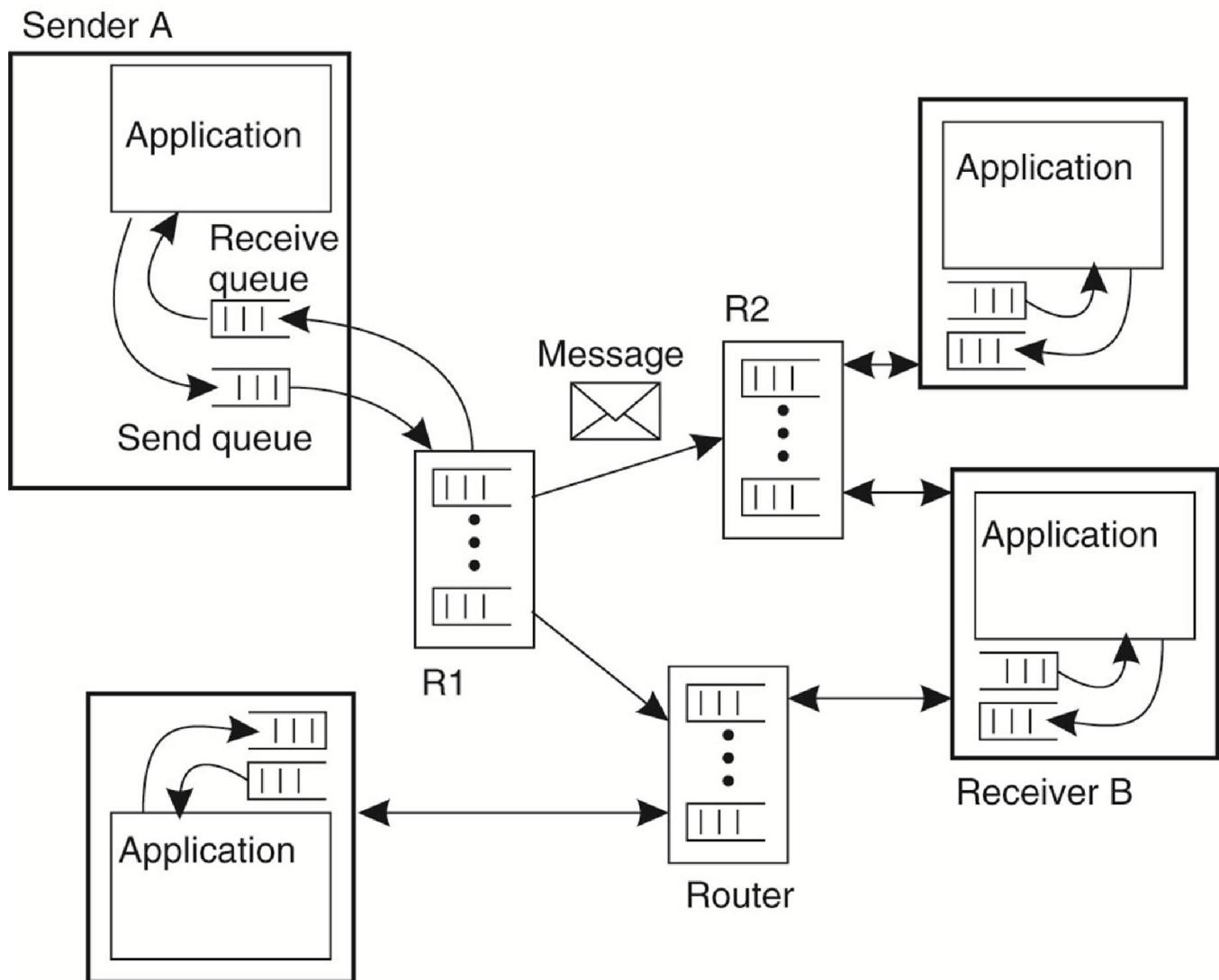
- Source queue
- Destination queue
- Queue manager
 - Manager the queue
 - May operate as routers, or relays
- Message broker is an application-level gateway to convert incoming messages so that they can be understood by the destination application.
- Overlay network

Source and Destination Queues

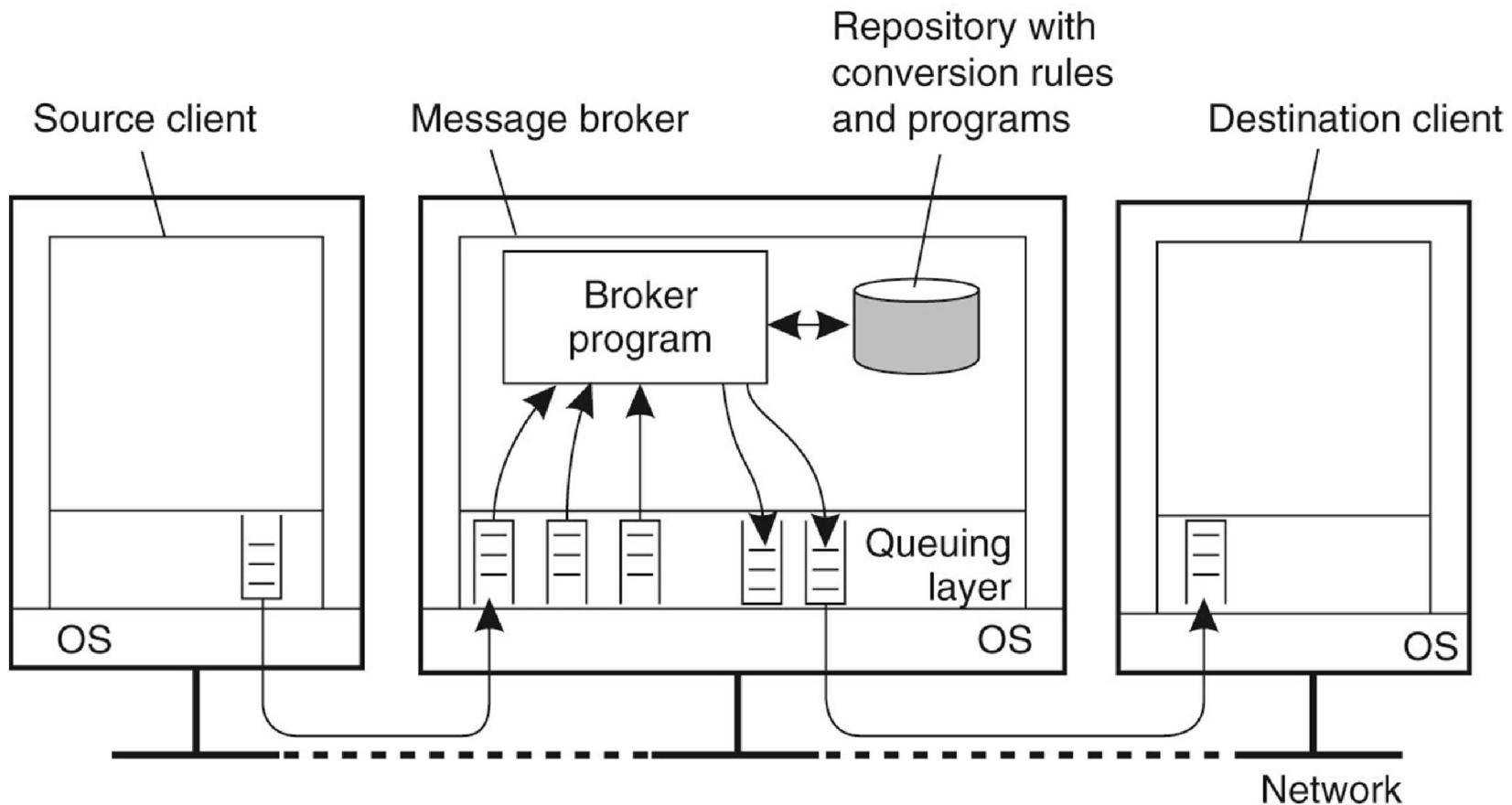


The relationship between queue-level addressing and network-level addressing.

Relays of a MQ System



Message Brokers

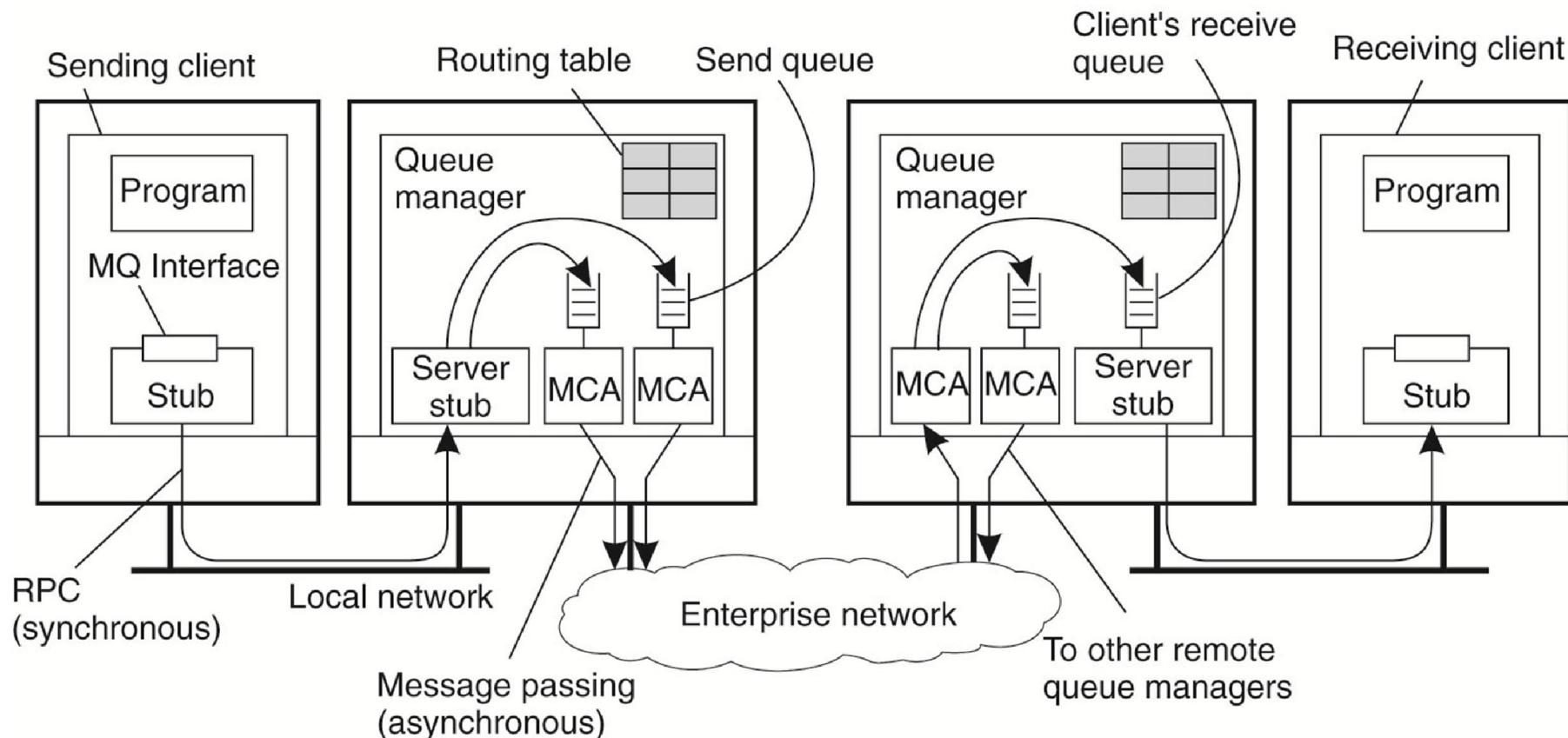


MQ System Applications

- E-mail
- WorkFlow
- Batch Processing
- EAI (Enterprise Application Integration)
- Integration of a (possibly widely-dispersed) collection of databases and applications into a federated information system

IBM's WebSphere MQ System 8.0

23 May 2014



MAC: Message Channel Agent

Outline

- Fundamentals
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication
- Multicast Communication

Stream-oriented Communication

- RPC and MQ
 - It does not matter at what particular point in time communication takes place.
 - Although a system may perform too slow or too fast, timing has no effect on correctness.
- Stream-oriented Communication
 - Distributed system offer to exchange time-dependent information such as audio and video streams.

Multimedia Streams

- **Discrete** (representation) media – temporal relationships not crucial
 - Text
 - Images
 - Code
- **Continuous** (representation) media – temporal relationships between data items crucial to correct interpretation
 - Audio
 - Video

Data Streams

- A data stream is a sequence of data units, it can be applied to discrete as well as continuous media.
 - UNIX pipes or TCP/IP connections are typical examples of (byte-oriented) discrete data streams.
 - Playing an audio file typically requires setting up a continuous data stream between the file and the audio device.

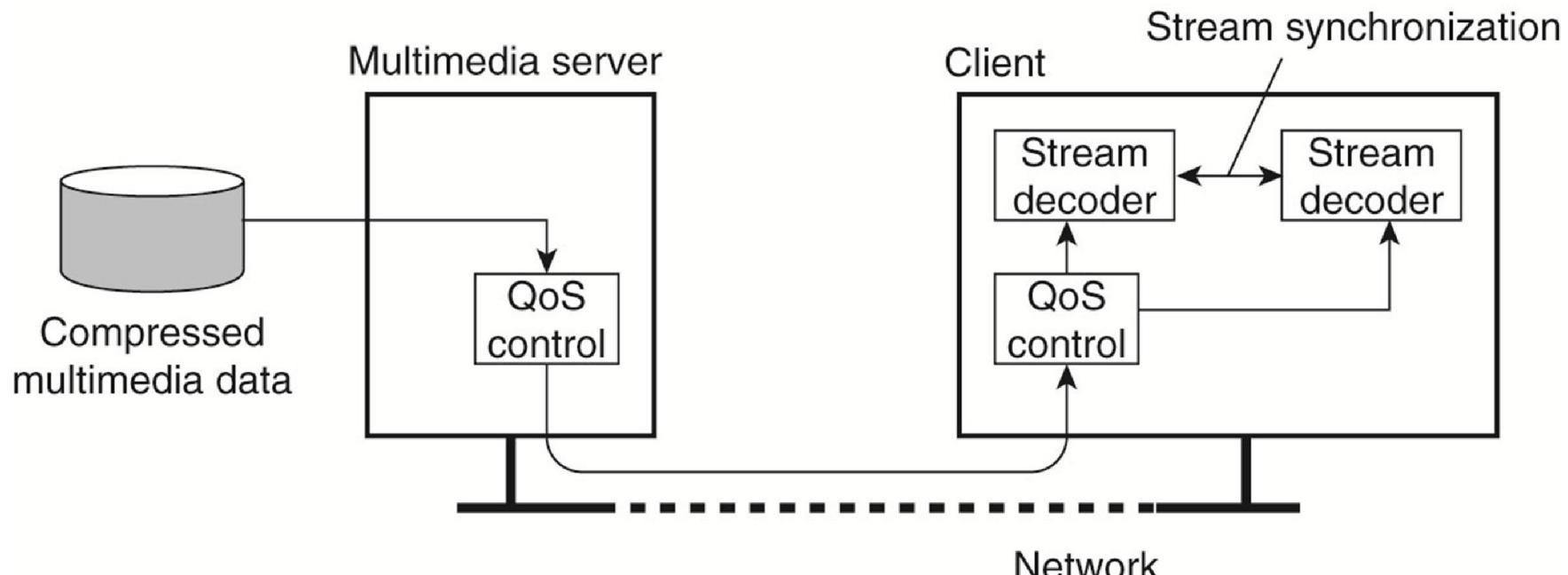
Data Streams

- The data items in a stream are transmitted one after the other, but
 - **Asynchronous** (异步) **transmission mode**, there are no further timing constraints on when transmission of items should take place.
 - **Synchronous** (同步) **transmission mode**, there is a maximum end-to-end delay defined for each unit.
 - **Isochronous** (等步) **transmission mode**, there are maximum and minimum end-to-end delay
 - Data units are transferred on time.
 - bounded (delay) jitter.
- In this section, continuous data streams using isochronous transmission, simply as streams.

Simple and Complex Streams

- A simple stream consists of only a single sequence of data
- A complex stream consists of several related simple streams, called substreams.
 - The relation between the substreams in a complex stream is often also time dependent.
 - Stereo audio is complex stream consisting of two substreams, each used for a single audio channel. Those two substreams are continuously synchronized.
 - Movie contains four streams, which of them?

Complex Streams Communication



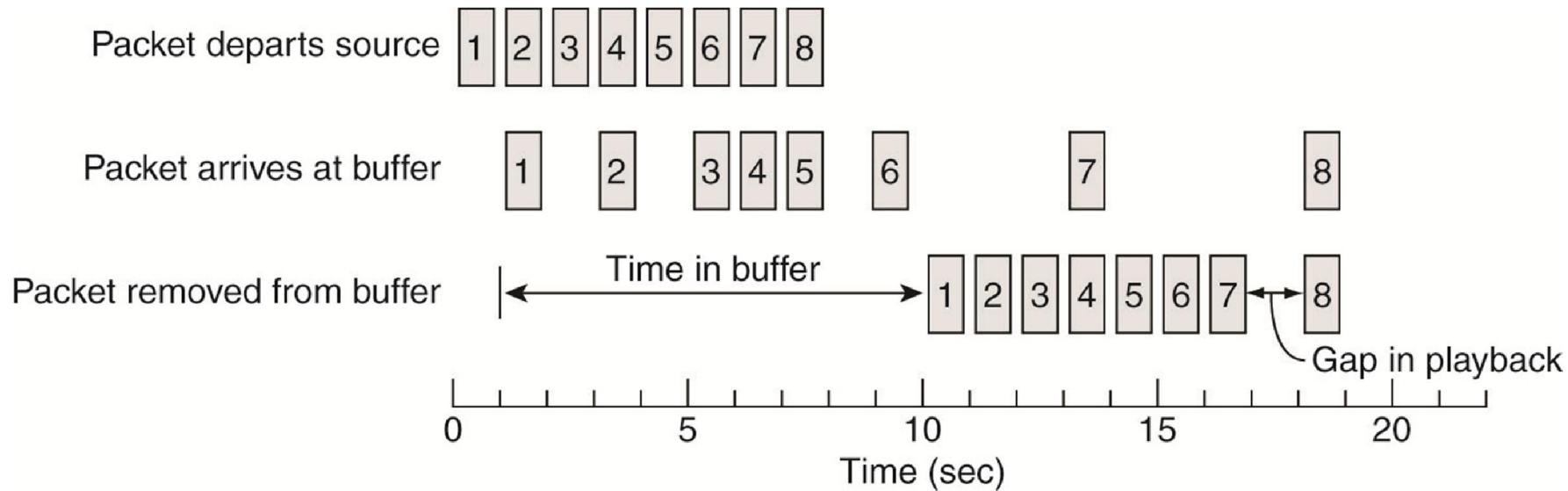
Complex stream: multiple sub-streams

Streams and QoS

- QoS: Quality of Service
- The required bit rate at which data should be transported
- The maximum delay until a session has been set up
- The maximum end-to-end delay
- The maximum delay variance, or jitter
- The maximum round-trip delay

Enforcing QoS (1)

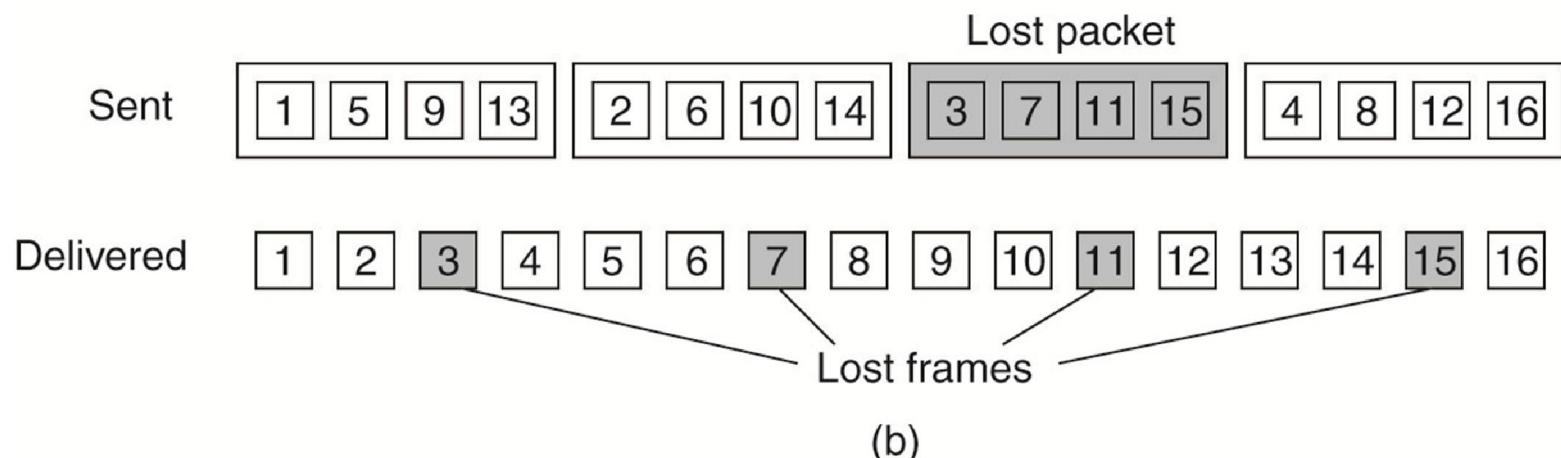
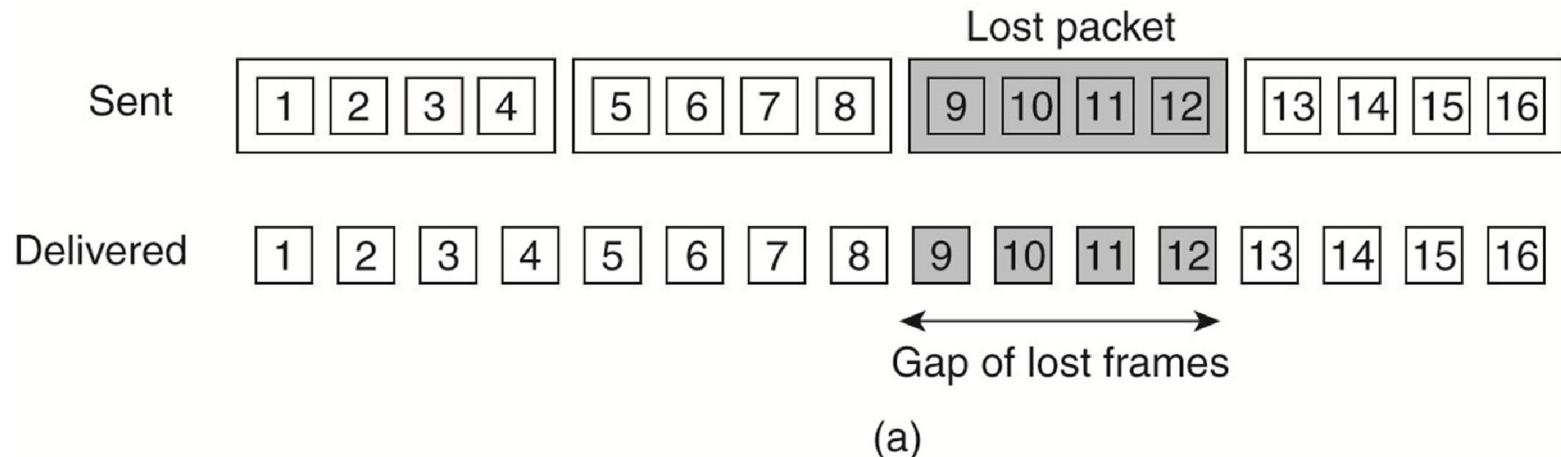
Using a buffer to reduce jitter.



The size of the receiver's buffer corresponds to 9 seconds of packets to pass to the application.
Unfortunately, packet #8 took 11 seconds to reach the receiver.
The result is a gap in the playback at the application.

Enforcing QoS (2)

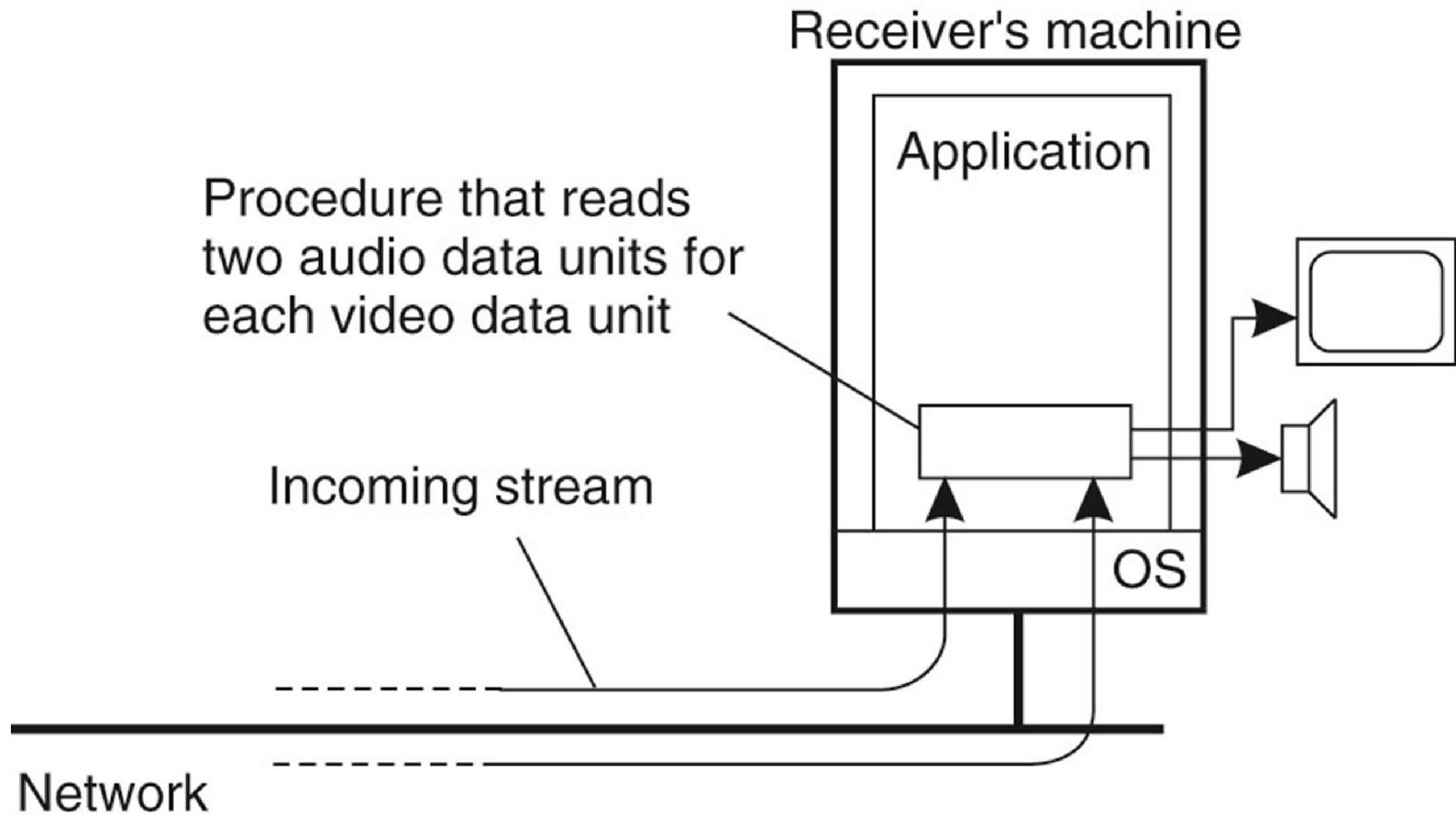
Interleaved transmission



Stream Synchronization

- The simplest form of synchronization is that between a discrete data stream and a continuous data stream.
 - A slide show on the Web that has been enhanced with audio
- Synchronization between continuous data streams.
 - Lip synchronization
 - Stereo synchronization (< 20 μ s)

Receiver-side Synchronization



There is a process that simply executes read and write operations on several simple streams, ensuring that those operations adhere to specific timing and synchronization constraints.

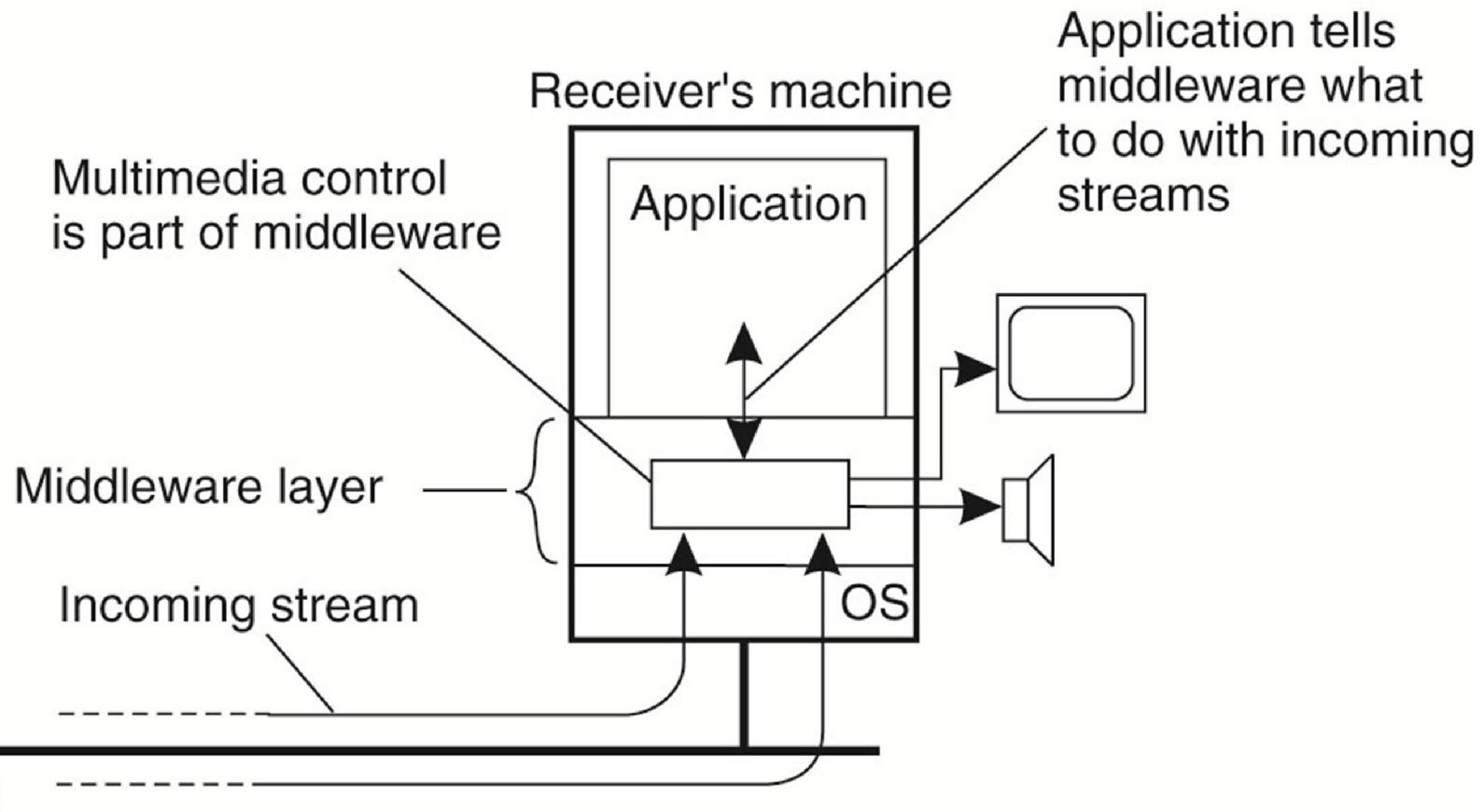
Example

- The video stream
 - Images of 320x240 pixels
 - Pixel encoded by a single byte, leading to video data units of 76,800 bytes each
 - Images are to be displayed at 30 Hz, or one image every 33 msec.
- The audio stream
 - audio units of 11760 bytes, each corresponding to 33 ms of audio,
- Achieve lip synchronization by simply reading an image and reading a block of audio in every 33 ms.

Sender-side Synchronization

- Merge the two substreams at the sender.
- The resulting stream consists of data units consisting of pairs of samples, one for each channel.
- The receiver now merely has to read in a data unit, and split it into a video and audio sample.
- Delays for both channels are now identical.

Middleware-side Synchronization



Multimedia middleware offers a collection of interfaces for controlling audio and video streams, including interfaces for controlling devices such as monitors, cameras, microphones. Such as let application specify the rate at which images should be displayed

Outline

- Fundamentals
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication
- Multicast Communication

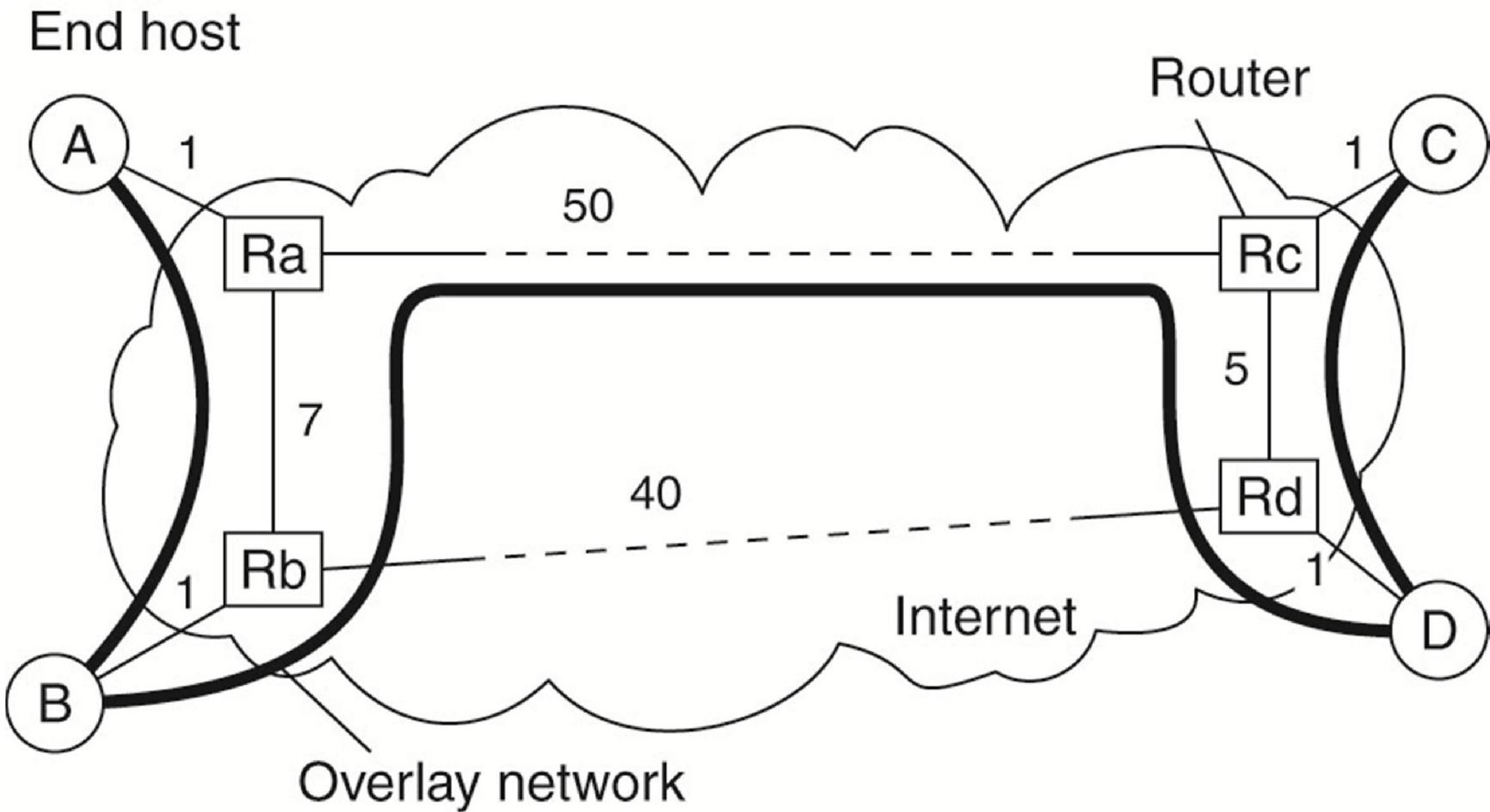
Multicast Communications

- Multicast Communication: communication in distributed systems is the support for sending data to multiple receivers.
- Layer at which done
 - MAC layer
 - Network layer
 - Application layer (overlays)
- Application layer multicast
 - Determine group membership
 - Determine overlay topology
 - Set up forwarding tables

Overlay Network

- Nodes organize themselves directly into a tree
 - There is a unique path between every pair of nodes.
- Nodes organize into a mesh network
 - There are multiple paths between every pair of nodes.
- The latter generally provides higher robustness
 - If a connection breaks, there will still be an opportunity to disseminate information without re-organize the entire overlay network.

Overlay Construction



Gossip

- Epidemic Behavior
- Anti-entropy
- Removing data
- Gossiping
- Spreading the deletion
- Applications

Epidemic_(流行病) Behavior

- **Infected** (已感染): a node that it is willing to spread to other nodes.
- **Susceptible** (易感染): a node that has not yet seen this data is called.
- **Removed** (已隔离): an updated node that is not willing or able to spread its data.

Anti-entropy Propagation Model

- A node P picks another node Q at random, and subsequently exchanges updates with Q .
- Three approaches to exchanging updates:
 - P only pushes its own updates to Q
 - *if many nodes are infected, the probability of each one selecting a susceptible node is relatively small.*
 - P only pulls in new updates from Q
 - works much better when many nodes are infected.
 - P and Q send updates to each other (i.e., a push-pull approach)
 - it can then be shown that the number of rounds to propagate a single update to all nodes takes $O(\log(N))$ rounds

A round as spanning a period in which every node will at least once have taken the initiative to exchange updates with a randomly chosen other node.

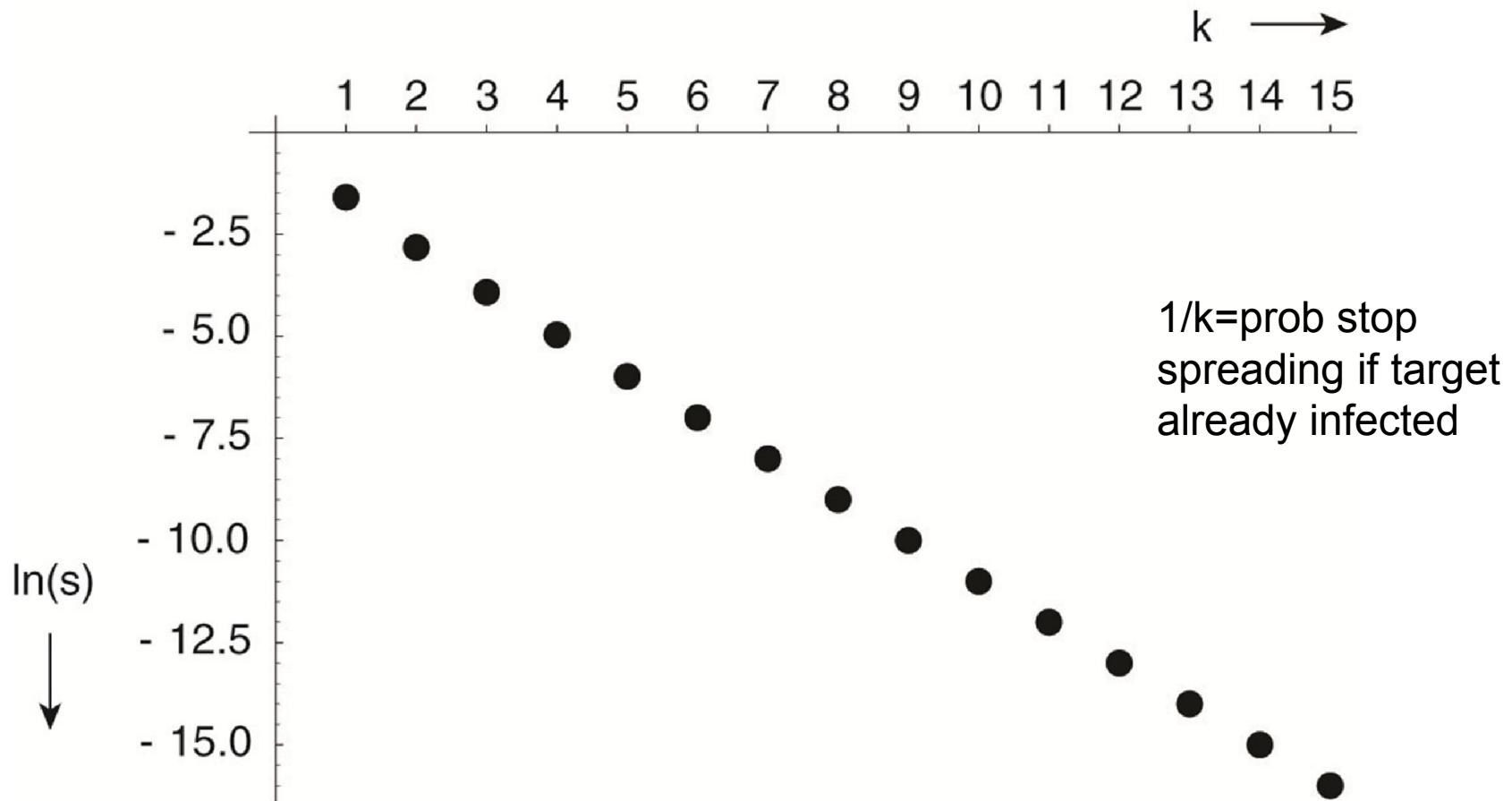
Gossiping (1)

- If node P has just been updated for data item x ,
 - it contacts an arbitrary other node Q and tries to push the update to Q .
 - if Q was already updated by another node.
 - P may lose interest in spreading the update any further, say with probability $1/k$.
 - P becomes removed.
- Gossiping turns out to be an excellent way of rapidly spreading news.

Gossiping (2)

- Gossiping cannot guarantee that all nodes will actually be updated
- s susceptible nodes (ratio) that will remain ignorant of an update:
$$s = e^{-(k+1)(1-s)}$$
- if $k = 4$, $\ln(51)=-4.97$, so that s is less than 0.007, meaning that less than 0.7% of the nodes remain susceptible.

Gossiping (3)



Application

- Every node i initially chooses an arbitrary number, say x_i . When node i contacts node j , they each update their value as:

$$x_i, x_j \leftarrow (x_i + x_j) / 2$$

after this exchange, both i and j will have the same value.

- Eventually all nodes will have the same value, namely, the average of all initial values
- Propagation speed is again exponential

Next Lesson...

DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 5
Naming

Naming System

- Names play a very important role in all computer systems.
 - Share resources;
 - Uniquely identify entities;
 - Refer to locations;
 - And more...
- Naming system
 - To resolve names;
 - Name resolution allows a process to access the named entity.

Outline

- Terminology
- Flat Naming
- Structured Naming
- Attribute-based Naming

Terminology

- **Names:** A name in a distributed system is a string of bits or characters that is used to refer to an entity.
- **Entity:** An entity in a distributed system can be practically anything
 - Resources: hosts, printers, disks, and files
 - Processes, users, mailboxes, newsgroups, Web pages, graphical windows, messages, network connections, and so on.

Terminology

- Entities can be operated on.
 - Printer offers an interface containing operations for printing a document, requesting the status of a print job.
- Access point is an special kind of entity, by which we can access and further operate on an entity.
 - An entity can offer more than one access point.
 - An entity may change its access points in the course of time.

Terminology

- **Address:** the name of an access point is called an address.
 - The address of an access point of an entity is also simply called an address of that entity.

Terminology

- **Identifier:** identifier is a name which satisfies:
 - Each identifier refers to at most one entity;
 - Each entity is referred to by at most one identifier;
 - An identifier always refers to the same entity.
- One-to-one and Persistent
- **Human-friendly name:** name which is tailored to be used by humans.

Terminology

- How do we resolve **names** and **identifiers** to **addresses**?
- A naming system maintains a **name-to-address binding**, and the simplest form is a table of *<name, address>* pairs.
- In distributed systems that span large networks and for which many resources need to be named, a centralized table does not work.

Outline

- Terminology
- Flat Naming
 - Simple Solutions
 - Home-Based (基于宿主位置) Approaches
 - Distributed Hash Tables (Chord)
 - Hierarchical Approaches
- Structured Naming
- Attribute-based Naming

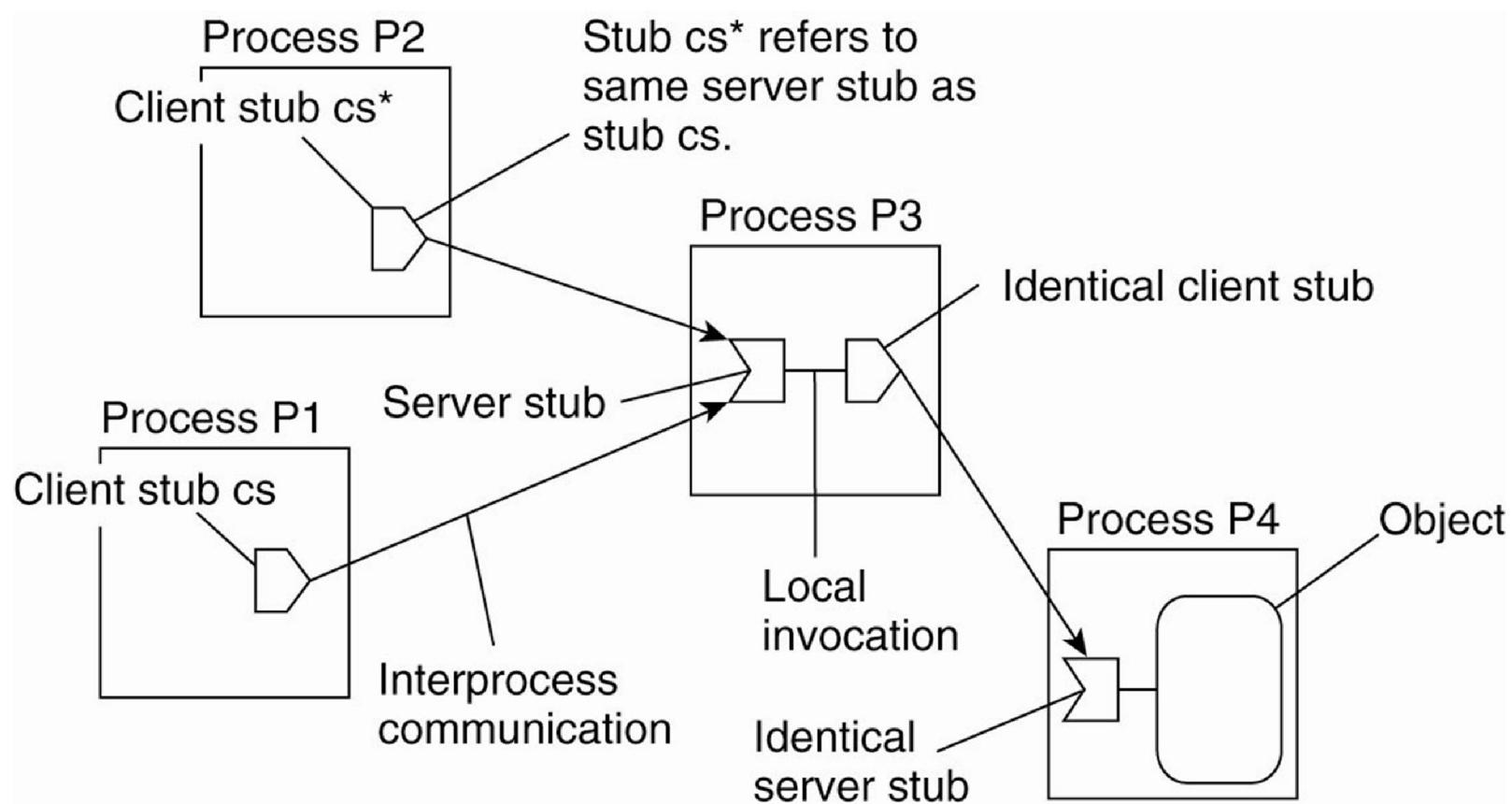
Simple Solutions

- **Broadcasting**: a message containing the identifier of the entity is broadcast to each machine and each machine is requested to check whether it has that entity.
 - Can never scale beyond local area networks (think of ARP/RARP) .
 - Requires all processes to listen to incoming location requests.
 - Broadcasting becomes inefficient when the network grows
- **Multicasting** can also be used to locate entities in point-to-point networks.

Simple Solutions

- **Forwarding pointers**: Each time an entity moves, it leaves behind a pointer telling where it has gone to.
 - So long that locating that entity is prohibitively expensive;
 - All intermediate locations in a chain will have to maintain their part of the chain of forwarding pointers as long as needed;
 - Vulnerability(脆弱性) to broken links.

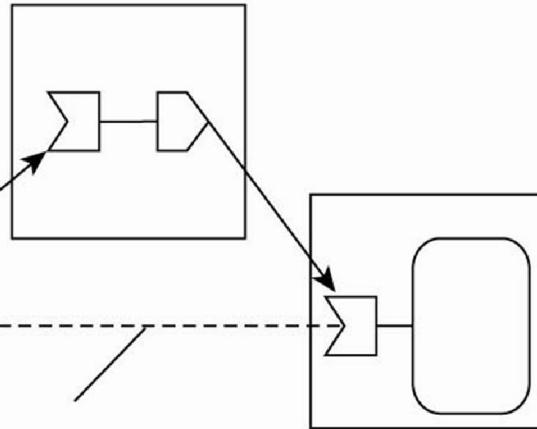
Forwarding Pointers (1)



The principle of forwarding pointers using (client stub, server stub) pairs.

Forwarding Pointers (2)

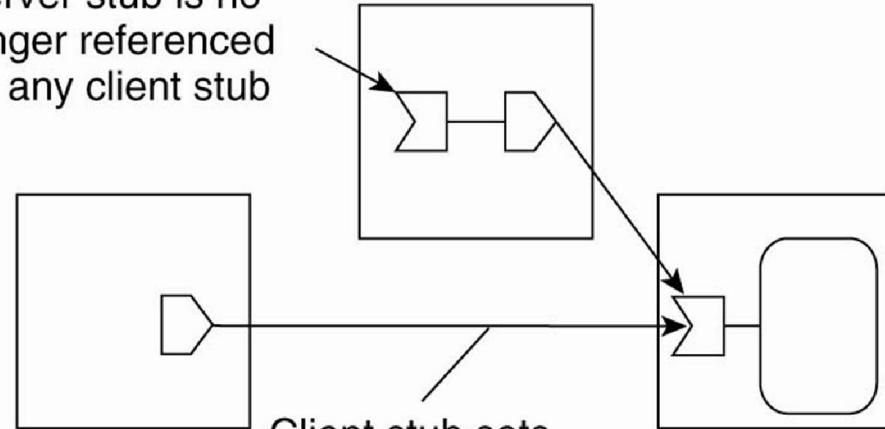
Invocation request is sent to object



Server stub at object's current process returns the current location

(a)

Server stub is no longer referenced by any client stub



Client stub sets a shortcut

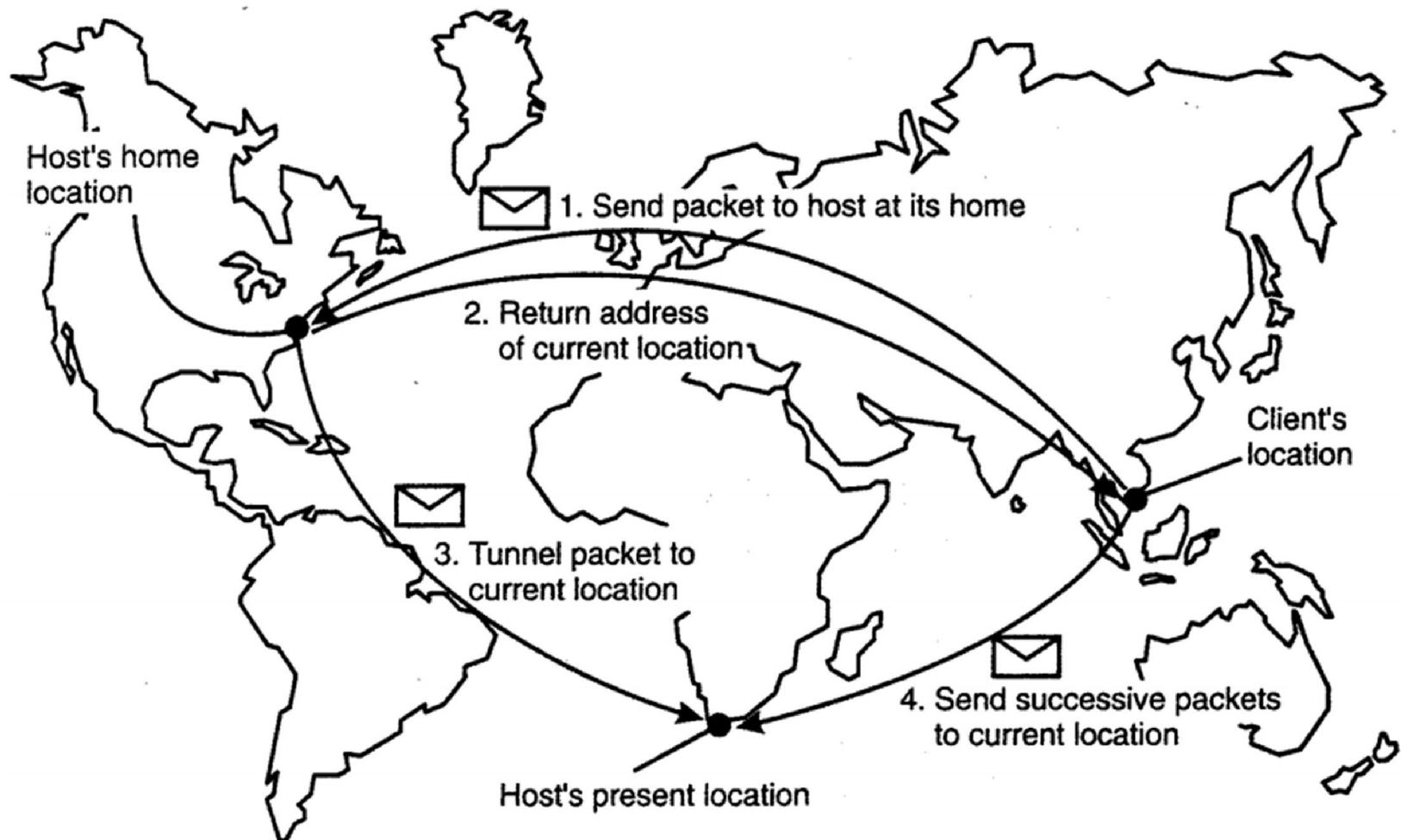
(b)

Redirecting a forwarding pointer by storing a shortcut in a client stub.

Home-Based Approaches

- Let a home keep track of where the entity is:
 - An entity's home address is registered at a naming service.
 - The home registers the foreign address of the entity
 - Clients always contact the home first, and then continues with the foreign location

Home-Based Approaches



The principle of Mobile IP

Problems

- The home address has to be supported as long as the entity lives.
- The home address is fixed, which means an unnecessary burden when the entity permanently moves to another location
- Poor geographical scalability (the entity may be next to the client)

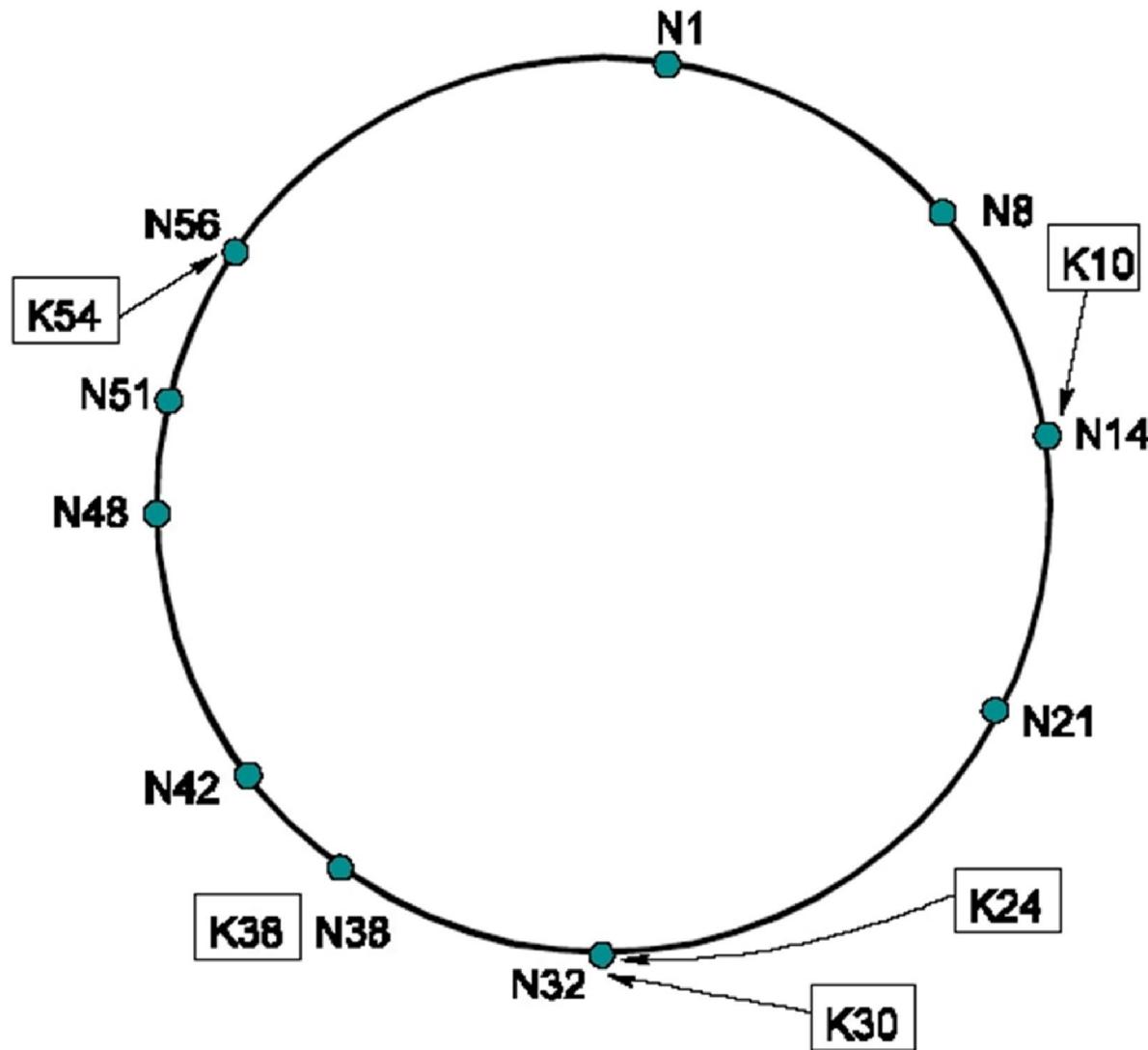
Distributed Hash Tables (Chord)

- Chord uses an m -bit (2^m) **identifier** space to assign identifiers to **nodes** as well as **keys** to specific **entities**.
- **ID:** from 0 to 2^m-1 , m is 128 or 160
- **Node ID (NID):** a NID is hashed by the IP address of a node;
- **Key ID (KID):** a entity is represent by a key, and the KID is hashed by the key;
- **Hash function:** SHA-1 (Secure Hash Algorithm 1)
<https://en.wikipedia.org/wiki/SHA-1>

Placement in Chord Ring

- **Node Placement:** a node is placed on Chord ring according to its NID
- **Successor and predecessor:** The successor to a node is the next node in the Chord ring in a clockwise direction. The predecessor is counter-clockwise.
 - Normally there are "holes" in the sequence
- **Key placement:** a key is stored in successor(KID), it is the first node whose NID equals to or follows KID in the ring in a clockwise direction.

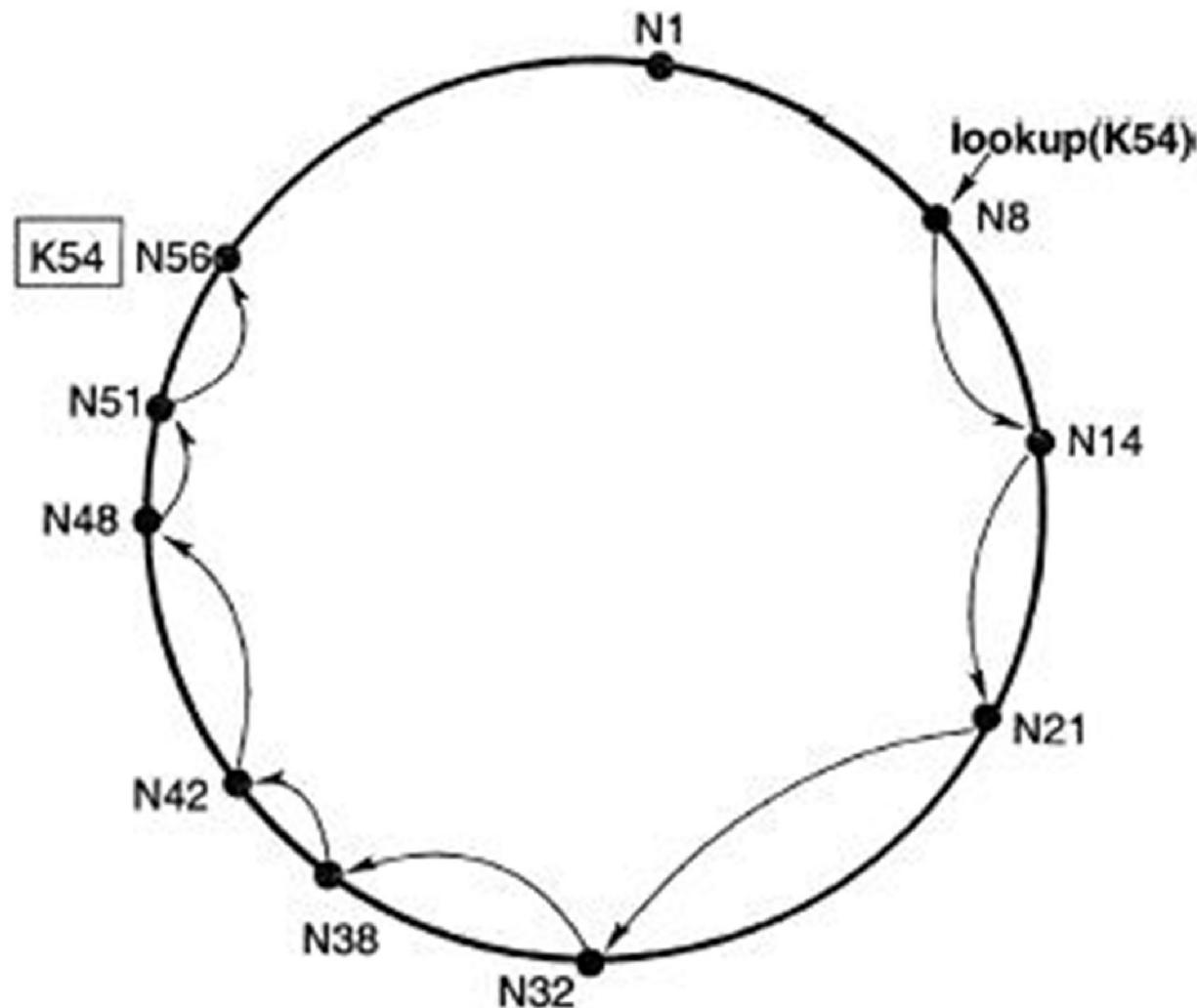
$$m = 6$$



Key Location

- **Basic query:** The basic approach is to pass the query to a node's successor. Query a key or a node is the same as well.
 - Query time is $O(N)$;
- To avoid the **linear search**, Chord implements a faster search method by requiring each node to keep a **finger table**.

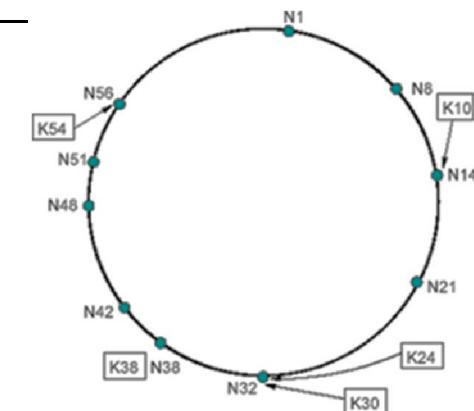
$$m = 6$$



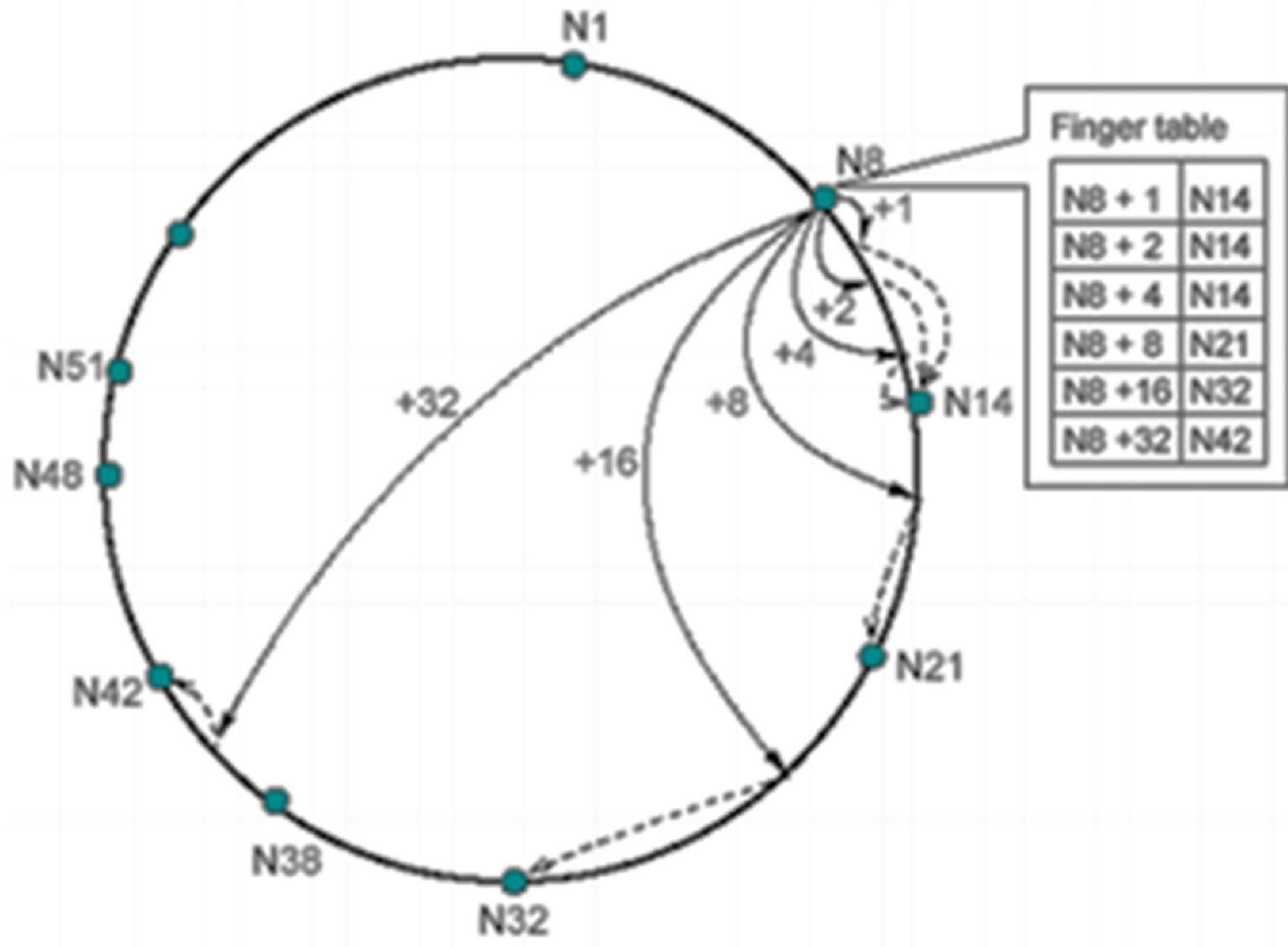
Finger table

- A finger table contains m entries.
- The i -th entry of node n contain a link to
 $\text{successor}((n+2^{i-1}) \bmod 2^m)$ ($i \in [1, m]$)
- Example: $n = 8$, $m = 6$
 - $i = 1$, $\text{successor}(9) = N14$
 - $i = 2$, $\text{successor}(10) = N14$
 - $i = 3$, $\text{successor}(12) = N14$
 - $i = 4$, $\text{successor}(16) = N21$
 - $i = 5$, $\text{successor}(24) = N32$
 - $i = 6$, $\text{successor}(40) = N42$

i	$((n+2^{i-1}) \bmod 2^m)$
1	9
2	10
3	12
4	16
5	24
6	40



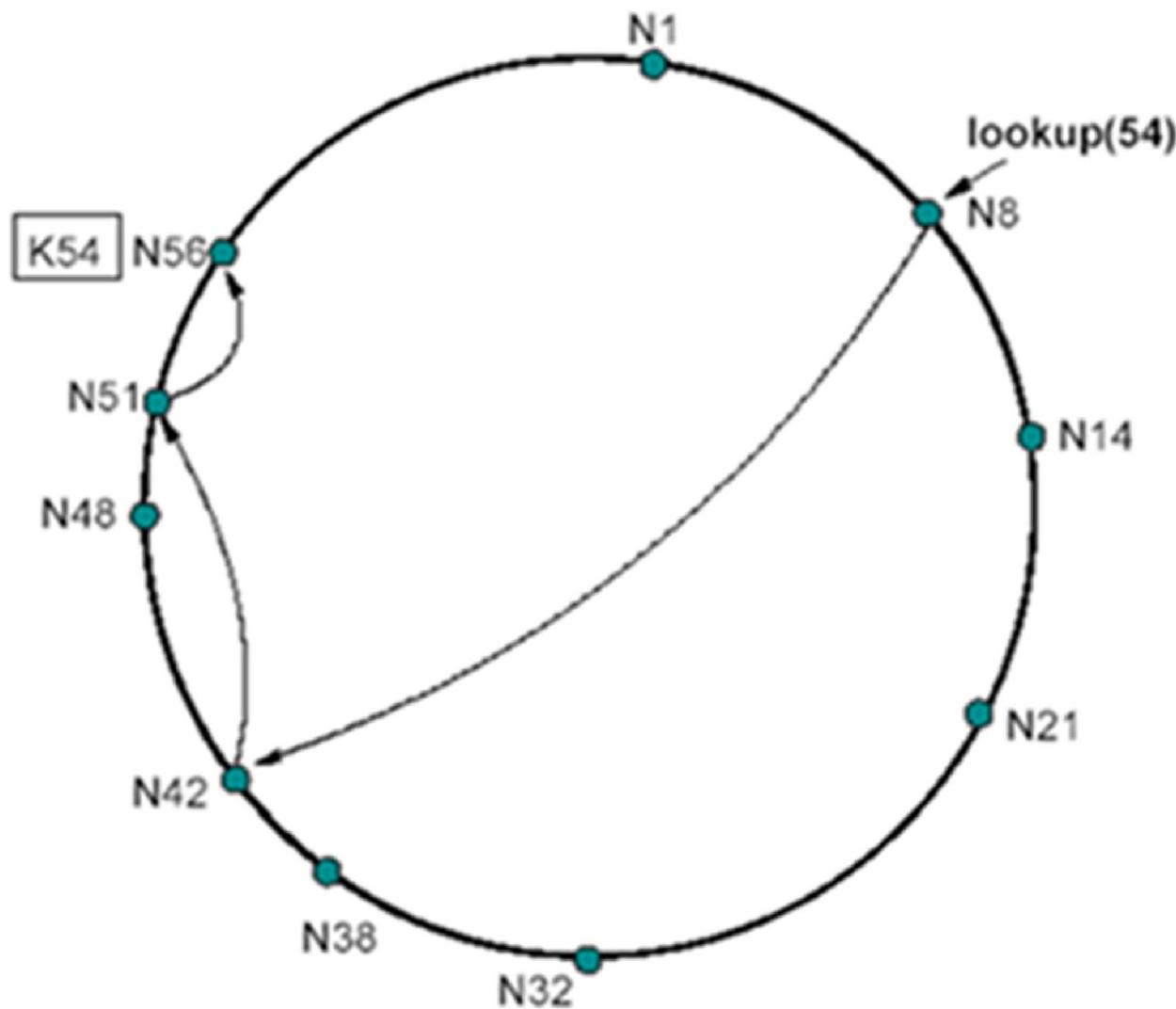
$$m = 6$$



Finger Table based Look Up

- Every time a node with NID wants to look up a key with KID,
 - IF $NID < KID \leq \text{successor}(NID)$, then $\text{successor}(NID)$ is the target;
 - Else if, it forwards the node whose NID is closest and no larger than KID in the Finger Table (this node is the closest predecessor of key).
- Query time is $O(\log N)$;

$$m = 6$$



Chord

General Mechanism

Resolving key 26
from node 1.

$$26 \geq 18$$

$$28 > 26 \geq 20$$

$$28 > 26 \geq 21$$

$$28 > 26$$

Resolving key 12
from node 28.

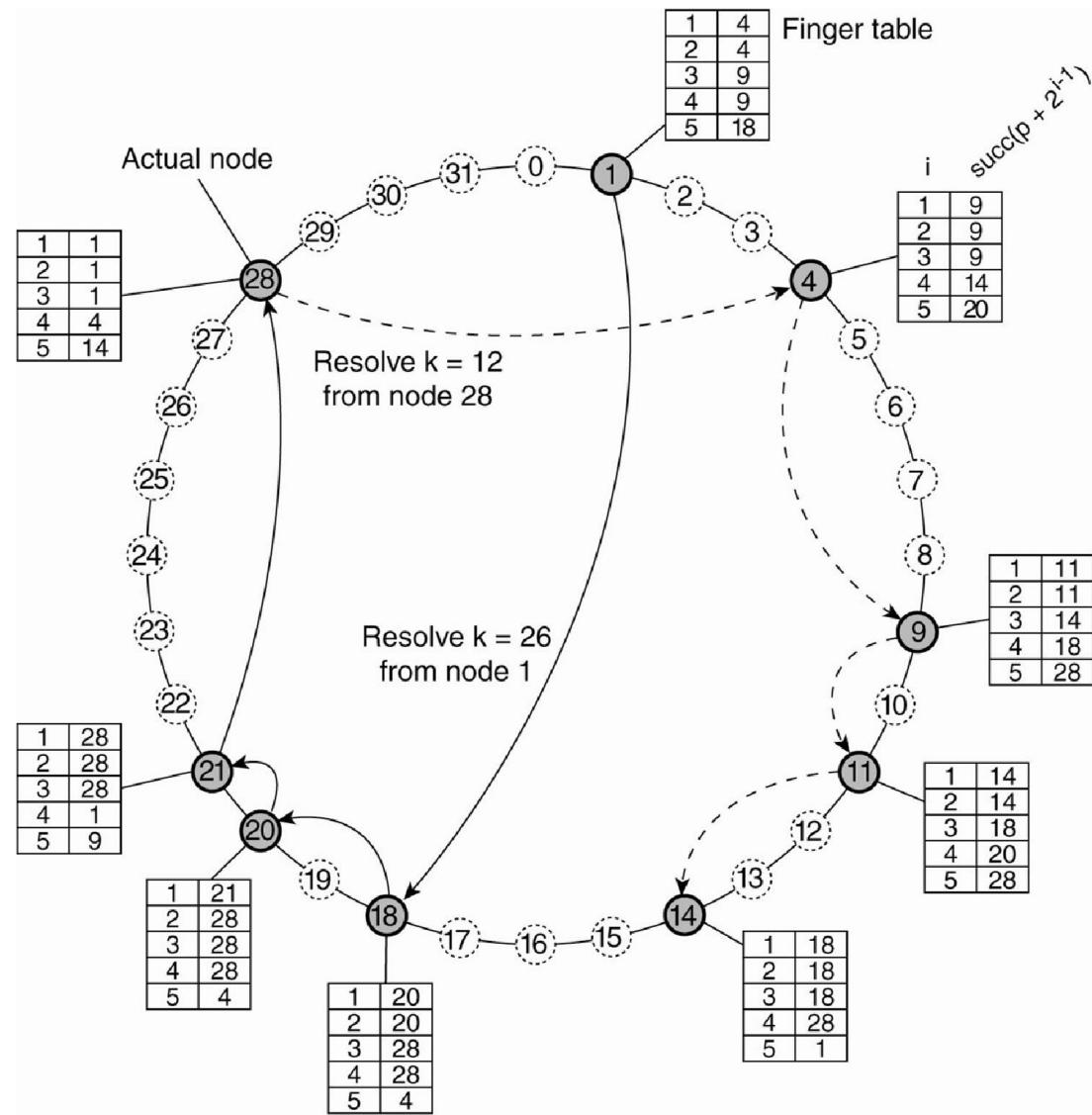
$$12 \geq 4$$

$$9 > 12 \geq 14$$

$$11 > 12 \geq 14$$

$$14 > 12$$

Why distances of
powers of 2?



Logarithmic search time performance

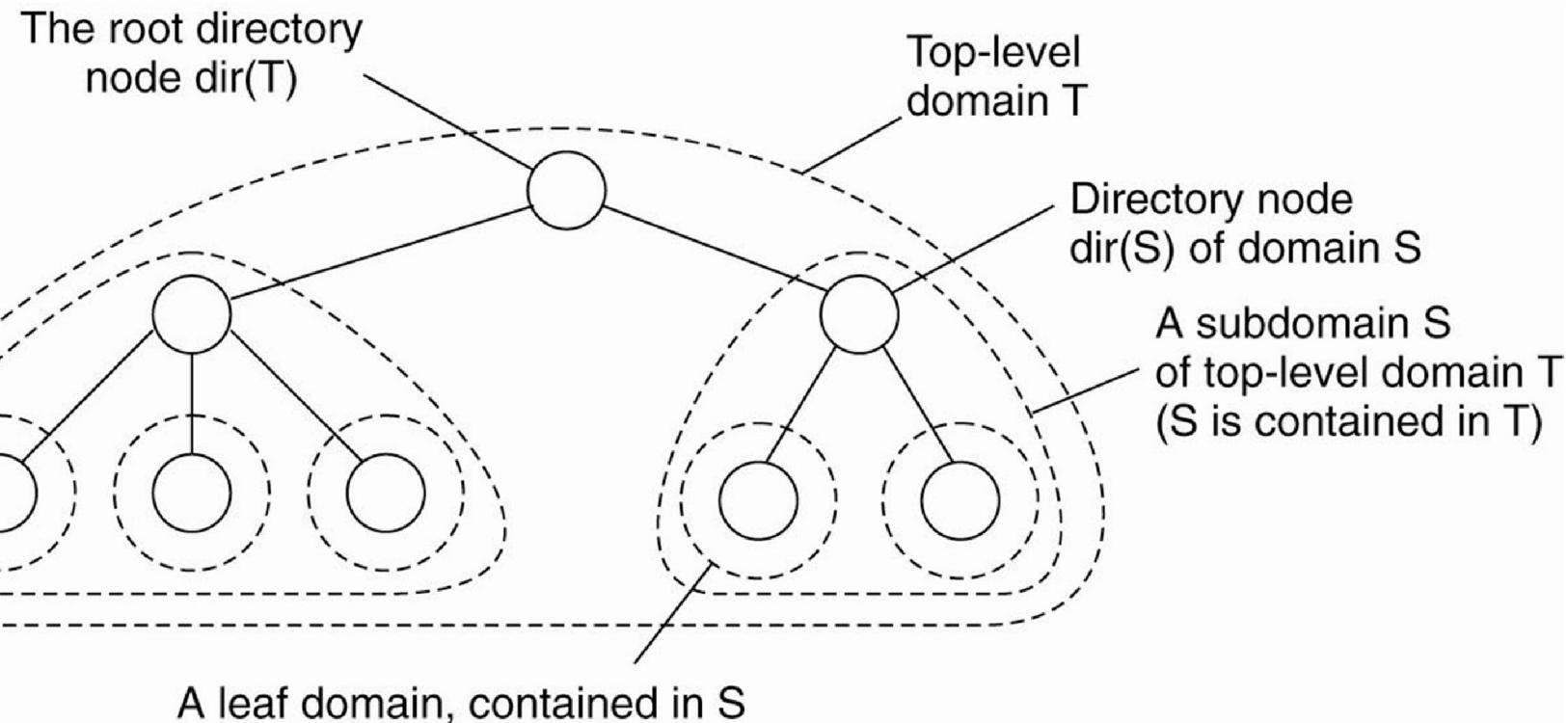
Hierarchical Approaches

- Basic idea is build a large-scale search tree for which the underlying network is divided into hierarchical domains. Each domain is represented by a separate directory node.

Domains

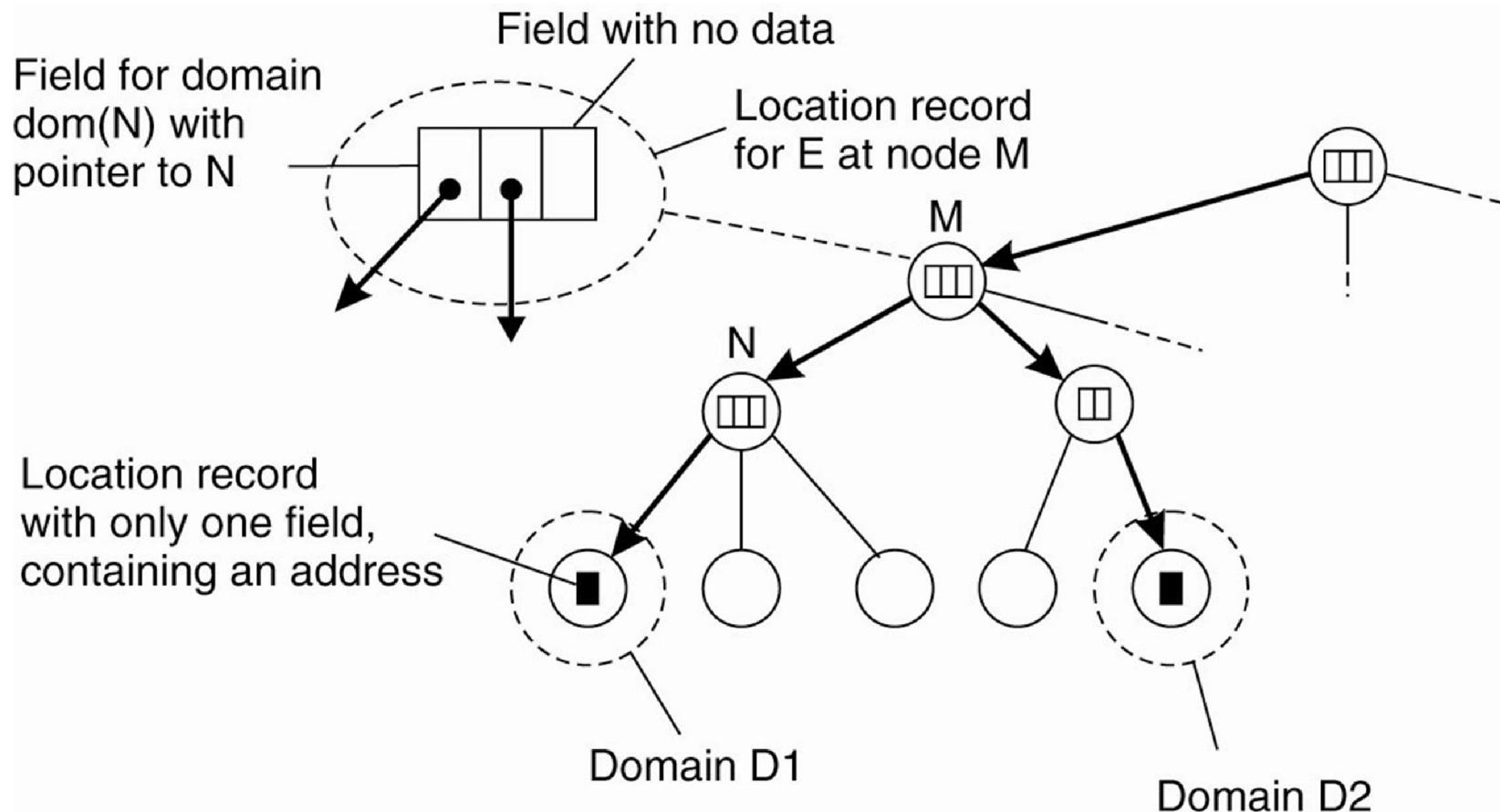
- A network is divided into a collection of **domains**.
- There is a single top-level domain that spans the entire network.
- Each domain can be subdivided into multiple, small **subdomains**.
- A lowest-level domain, called a **leaf domain**
- Each domain D has an associated **directory node** $dirt(D)$

Hierarchical Domains

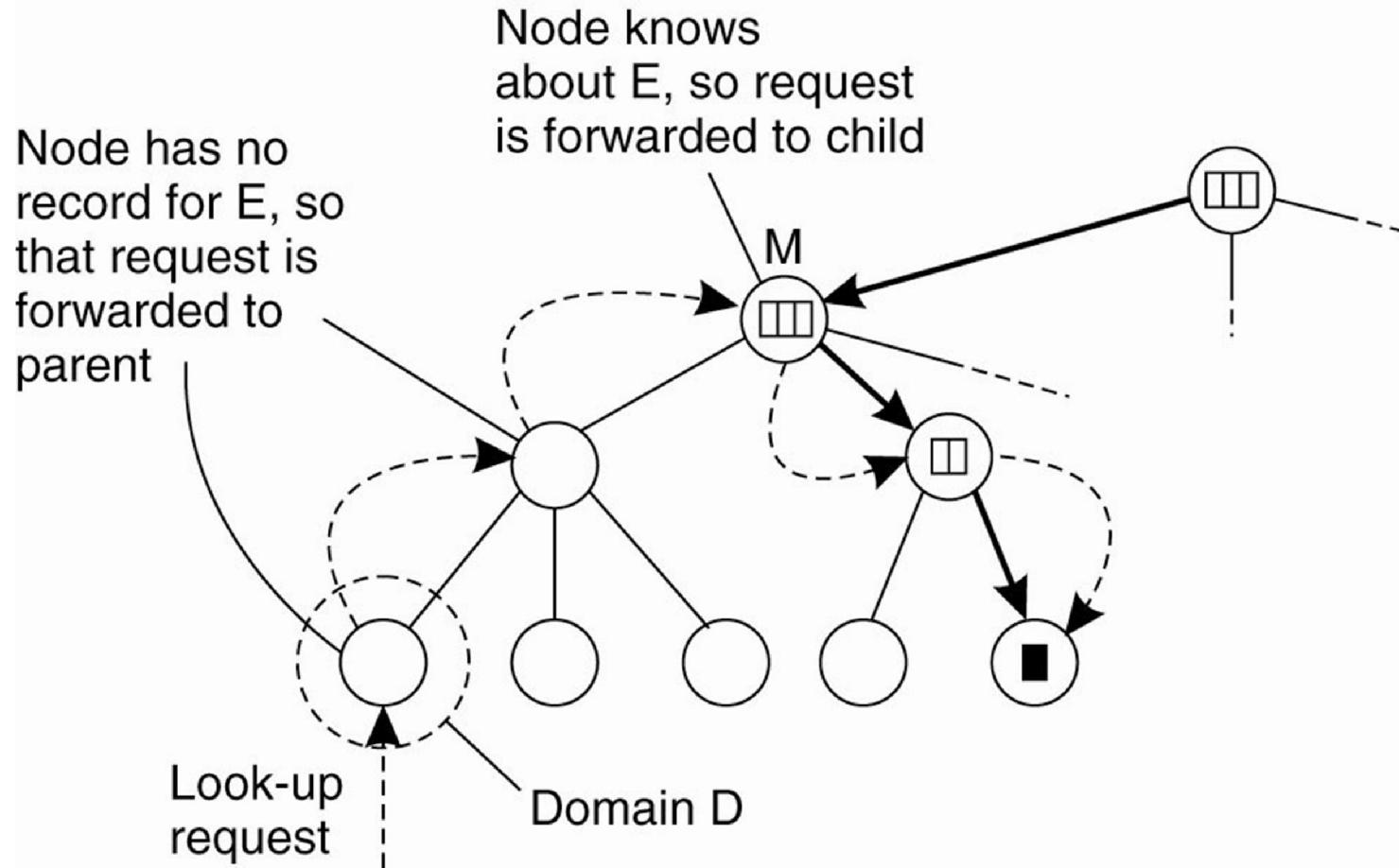


The root node will have a location record for each entity, where each location record stores a pointer to the directory node of the next lower-level subdomain where that record's associated entity is currently located.

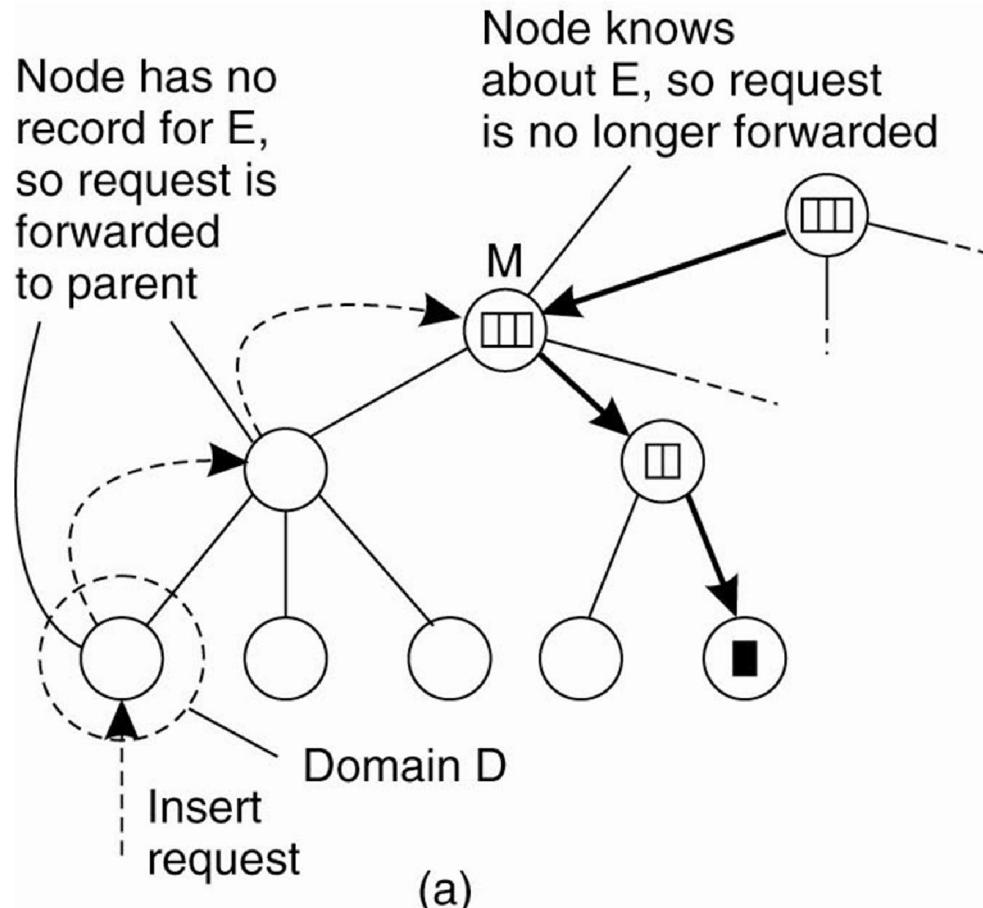
Entity Having Two Addresses in Different Leaf Domains



Looking Up a Location in a Hierarchically Organized Service.

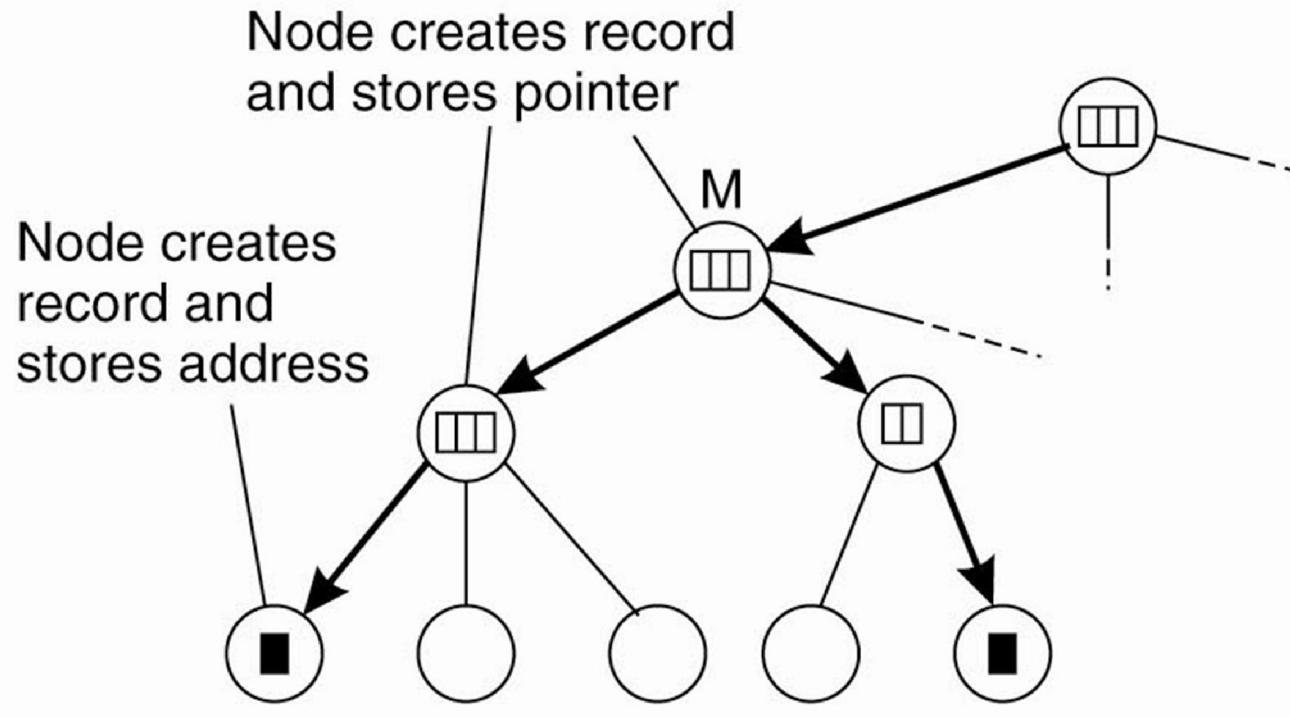


Update Operation (1)



An insert request is forwarded to the first node that knows about entity E .

Update Operation (2)



A chain of forwarding pointers to the leaf node is created.

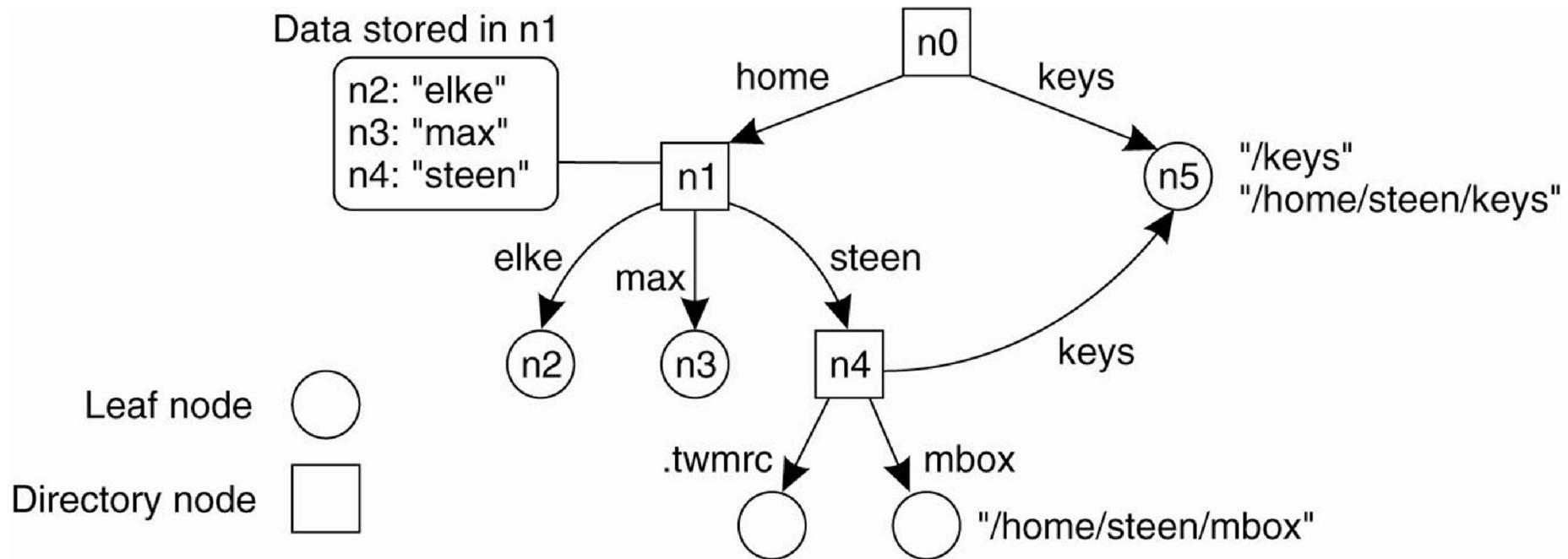
Outline

- Terminology
- Flat Naming
- Structured Naming
 - Name Spaces
 - Name Resolution
- Attribute-based Naming

Name Spaces

- The name space is the way that names in a particular system are organized. This also defines the set of all possible names.
- A name space can be represented as a **labeled, directed graph with two types of nodes**.
 - Leaf node: represents a (named) entity.
 - Directory node: an entity that refers to other nodes, a directory node contains a (directory) table of <**edge label, node identifier**> pairs

A General Naming Graph with a Single Root Node.



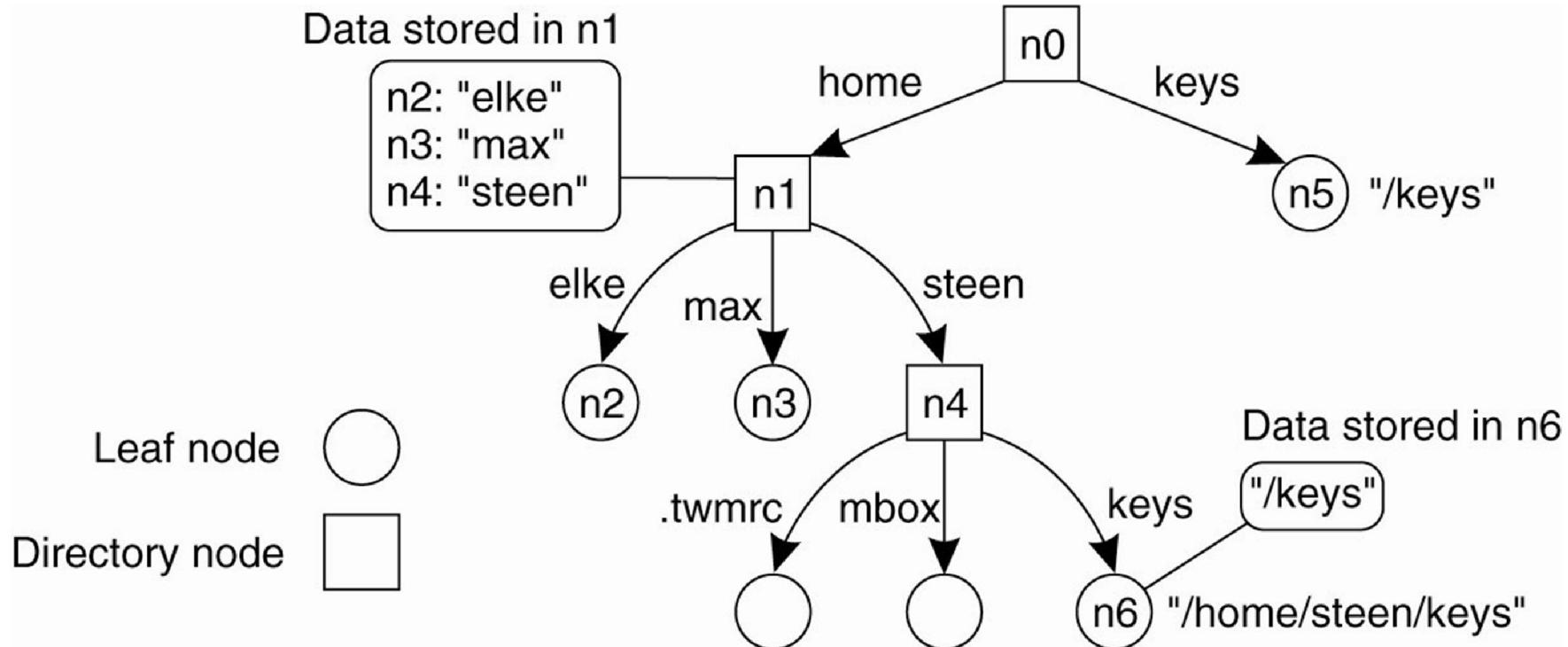
Root

- A node, which has only outgoing and no incoming edge, is called the root (node) of the naming graph.
- A naming graph may have several root nodes.
 - For simplicity, many naming systems have only one.

Path Name

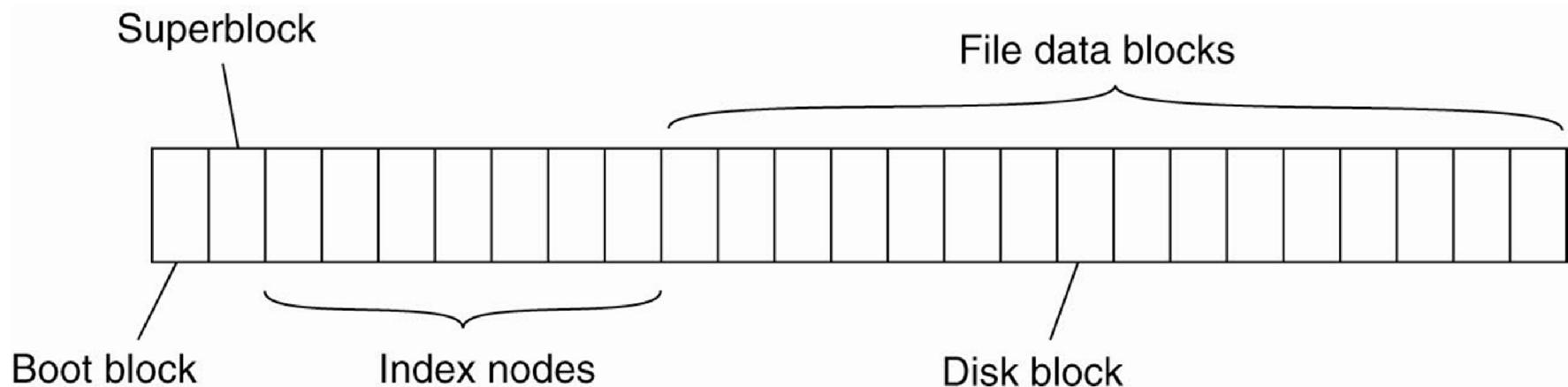
- Each path in a naming graph can be referred to by the sequence of labels corresponding to the edges in that path, such as
 $/label-1/label-2/\dots /label-n$
Such a sequence is called a **path name**.
- A **global name** is a name that denotes the same entity, no matter where that name is used in a system.
- A **local name** is essentially a relative name whose directory in which it is contained is (implicitly) known.

The concept of a symbolic link explained in a naming graph



Example

- The general organization of the UNIX file system implementation on a logical disk of contiguous disk blocks.



Name Resolution

- Name resolution: the process of looking up a name to find the “value”.
- Name resolution can take place only if we know how and where to start;
- A closure mechanism deals with selecting the initial node in a name space from which name resolution is to start.

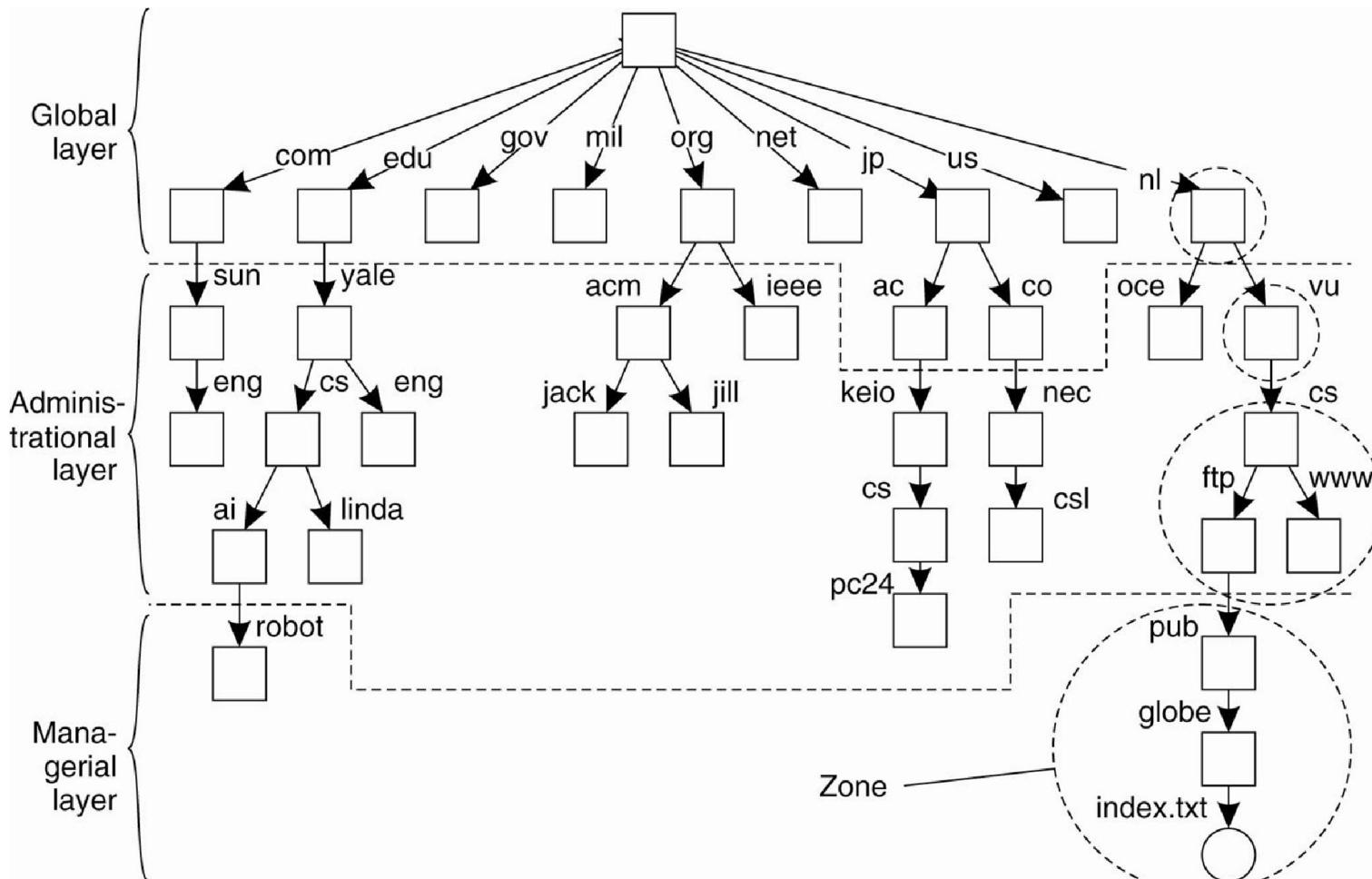
Closure Mechanism

- A closure mechanism (终止机制) deals with selecting the initial node in a name space from which name resolution is to start.
- The mechanism to select the implicit context from which to start name resolution. Examples, file systems, ZIP code, DNS
 - www.cs.vu.nl: start at a DNS name server
 - /home/steen/mbox: start at the local NFS file server (possible recursive search)
 - 240031204447784: dial a phone number
 - 130.37.24.8: IP of Web server

Name Space Distribution

- Name spaces always map names to something.
 - DNS maps what to what?
- Can be divided into three layers:
 - Global layer: Doesn't change very often.
 - The root node and other directory nodes logically close to the root.
 - Administrational layer: Single organization
 - Directory nodes in the administrational layer is that they represent groups of entities that belong to the same organization or administrative unit.
 - Managerial layer: Change regularly
 - Represent shared files, user-defined directories and files.

Name Space Distribution



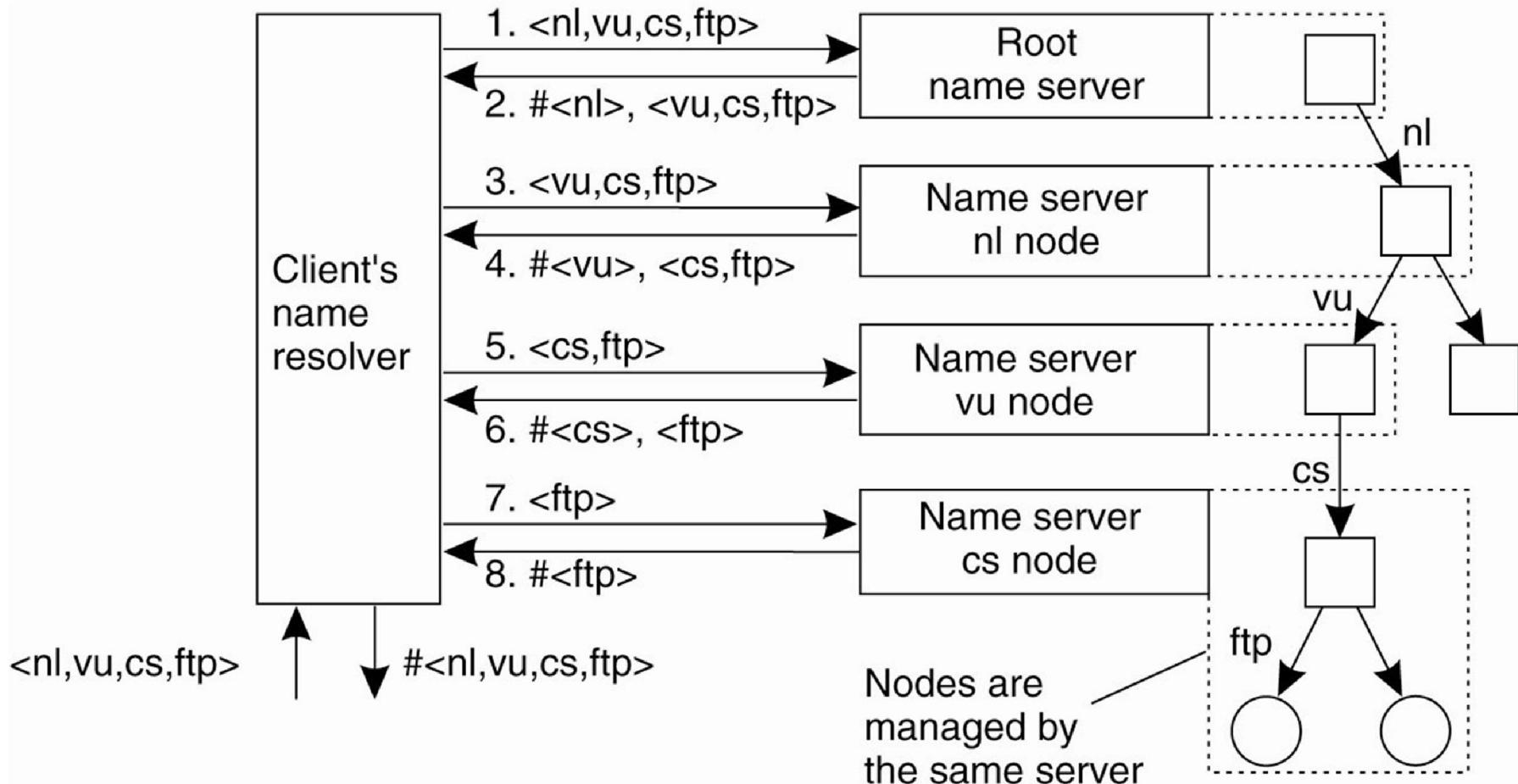
An example partitioning of the DNS name space, including Internet-accessible files, into three layers.

Name Space Distribution

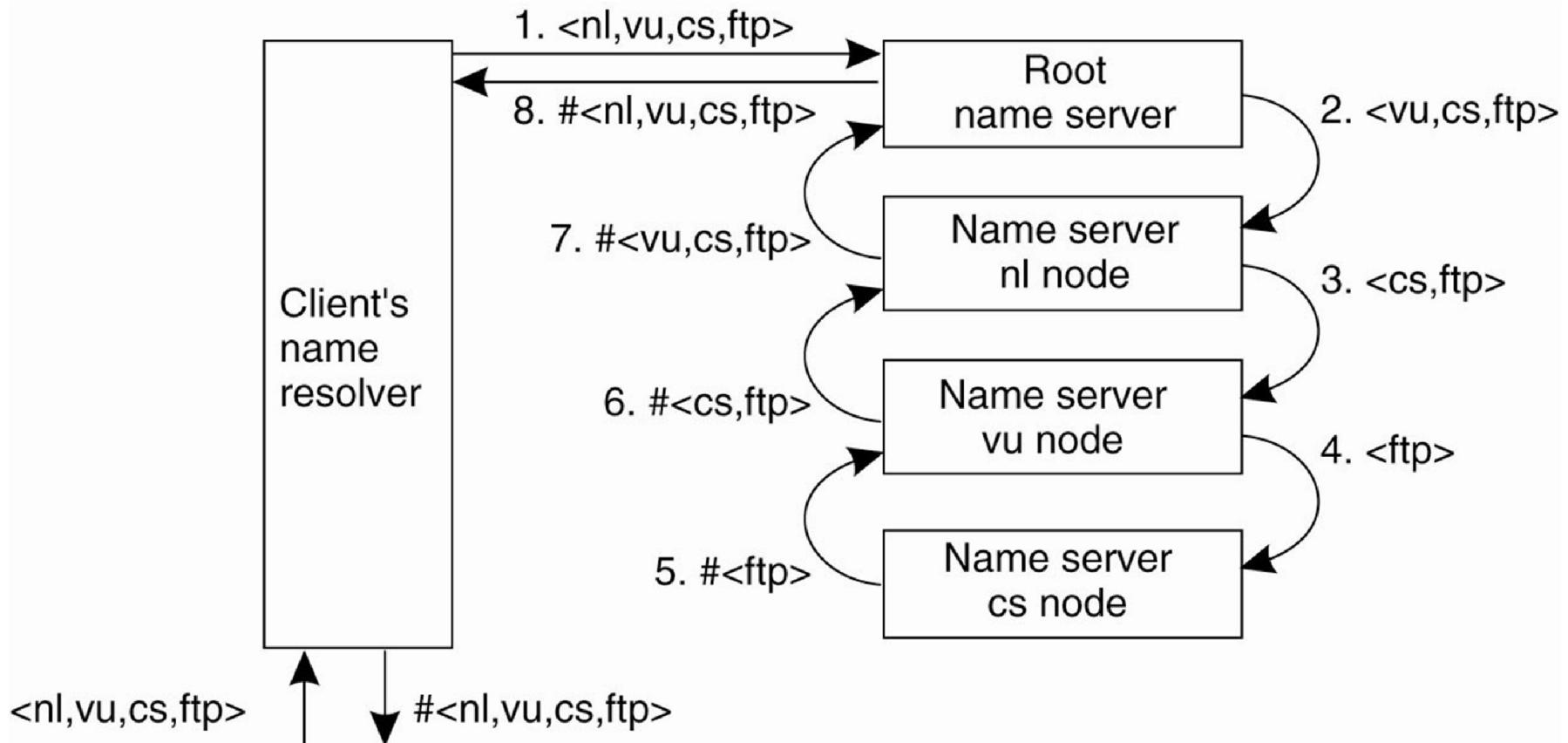
Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side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

A comparison between name servers for implementing nodes from a large-scale name space partitioned into a global layer, an administrative layer, and a managerial layer.

Iterative Name Resolution



Recursive Name Resolution (1)



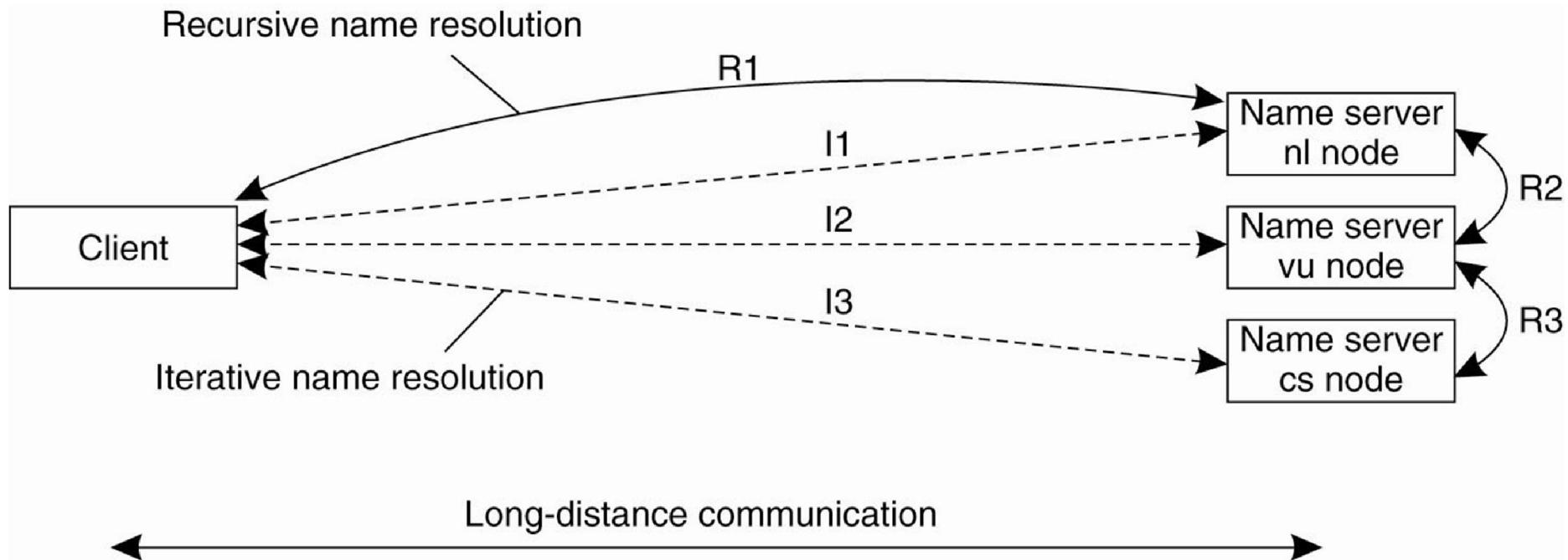
Recursive Name Resolution (2)

Server for node	Should resolve	Looks up	Passes to child	Receives and caches	Returns to requester
cs	<ftp>	#<ftp>	—	—	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Recursive name resolution of <nl, vu, cs, ftp>.

Name servers cache intermediate results for subsequent lookups.

The comparison between recursive and iterative name resolution with respect to communication costs.



Outline

- Terminology
- Flat Naming
- Structured Naming
- Attribute-based Naming
 - Directory Services
 - LDAP

Intention

- Flat and structured names generally provide a unique and **location-independent** way of referring to entities.
 - Structured names have is more **human-friendly**.
- Effectively searching for entities requires that a user can provide merely a description of what he is looking for, named attribute.

Attribute-based Naming

- Distributed systems is to describe an entity in terms of $\langle \text{attribute}, \text{value} \rangle$ pairs, generally referred to as attribute-based naming.
- User specifies attributes and values a specific , then naming system to return one or more entities that meet the user's description.

Directory Services

- Naming Systems: systems that support structured naming.
- Directory services: attribute-based naming systems.
 - Entities have a set of associated attributes that can be used for searching

Resource Description Framework

- Design an appropriate set of attributes is not trivial, it has to be done manually.
- Research has been conducted on unifying the ways that resources can be described.
- Resource Description Framework (RDF).
 - resources are described as triplets consisting of a **subject** (主题), a **predicate** (断言), and an **object** (对象).
 - *<Person, Name, Alice>* describes a resource Person whose name is Alice.

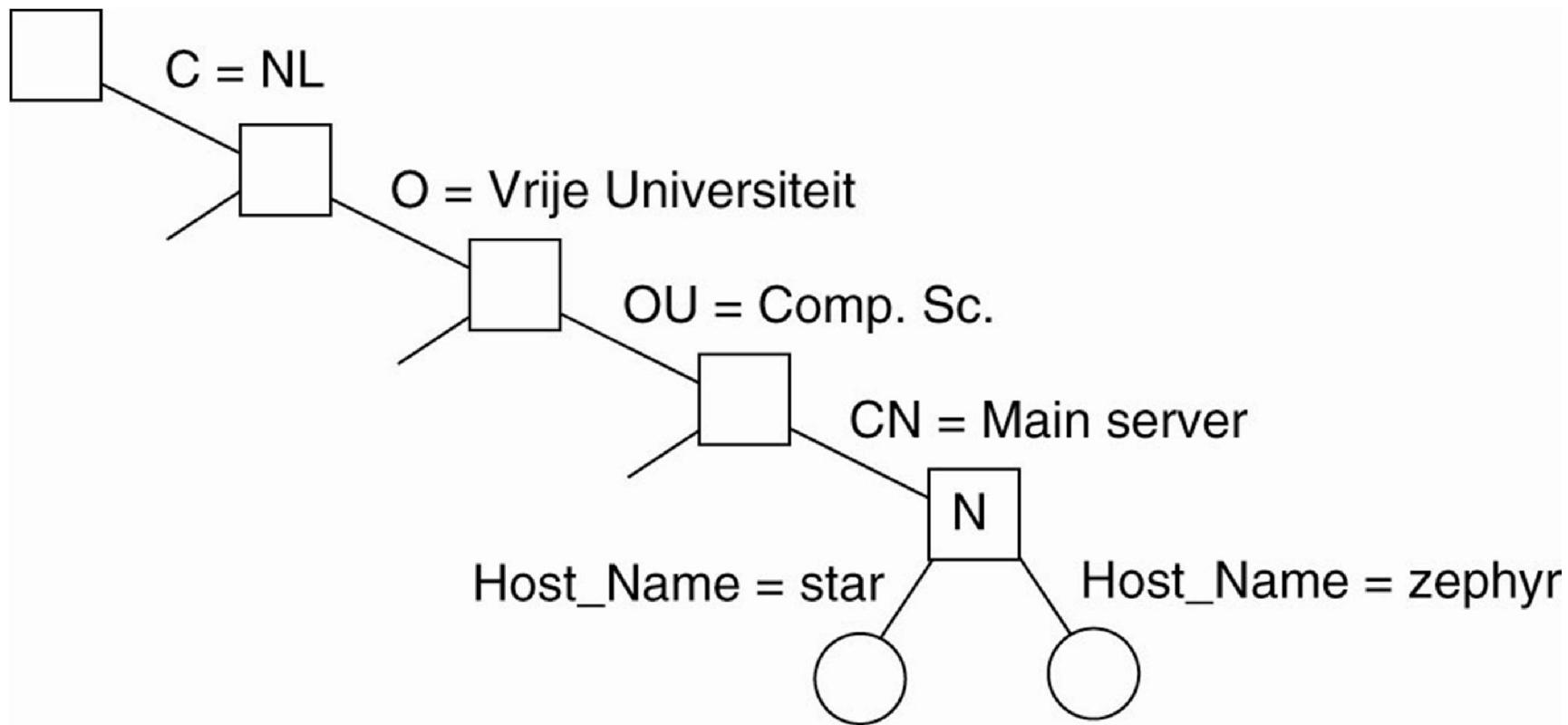
LDAP

- A common approach to tackling distributed directory services is to **combine structured naming with attribute-based naming.**
- LDAP : Lightweight Directory Access Protocol
- LDAP consists of
 - A number of records referred to directory entries;
 - Each record is made up of a collection of *<attribute, value>* pairs;
 - Each attribute has an associated type.

A Simple Example of an LDAP Directory Entry Using LDAP Naming Conventions

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Comp. Sc.
CommonName	CN	Main server
Mail_Servers	—	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	—	130.37.20.20
WWW_Server	—	130.37.20.20

Part of a Directory Information Tree



Two Directory Entries Having Different *Host_Name*

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

Next Lesson...

DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 6

Synchronization

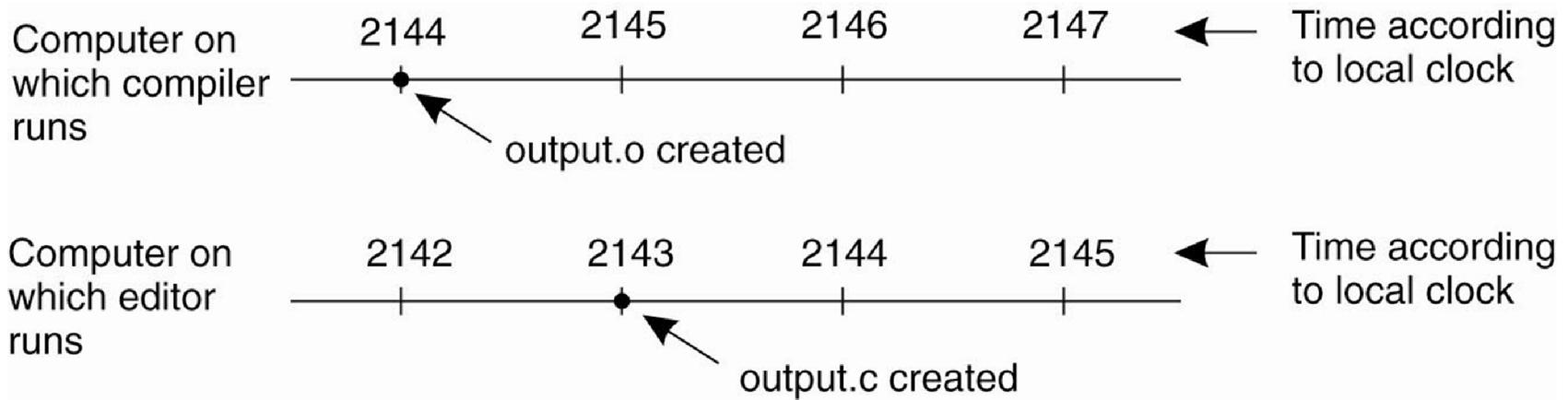
Outline

- Physical Clock Synchronization
- Logical Clocks
- Mutual Exclusion
- Election Algorithms

Problem

- In a no-distributed system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it.
 - If process *A* asks for the time before process *B* do so, then the value that *B* gets will be higher than (or possibly equal to) the value *A* got.
 - It will certainly not be lower.
- If *A* and *B* are two processes on two server, it is difficult to make sure which one is later.
 - Time is ambiguous

Clock Synchronization



- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
- Why not synchronized with the global physical clock?

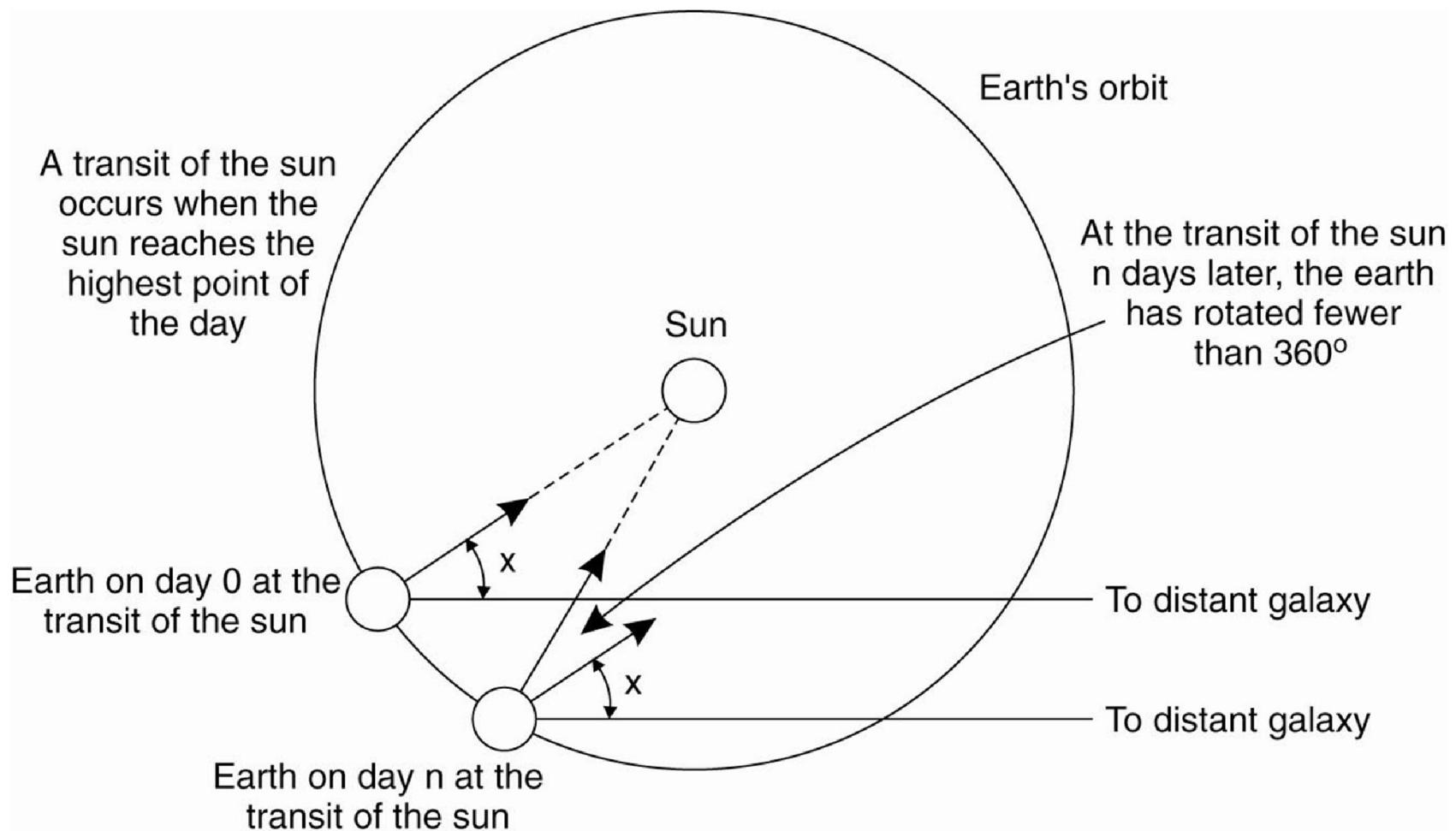
Astronomical Clock

- The event of the sun's reaching its highest apparent point in the sky is called the **transit of the sun** (中天).
- The interval between two consecutive transits of the sun is called the **solar day** (太阳日).
- Since there are 24 hours in a day, each containing 3600 seconds, the **solar second** (太阳秒) is defined as exactly 1/86400 of a solar day.

Mean Solar Second

- In the 1940s, it was established that the period of the earth's rotation is not constant.
 - The earth is slowing down due to tidal friction and atmospheric drag.
 - 300 million years ago there were about 400 days per year.
- The length of the year is not thought to have changed.
 - Compute the length of the day by measuring a days of year, and taking the average before dividing by 86,400, it is mean solar second.

Mean Solar Day



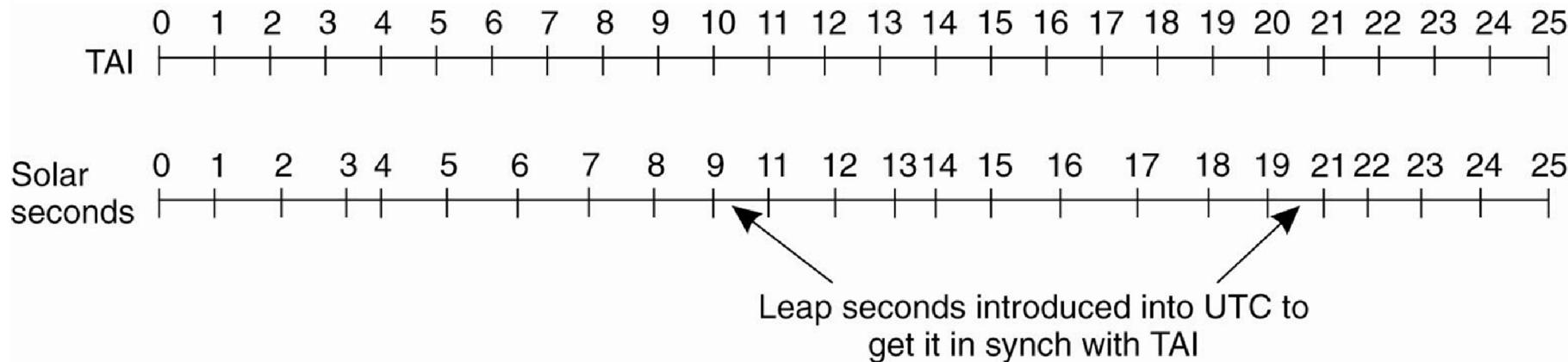
Atomic Clock

- Counting transitions(跃迁) of the cesium 133 atom (铯 133原子).
- Physicists took over the job of timekeeping from the astronomers
 - A second is the time it takes the cesium 133 atom to make exactly 9,192,631,770 transitions
 - To make the atomic second equal to the mean solar second in the year of its introduction

International Atomic Time

- Currently, several laboratories around the world have cesium 133 clocks.
- Periodically, each laboratory tells the Bureau International de l'Heure (BIR) in Paris how many times its clock has ticked.
- The BIR averages these to produce **International Atomic Time**, which is abbreviated TAI.
- Thus TAI is just the mean number of ticks of the cesium 133 clocks since midnight on Jan. 1, 1958 (the beginning of time) divided by 9,192,631,770.

Leap seconds



- 86,400 TAI seconds is now about 3 *usec* less than a mean solar day.
 - Using TAI, noon would get earlier.
- BIR solves the problem by introducing **leap seconds** (闰秒) whenever the discrepancy between TAI and solar time grows to 800 msec.
 - Have been used about 30 times

Universal Coordinated Time

- TAI with leap seconds is Universal Coordinated Time (UTC).
- UTC is the basis of all modern civil timekeeping.
- UTC replaced the old standard Greenwich Mean Time (GMT). which is astronomical time.
- Geostationary Environment Operational Satellite can provide UTC accurately to 0.5 msec

WWV

- To provide UTC to people who need precise time, the National Institute of Standard Time (NIST) operates a shortwave radio station with call WWV from Fort Collins(柯林斯堡), Colorado(科罗拉多州).
- WWV broadcasts a short pulse at the start of each UTC second.
 - The accuracy of WWV itself is about ± 1 msec, but due to random atmospheric fluctuations that can affect the length of the signal path, in practice the accuracy is no better than ± 10 msec.

Computer Timer (Clock)

- A computer timer is usually a precisely machined quartz crystal (石英晶体).
- Quartz crystals oscillate (振荡) at a well-defined frequency when kept under tension
 - Frequency that depends on the kind of crystal, how it is cut, and the amount of tension.
- Two registers, a counter and a holding register
 - Each oscillation of the crystal decrements the counter by one.
 - When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register.
- Program a timer to generate an interrupt any times a second

Computer Timer (Clock)

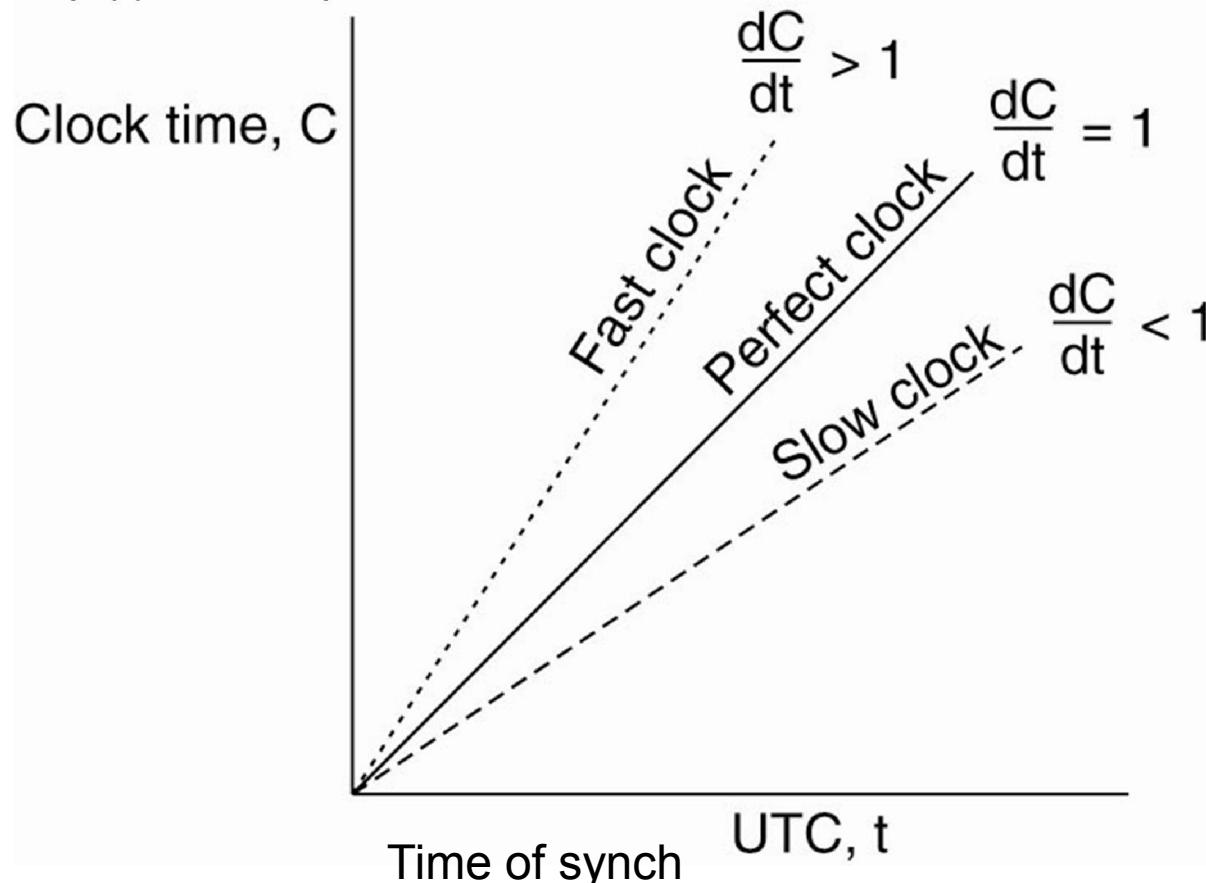
- Different initial time if it is not accurate enough.
- Although the frequency at which a crystal oscillator runs is usually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency.

Clock Synchronization Algorithms

- Goal is to keep all the computer together as well as possible no matter they have or have not WWV receiver.

Underlying Model (assumption)

- Each computer is assumed to have a timer that causes an interrupt H times a second.
- UTC time t , Computer p 's *time* $C_p(t)$
- Ideally, $C_p(t) = t$, $dC_p/dt=1$



Maximum Drift Rate

- $H = 60$ should generate 216,000 ticks per hour
- Relative error of manufacturer is about 5-10, means in the range 215,998 to 216,002 ticks per hour.
- If there exists some constant ρ such that:

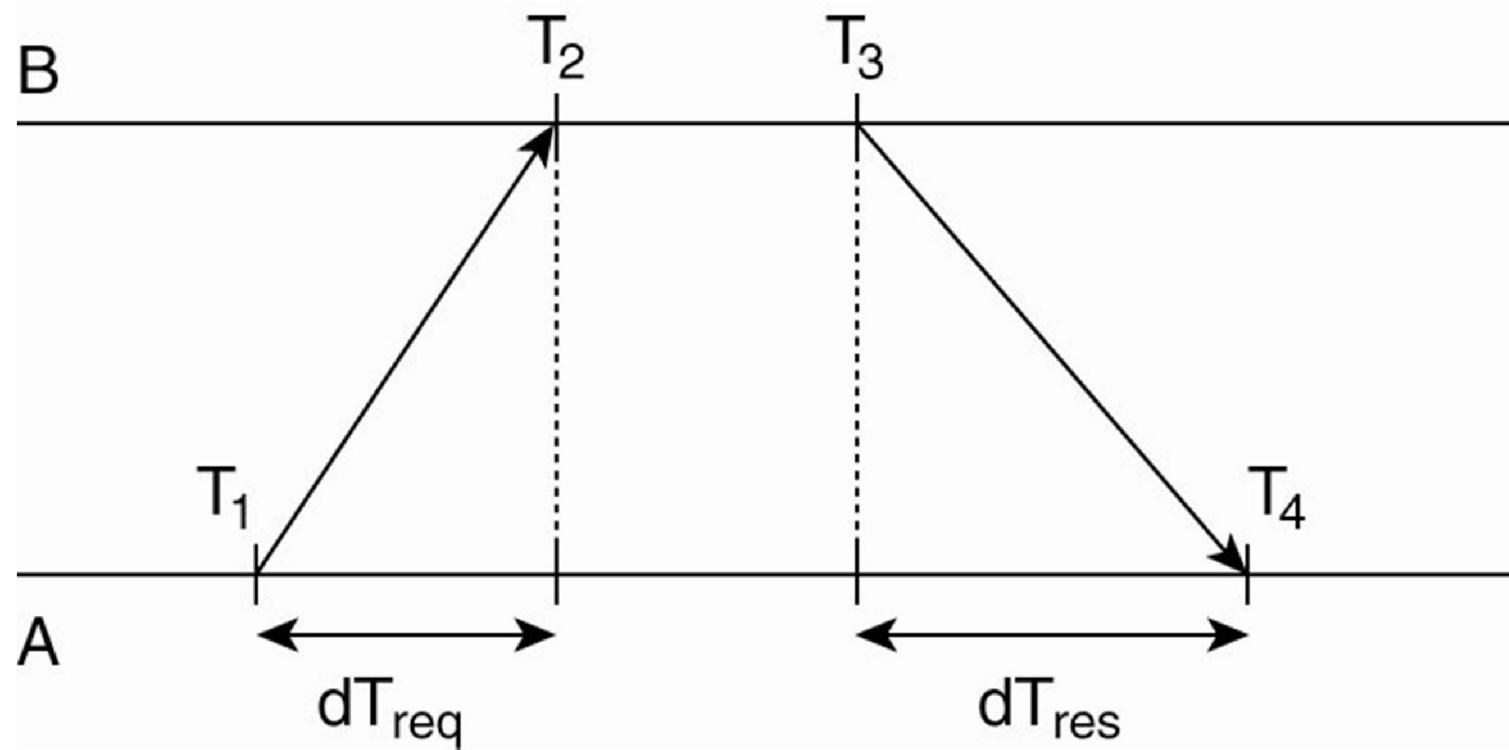
$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

- Maximum drift rate (ρ) is specified by the manufacturer.

Synchronization Frequency

- 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 $2 \rho \Delta t$.
- If the operating system designers want to guarantee that no two clocks ever differ by more than δ
 - Clocks must be resynchronized (in software) at least every $\delta / 2 \rho$ seconds.

Network Time Protocol

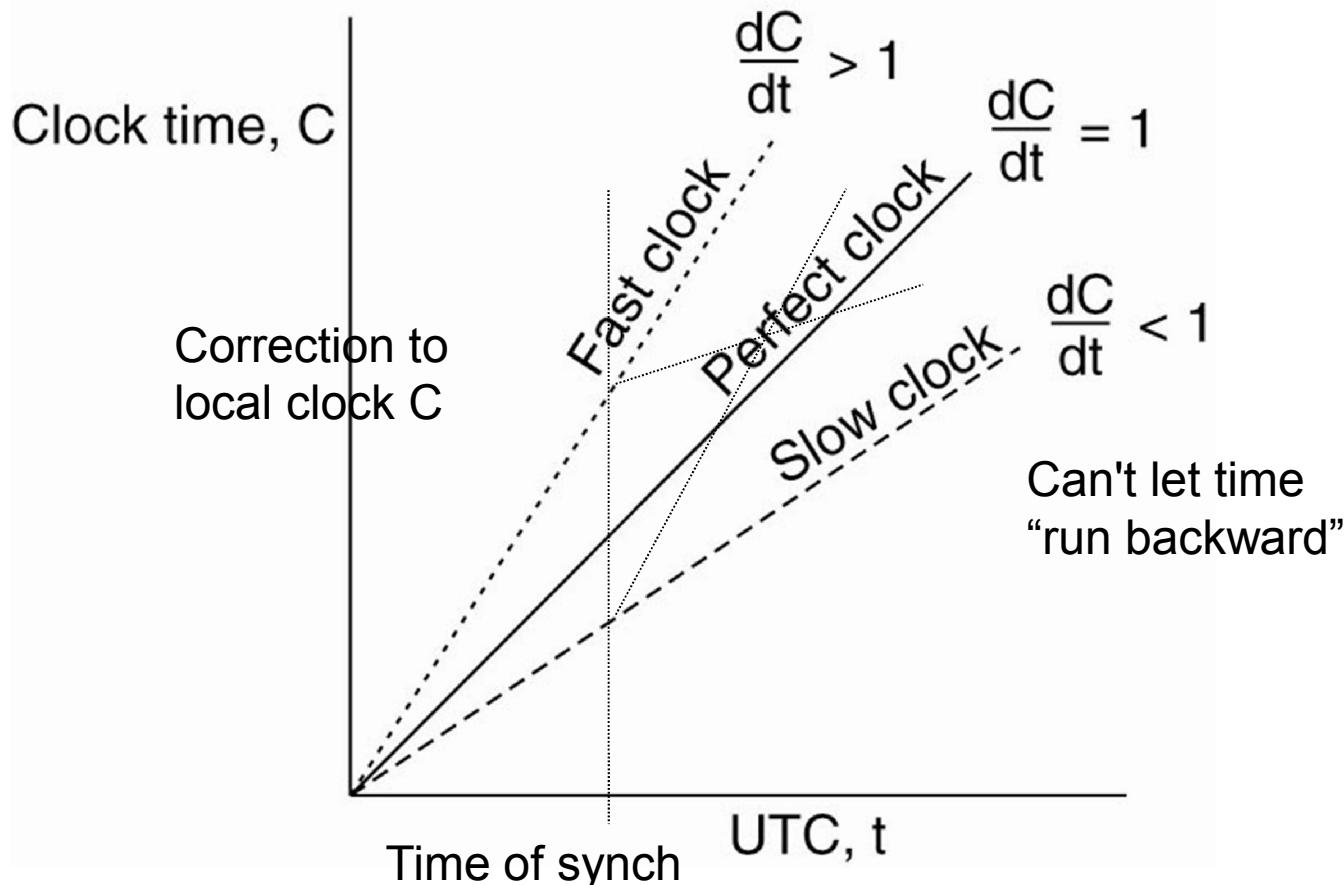


$T_2 - T_1 \approx T_4 - T_3$ A can estimate its offset relative to B

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

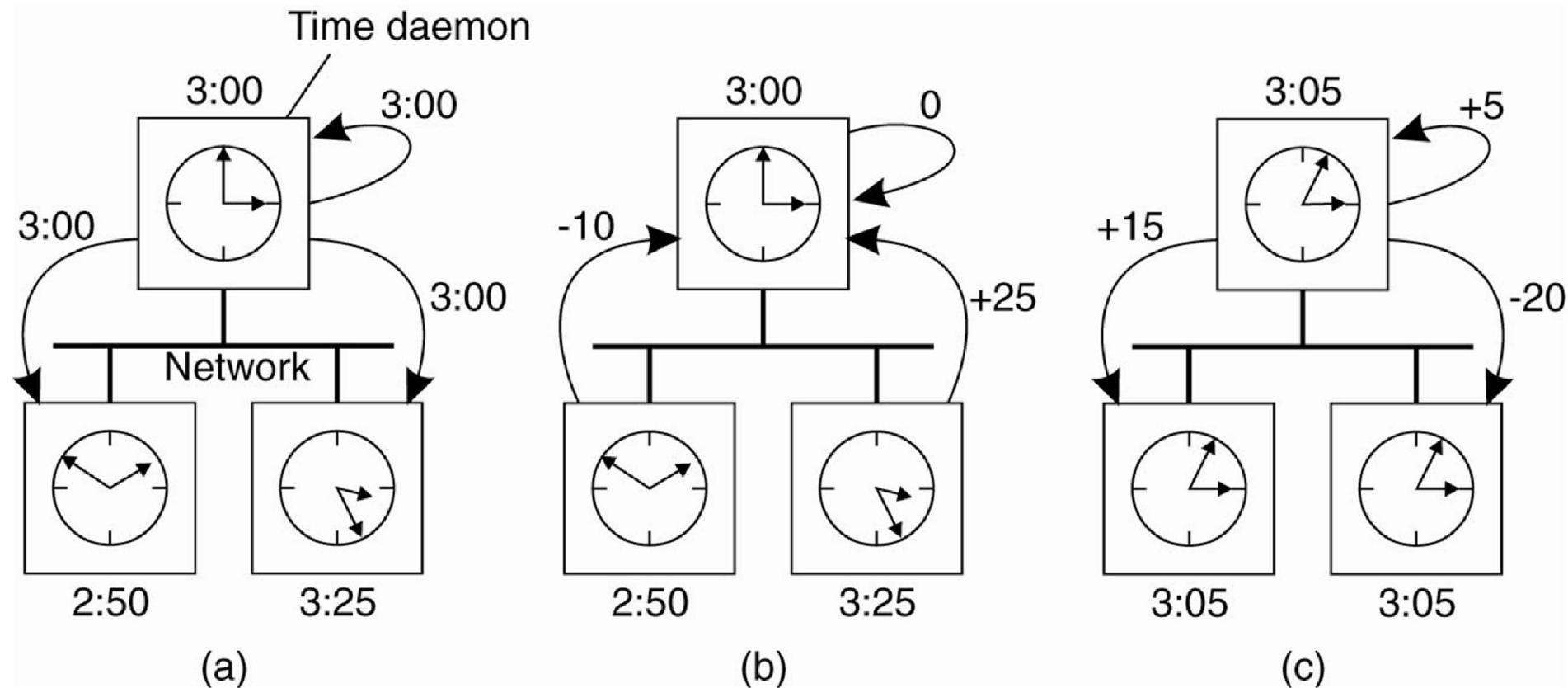
Synchronizing

- Such a change must be introduced gradually, can not set its clock backward, but change H value



The Berkeley Algorithm

Note that for many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agrees with UTC.



time server is active, polling every machine from time to time to ask what time it is there. Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved.

Outline

- Physical Clock Synchronization
- Logical Clocks
- Mutual Exclusion
- Global Positioning of Nodes
- Election Algorithms

Lamport (1978)

- All processes agree on exactly what time it is, but rather than they agree on the order in which events occur.
- That is Logic Clocks

Happens-before

- Expression $a \rightarrow b$ is read "a happens before b" and means that all processes agree that first event a occurs, then afterward, event b occurs.
 - If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$ is true.
 - If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then $a \rightarrow b$ is also true.
 - if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

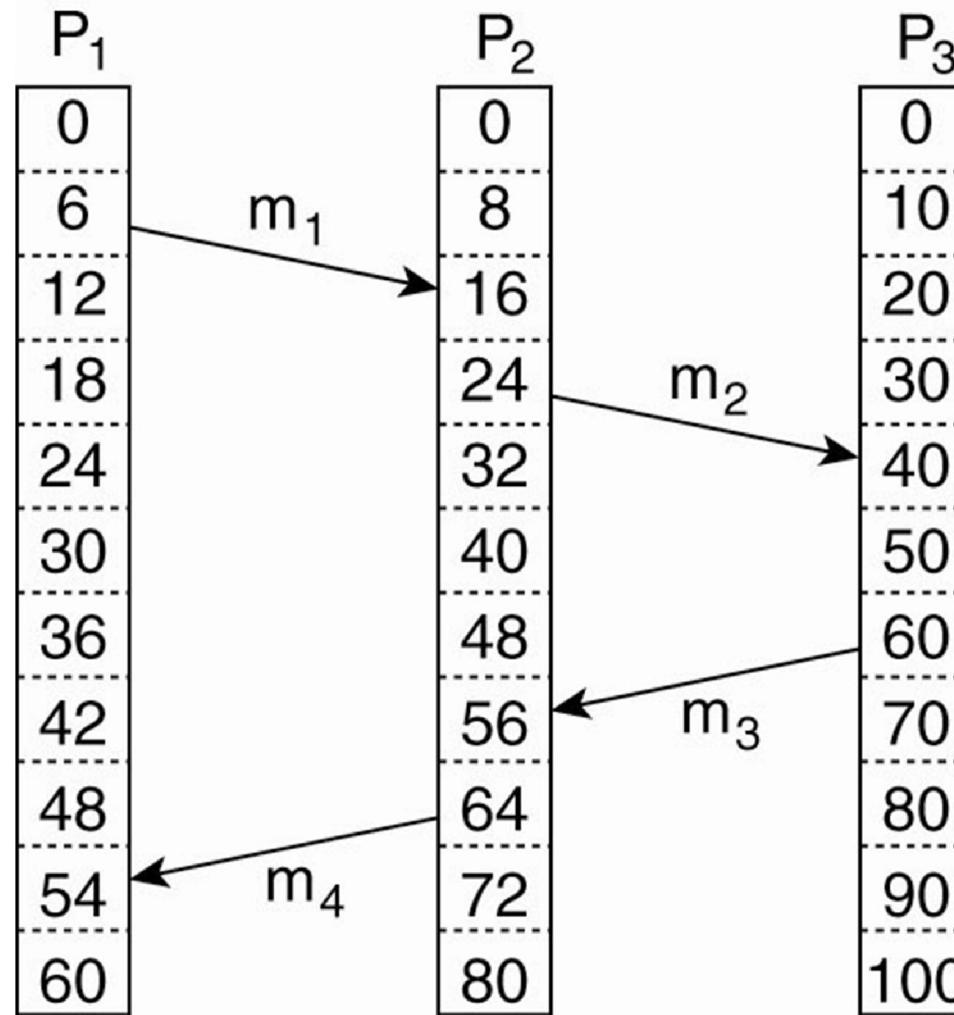
Concurrent

- If two events, x and y , happen in different processes that do not exchange messages (not even indirectly via third parties), then $x \rightarrow y$ is not true, but neither is $y \rightarrow x$. These events are said to be concurrent, which simply means that nothing can be.
- Assumption: no two events ever occur at exactly the same time

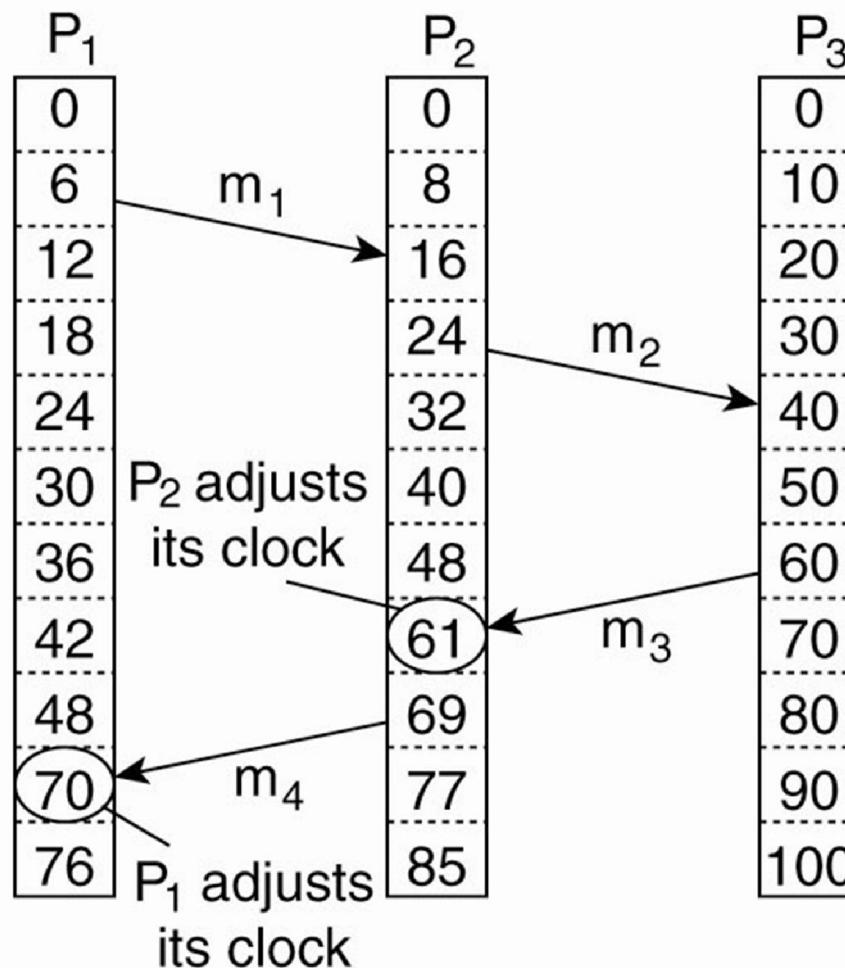
Lamport's Logic Clocks

- Lamport Logic Clocks C
- For every event, a , we can assign it a time value $C(a)$ on which all processes agree. These time values must have the property that if $a \rightarrow b$, then $C(a) < C(b)$.
- the clock time, C , must always go forward (increasing), never backward (decreasing).
- Corrections to time can be made by adding a positive value, never by subtracting one.

Three processes, each with its own clock.
The clocks run at different rates.



Three processes, each with its Lamport algorithm correcting the clocks

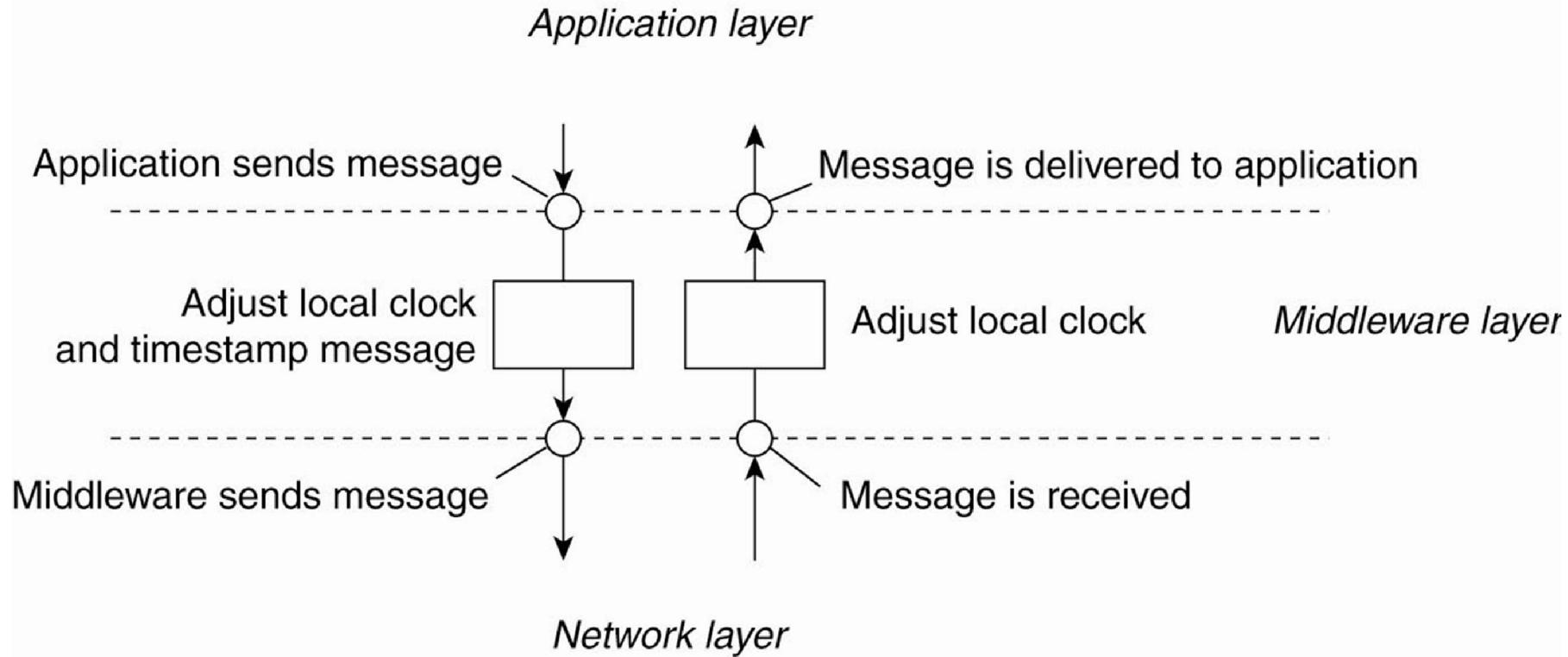


Each message carries the sending time according to the sender's clock

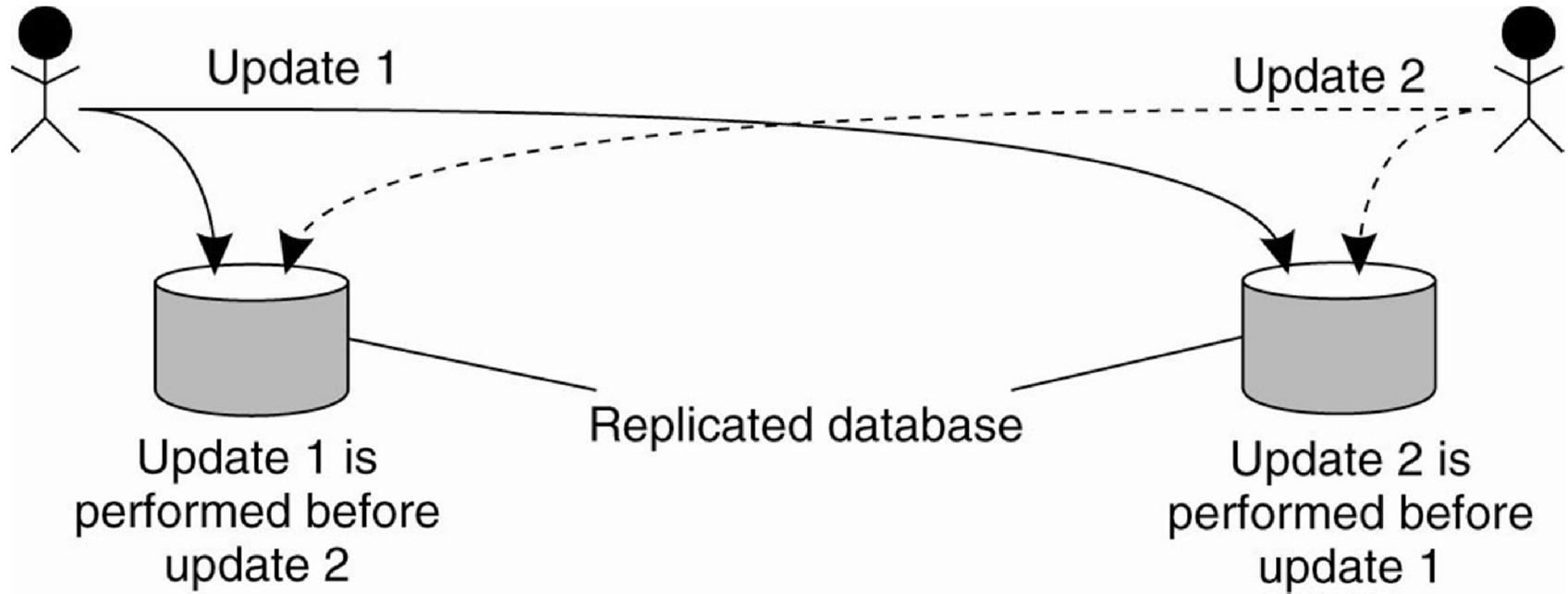
Lamport Algorithm

- Updating counter C_i for process P_i
 - Before executing an event P_i executes $C_i \leftarrow C_i + 1$. (or $+ \Delta i$ in general)
 - When process P_i sends a message m to P_j , it sets m 's timestamp $ts(m)$ equal to C_i after having executed the previous step.
 - Upon the receipt of a message m , process P_j adjusts its own local counter as $C_j \leftarrow \max\{C_j, ts(m)\}$, after which it then executes the first step and delivers the message to the application.

Lamport's Logical Clock



Example: Totally Ordered Multicasting



Key concerns are: do operations commute (final state same); what values do clients see (intermediate states)

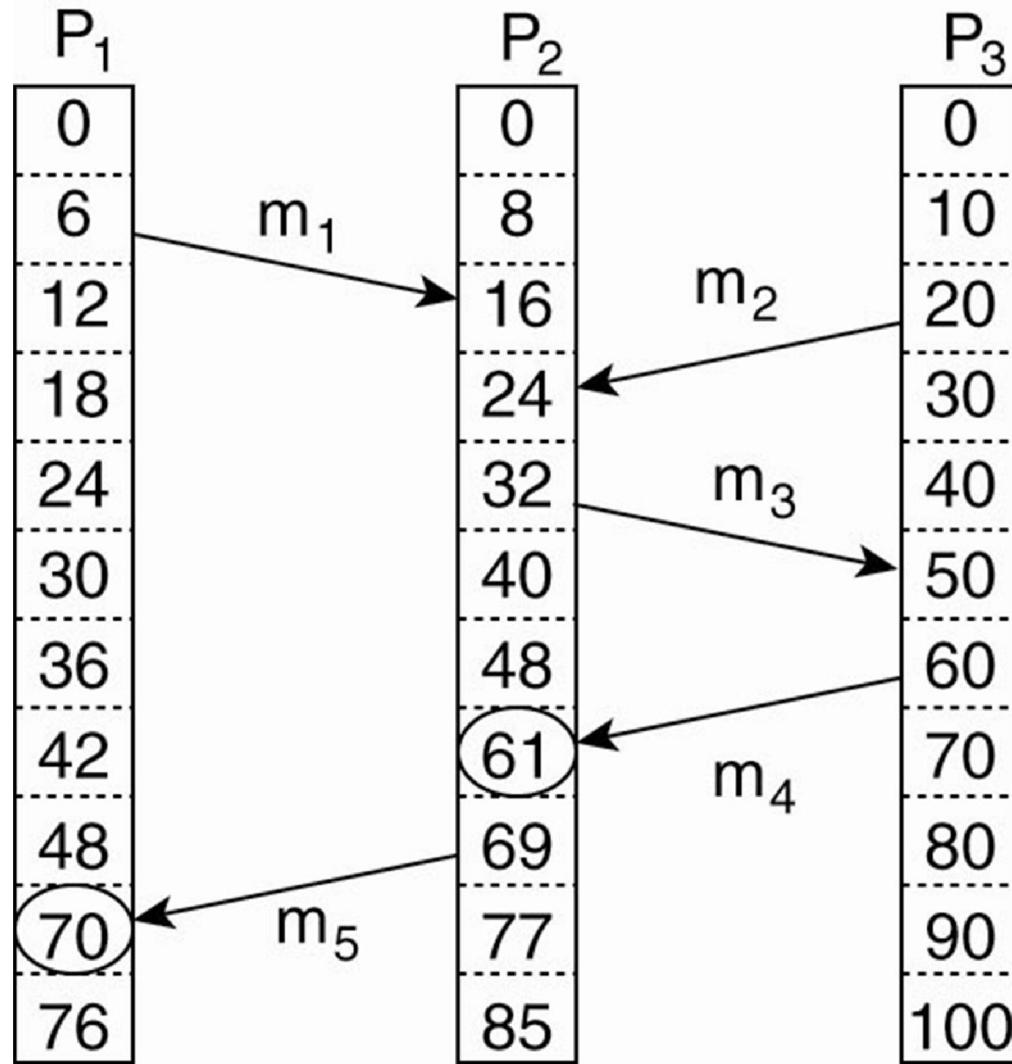
Causal Message Delivery

- Want a method by which messages can be delivered in causal order, not necessarily in some total order across all processes
 - Lamport logical clocks impose a total order that obeys causal order, but this is too restrictive
 - Messages that are not causally related have a delivery order imposed on them unnecessarily
- Vector clocks can capture causality information precisely

Concurrent message transmission using logical clocks

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

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



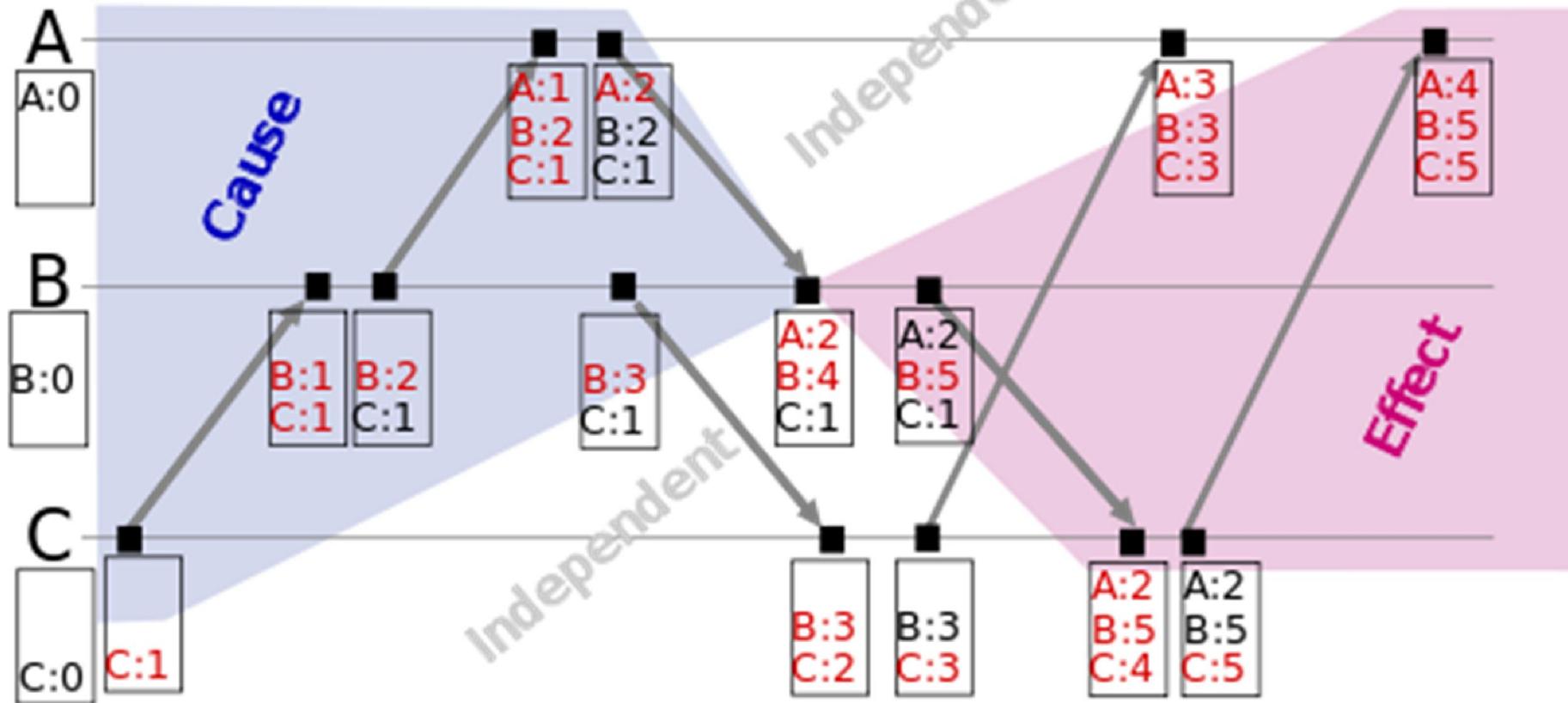
Vector Clocks

- If $VC(a) < VC(b)$ for some event b , then event a is known to causally precede event b .
- Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:
 - $VC_i[i]$ is the number of events that have occurred so far at P_i . In other words, $VC_i[i]$ is the local logical clock at process P_i .
 - If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j . It is thus P_i 's knowledge of the local time at P_j .

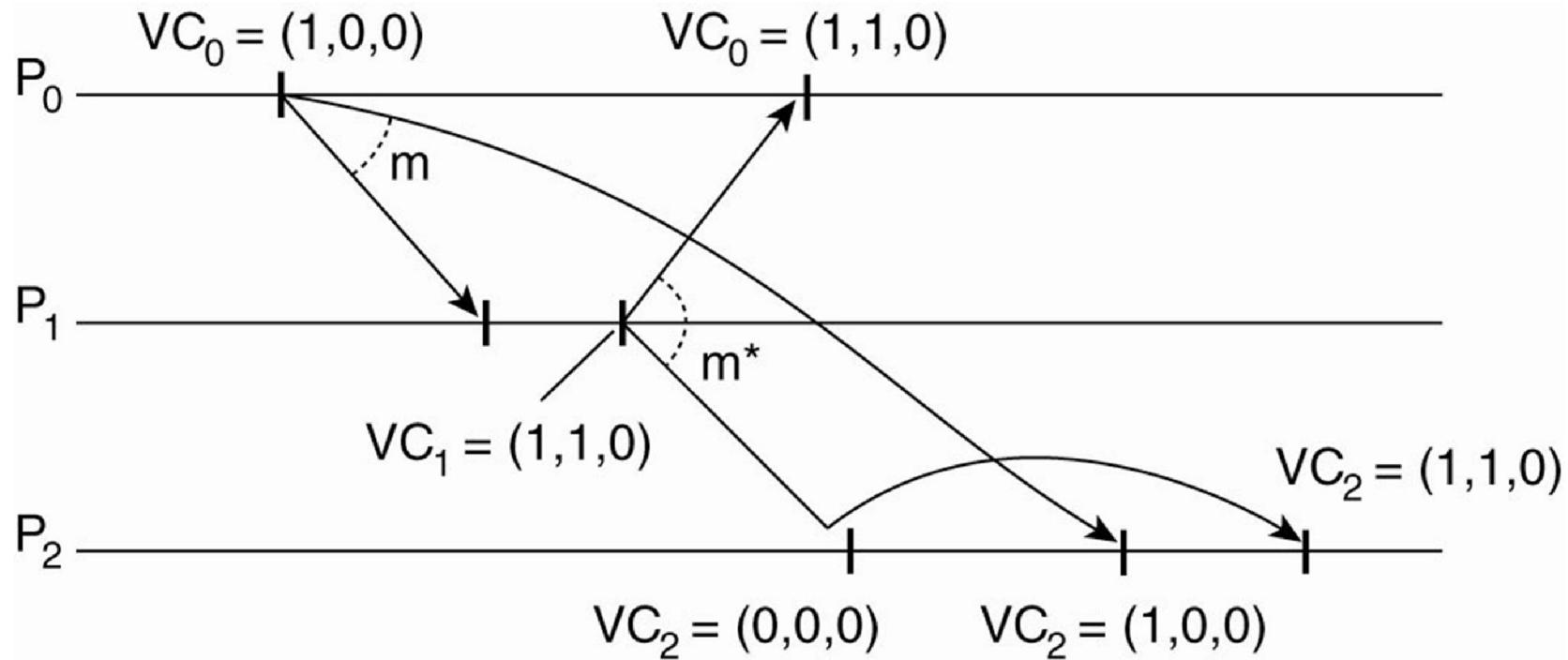
Vector Clocks

- Before executing an event (e.g., message send) P_i executes $VC_i[i] = VC_i[i] + 1$.
- When process P_i sends a message m to P_j , it sets m 's (vector) timestamp $ts(m) = VC_i$.
- Upon the receipt of a message m from P_i , process P_j adjusts its own vector by setting $VC_j[j] = \max\{ VC_j[j], ts(m)[j] \}$
- Comparing:
 - $V = V'$ iff all $V[k] = V'[k]$ of any k
 - $V < V'$ iff any $V[k] < V'[k]$ of any k

Time



Enforcing Causal Communication



Outline

- Physical Clock Synchronization
- Logical Clocks
- Mutual Exclusion
- Election Algorithms

Mutual Exclusion

- Processes will need to simultaneously access the same resources.
- To prevent that such concurrent accesses corrupt the resource, or make it inconsistent, solutions are needed to grant **mutual exclusive access by processes**.

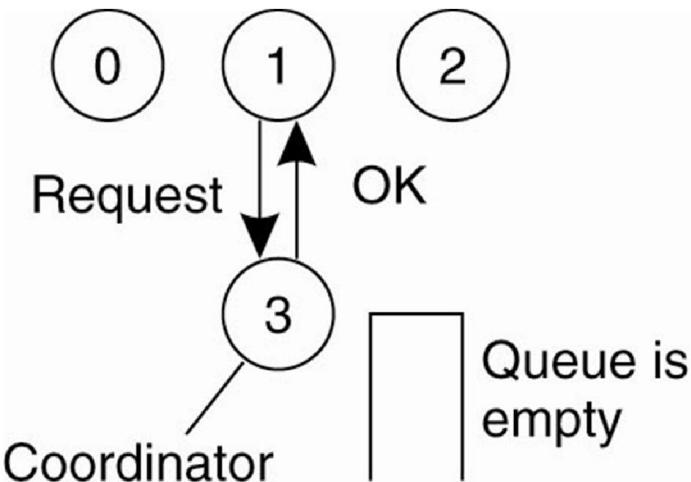
Token-based approach

- Achieving by passing a special message between the processes, known as a token.
 - There is only one token available and who ever has that token is allowed to access the shared resource.
 - The token is passed on to a next process when finished.
 - If a process having the token is not interested in accessing the resource, it simply passes it on.
- Token-based approach avoid starvation.
- Deadlocks by which several processes are waiting for each other to proceed, can easily be avoided, contributing to their simplicity.
- The main drawback of token-based solutions is a rather serious one: when the token is lost.

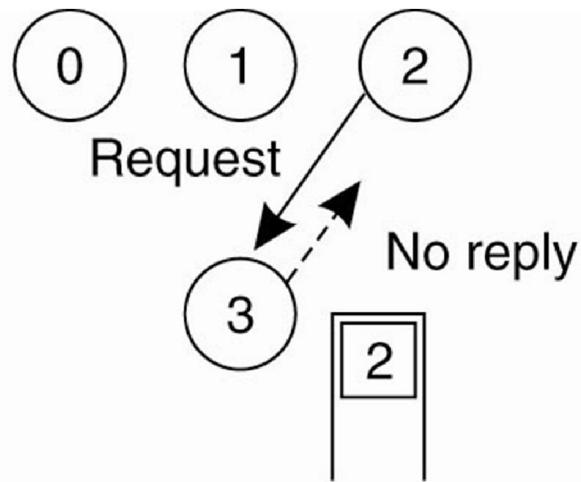
Permission-based approach

- A process want to access the resource first requires the permission of other processes.
 - Centralized Algorithms
 - Decentralized Algorithms
 - Distributed Algorithms
 - Token Ring Algorithm (pervious slide)

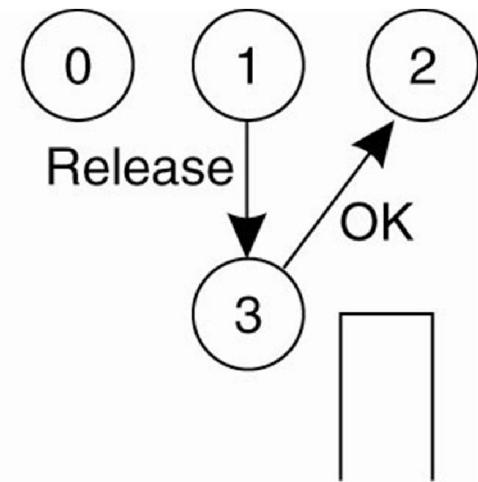
Centralized Algorithms



(a)



(b)



(c)

- The coordinator is a single point of failure, so if it crashes, the entire system may go down.
- If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back.

Decentralized Algorithm

- Each resource is assumed to be replicated n times. Every replica has its own coordinator for controlling the access by concurrent processes
- Whenever a process wants to access the resource, it will simply need to get a majority vote from $m > n/2$ coordinators.
 - Failures of a single coordinator
 - Problem of recovery
 - Problem of concurrency

Probability

- Let p be the probability that a coordinator resets during a time interval Δt .
- $P(k)$ that k out of m coordinators reset during the same interval

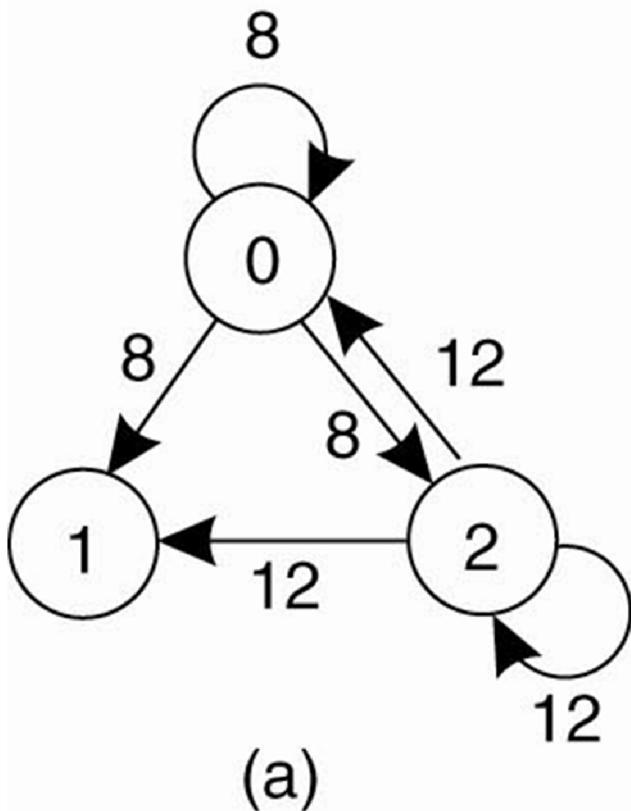
$$P[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- At least $2m - n$ coordinators need to reset in order to violate the correctness of the voting mechanism

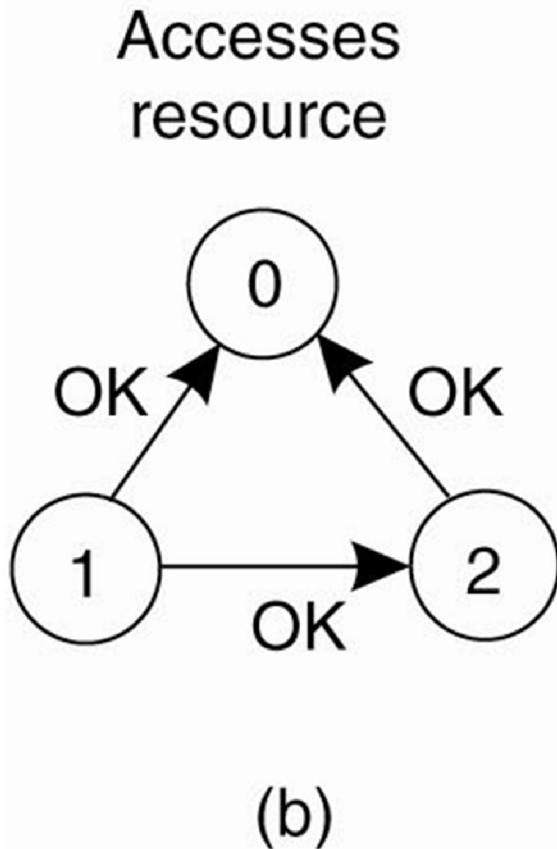
$$\sum_{k=2m-n}^n P(k)$$

Distributed Algorithm

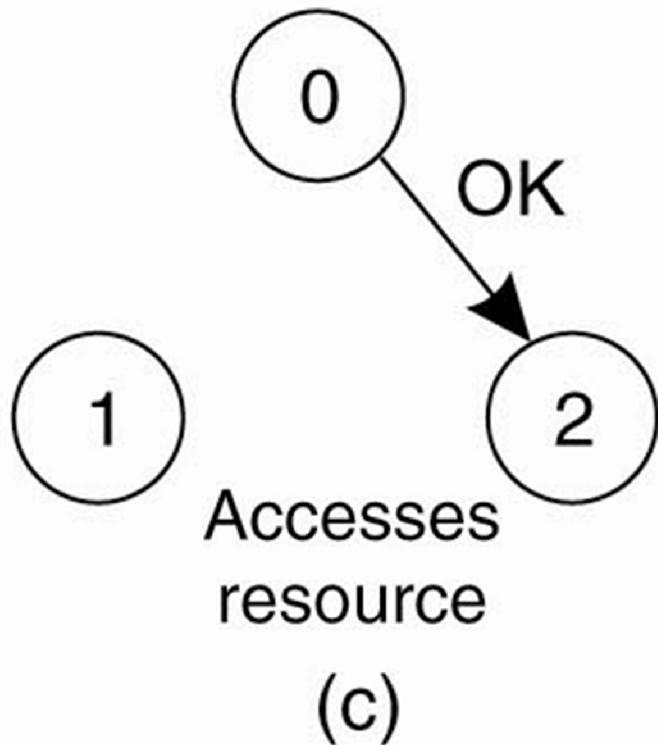
- Lamport Timestamp DME(Distributed Mutual Exclusion) algorithm
- When a process wants to access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time.
- When a process receives a request message from another process,
 - If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
 - If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
 - If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.



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



(b) Process 0 has the lowest timestamp, so it wins.



(c) When process 0 is done, it sends an OK also, so 2 can now go ahead.

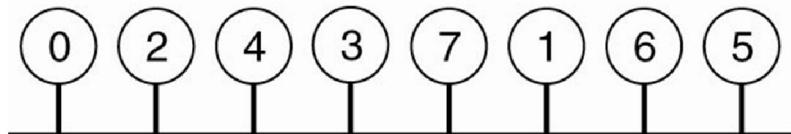
Lamport Timestamp DME algorithm

- It is guaranteed without deadlock or starvation.
 - The number of messages required per entry is now $2(n - 1)$.
- No single point of failure exists, but N points of failure
 - If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as **denial of permission**, thus blocking all subsequent attempts by all processes to enter all critical regions.
- Multicast communication, small groups of processes that never change their group memberships.
- Bottleneck, all processes are involved in all decisions.

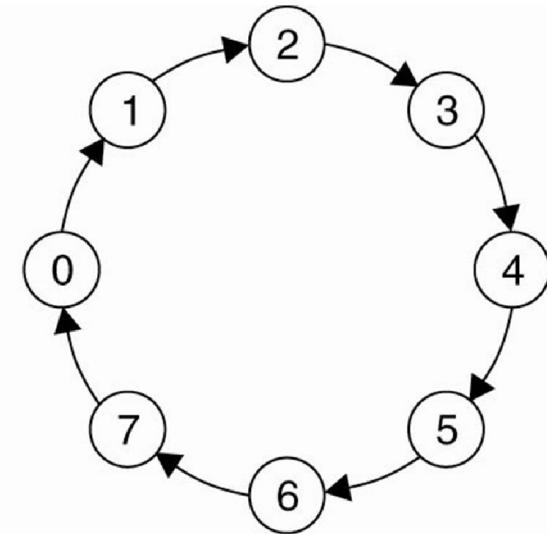
Lamport Timestamp DME algorithm

- Nevertheless, this algorithm is **slower**, more **complicated**, more **expensive**, and less **robust** than the original centralized one.
- It shows that a distributed algorithm is at least possible.
- It is updated many times.

A Token Ring Algorithm



(a)



(b)

- It does not matter what the ordering is.
- All that matters is that each process knows who is next in line after itself.
- Detecting that the token is lost is difficult

A Comparison of the Four Algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	$3mk$, $k = 1, 2, \dots$	$2m$	Starvation, low efficiency
Distributed	$2(n - 1)$	$2(n - 1)$	Crash of any process
Token ring	1 to ∞	0 to $n - 1$	Lost token, process crash

m coordinators try k times

- It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one

Outline

- Physical Clock Synchronization
- Logical Clocks
- Mutual Exclusion
- Election Algorithms

Election Algorithms

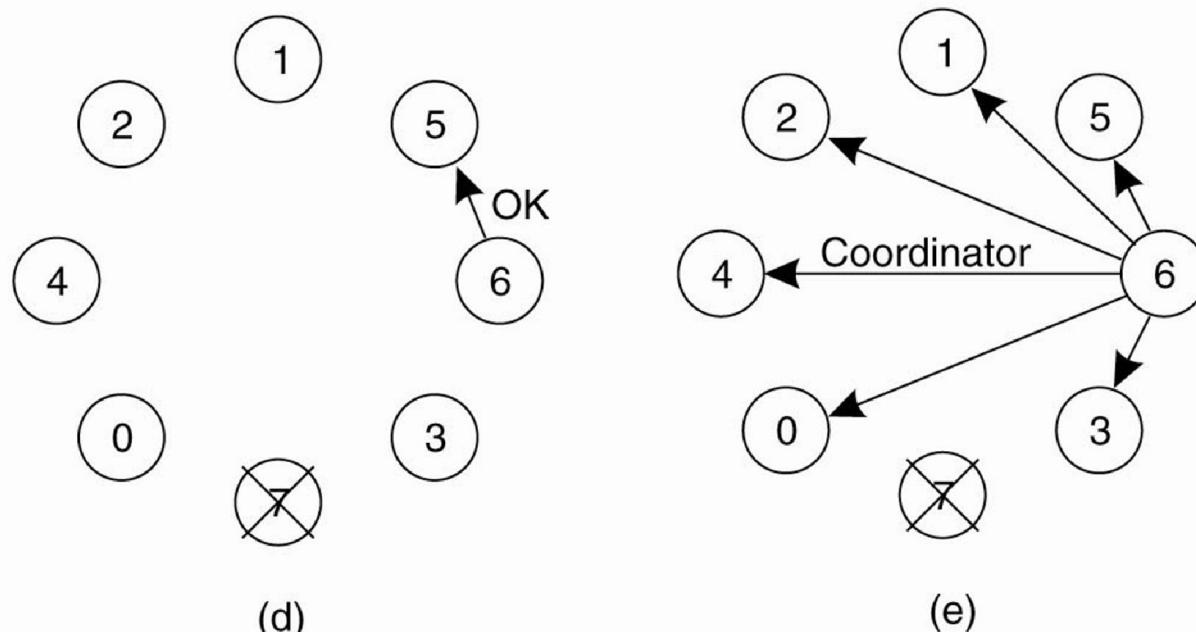
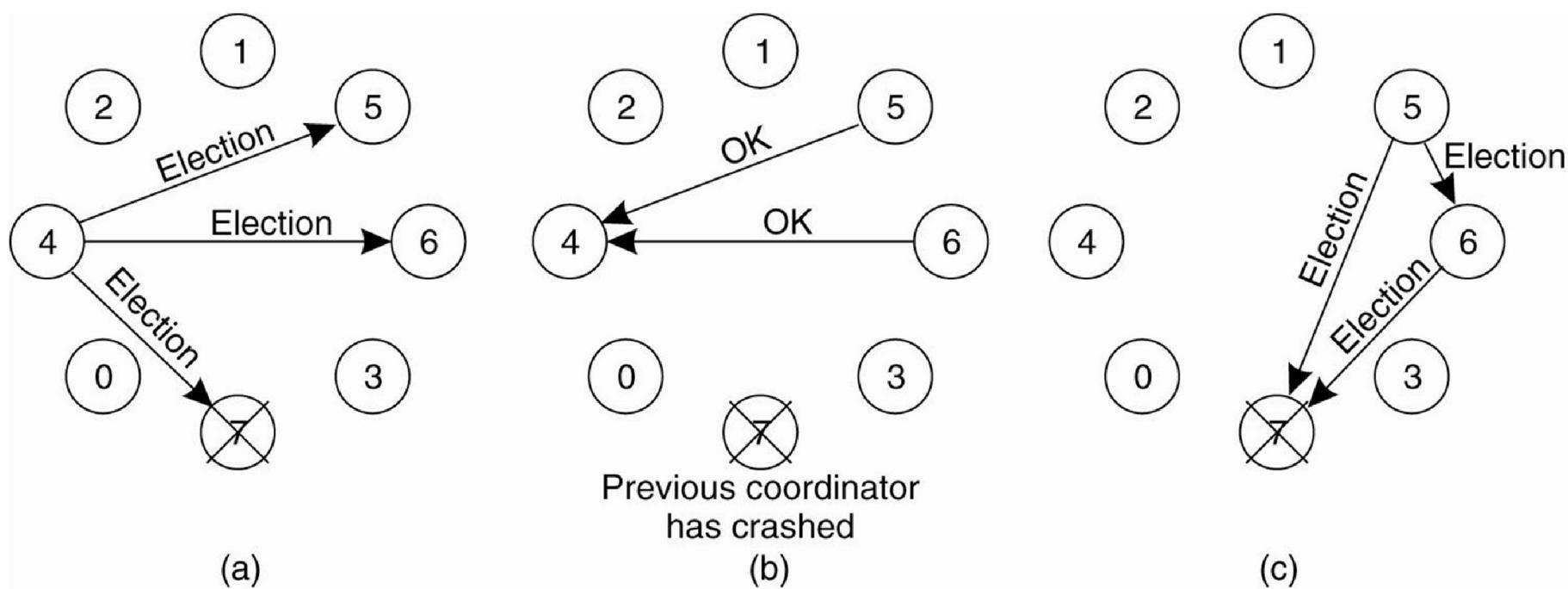
- Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role.
- In general, it does not matter which process takes on this special responsibility, but one of them has to do it.
- **Algorithms for electing a coordinator**

Algorithms for electing a coordinator

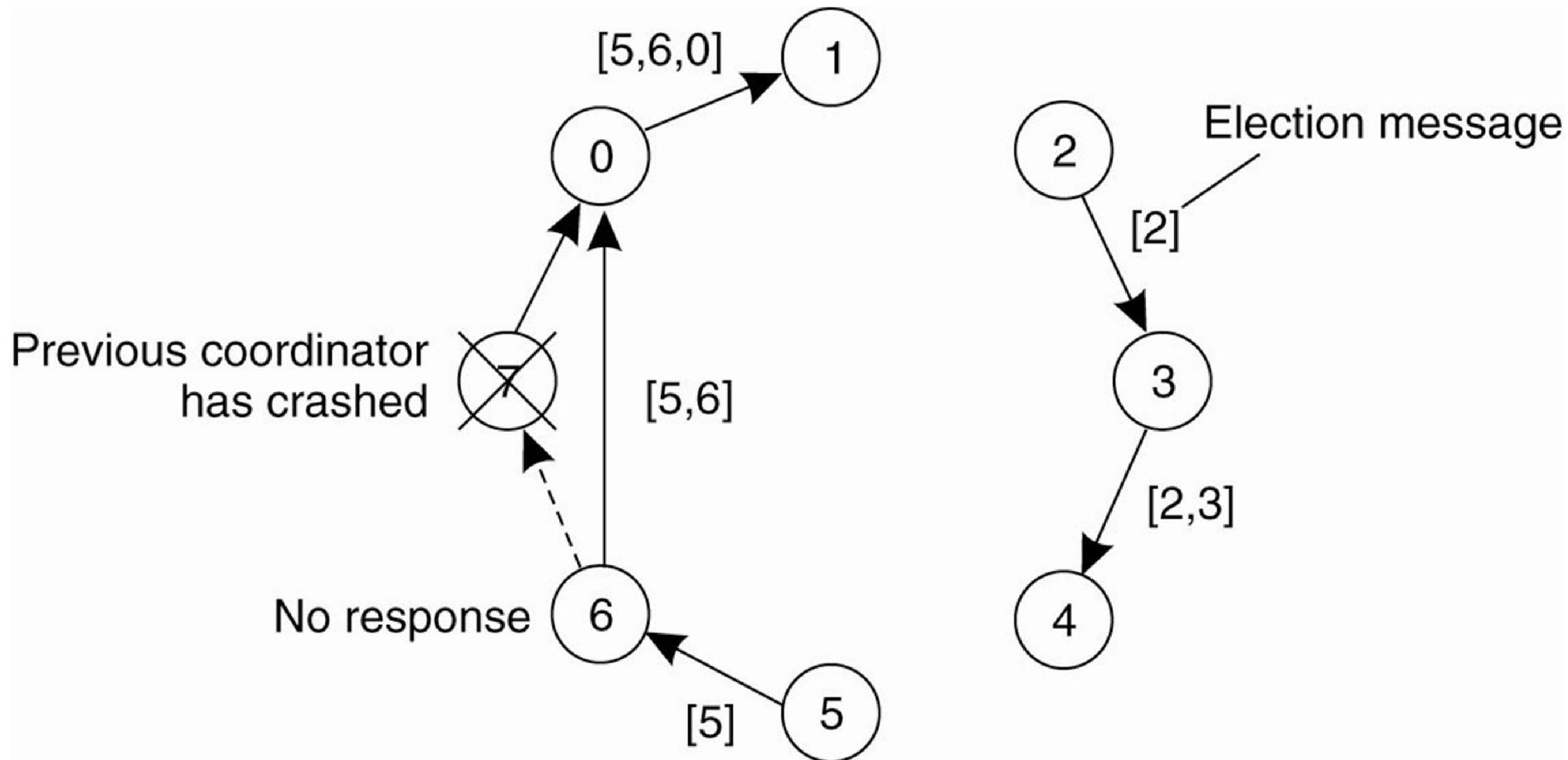
- Assuming all processes are exactly the same except their process numbers.
- Assuming every process knows the process number of every other process.
- The process with the **largest process numbers** should be coordinator.

Bully Algorithm (1982)

- When any process notices that the coordinator is no longer responding to requests, it initiates an election.
- A process, P , holds an election as follows:
 - P sends an ELECTION message to all processes with higher numbers.
 - If no one responds, P wins the election and becomes coordinator.
 - If one of the higher-ups answers, it takes over. P 's job is done.
- If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job.

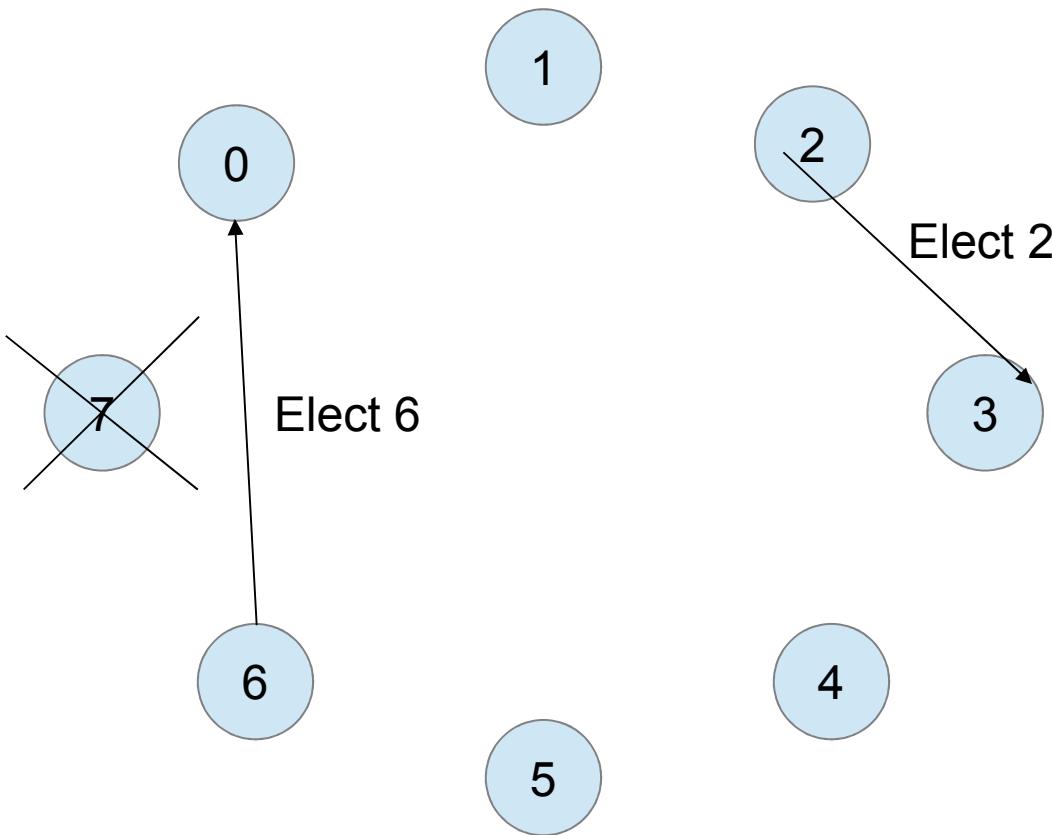


A Ring Algorithm



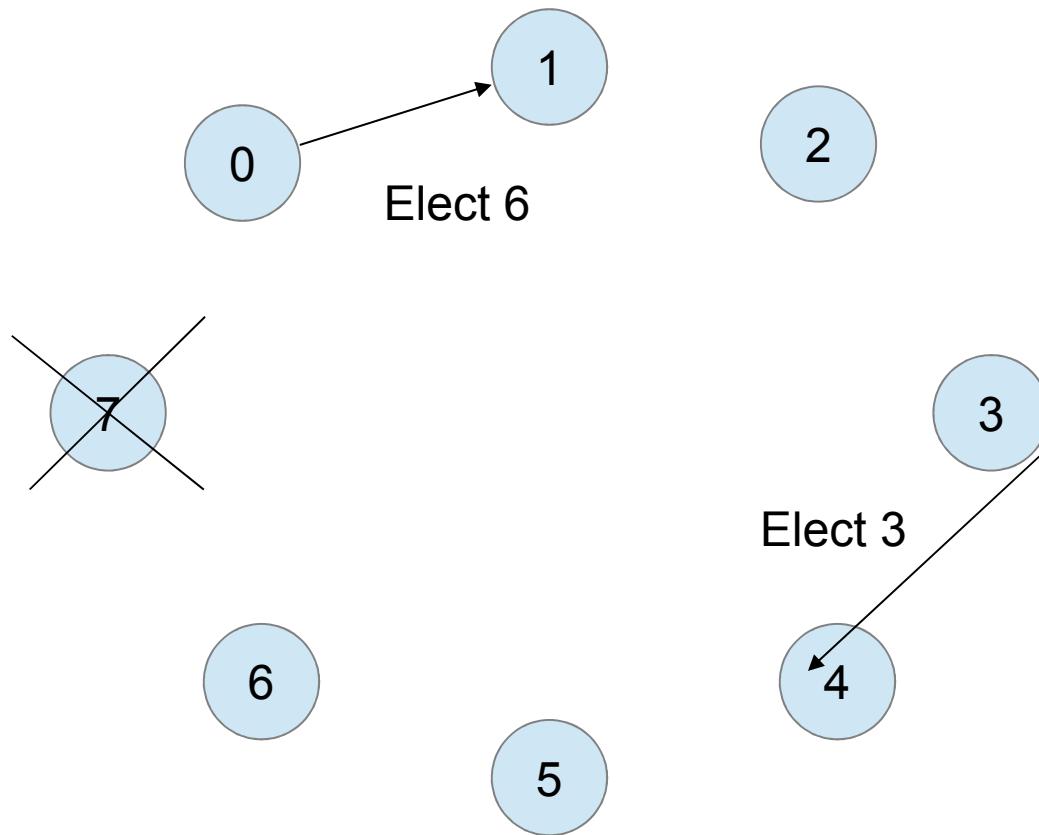
Note that here, each message collects node IDs as it circulates the ring – the winner will know all live nodes at end; only the best candidate ID must be circulated, and new message propagated only if its candidate is better

Ring Algorithm



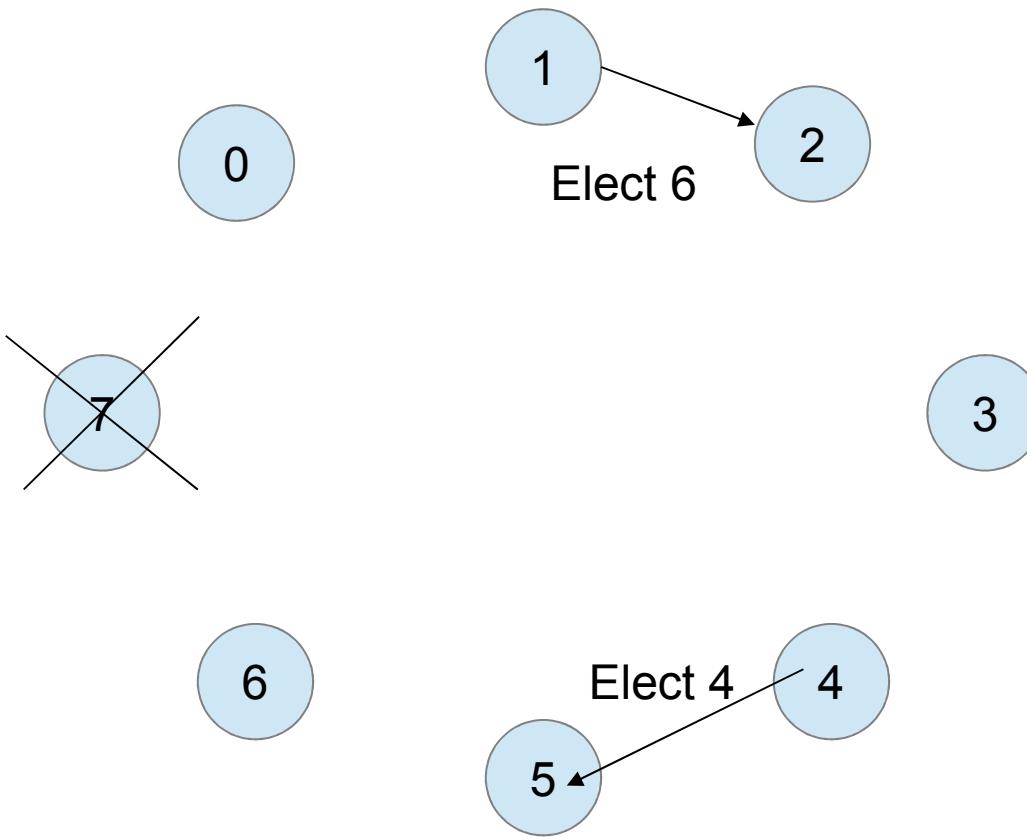
Nodes 6 and 2 initiate elections, believing node 7 (previous leader) to be dead. Here, only the best candidate ID is circulated. A node enters election if it detects leader failure or if it receives an election message.

Ring Algorithm



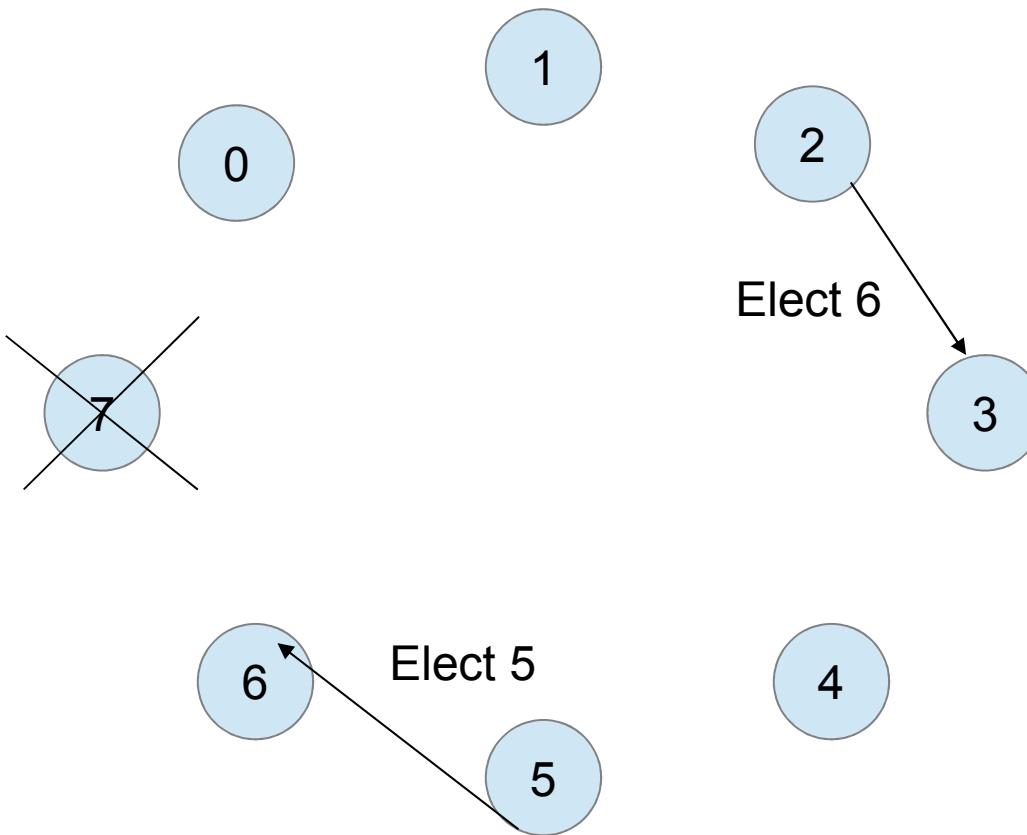
Nodes 0 and 3 receive election messages, enter election. A node sends an election message if it initiates or if it receives a message with a better candidate. Node 0 continues with 6 as the best candidate, while node 3 considers itself the best candidate.

Ring Algorithm



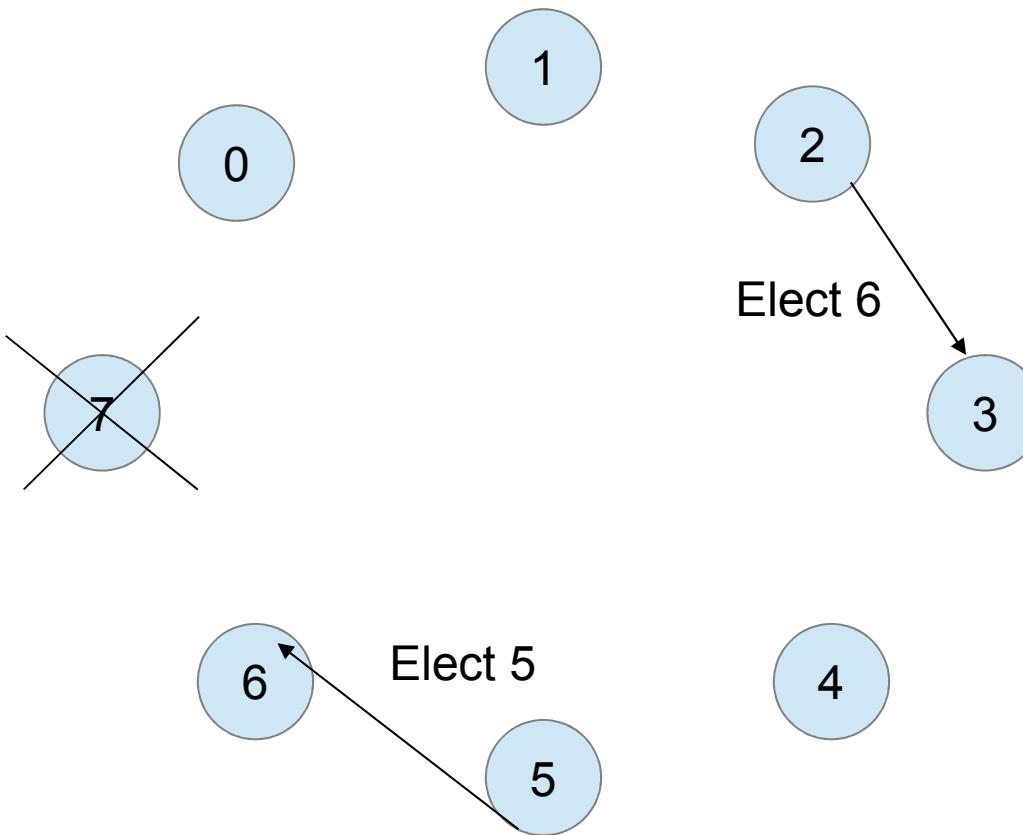
Nodes 1 and 4 receive election messages, enter election. Node 1 continues with 6 as the best candidate, while node 4 considers itself the best candidate.

Ring Algorithm



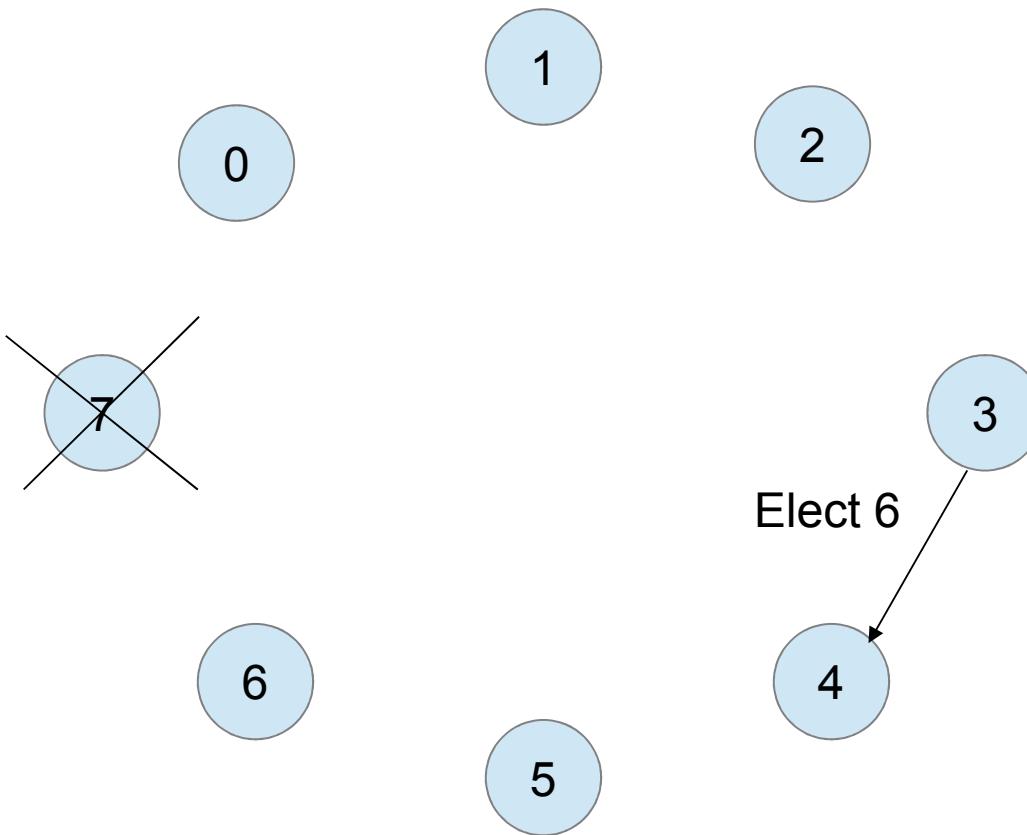
Nodes 5 and 2 receive election messages, 5 enters election (2 is already in). Node 2 realizes that 6 is a better candidate than the best it has seen (i.e., 2) so forwards message, while node 5 considers itself the best candidate.

Ring Algorithm



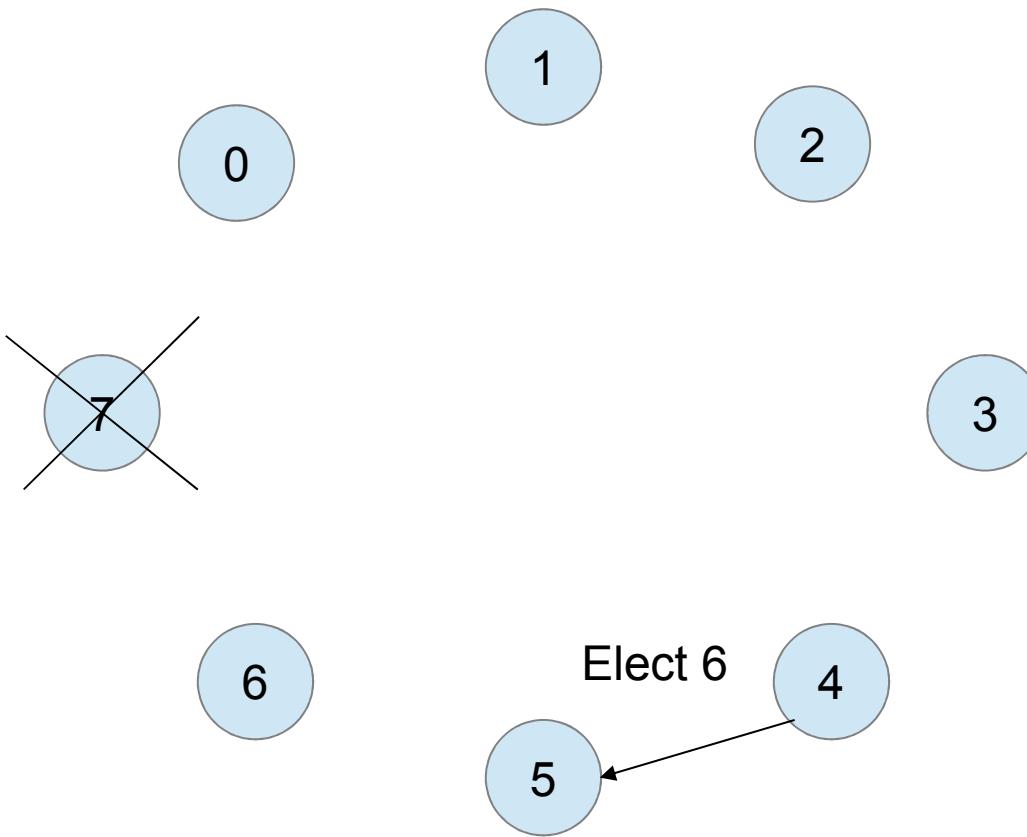
Nodes 6 and 3 receive election messages, are both already in. Node 3 realizes that 6 is a better candidate than the best it has seen (i.e., 3) so forwards message, while node 6 knows it has seen a better candidate (itself).

Ring Algorithm



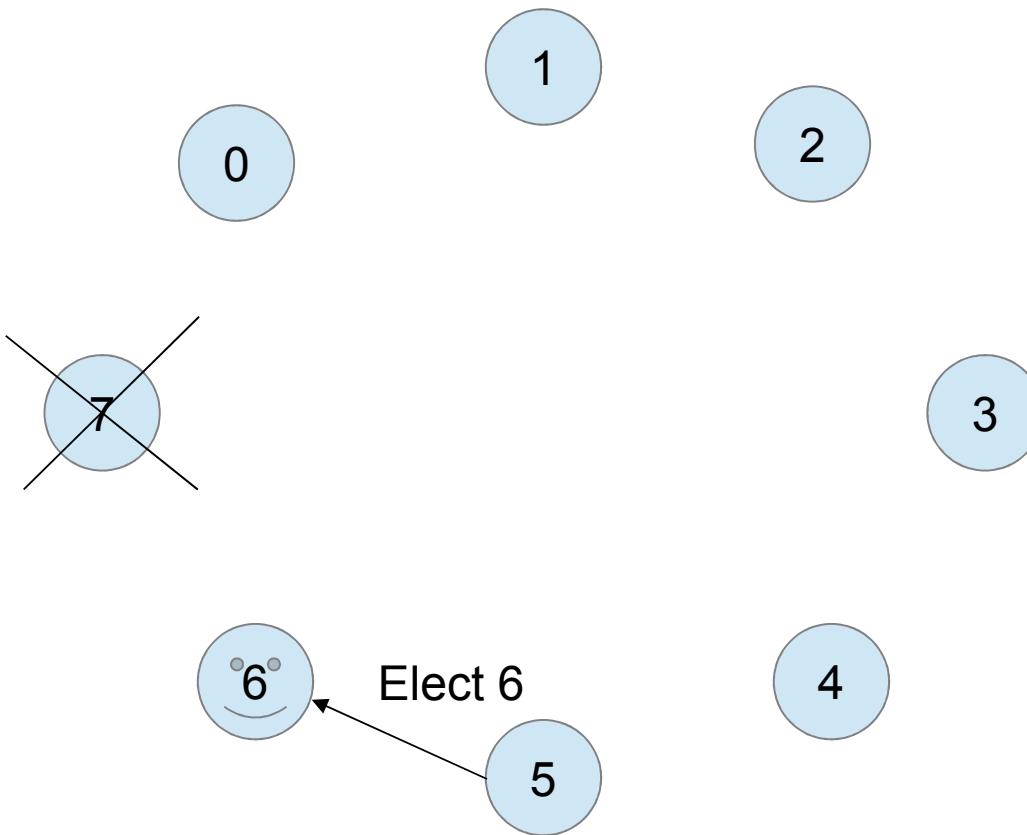
Node 4 receives election message, but is already in. Node 4 realizes that 6 is a better candidate than the best it has seen (i.e., 4) so forwards message, while node 6 waits, having suppressed the election message from 5.

Ring Algorithm



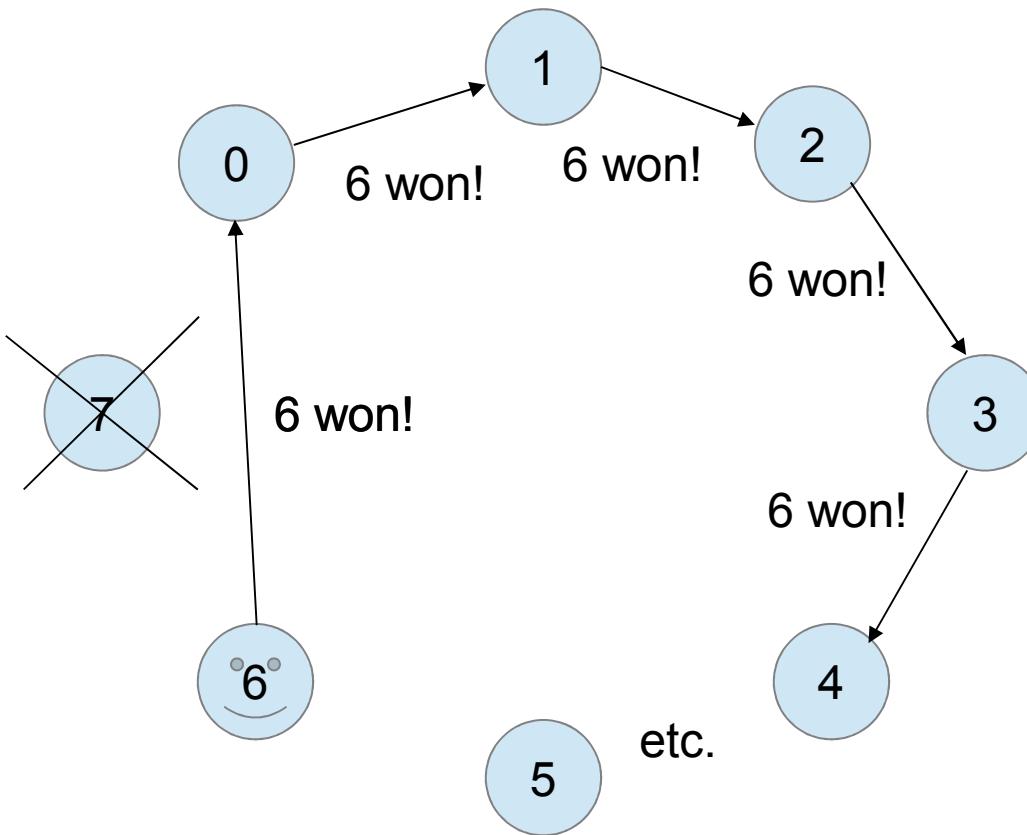
Node 5 receives election message, but is already in. Node 5 realizes that 6 is a better candidate than the best it has seen (i.e., 5) so forwards message.

Ring Algorithm



Node 6 receives election message with its own ID as the best candidate seen. Since only the candidate itself will start an election message with its own ID, it knows there is no better candidate in the ring and it has won!

Ring Algorithm



The winner knows it won when it sees its ID on a message, and then circulates an election results message around the ring with its ID. The results message causes each node to exit the election and return to normalcy.

Improving Elections

- Use of heartbeats to detect failure – make “better” candidates more sensitive (i.e., time out sooner) so only one node starts an election
- Node starting an election can just contact the best candidate it believes to be alive (to be sure)
 - To avoid lengthy delays, the initiator can contact the K best candidates, or
 - If the best “live” candidate(s) don't respond, it can escalate to contact more, lower-tier candidates

Elections vs. Distributed Mutual Exclusion

- While both types of protocol end up selecting a **single winner**, there are some important differences:
 - A DME participant keeps trying until it wins, this is not needed in leader election;
 - There is no issue of fairness in leader election;
 - All nodes need to know the winner when leader election is done, this is not needed in DME;
 - DME can be more message-efficient (i.e., sublinear) compared to leader election

Next Lesson...

DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 7
Consistency and Replication

Replication

- An important issue in distributed systems is the replication of data which enhance **reliability** or improve **performance**
- One of the major problems is keeping replicas consistent
 - One copy is updated we need to ensure that the other copies are updated as well;

Reliability

- Read: one replica crashes by simply switching to one of the other replicas.
- Write: against a single, failing write operation, by considering the value that is returned by at least two copies as being the correct one (3 copies).

Performance

- By replicating the server and subsequently dividing the work.
- By placing a copy of data in the proximity of the process using them, the time to access the data decreases.

Cost

- Whenever a copy is modified, that copy becomes different from the rest. Consequently, modifications have to be carried out on all copies to ensure consistency.
- **When and how** the modifications need to be carried out determines the price of replication.

Example: Web pages

- If no special measures are taken, fetching a page from a remote Web server is time-consuming.
- To improve performance, Web browsers often locally store a copy of a previously fetched Web page.
- However, if the user always wants to have the latest version of a page, he gets old pages.
- When and how the modifications can be propagated to cached copies?

Problems in detail

- Process P accesses a local replica N times per second, whereas the replica itself is updated M times per second. $N > M$? $N < M$? $N \ll M$?
 - Access-to-update ratio
- Keep all copies consistent generally requires global synchronization, which is inherently costly in terms of performance
 - Cost of Lamport timestamps

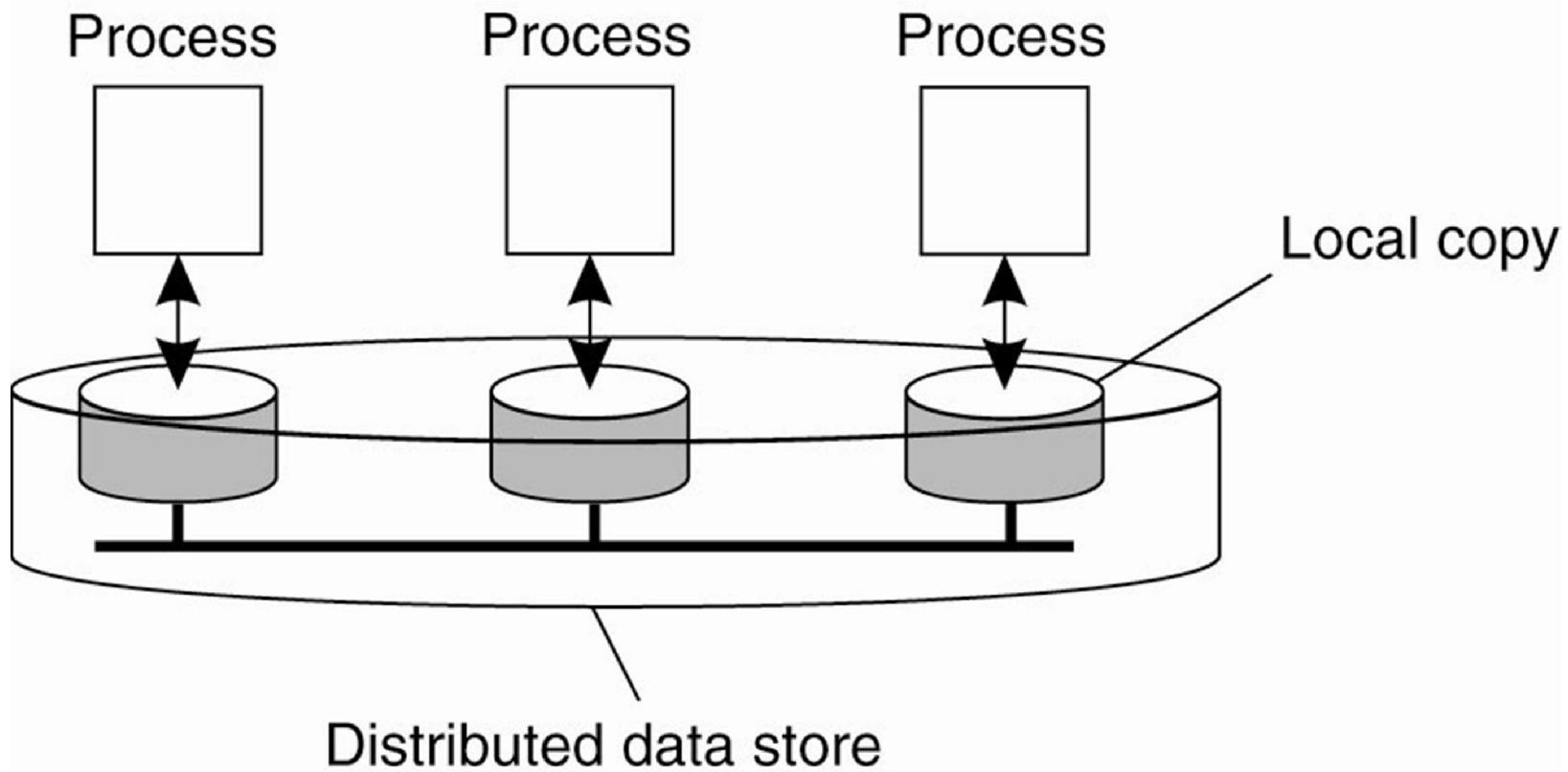
Outline

- Data-centric Consistency Models
 - Sequential Consistency
 - Causal Consistency
 - Eventual Consistency
- Client-centric Consistency Models
- Replica Management
- Consistency Protocols

Terminologies

- **Data store**: a data store may be physically distributed across multiple machines. Each process that can access data from the store is assumed to have a local (or nearby) copy available of the entire store.
- **Write and read**: a data operation is classified as a write operation when it changes the data, and is otherwise classified as a read operation.
- **Consistency model**: a consistency model is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly.

Data store



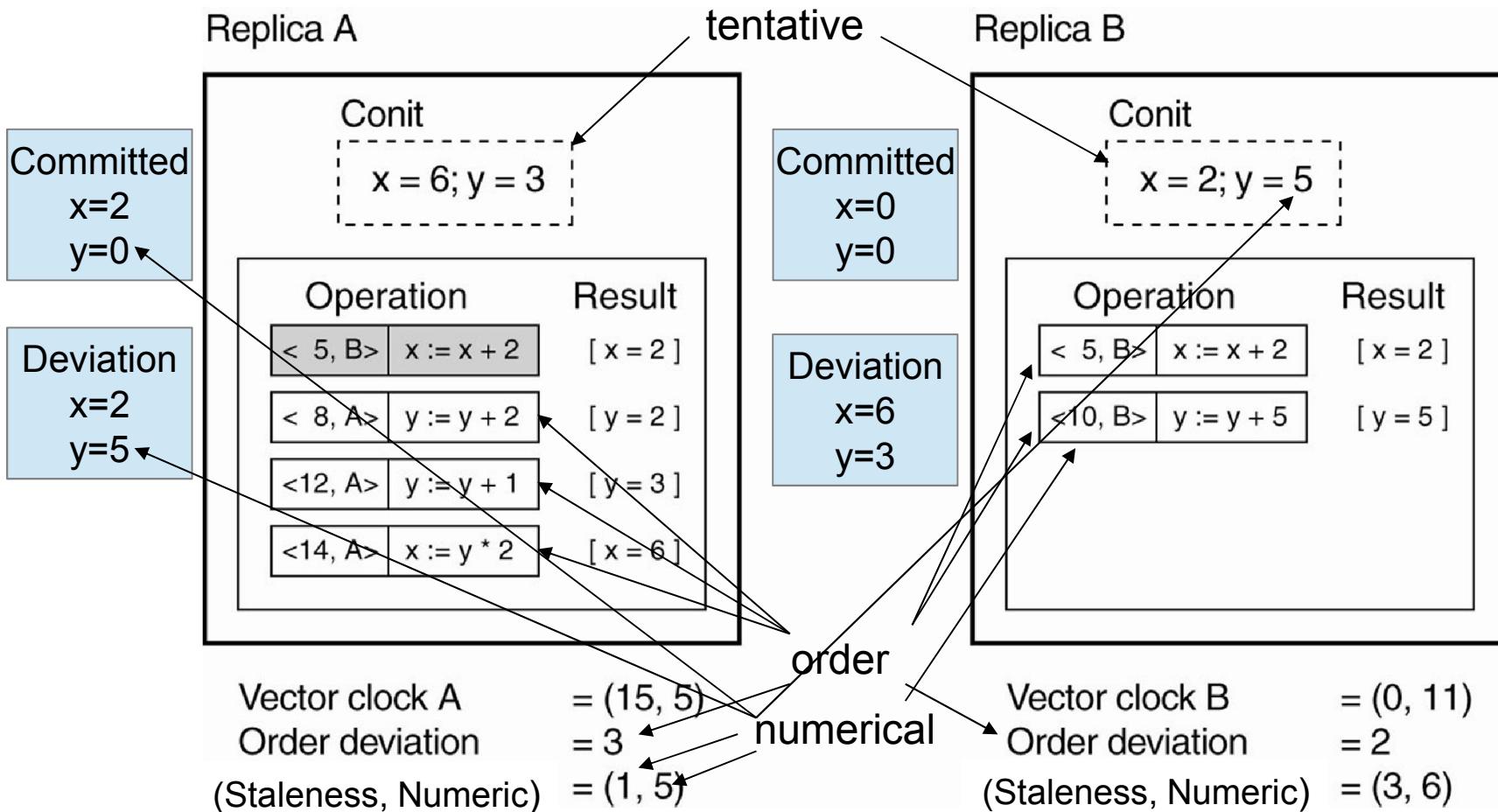
Continuous Consistency

- They refer to three deviations as forming continuous consistency ranges
 - Numeric: deviation in numerical values between replicas
 - Staleness: deviation in staleness between replicas
 - Order: deviation with respect to the ordering of update operations
- Absolute deviation (by $|v-v'|$)
- Relative deviation (by $|v-v'|/v$)

Consistency Unit

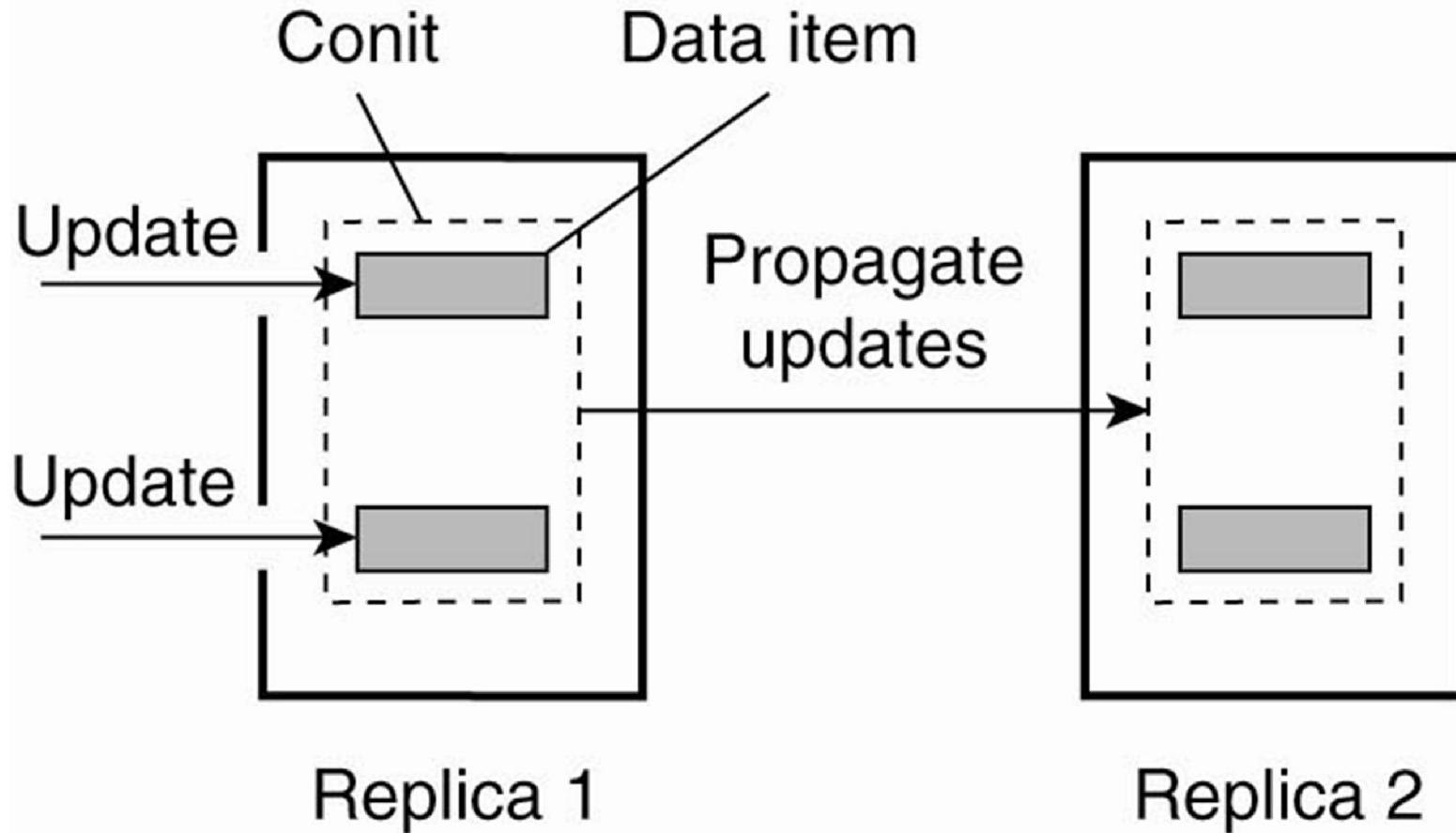
- Conit (CONSistency uNIT): a conit specifies the unit over which consistency is to be measured.
 - For example, in our stock-exchange example, a conit could be defined as a record representing a single stock.

init: $x=0, y=0$

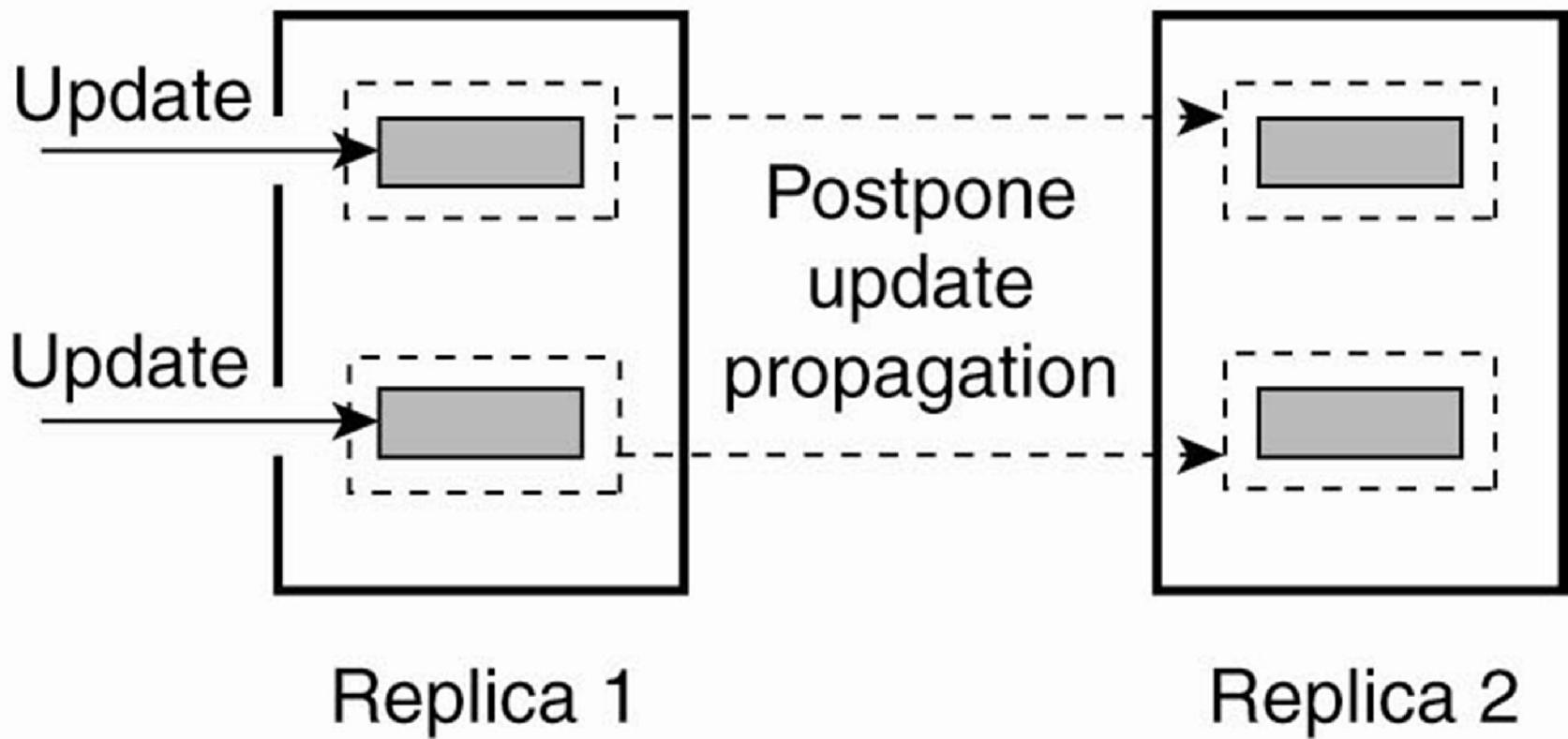


Order = # uncommitted local updates; Numerical (#unseen remote updates, max dev. between remote tentative and local committed); gray = committed

Granularites of conits (1)



Granularites of conits (2)



Tradeoffs:

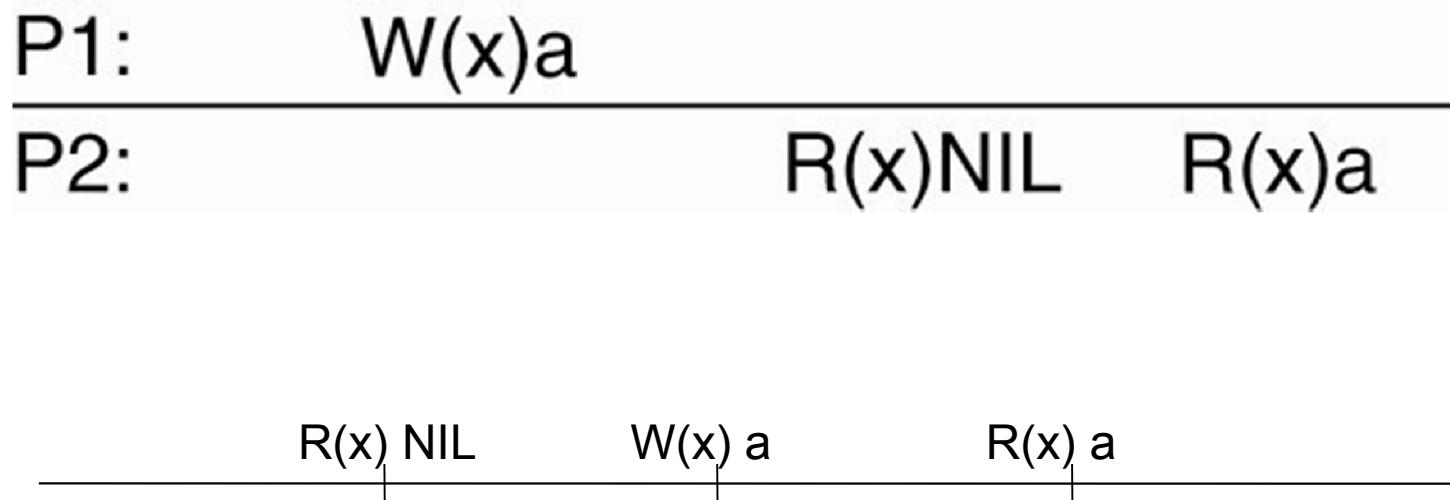
high traffic, false sharing if conit too large;
too stale, overhead too great if conit too small.

Access Consistency

- Forms of access consistency
 - Sequential (顺序) Consistency
 - Causal (因果) Consistency
 - Eventual Consistency
- Notation:
 - $W_i(x)a$ = Process i wrote object x with value a
 - $R_i(x)a$ = Process i read object x with value a
 - In slides, \square_i is abbreviated, i is a process ID.

Sequential Consistency

Behavior of two processes operating on the same data item. The horizontal axis is time.



A sequential order satisfying the observations

The result of any execution is the same **as if the (read and write) operations by all processes on the data store were executed in some sequential order **and** the operations of each individual process appear in this sequence in the order specified by its program.**

- Lamport

- Any valid interleaving of read and write operations is acceptable behavior;
- All processes see the same interleaving of operations;
- A process "sees" **writes** from all processes (may not in-time) but only its own **reads**.
- Nothing is said about time, e.g. "most recent" operation.

Sequential Consistency (1)

- Time does not play a role

P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

P4: R(x)b R(x)a

(a)

P3: $W_2(x)b, R_3(x)b, W_1(x)a, R_3(x)a$

P4: $W_2(x)b, R_4(x)b, W_1(x)a, R_4(x)a$

P3: $W_2(x)b, R_3(x)b, W_1(x)a, R_3(x)a$

P4: $W_1(x)a, R_4(x)a, W_1(x)b, R_4(x)b$

P1: W(x)a

P2: W(x)b

P3:

P4:

Forces $W(x)b$ first

(b)

Forces $W(x)a$ first

Sequential Consistency (2)

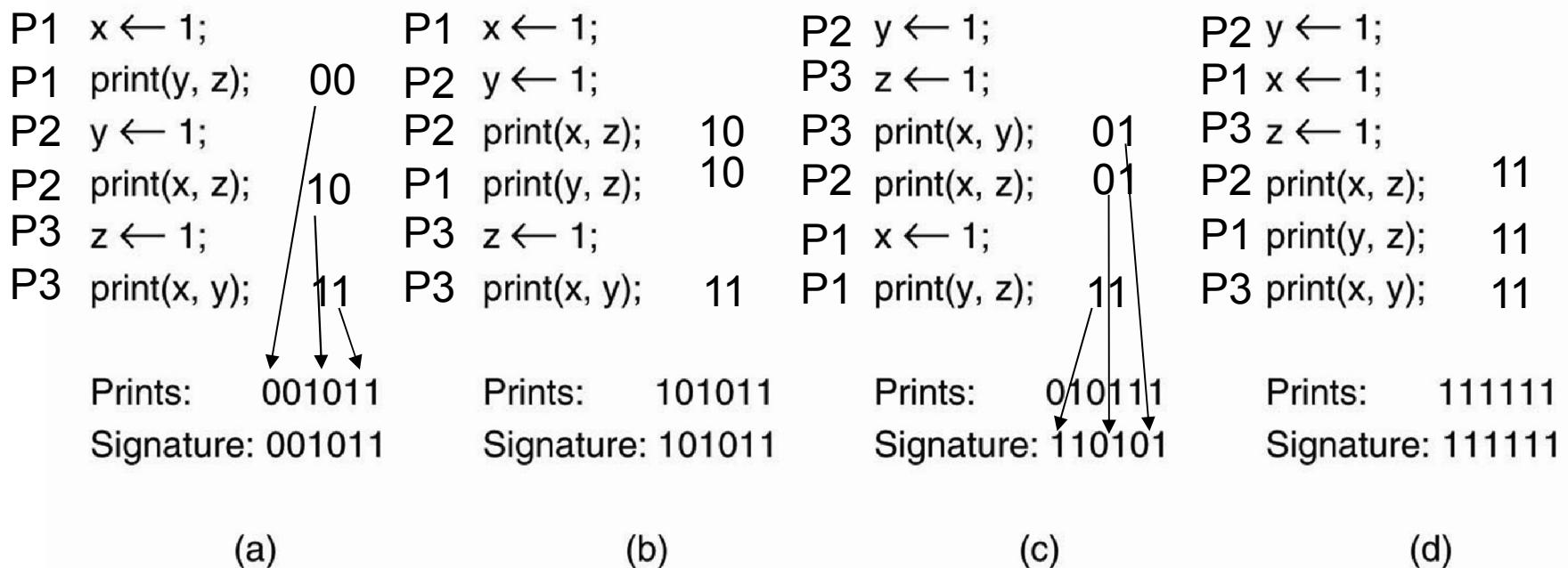
Various interleaved execution sequences are possible

Process P1	Process P2	Process P3
$x \leftarrow 1;$ <code>print(y, z);</code>	$y \leftarrow 1;$ <code>print(x, z);</code>	$z \leftarrow 1;$ <code>print(x, y);</code>

- With six independent statements, there are potentially 720 (6!) possible execution sequences,
 - Some of these violate program order.
- 90 ($720 \div 2 \div 2 \div 2$) valid execution sequences.

Sequential Consistency (3)

Initial values all 0



- Signature is output of processes in process ID order: $P1 \rightarrow P2 \rightarrow P3$
- 000000 is not permitted, because that would imply that the print statements ran before the assignment statements
- An other impossible Example is 001001, 00 means P1 before P2 and P3, 10 means P2 before P3, 01 means P3 before P1

Sequential Consistency (4)

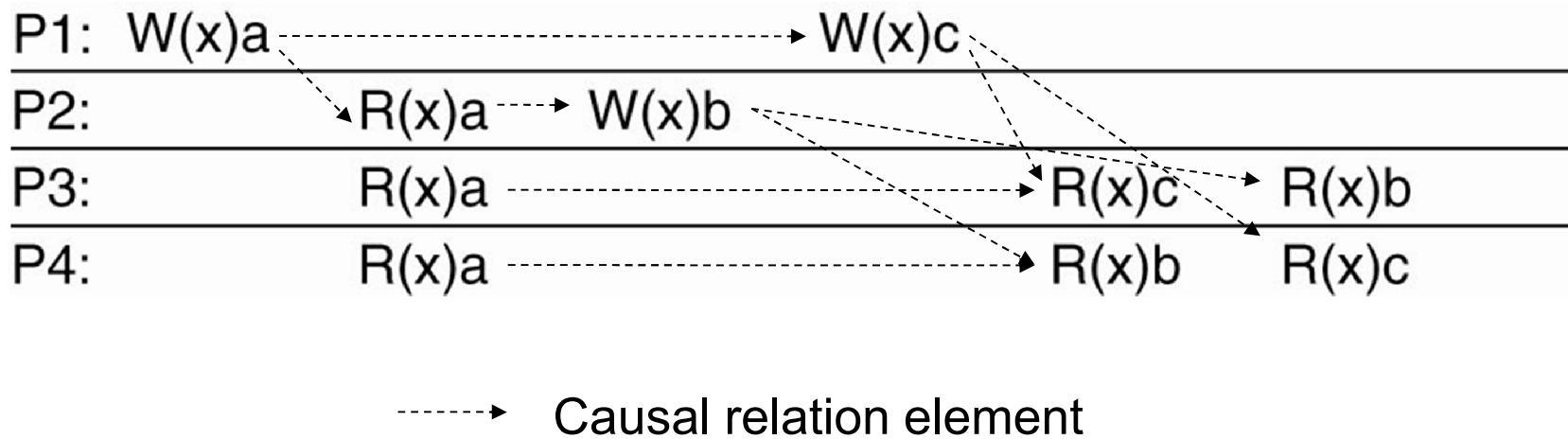
- There is possible 64 statement;
- 90 different valid statement orderings produce less than 64 results, and the processes must accept all of these as valid results;
- The contract between the processes and the distributed data store is that **the processes must accept all of these as valid results.**

Causal Consistency

Writes that are potentially causally related must be seen by all processes in the same order.
Concurrent writes may be seen in a different order on different machines.

- Concurrent: not causally related
- Writes that are potentially causally related ...
 - must be seen by all processes
 - in the same order.
- Concurrent writes ...
 - may be seen in a different order
 - On different machines.

Example

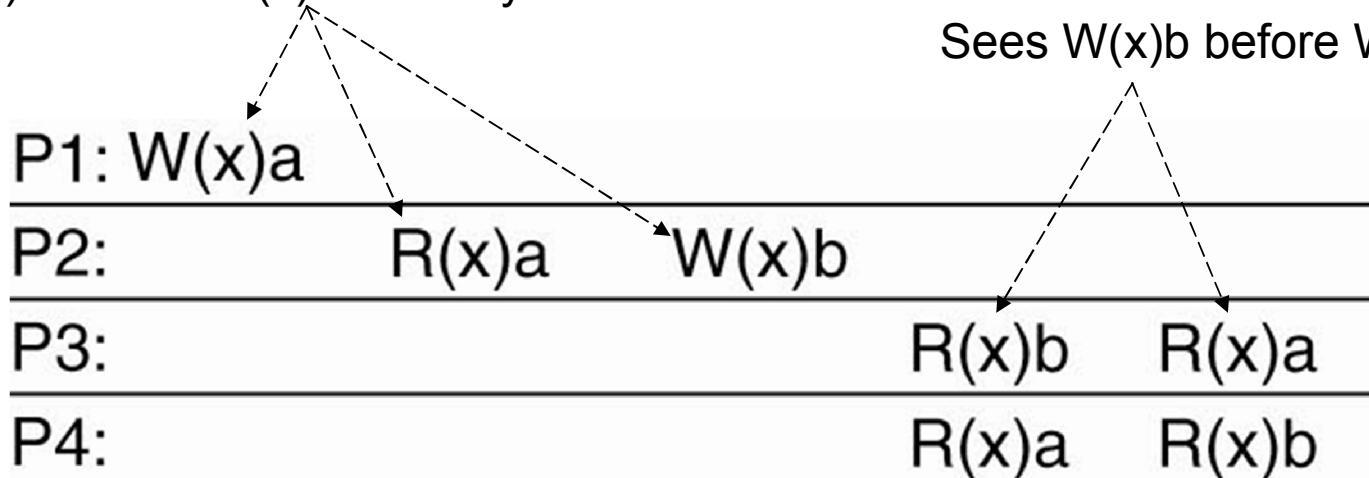


This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

Induced digraph is acyclic

Example

$W(x)a$ before $W(x)b$ causally



- Now consider a second example. We have $W_2(x)b$ potentially depending on $W_1(x)a$ because the b may be a result of a computation involving the value read by $R_2(x)a$. The two writes are causally related, so all processes must see them in the same order.

Example

$W(x)a$ not causally related to $W(x)b$

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

Eventual Consistency

- The form of consistency is called **eventual consistency** if no updates take place for a long time, all replicas will gradually become consistent.
 - Updates are guaranteed to propagate to all replicas sooner or later.
 - Write-write conflicts are often relatively easy to solve when assuming that only a small group of processes can perform updates.
 - Clients always access the same replica.

Difference

- Strong Consistency
 - Reading is mutual exclusive, and the a copy of the object provided by the process reflects all changes.
- Eventual Consistency
 - The data stores we consider are characterized by the lack of simultaneous updates, or when such updates happen, they can easily be resolved.
 - Most operations involve reading data.

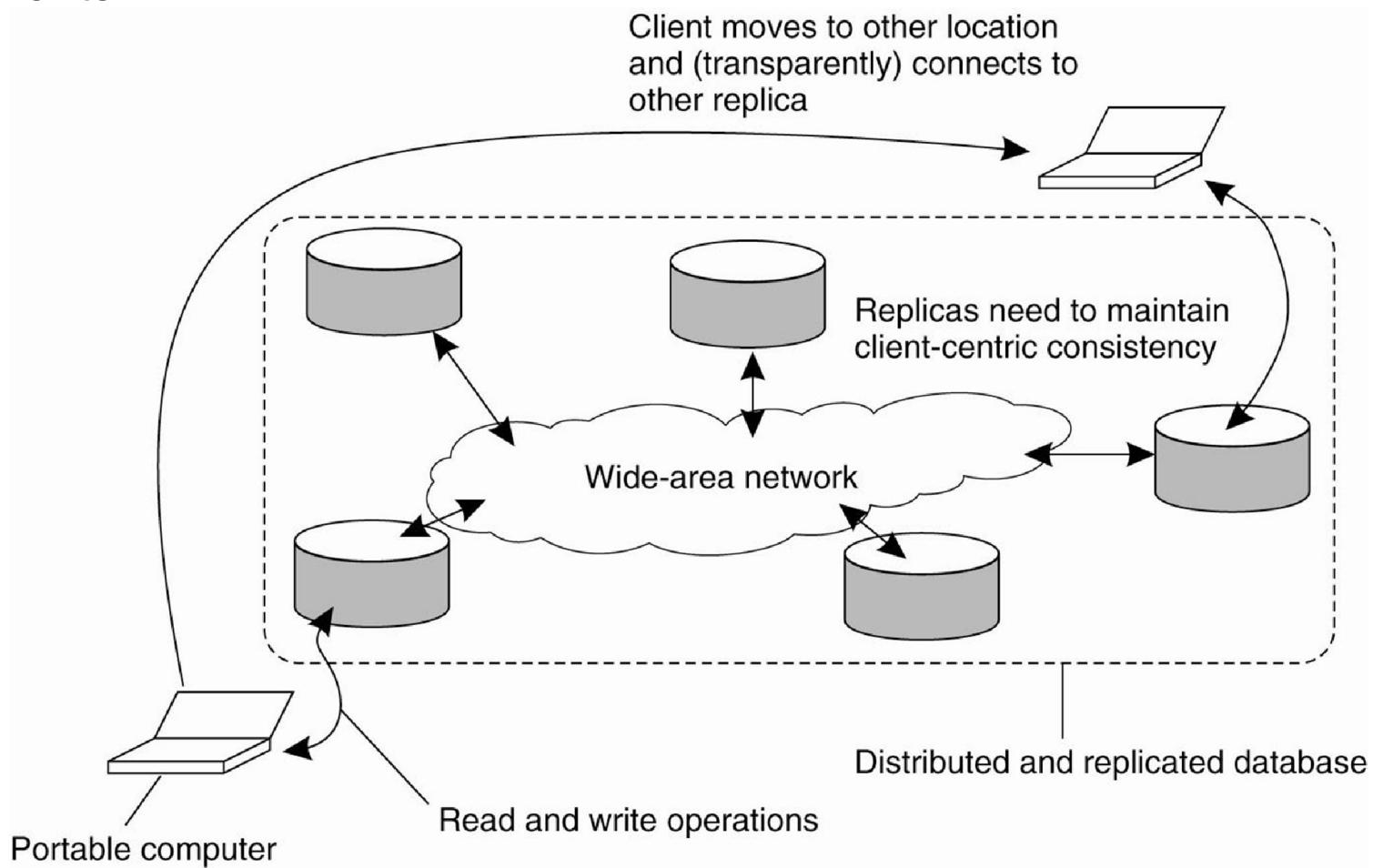
Example: DNS Name Space

- Write-write conflicts: conflicts resulting from two operations that both want to perform an update on the same data, never occur.
- Read-write conflicts: one process wants to update a data item while another is concurrently attempting to read that item.
- Lazy fashion: a reading process will see an update only after some time has passed since the update took place.

Example: World Wide Web

- Web pages are updated by a single authority, such as a webmaster or the actual owner of the page. There are normally no write-write conflicts to resolve.
- On the other hand, to improve efficiency, browsers and Web proxies are often configured to keep a fetched page in a local cache and to return that page upon the next request.

- The principle of a mobile user accessing **different replicas** of a distributed database.
- **Client-centric consistency** provides guarantees for a single client concerning the consistency of accesses to a data store by that client.
- No guarantees are given concerning concurrent accesses by different clients.



Others

一致性	描述
严格一致性	所有共享访问事件必须按绝对时间严格排序
线性一致性	所有进程看到的共享访问必须是同一顺序。访问则是按照全局时间戳排序。
FIFO	各进程按自己的队列方式对写操作排序，这个顺序对所有进程可见，但来自不同进程的对共享写操作的排序有可能不一样。

(a)

一致性	描述
弱一致性	至少执行一次同步后，共享数据才被认为是一致的。
释放一致性	退出临界区后，执行共享数据一致性操作
入口一致性	进入临界区时，执行属于该临界区的共享数据的一致性操作。

(b)

(a) 不使用同步操作的一致性模型 (b) 使用同步操作的一致性模型

Outline

- Data-centric Consistency Models
- Client-centric Consistency Models
 - Monotonic Reads
 - Monotonic Writes
 - Read Your Writes
 - Writes Follow Reads
- Replica Management
- Consistency Protocols

Notation

- $x_i[t]$: value of object x at store L_i at time t
- $WS(x_i[t])$: set of writes to object x at store L_i at time t
- $WS(x_i[t_1]; x_j[t_2])$ = writes to object x at store L_i at time t_1 have been applied at store L_j at time t_2
- $R(x_i[t])$ = read object x at store L_i at time t
- In slides, $[t]$ is abbreviated, and In slides, i is a replica ID.

Monotonic Read Consistency

If a process reads the value of a data item x , any successive read operation on x by that process will always return that same value or a more recent value.

- Monotonic-read consistency guarantees that if a process has seen a value of x at time t , it will never see an older version of x at a later time.

Example: Distributed Email System

- Each user's mailbox may be distributed and replicated across multiple machines
- Updates are propagated in a lazy
- When you read the new email in a city, you can read it again in any other cities.

L1: $WS(x_1)$

$R(x_1)$

L2:

$WS(x_1; x_2)$

$R(x_2)$

(a)

All operations in $WS(x_1)$ should have been propagated to L₂ before the second read operation takes place

Operations of $WS(x_1)$ performed at L2 before those of $WS(x_2)$

L1: $WS(x_1)$

$R(x_1)$

L2:

$WS(x_2)$

$R(x_2)$

(b)

Write operations in $WS(x_2)$ have been performed at L₂. No guarantees are given that this set also contains all operations contained in $WS(x_1)$

Operations of $WS(x_1)$ not performed at L2 before those of $WS(x_2)$

Monotonic Writes Consistency

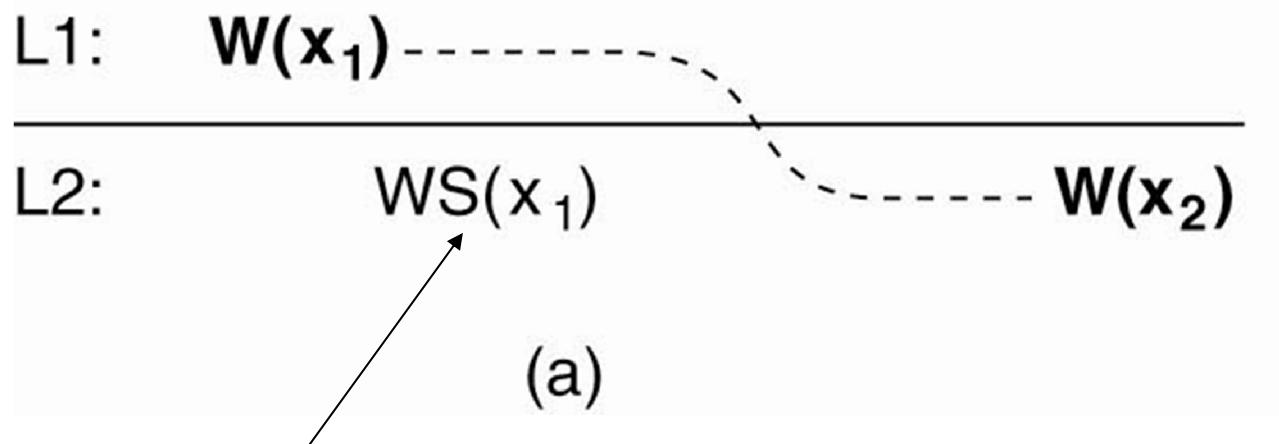
A write operation by a process on a data item x is completed before any successive write operation on x by the same process.

- Completing a write operation means that the copy on which a successive operation is performed reflects the effect of a previous write operation by the same process, no matter where that operation was performed.
- A write operation on a copy of item x is performed only if that copy has been brought up to date by means of any preceding write operation, which may have taken place on other copies of x .

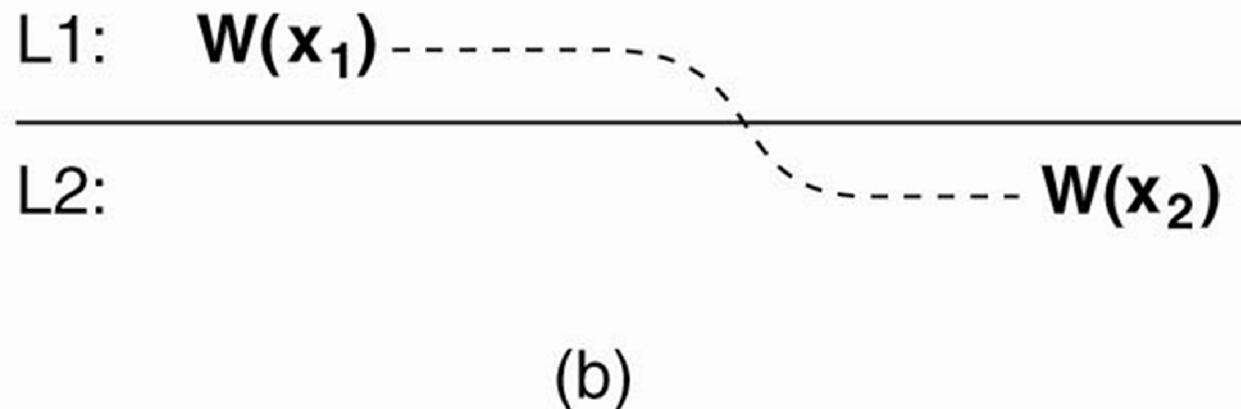
Example: Distributed Library

(version control system)

- Updating such a library is done by replacing one or more functions, leading to a next version.
- With monotonic-write consistency, guarantees are given that if an update is performed on a copy of the library, all preceding updates will be performed first.
- The resulting library will then indeed become the most recent version and will include all updates that have led to previous versions of the library.



Operations of $WS(x_1)$ performed at L2 before write of x at L2, $W(x_2)$



Write of x at L1, $W(x_1)$, not performed at L2 before write of x at L2

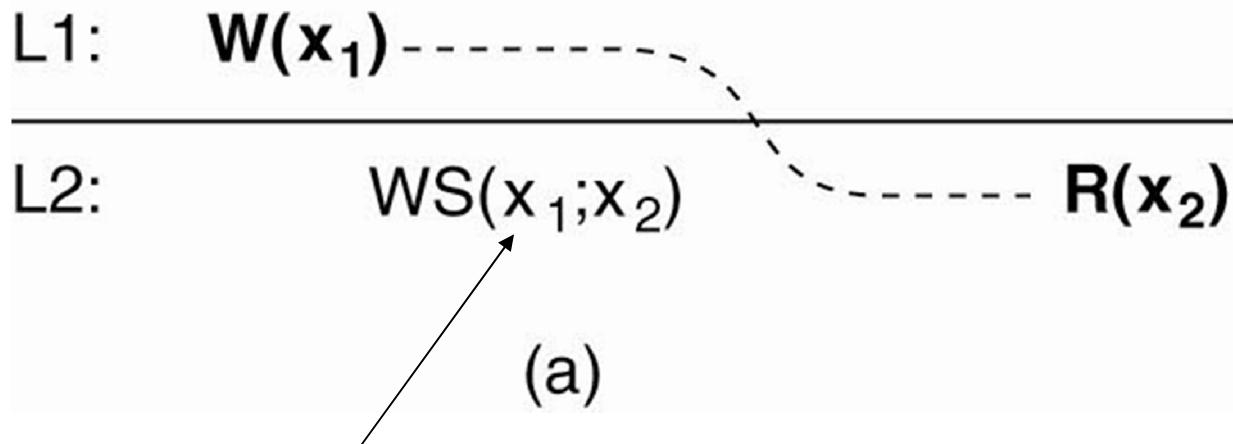
Read-your-writes Consistency

The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

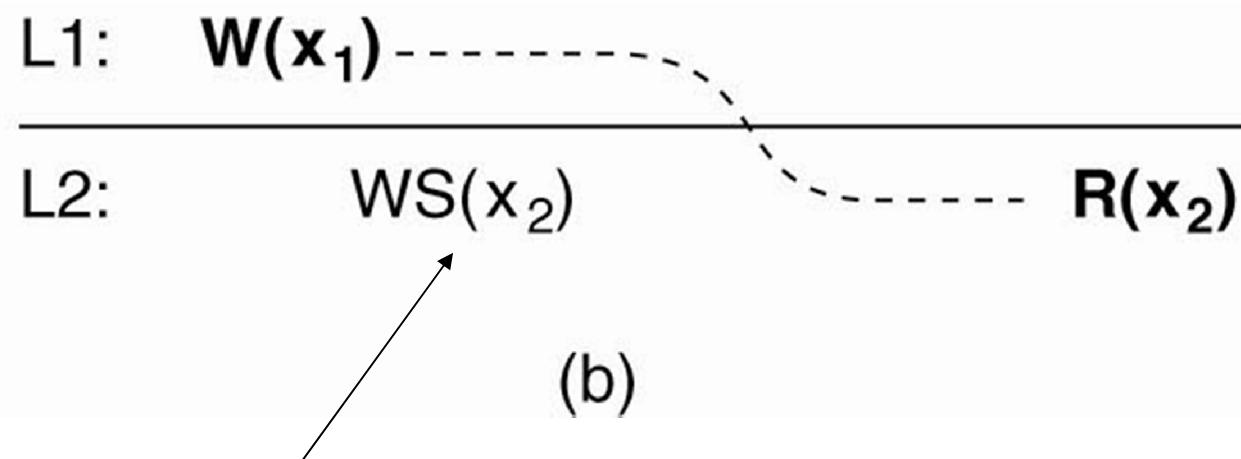
- A write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

Example: IM System

- To enter a IM on the Web, it is often necessary to have an account with a password.
- Changing a password make take no time to come into effect if you re-login again.



Operations of $WS(x_1)$ performed at L2 before those of $WS(x_2)$,
and subsequent read at L2, $R(x_2)$



Operations of $WS(x_1)$ not performed at L2

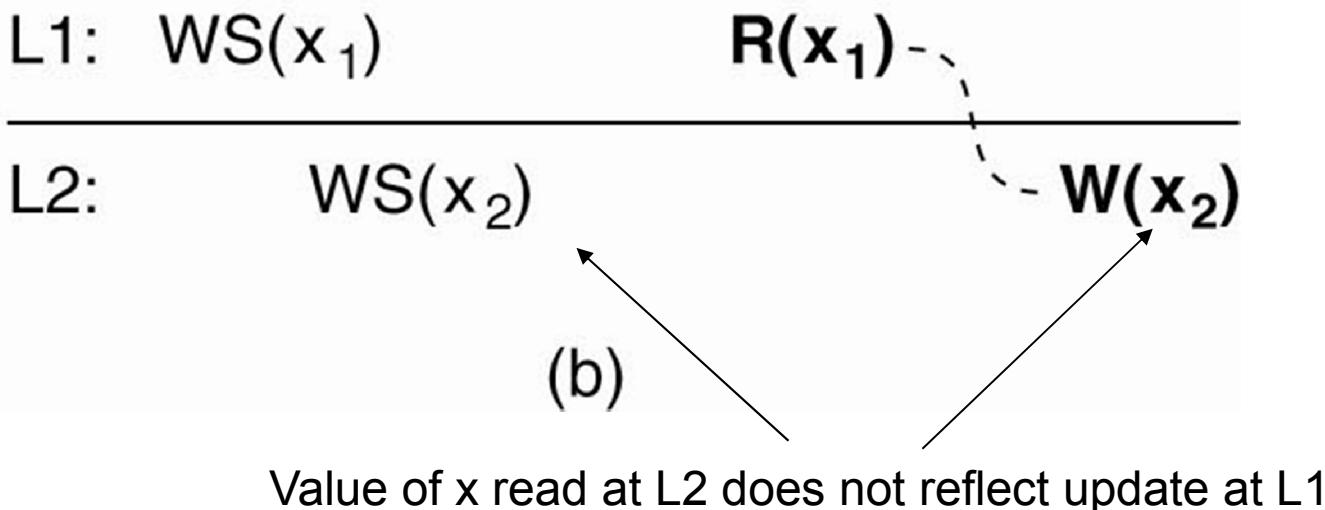
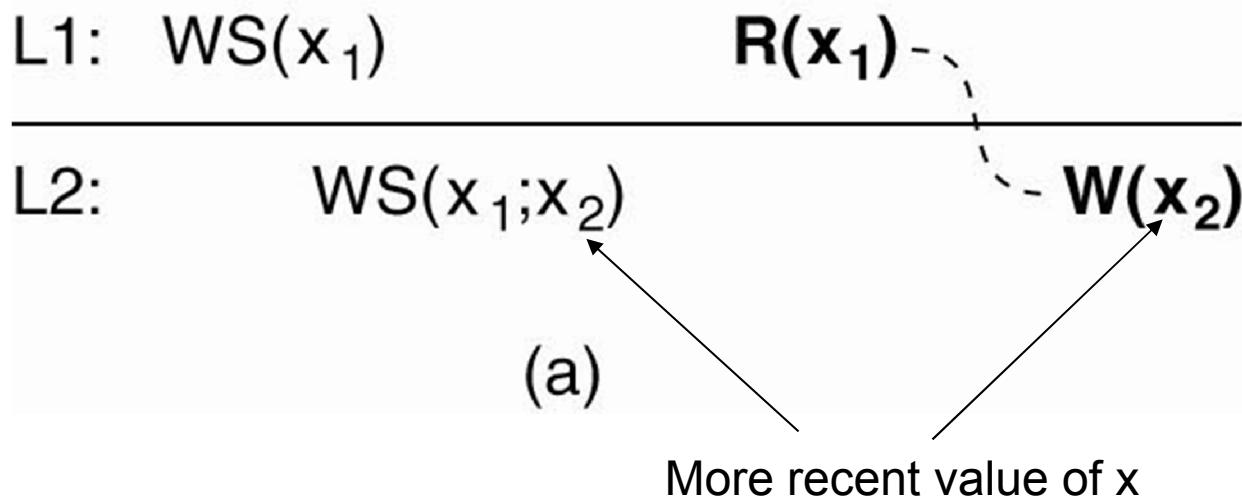
Writes-follow-reads Consistency

A write operation by a process on a data item x following a previous read operation on x by the same process **is guaranteed to take place on the same or a more recent value of x that was read.**

- Any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process.

Example: BBS on Web

- Assume that a user first reads an article A. Then, he reacts by posting a response B.
 - Note that users who only read articles need not require any specific client-centric consistency model.
- By requiring writes-follow-reads consistency, B will be written to any replica only after A has been written as well.
- The writes-follow-reads consistency assures that reactions to articles are stored at a local copy only if the original is stored there as well.



Outline

- Data-centric Consistency Models
- Client-centric Consistency Models
- **Replica Management**
- Consistency Protocols

Placement

- A key issue for any distributed system that supports replication is to decide **where**, **when**, and by **whom** replicas should be placed, and subsequently **which** mechanisms to use for keeping the replicas consistent.
 - Placing replica servers
 - Placing content
 - Distributing content (update)

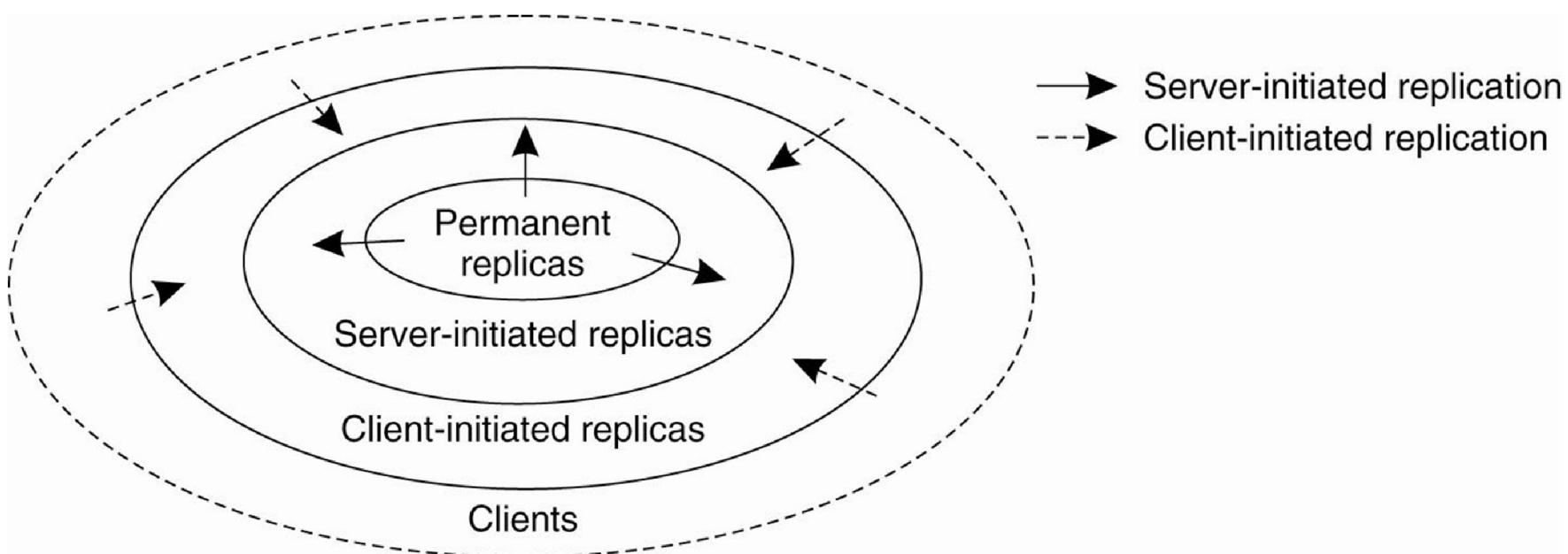
Replica-Server Placement

- Replica-server placement is concerned with finding the best locations to place a server that can host (part of) a data store.
- The best K out of N locations need to be selected ($K < N$).
- There are various ways to compute the best placement of replica servers
 - $O(N^2)$
 - $O(N * \max(\log N, K))$

Content Placement

- Permanent replicas can be considered as the initial set of replicas that constitute a distributed data store.
- Server-initiated replicas are copies of a data store that exist to enhance performance and which are created at the initiative of the (owner of the) data store.
- Client-initiated replicas are ones initiated by a client, they are more commonly known as (client) caches.

Content Placement



Content Distribution (1)

- An important design issue concerns what is actually to be propagated, state or operations?
 - Propagate only an **update notification** (invalidation protocols), use little network bandwidth
 - Propagate **update data** from one **copy** to another, and is useful when the read-to-write ratio is relatively high
 - Propagate the **update operation** to other copies. also referred to as active replication.

Content Distribution (2)

- Pull Protocol versus Push Protocol

Issue	Push-based	Pull-based
State at server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

- Unicasting Protocol or Multicasting Protocol

Outline

- Data-centric Consistency Models
- Client-centric Consistency Models
- Replica Management
- **Consistency Protocols**

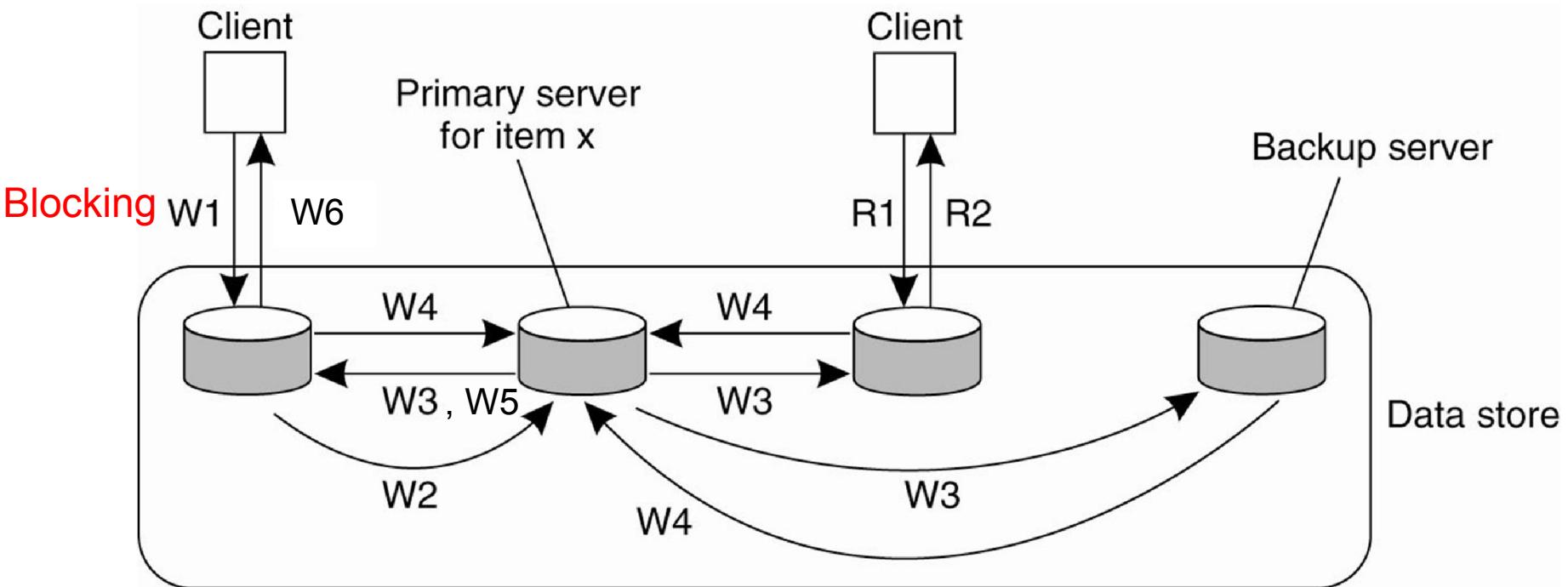
Consistency Protocol

- A consistency protocol describes an implementation of a specific consistency model.
- Protocols for data-centric consistency
 - Primary-Based Protocols (data based)
 - Remote-Write
 - Local-Write
 - Replicated-Write Protocols (operations based)
 - Active Replication
 - Quorum-based
- Protocols for client-centric consistency

Primary-Based Protocols

- Each data item x in the data store has an associated primary, which is responsible for coordinating write operations on x .
- Remote-Write: the primary is fixed at a remote server
- Local-Write: write operations can be carried out locally after moving the primary to the process where the write operation is initiated.

Remote-Write (Primary Backup Protocol 1993)



W1. Write request

W2. Forward request to primary

W3. Tell backups to update

W4. Acknowledge update

W5. Acknowledge write completed

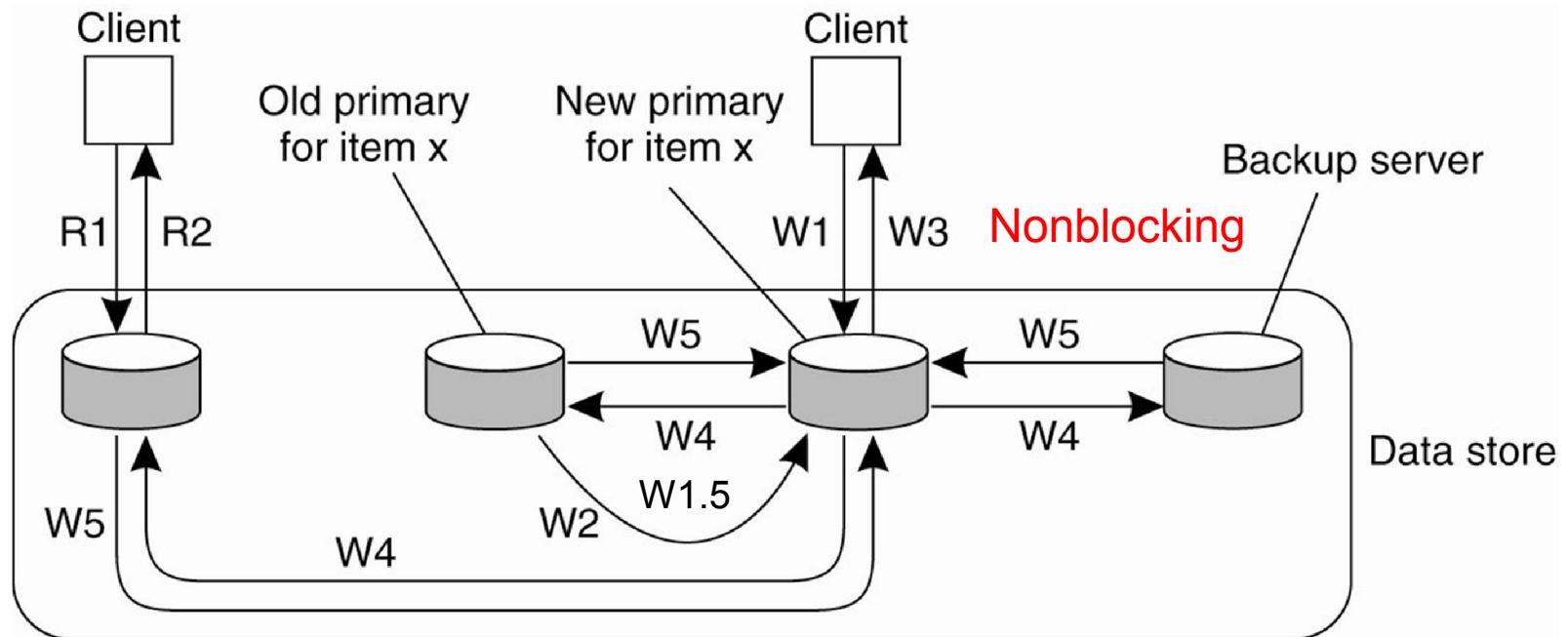
R1. Read request

R2. Response to read

W6 – tell client write completed (opt)

It is simple but may take a relatively long time

Local-Write



W1. Write request W1.5: Tell primary
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read
W4 – often notification, not update
So R1.1 update.req, R1.2 update.cnf

- Primary copy migrates between processes that wish to perform a write operation.
- Multiple, successive write operations can be carried out locally, while reading processes can still access their local copy.

Replicated-Write Protocols

- Write operations can be carried out at multiple replicas instead of only one, as in the case of primary-based replicas.
- Active Replication
 - an operation is forwarded to all replicas
- Quorum-based Protocols
 - an operation is forwarded to majority replicas

Active Replication

- Each replica has an associated process that carries out update operations.
- Operations need to be carried out in the same order everywhere.
 - A totally-ordered multicast mechanism
 - Lamport logical clocks
 - A central coordinator

Quorum-based

- Require clients to request and acquire the permission of multiple servers before either reading or writing a replicated data item.

Example: To Update

- A client must first contact at least half the servers plus one (a majority) and get them to agree to do the update.
- Once they have agreed, the file is changed and a new version number is associated with the new file.

Example: To Read

- A client must also contact at least half the servers plus one and ask them to send the version numbers associated with the file.
- If all the version numbers are the same, this must be the most recent version
 - An attempt to update only the remaining servers would fail because there are not enough of them.

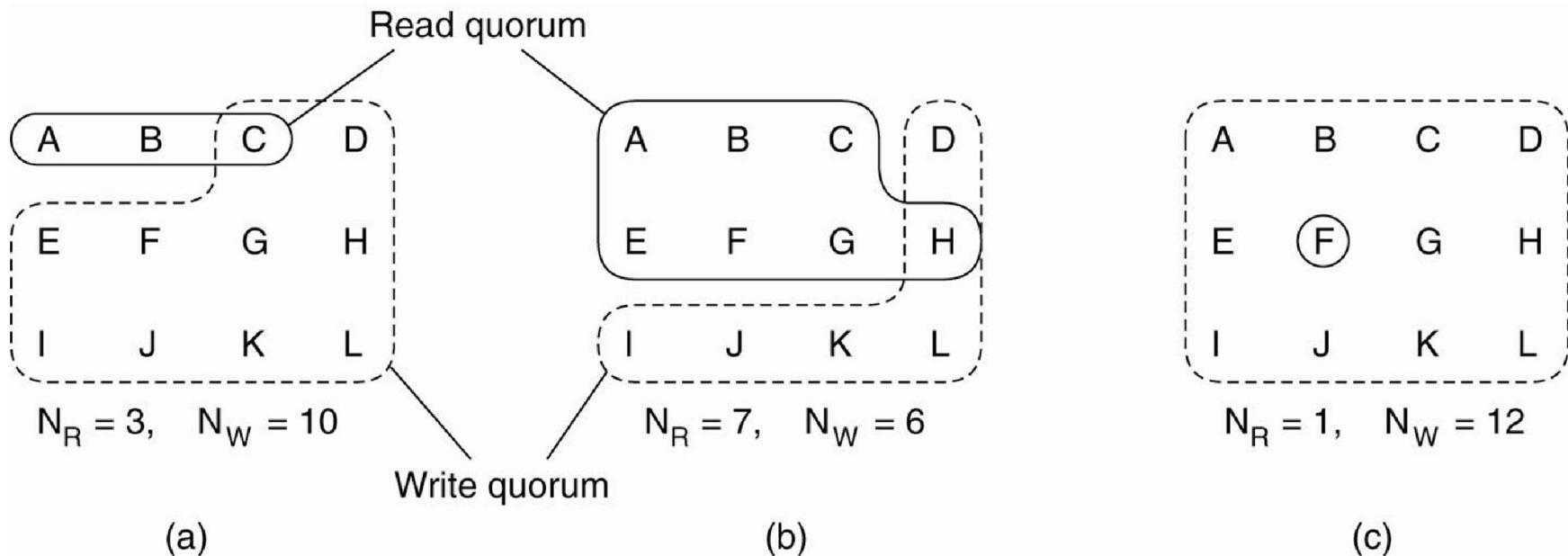
Example

- If there are 5 servers and a client determines that 3 of them have version 8, it is impossible that the other two have version 9.
- After all, any successful update from version 8 to version 9 requires getting 3 servers to agree to it, not just 2.

Quorum-based (sophisticated)

- To read a file of which N replicas exist, a client needs to assemble a read quorum, an arbitrary collection of any N_R servers, or more.
- To modify a file, a write quorum of at least N_W servers is required.
 - $N_W > N/2$ (prevent read-write conflicts)
 - $N_R + N_W > N$ (prevents write-write conflicts)
 - Must ensure any two write sets **intersect** (正交); and any read set **intersects** every write set.
- Any N_W copies will do for a write, and any N_R copies for a read
- May have witnesses, witness can alert querier that new version # exists

Quorum-based (sophisticated)



Next Lesson...

DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 8
Fault Tolerance

Outline

- Basic Concepts
- Process Resilience(恢复)
- Reliable Client-server Communication
- Reliable Group Communication
- Recovery
- Distributed Commit

Partial Failure

- An important goal in distributed systems design is
 - it can automatically recover from partial failures without seriously affecting the overall performance
 - or, whenever a failure occurs, the distributed system should continue to operate in an acceptable way while repairs are being made,
 - or, it should **tolerate faults** and continue to operate to some extent even in their presence.

Dependable Systems (1993)

- MARS
 - **Maintainability**: how easy a failed system can be repaired
 - **Availability**: a system is ready to be used immediately
 - **Reliability**: system can run continuously without failure
 - **Safety**: system temporarily fails to operate correctly, nothing catastrophic(灾难性的) happens.

Availability and Reliability

- The two are not the same.
 - A system goes down for one millisecond every hour, it has an availability of over 99.9999 percent, but is still highly unreliable.
 - A system that never crashes but is shut down for two weeks every August has high reliability but only 96 percent availability.

Terminology

- **Failure(Fail)**: system cannot meet its promises.
- **Error**: it is a part of a system's state that may lead to a failure.
- **Fault**: it is the cause of an error. **Fault -> Error -> Failure**
- **Fault Tolerance**: the system can tolerate faults and continue to operate normally
 - Does not **fail** in condition of occurring **faults**
- **Transient faults** occur once and then disappear. If the operation is repeated, the fault goes away.
- **Intermittent fault** occurs, then vanishes of its own accord, then reappears, and so on.
- **Permanent fault** is one that continues to exist until the faulty component is replaced.

Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure <i>Receive omission</i> <i>Send omission</i>	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages or can not (hang)
Timing failure <small>Performance failure</small>	A server's response lies outside the specified time interval
Response failure <i>Value failure</i> <i>State transition failure</i>	A server's response is incorrect The serious type of failure The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

The most serious type of failure

Byzantine Failure

- Byzantine_(拜占庭) Fault: any fault presenting different symptoms to different observers.
- Byzantine Failure: the loss of a system service due to a Byzantine fault;
- Byzantine Fault Tolerance: the characteristic of a system that tolerates the class of failures known as the Byzantine Generals' Problem

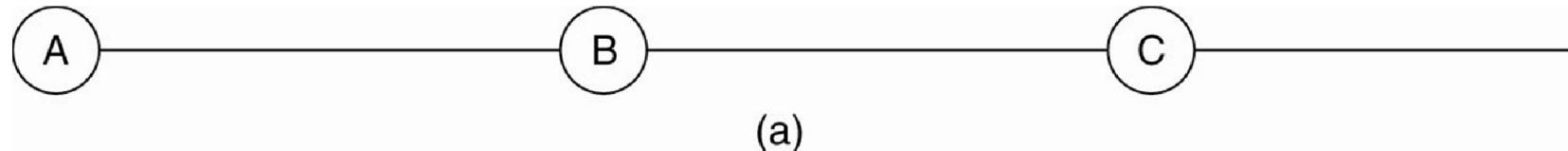
Failure Mode

- **Fail-stop** system: A server friendly announce it is about to crash, then crashes.
- **Fail-silent** systems: If other processes is needed to decide that a server has prematurely halted
 - The server may just be unexpectedly slow.
- **Fail-safe systems**: a fail server is producing random output, but this output can be recognized by other processes as garbage.

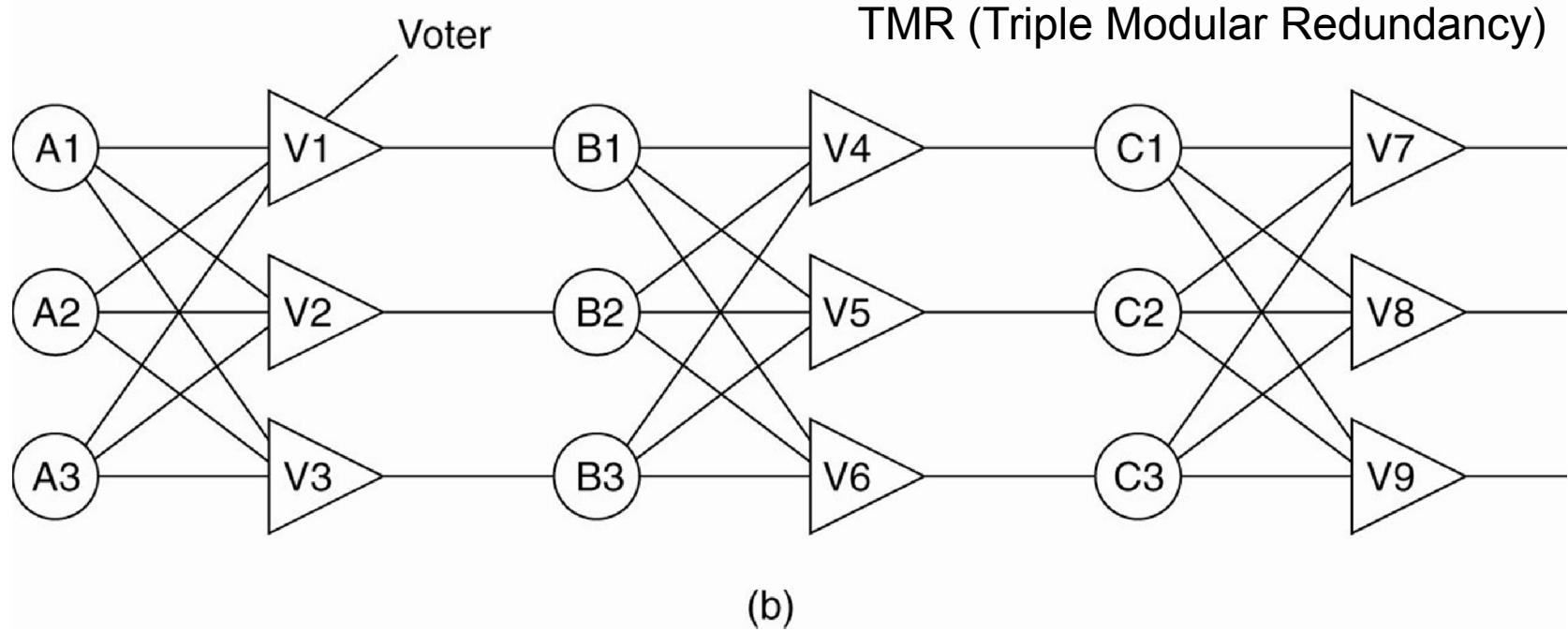
Redundancy

- If a system is to be fault tolerant, the best it can do is to try to **hide the occurrence of failures** from other processes.
 - **Information redundancy**, extra bits are added to allow recovery from garbled bits.
 - **Time redundancy**, an action is performed, and then, if need be, it is performed again.
 - **Physical redundancy**, extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components.

Failure Masking by Redundancy



(a)



(b)

Biology: mammals have two eyes, two ears, two lungs, etc
Aircraft: 747s have four engines but can fly on three
Sports: multiple referees in case one misses an event

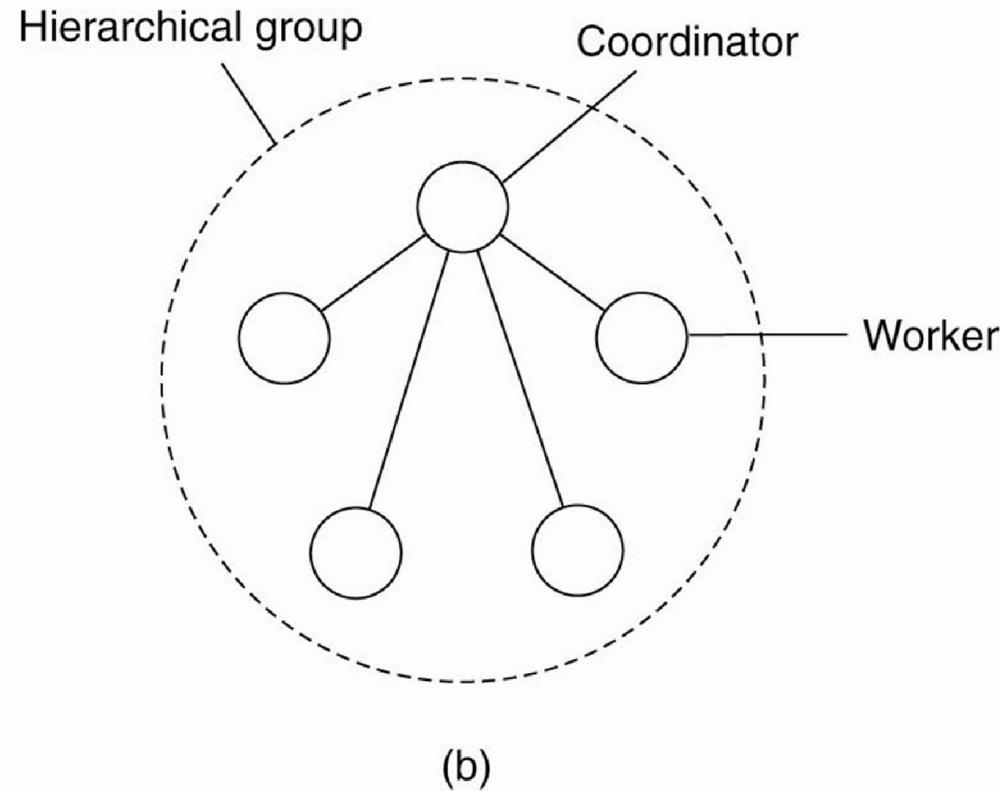
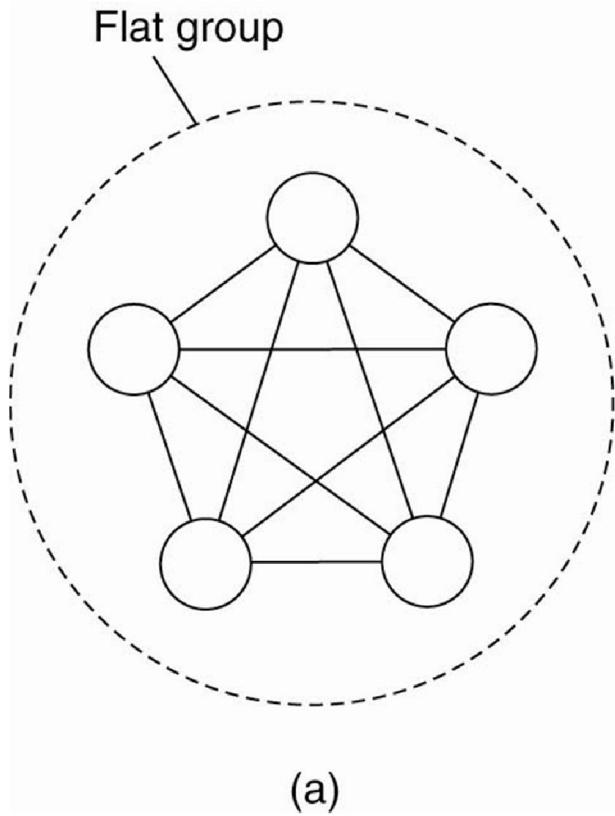
Outline

- Basic Concepts
- Process Resilience
 - Processes Group and k -Fault Tolerant
 - Distributed Agreement
 - Failure Detection
- Reliable Client-server Communication
- Reliable Group Communication
- Recovery
- Distributed Commit

Fault-tolerant of Processes

- Tolerating a faulty process is to organize several identical processes into a group.
 - When a message is sent to the group, all members of the group receive it.
 - One process in a group fails, some other process can take over.

Flat Groups versus Hierarchical Groups



Group Membership

- Creating and deleting groups
- Join and leave groups
 - Group server
 - Distributed management
- Three issues
 - To leave a group, a member just sends a goodbye message to everyone.
 - Leaving and joining have to be synchronous with data messages being sent
 - Many machines go down that the group can no longer function at all.

k-Fault Tolerant

- A system is said to be k fault tolerant if it can survive faults in k components and still meet its specifications
 - If processes fail silently, then having $k + 1$ of them is enough to provide k fault tolerance.
- If processes exhibit Byzantine failures, minimum of $2k + 1$ processors are needed to achieve k fault tolerance.
 - k failing processes could accidentally (or even intentionally) generate the same reply

Distributed Agreement (协定)

The general goal of distributed agreement algorithms is to have all the non-faulty processes reach consensus_(一致) on some issue, and to establish that consensus within a finite number of steps.

- Synchronous versus asynchronous systems.
- Communication delay is bounded or not.
- Message delivery is ordered (reliable) or not.
- Message transmission is done through unicasting or multicasting.

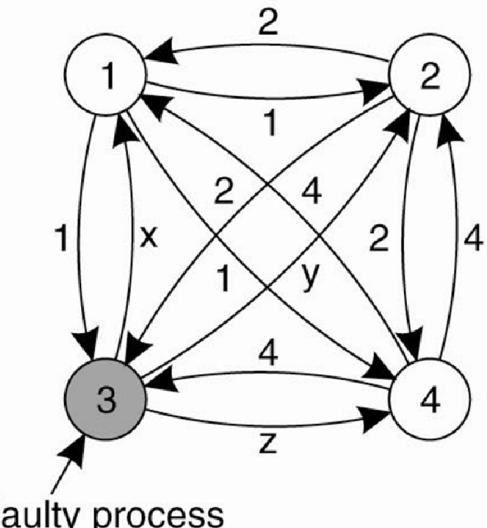
Circumstances under which distributed agreement can be reached.

		Message Order						
		Unordered		Ordered		Communication		
Processors	Asynchronous	No	No	Yes	No	Unbounded		
		No	No	Yes	No	Bounded		
Processors	Synchronous	Yes	Yes	Yes	Yes	Unbounded		
		No	No	Yes	Yes			
Point-to-Point		Broadcast			Point-to-Point			
Transmission								

Figure 8-4. in the book is wrong

Byzantine Agreement Problem (1982)

- Each General is either loyal or a traitor to the Byzantine state.
- All Generals communicate by sending and receiving messages.
- There are only two commands: attack and retreat.
- All loyal Generals should agree on the same plan of action: attack or retreat.
- A small linear fraction of bad Generals should not cause the protocol to fail (less than a 1/3 fraction).



Faulty process

1. each process sends their value to the others

1 Got(1, 2, x, 4)

2 Got(1, 2, y, 4)

3 Got(1, 2, 3, 4)

4 Got(1, 2, z, 4)

Each process constructs a vector V of length N , such that if process i is non-faulty, $V[i] = V_i$. Otherwise, $V[i]$ is undefined. There are at most k faulty processes.

$N=4$ $k=1$

assume: processes are synchronous
messages are unicast while preserving ordering. communication delay is bounded.

1 Got

From 2 (1, 2, y, 4)

From 3 (a, b, c, d)

From 4 (1, 2, z, 4)

2 Got

(1, 2, x, 4)

(e, f, g, h)

(1, 2, z, 4)

4 Got

(1, 2, x, 4) From 1

(1, 2, y, 4) From 2

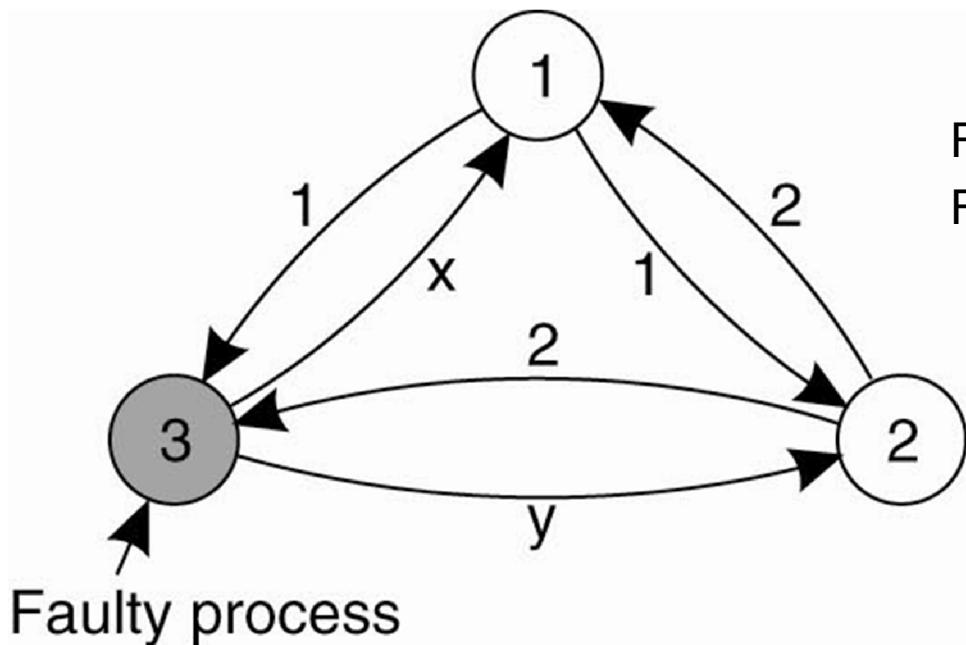
(i, j, k, l) From 3

2. the vectors that each process assembles

3. the vectors that each process receives

4. each process examines the i -th element of each of the newly received vectors. If any value has a majority, that value is put into the result vector. If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

$N=3 \ k=1$



From 2	$\frac{1 \text{ Got}}{(1, 2, y)}$	From 1
From 3	(a, b, c)	From 3

1 Sees:
From 1: (1,2,x)
From 2: (1,2,y)
From 3: (1,b,x)

So which process should 1 suspect?

Suppose $a=1$ and $c=x$...

Majority: (1,2,x) and ?? is bad

Lamport's Solution

- In a system with k faulty processes
 - Agreement can be achieved only if $2k + 1$ correctly functioning processes are present, for a total of $3k + 1$.
- Agreement is possible only if more than two-thirds of the processes are working properly.

- **k -Fault Tolerant of a process group**
 - If processes exhibit Byzantine failures, minimum of $2k + 1$ processors are needed to achieve k fault tolerance.
- **Distributed Agreement of several process**
 - Agreement can be achieved only if $2k + 1$ correctly functioning processes are present, for a total of $3k + 1$.

Failure Detection

- A timeout mechanism is used to check whether a process has failed, In practice, actively pinging processes is usually followed.
 - Simply stating that “**a process has failed because it does not return an answer to a ping message**” may be wrong.
 - There is **few work** take more into account than only the lack of a reply to a single message.
- Regularly exchanging information with neighbors, process with an old information may be failure.

Failure Detection

- Failure detection should ideally be able to distinguish network failures from node failures?
 - Neighbors
- When a member failure is detected, how should other non-faulty processes be informed?
 - Multicasting

Outline

- Basic Concepts
- Process Resilience
- Reliable Client-server Communication
- Reliable Group Communication
- Recovery
- Distributed Commit

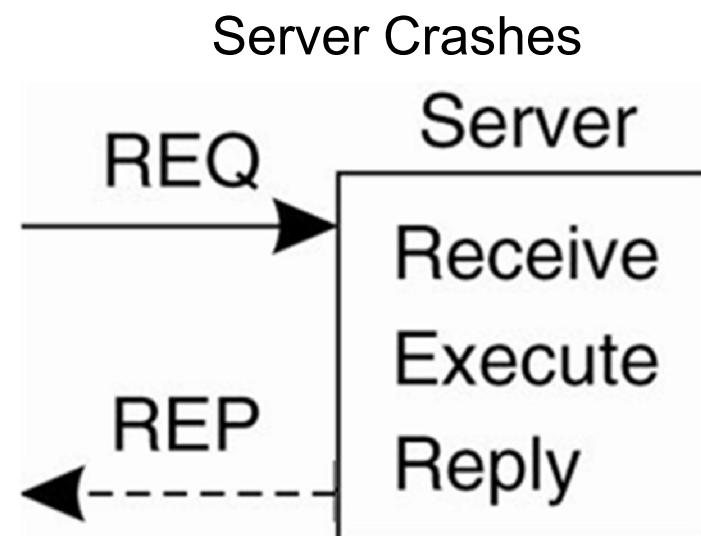
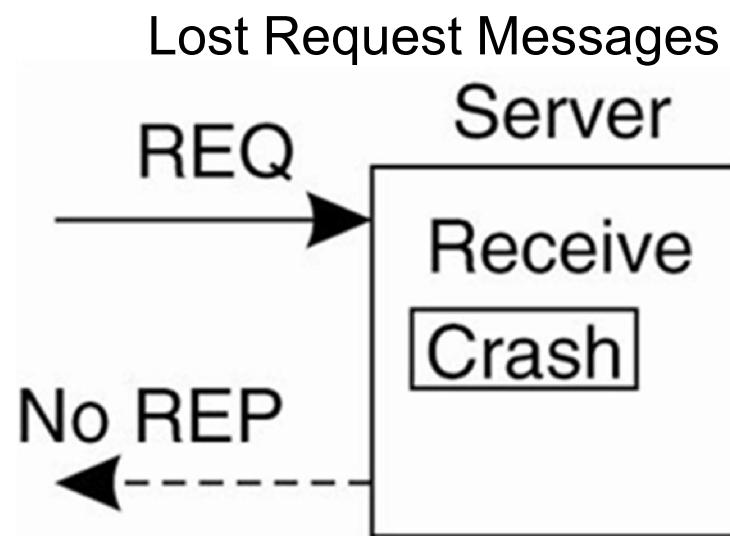
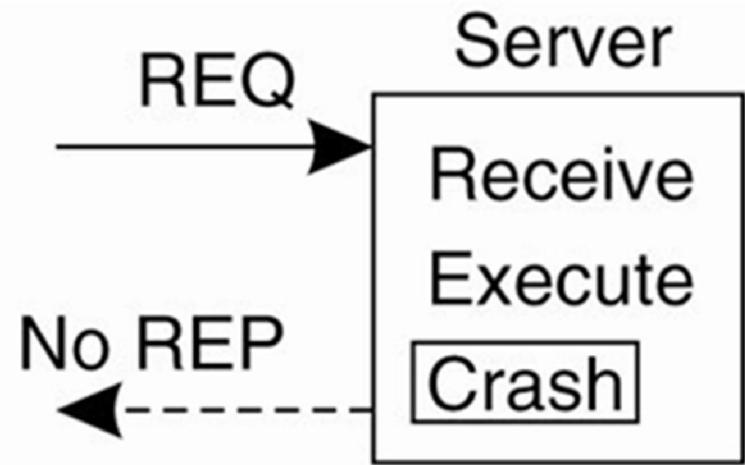
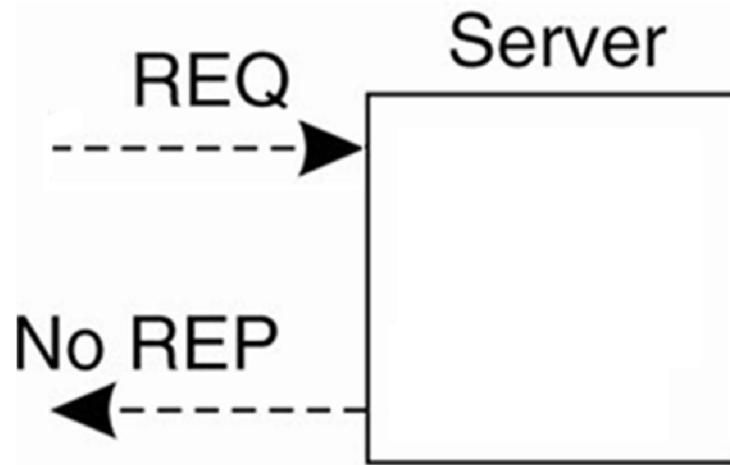
RPC Semantics in the Presence of Failures

- The client is unable to locate the server.
- The request message from the client to the server is lost.
- The server crashes after receiving a request.
- The reply message from the server to the client is lost.
 - The former three are same to the client
- The client crashes after sending a request.

Client Cannot Locate the Server

- Reason:
 - Servers might be down
 - Client stub might be out of date
- Solution:
 - Raise an exception in the client
 - Not every language has exceptions or signals
 - To write an exception handler destroys the transparency we have been trying to achieve

Three Failures



Server Crashes

Lost Reply Messages

Three Failures

- Client-side solution
 - Reissue the request
 - Let the request be idempotent (幂等的)
 - Assign each request a sequence number
- Server-side solution
 - Omit retransmission (for reply is lost)
 - Reply before/after execution (for server crashes)
 - Other solutions (for server crashes)
 - Process Resilience (not include in this section)
 - Recovery and redo with logs (not include in this section)

Reissue the request

- Client stub start a timer. If the timer expires before a reply comes back, the request is sent again.
- If the request was truly lost, everything will work fine.
- If many request are lost or server is crash. the client gives up and concludes that the server is down. "Cannot locate server"
- If the reply was lost.
 - Idempotent
 - Dealing with a reissued request at server side.

Sequence Number and Idempotent

- Client assign each request a sequence number
 - The sever logs all executed requests with their sequence number
- The request should be idempotent (幂等的)
 - Try to structure all the requests in an idempotent way.
 - Insert a new records (non-idempotent) → try to delete the record before insert it (idempotent)
 - CURD of database operation
 - Create, Update, Read, Delete

Semantics

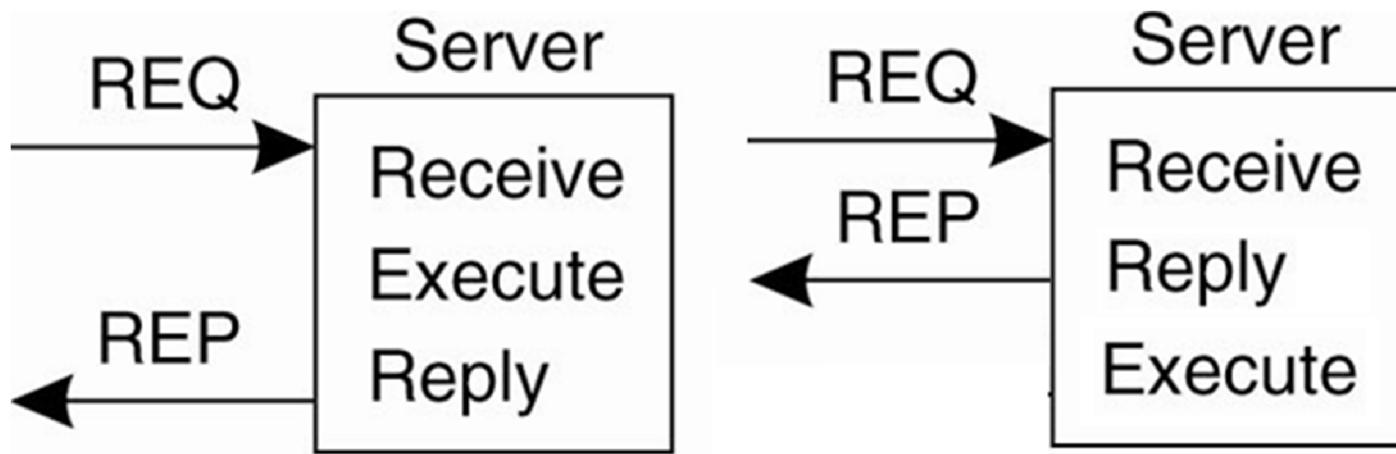
- At-least-once semantics: to keep trying until a reply has been received.
- At-most-once semantics: gives up immediately and reports failure back.
- Nothing semantics: a server crashes, the client gets no help and no promises about what happened.
- Exactly-once semantics: there is no way to arrange this.

Reissue Strategies

- The client can decide to **never reissue a request**, at the risk that the request will not be executed.
- The client decide to **always reissue a request**, but this may lead to its request being executed twice or even more.
 - The client decide to **reissue a request only if it did not yet receive an acknowledgment that its request had been delivered to the server**
 - The client decide to **reissue a request only if it has received an acknowledgment for the print request**.

Server-side Solution

- Sending the completion message to client before/after execution



- Sending the completion message to client before/after execution

Server Crashes: Example (1)

- Three events that can happen at the server:
 - Send the completion message (M),
 - Print the text (P),
 - Crash (C).

Server Crashes: Example (2)

- $M \rightarrow P \rightarrow C$: A crash occurs after sending the completion message and printing the text.
- $M \rightarrow C (\rightarrow P)$: A crash happens after sending the completion message, but before the text could be printed.
- $P \rightarrow M \rightarrow C$: A crash occurs after sending the completion message and printing the text.
- $P \rightarrow C (\rightarrow M)$: The text printed, after which a crash occurs before the completion message could be sent.
- $C (\rightarrow P \rightarrow M)$: A crash happens before the server could do anything.
- $C (\rightarrow M \rightarrow P)$: A crash happens before the server could do anything.

Server Crashes: Example (3)

Client Reissue strategy	Strategy M → P			Strategy P → M		
	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

Assuming the server is crash, restart, and recovery without any log information.

Client Crashes

- **Orphan:** unwanted computation which is active and no one is waiting for the result.
- Orphans can cause a variety of problems
 - Waste CPU cycles
 - Lock files
 - Tie up valuable resources
 - Confusion can result if client reboots and does the RPC again, but the reply from the orphan comes back immediately

Solutions

- Extermination (消灭)
 - Client logs entry telling what the request is about
 - After client's reboot, the log is checked and the orphan is explicitly killed off
- Reincarnation (转世,再生)
 - When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch ([ɛpək] 纪元)
 - All remote computations on behalf of that client are killed
- Expiration (过期)
 - If it cannot finish in T time, it must explicitly ask for another quantum.
 - If after a crash the client waits a time T before rebooting, all orphans are sure to be gone.

Outline

- Basic Concepts
- Process Resilience
- Reliable Client-server Communication
- Reliable Group Communication
- Recovery
- Distributed Commit

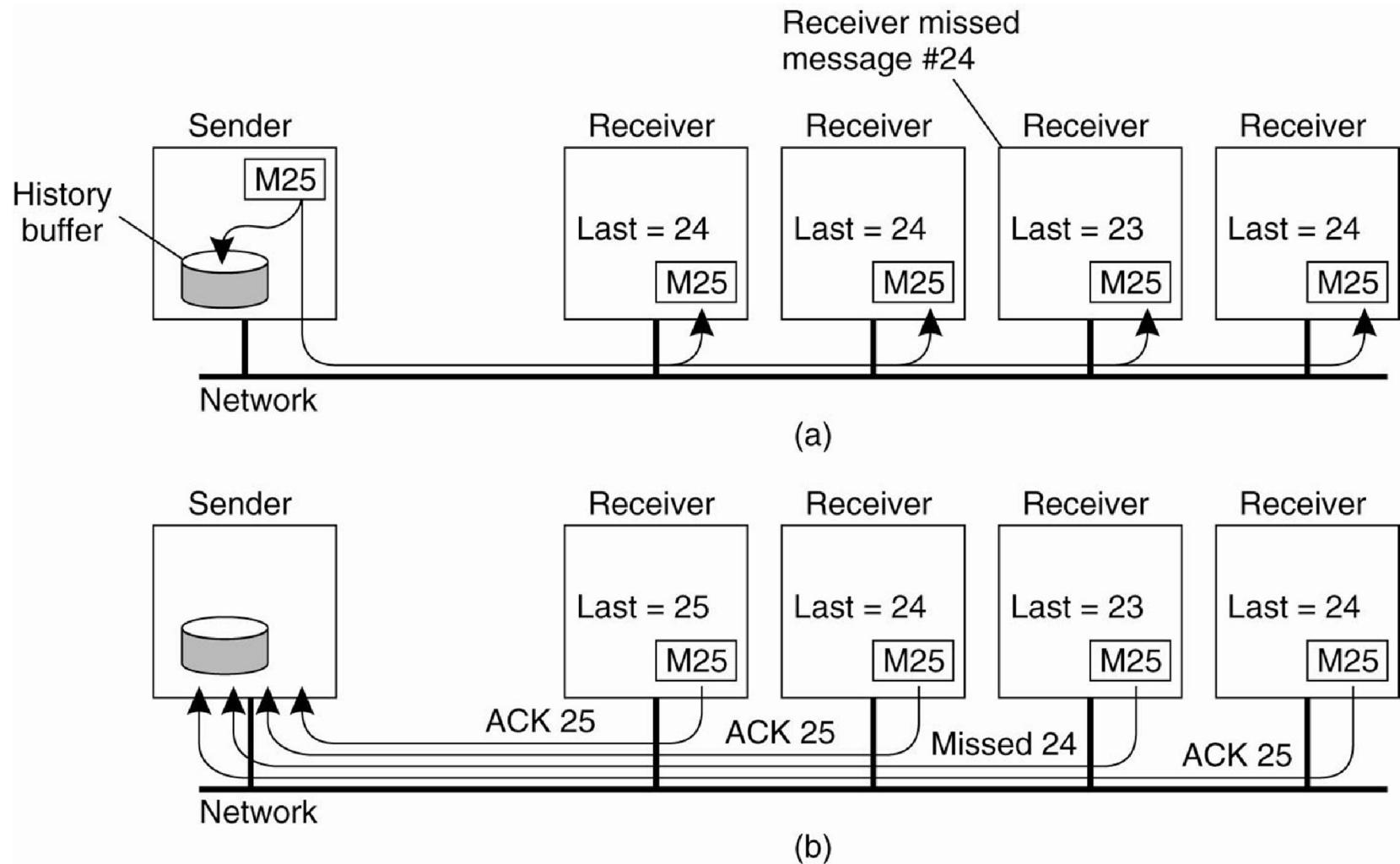
Reliable Group Communication

- Reliable multicasting turns out to be surprisingly tricky.
- Reliable group communication means that a message that is sent to a process group should be delivered to each member of that group.
 - What happens if during communication a process joins the group?
 - What happens if a (sending) process crashes during communication.

Simplified Solution with Assumptions

- Multiple reliable unicast to implements reliable multicast if:
 - Reliable communication when processes are assumed to operate reasonably;
 - Agreement exists on who is a member of the group and who is not, group member does not change;
 - No requirement that all group members receive messages in the same order.
 - sometime we need (atomic multicast)

Basic Reliable-Multicasting Schemes



The sending process assigns a sequence number to each message it multicasts. We assume that messages are received in the order they are sent. In this way, it is easy for a receiver to detect it is missing a message.

Problems

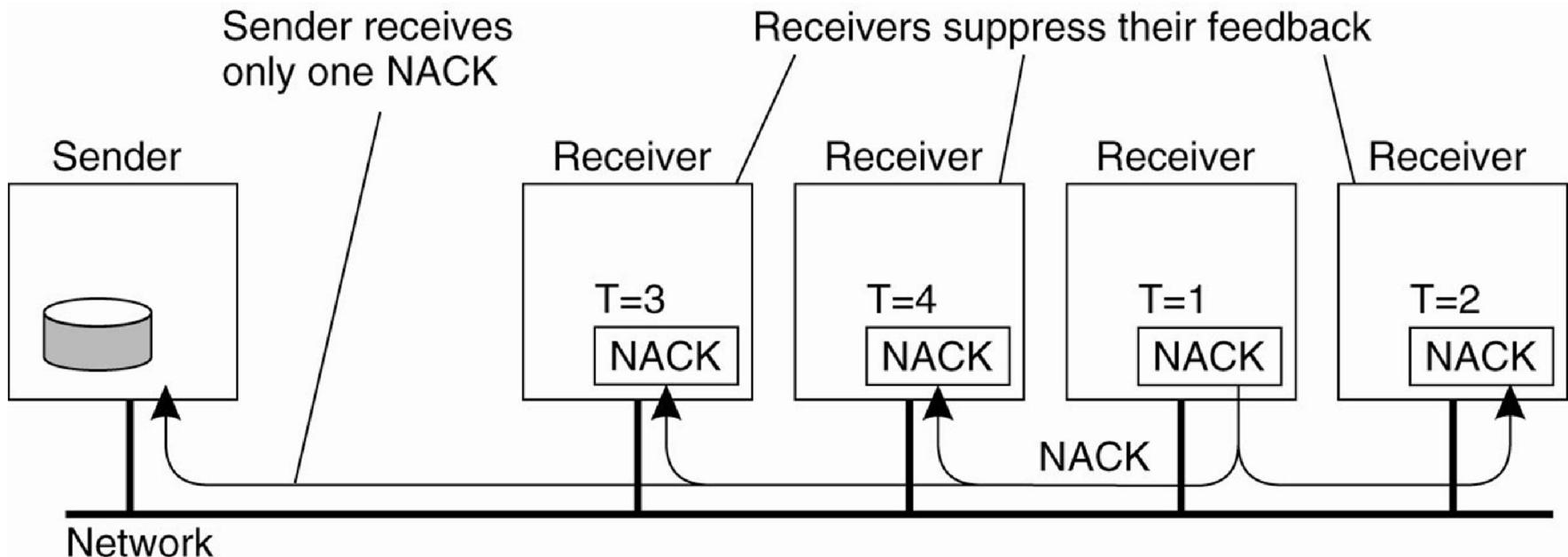
- ACK implosion
 - A receiver returns a feedback message only to inform the sender it is missing a message.
 - NACK but not ACK, thinking how a node know the message is missing?
- Buffer
 - Be forced to keep a message in its history buffer forever, do not know when it should be removed.
 - Sender has to remove a message from its history to prevent the buffer from overflowing

Feedback Suppression

(抑制)

- Receivers report only when they are missing a message;
- The receiver multicasts its feedback to the rest of the group;
- Other members, which get the feedback, suppress its own feedback.

Non-hierarchical Feedback Control



- A receiver R that did not receive message m schedules a feedback message with some random delay.
- If, in the meantime, another request for retransmission for m reaches R , R will suppress its own feedback, knowing that m will be retransmitted shortly.
- Only a single feedback message will reach S , which in turn subsequently retransmits m .

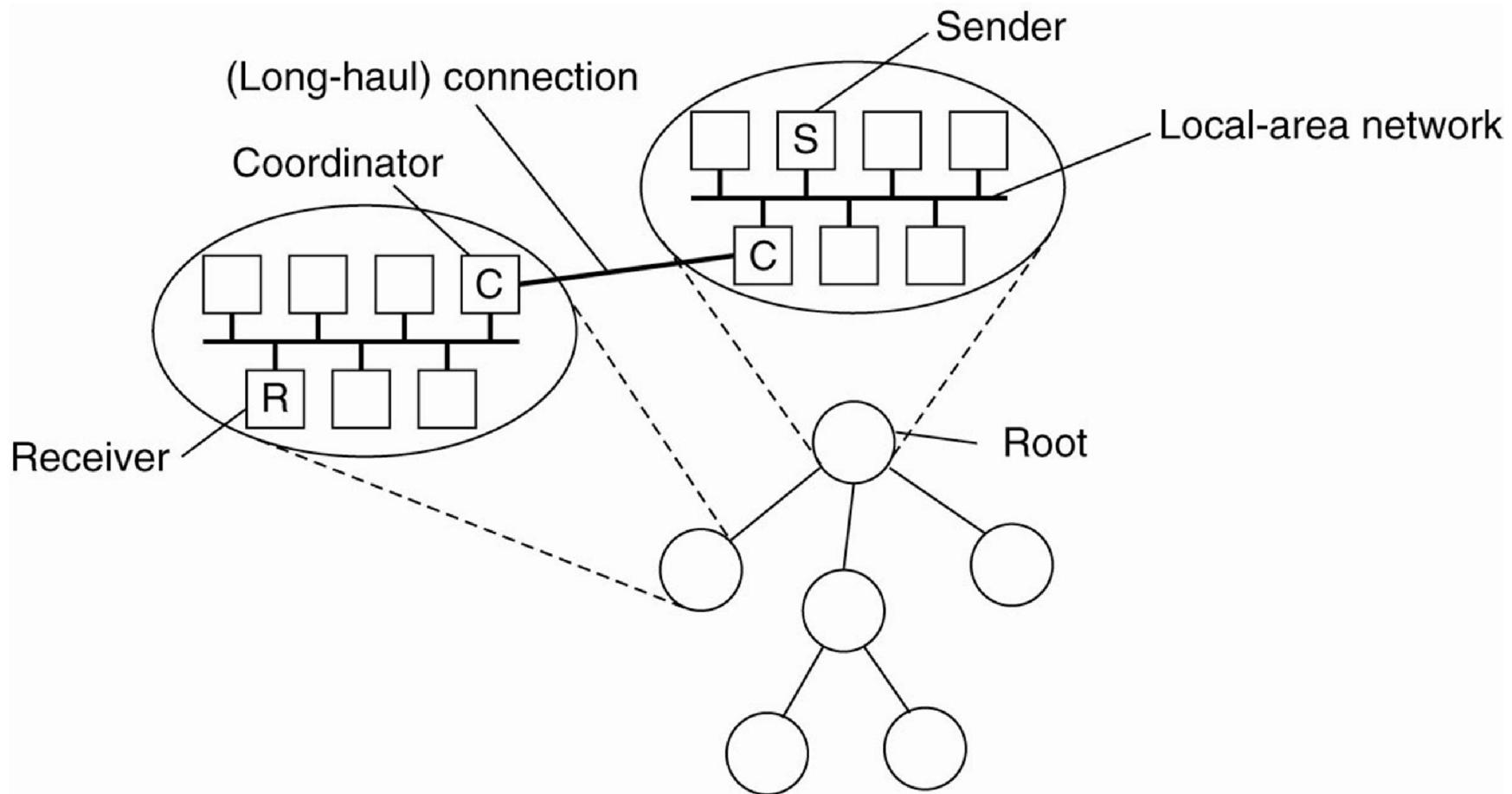
Problem

- Setting timers to all processes is not that easy.
- Multicasting feedback also interrupts those processes to which the message has been successfully delivered
 - Have multicast channel per NACK number....

Hierarchical Feedback Control

- A single sender that needs to multicast messages to a very large group of receivers.
 - The group of receivers is partitioned into a number of subgroups, which are subsequently organized into a tree.
- Each subgroup appoints a local coordinator which is responsible for handling retransmission requests of receivers contained in its subgroup. The local coordinator will thus have its own history buffer.
- If the coordinator itself has missed a message m , it asks the coordinator of the parent subgroup to retransmit m .
- The main problem with hierarchical solutions is the construction of the tree

Hierarchical Feedback Control



Atomic Multicast

- A distributed system is guaranteed:
 - A message is delivered to either all processes or to none at all in condition of the group members may be changed.
 - All messages are delivered in the same order to all processes.
- Example: protocols for data-centric consistency, replicated-write Protocols, such as active protocols.

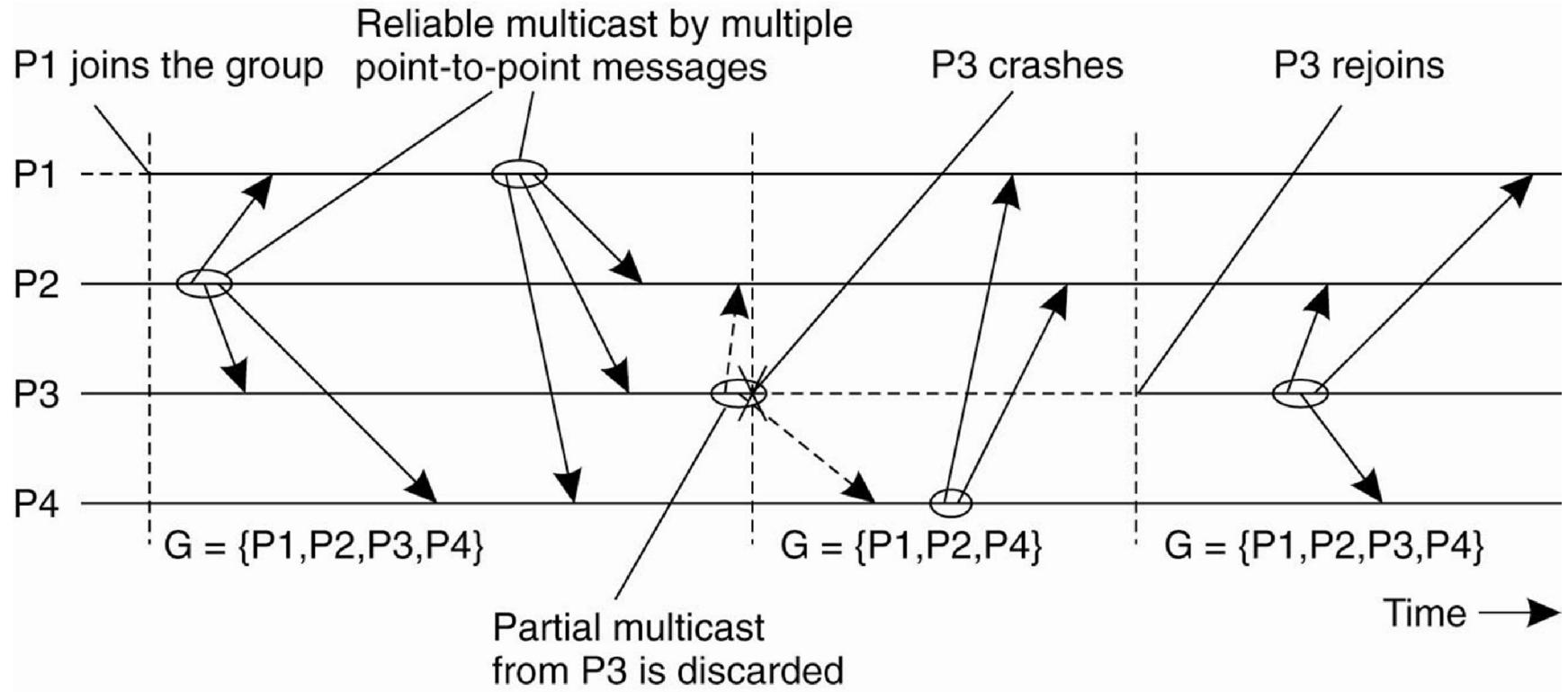
Group View

- Group view is uniquely associated to a multicast message m , it is a **list of processes** to which m should be delivered.
- Processes on that list all agree that m should be delivered to each one of them and to no other process.
- BUT, how about the Group View is changed?
 - Assume that while the multicast is taking place, another process joins or leaves the group.

Virtual Synchrony

- **Virtually synchronous multicast** guarantees that a message multicast to group view G is delivered to each non-faulty process in G , or none of them.
- If the sender of the message crashes during the multicast, the message may:
 - either be delivered to all remaining processes,
 - or ignored by each of them.

Virtual Synchrony



- The principle of virtual synchrony comes from the fact that all multicasts take place between view changes.

Message Ordering

- Unordered multicasts
- FIFO-ordered multicasts
 - FIFO local to sender only
- Causally-ordered multicasts
 - Causality determined by “happen-before” relation (local and message send-receive) on events,
- **Totally-ordered multicasts**
 - Totally-ordered multicast which guarantees any order of messages in different processes are same.
- Virtually synchronous reliable multicasting offering totally-ordered delivery of messages is called **atomic multicasting**.

Unordered multicasts

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

- Order of receipt does not have to be order of delivery.
- If delivered in order of receipt, would violate FIFO ordering at P3 (hence also violate causal order)

FIFO-ordered

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

- Each delivery order only has to obey restriction that m1 is delivered before m2, and m3 is delivered before m4.
- Note that since there is no causal relationship between the messages sent at P1 and those sent at P4, this also satisfies causal order.

Six Different Versions of Virtually Synchronous Reliable Multicasting

Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

- Atomic only implies all-or-nothing delivery with some total order;
- Constraints on order is orthogonal to this.

Outline

- Basic Concepts
- Process Resilience
- Reliable Client-server Communication
- Reliable Group Communication
- Recovery
- Distributed Commit

Terminology

- **Backward recovery**: bring the system from its present erroneous state back into a previously correct state.
 - **Checkpoint**: the system's present state is recorded at the checkpoint.
- **Forward recovery**: bring the system in a correct new state from which it can continue to execute.
 - **Reissuing** the message.

Problems

- Restoring a system or process to a previous state is generally a relatively costly operation in terms of performance.
- No guarantees can be given that once recovery has taken place, the same or similar failure will not happen again.
- Although backward error recovery requires checkpoint, some states can simply never be rolled back to.

Logs

- Combining checkpoints with message logging makes it possible to restore a state that lies beyond the most recent checkpoint without the cost of checkpointing.
 - Sender-based logging
 - Receiver-based logging (better)
- The combination of having fewer checkpoints and message logging is more efficient than having to take many checkpoints

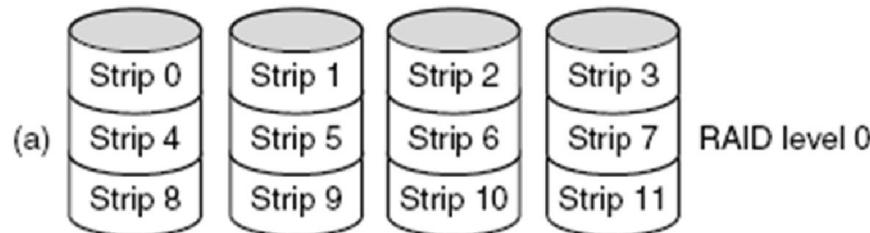
Stable Storage

- Storage comes in three categories
 - Ordinary RAM memory,
 - Disk storage
 - Stable storage, which is designed to survive anything except major calamities such as floods and earthquakes.
- Stable storage can be implemented with a pair of ordinary disks,

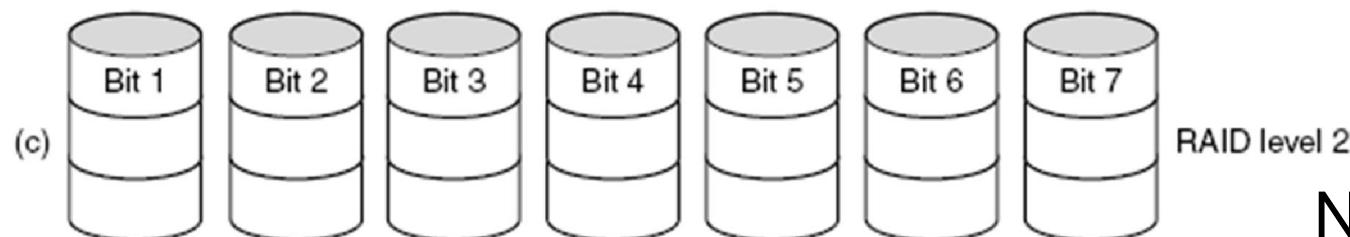
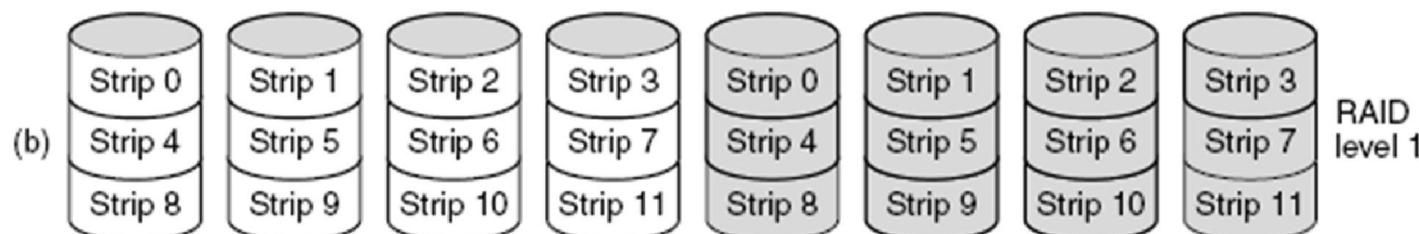
RAID

- RAID (Redundant Array of Independent Disks) is a data storage virtualization technology that combines multiple disk drive components into a logical unit for the purposes of data redundancy or performance improvement.
- RAID 1-7, 10, 50,60
- RAID 5 is widely applied because it includes the RAID 2,3,4

RAID Storage (1)



No redundancy,
pipelined for speed

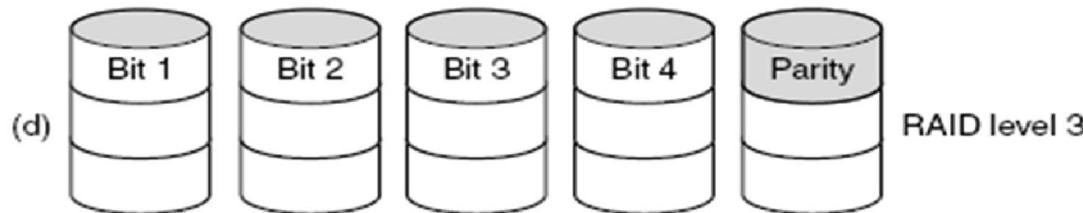


No Strip
No redundancy,
Hamming code (linear error-correcting codes)

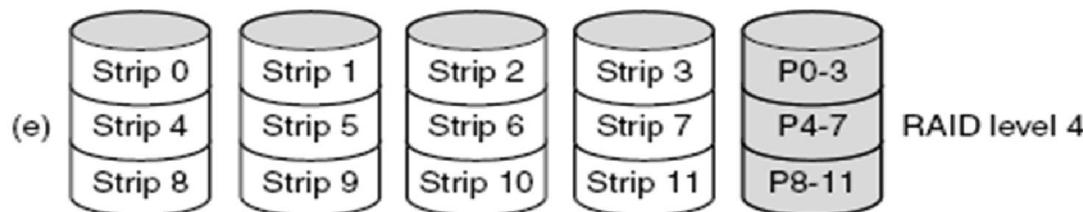
RAID levels 0 through 2.

Backup and parity drives are shown shaded

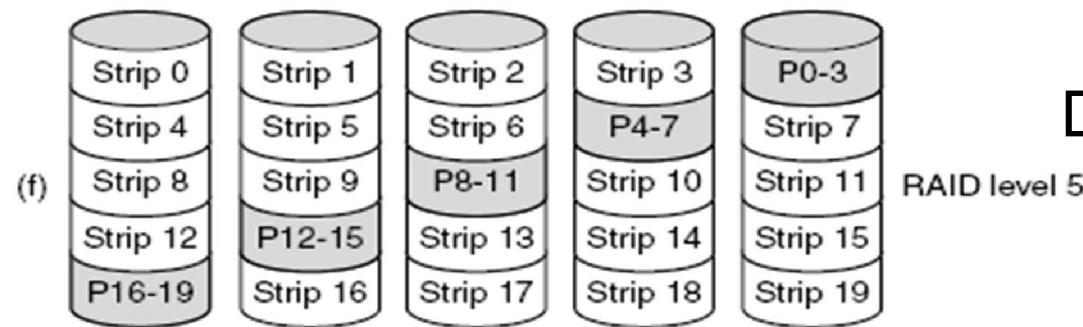
RAID Storage (2)



Bit level parity(奇偶性)



Strip level parity(奇偶性)



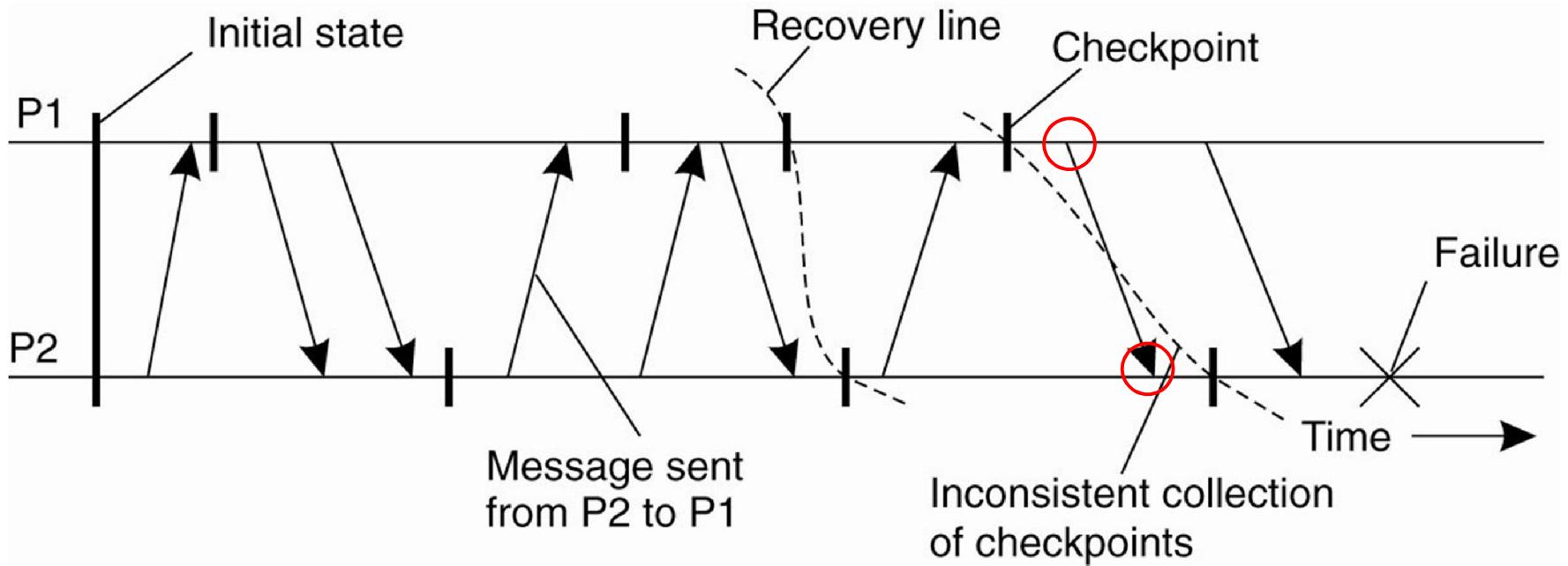
Distribute parity strips

RAID levels 3 through 5.
Backup and parity drives are shown shaded

Checkpoint

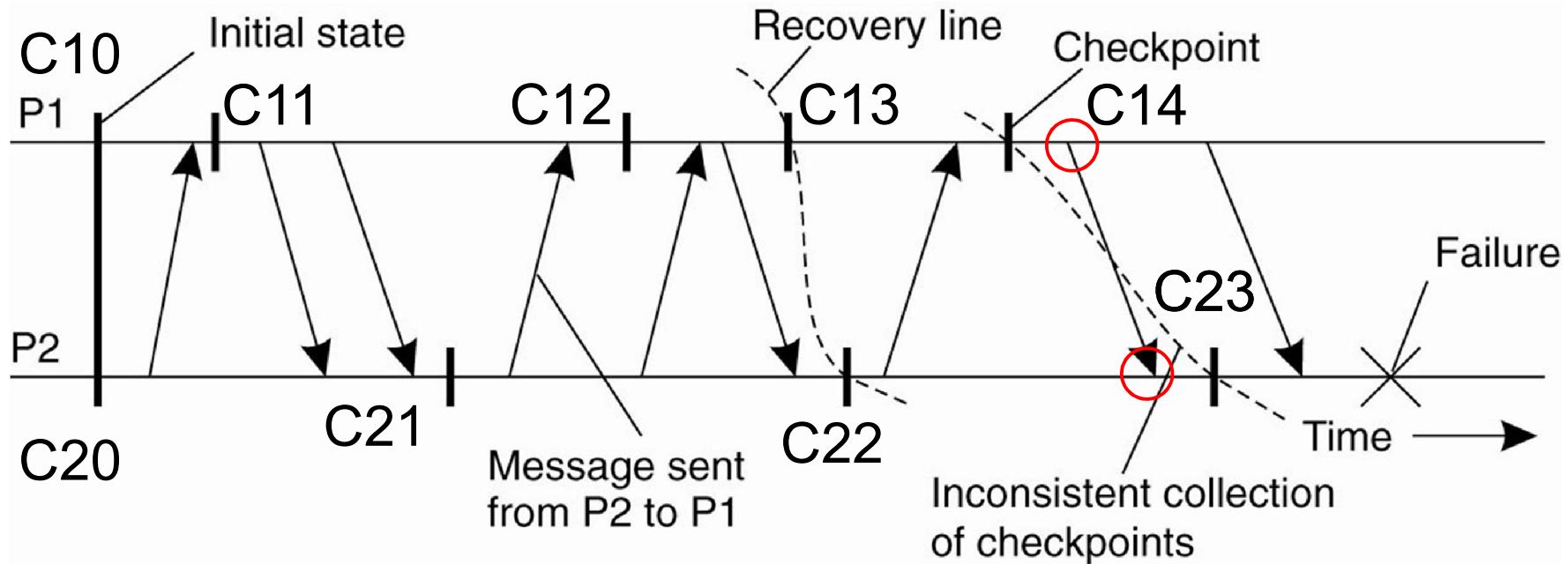
- At the **checkpoint**, a consistent global state, also called a **distributed snapshot**, is made.
- **Recovery line** is the way of recovering to the most recent distributed snapshot.
 - The most recent consistent collection of checkpoints

Checkpoint based Rollback (1)



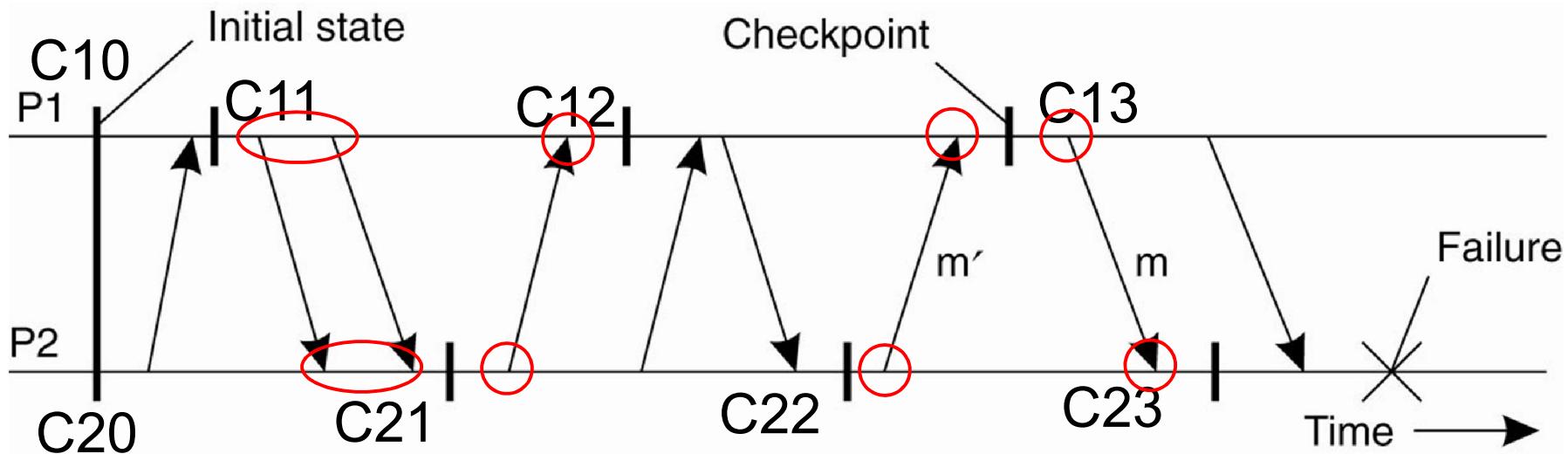
- Can't have a received message that wasn't sent!

Checkpoint based Rollback (2)



P2 crashes, rolls back to C23, but this requires P1 to return to C14, which causes P2 to return to C22, causing P1 to return to C13. Now we are OK, and the system can resume.

Independent Checkpoints



The **domino effect**, a.k.a. cascading rollbacks.

P2 crashes, rolls back to C23, but this requires P1 to return to C13, which causes P2 to return to C22 (due to m'), causing P1 to return to C12 (due to m). Now P2 has to return to C21, then P1 has to return to C11, ... until both are at initial state!

Globally Coordinate Checkpoint

- Globally Coordinate Checkpoint is to solve the problems of cascading rollbacks.
 - Coordination requires global synchronization, which may introduce performance problems.
- Independent Checkpoints have the problems of:
 - Computing the recovery line.
 - Each local storage needs to be cleaned up periodically

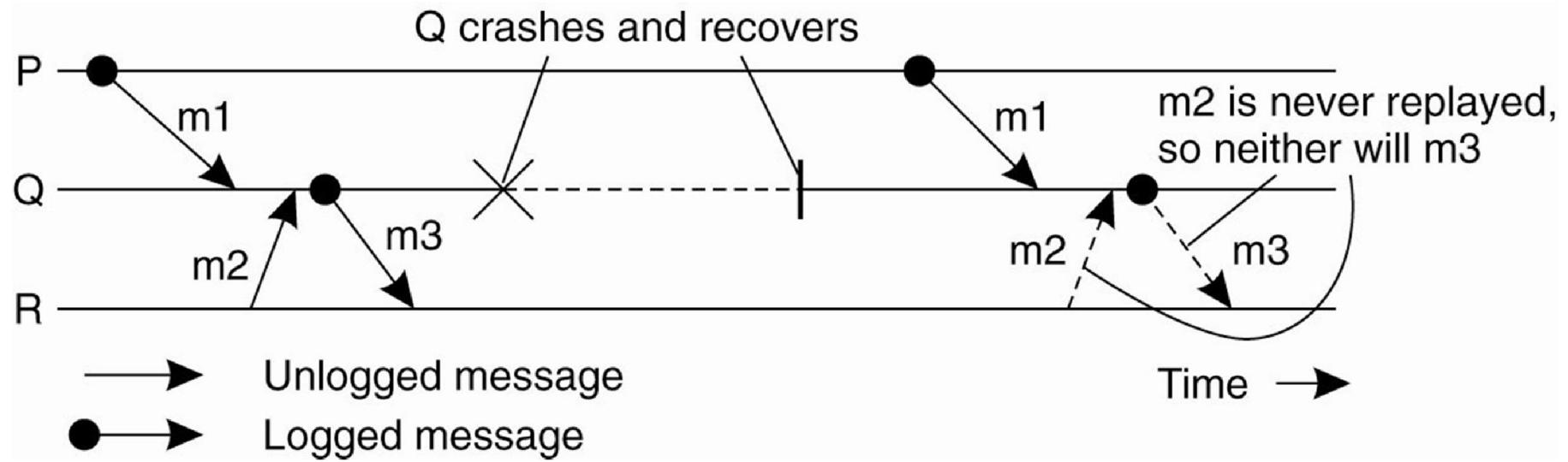
Message Logging

- If the transmission of messages can be replayed, we can still reach a globally consistent state but without having to restore that state from stable storage.
- A checkpointed state is taken as a starting point, and all messages that have been sent since are simply retransmitted and handled accordingly.

Piecewise Deterministic Model (分段确定模式)

- The execution of each process is assumed to take place as a series of **intervals** in which events take place.
- An interval can be replayed with a known result, that is, in a completely deterministic way, provided it is replayed starting with the same nondeterministic event as before.

Incorrect replay of messages after recovery



An orphan process is a process that survives the crash of another process, but whose state is inconsistent with the crashed process after its recovery.

Outline

- Basic Concepts
- Process Resilience
- Reliable Client-server Communication
- Reliable Group Communication
- Recovery
- **Distributed Commit**

Distributed Commit

- The distributed commit problem involves having an operation being performed by each member of a process group, or none at all.
- Distributed commit is often established by means of a coordinator and participants.
 - Coordinator tells all other processes that are also involved, called participants,

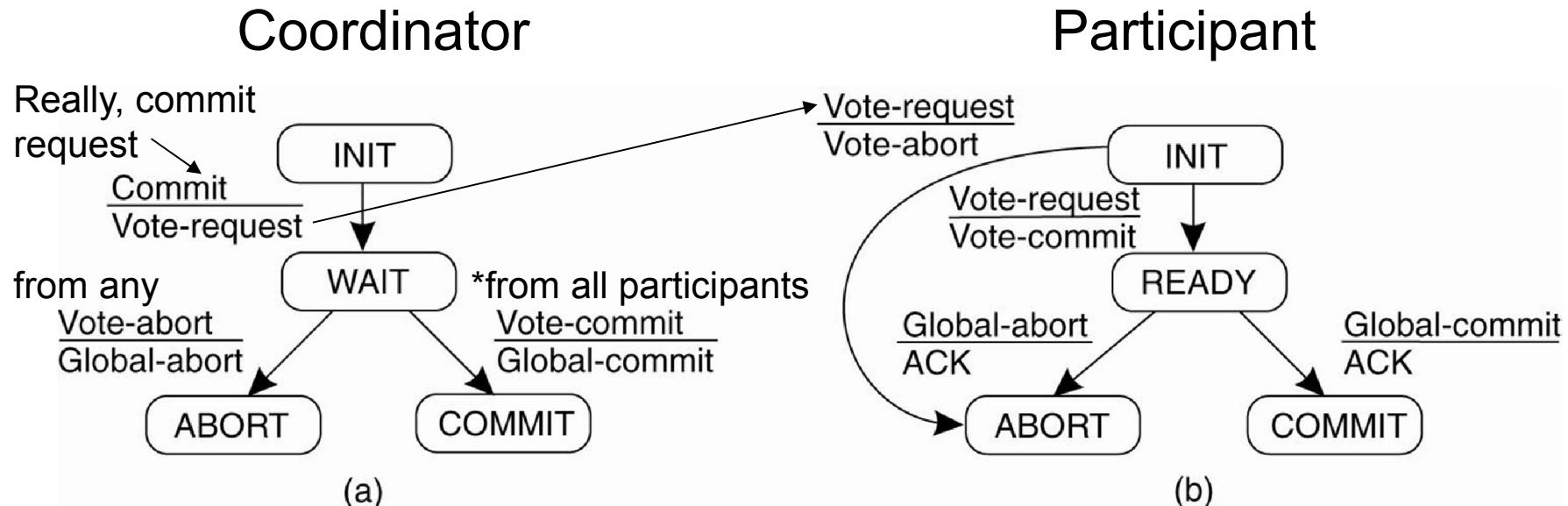
One-Phase Commit

- Coordinator simply tells all the participants to commit in a current situation.
 - Participants can not tell coordinator their status

Two-Phase Commit (2PC)

- The coordinator sends a **VOTE-REQUEST** message to all participants.
- When a participant receives a **VOTE-REQUEST** message, it returns either a **VOTE_COMMIT** message to the coordinator telling the coordinator that it is prepared to locally commit its part of the transaction, or otherwise a **VOTE-ABORT** message.
- The coordinator collects all votes from the participants. If all participants have voted to **commit the transaction**, then so will the coordinator. In that case, it sends a **GLOBAL_COMMIT** message to all participants. However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a **GLOBAL_ABORT** message.
- Each participant that voted for a commit waits for the final reaction by the coordinator. If a participant receives a **GLOBAL_COMMIT** message, it locally commits the transaction. Otherwise, when receiving a **GLOBAL_ABORT** message, the transaction is locally aborted as well.

Finite State Machine of 2PC

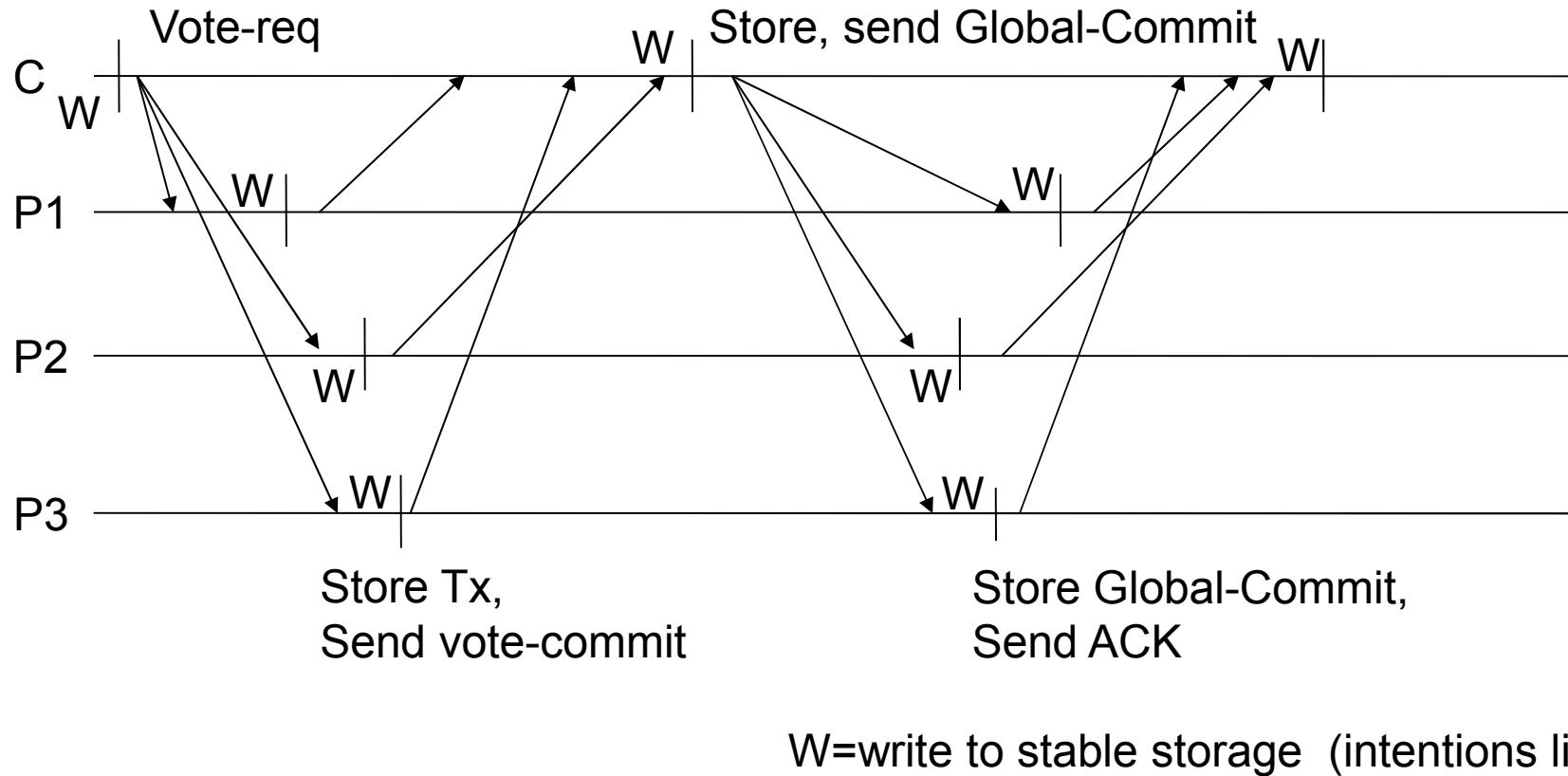


*Commit is like AND of participant votes except coordinator may elect to abort anyway!

Another better solution is to let a participant P contact another participant Q to see if it can decide from Q's current state what it should do.

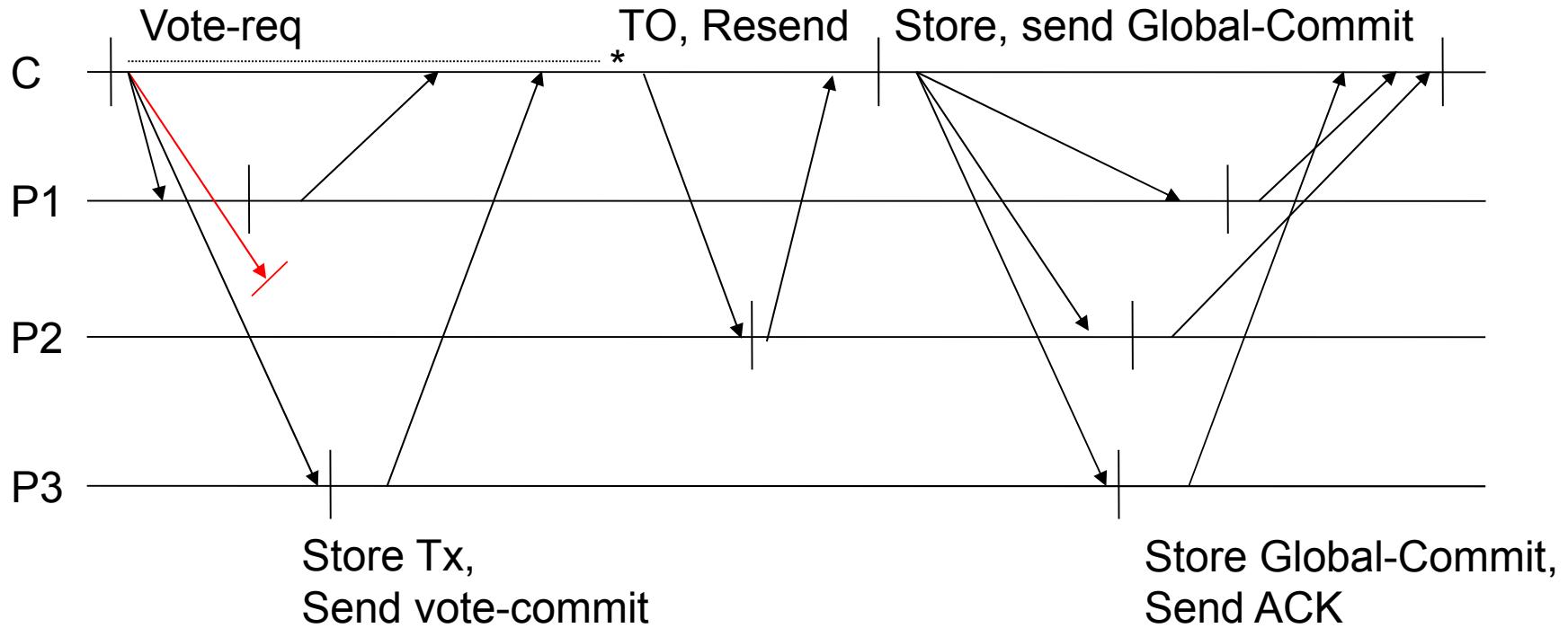
State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Two-Phase Commit (1)



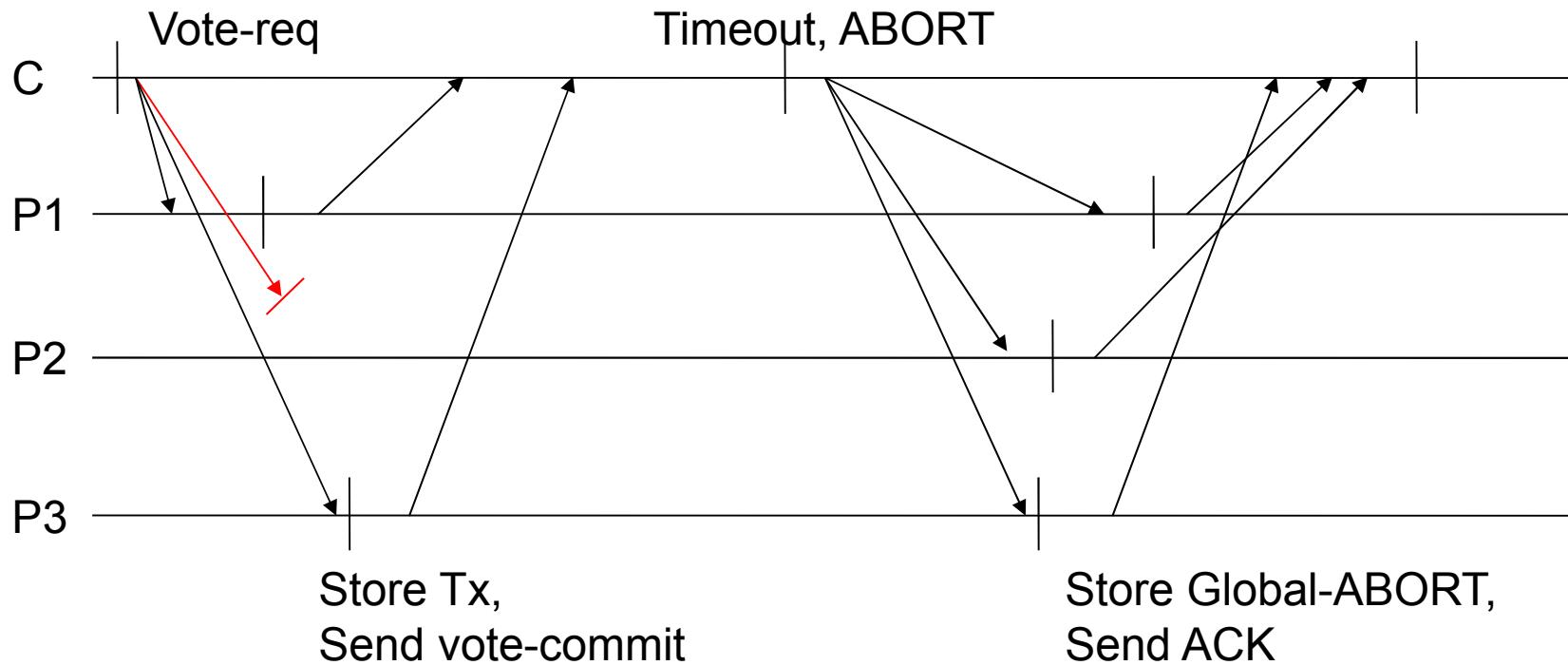
- Normal behavior of 2PC.
- All participants vote to commit, no messages lost.

Two-Phase Commit (2.1)



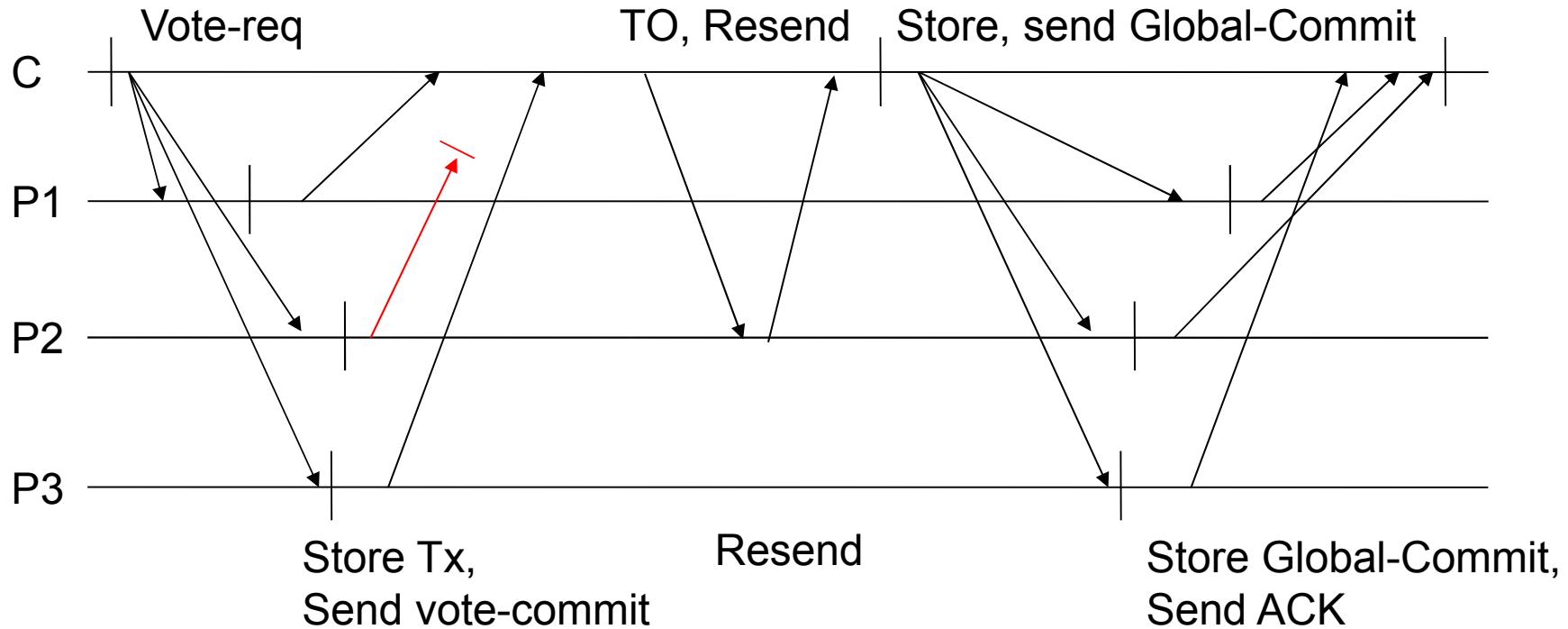
- Normal behavior of 2PC
- All participants vote to commit, Vote-req message lost.

Two-Phase Commit (2.2)



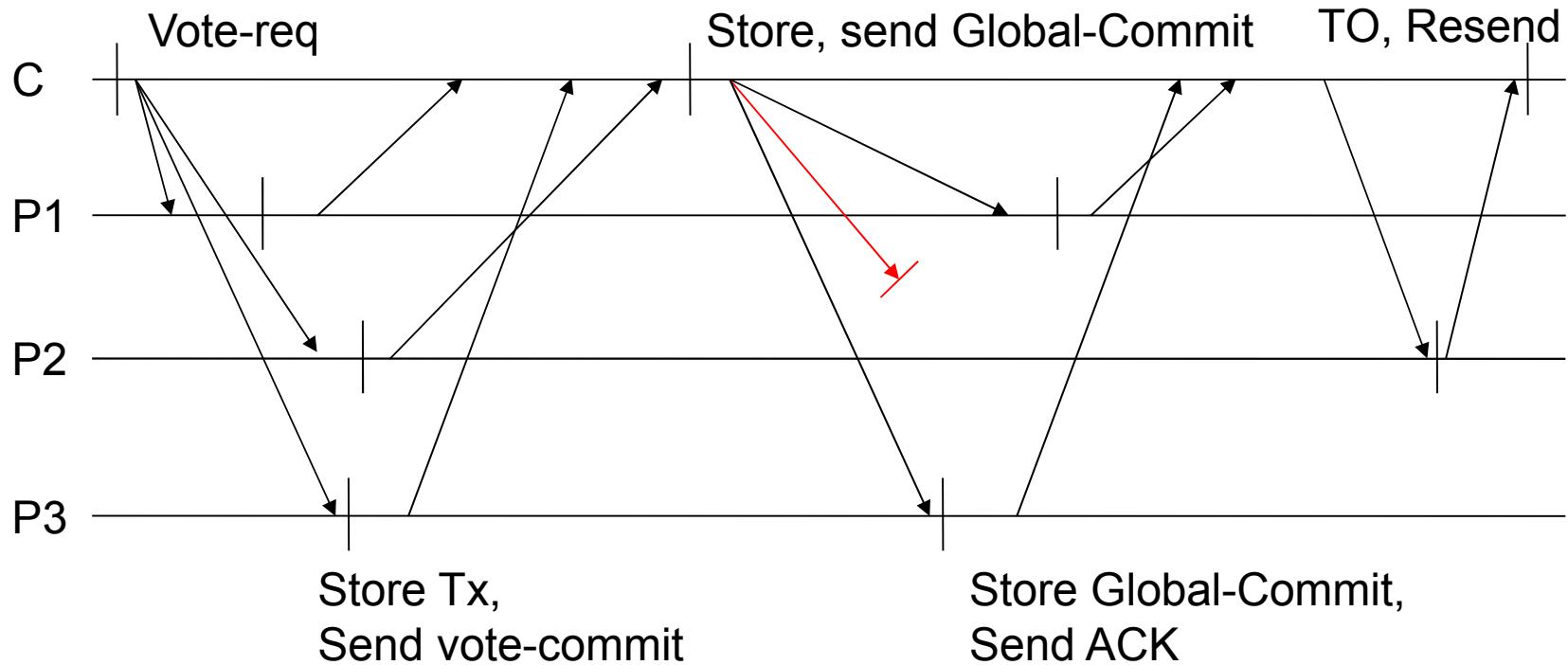
- Normal behavior of 2PC
- All participants vote to commit, Vote-req message lost.
- Coordinator elects to abort.

Two-Phase Commit (3)



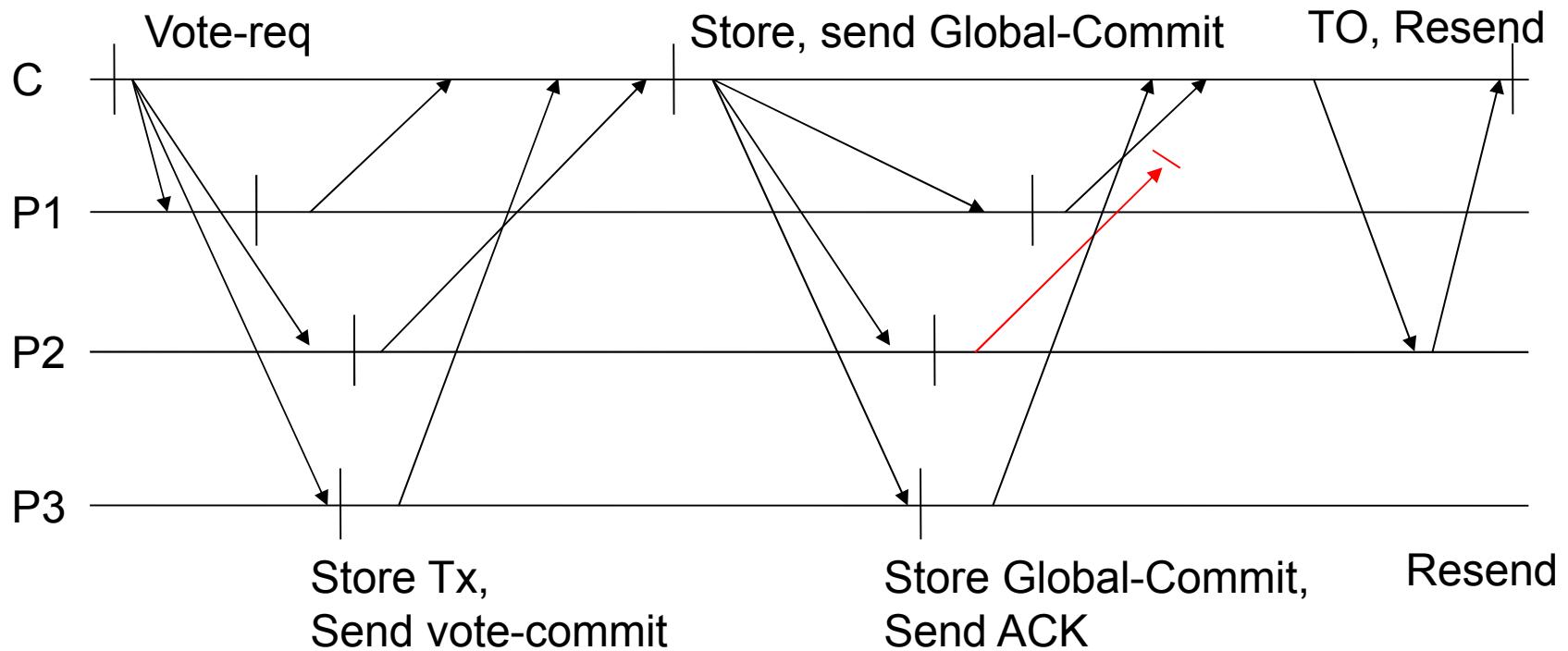
- Normal behavior of 2PC
- All participants vote to commit, YES message lost.
- Coordinator could also decide to abort (not shown here).

Two-Phase Commit (4)



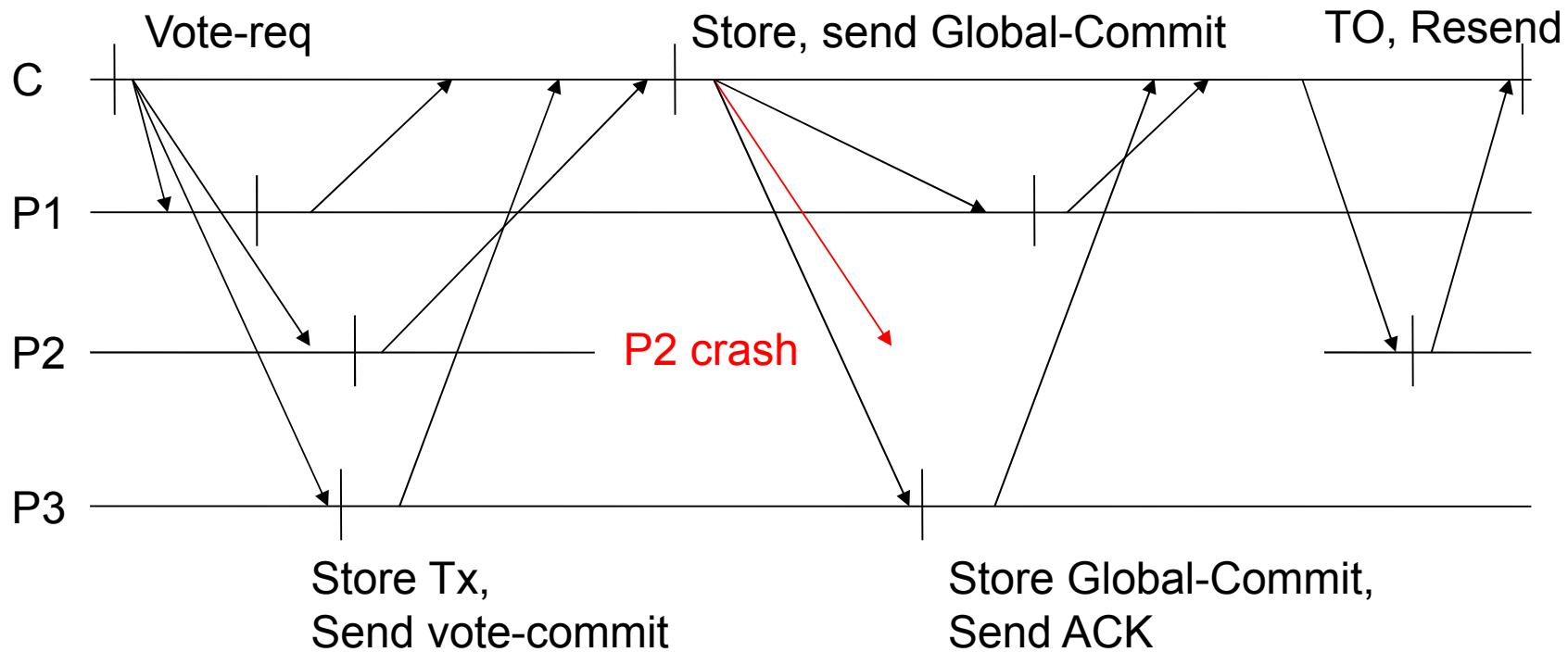
- Normal behavior of 2PC
- All participants vote to commit, Global-Commit message lost.

Two-Phase Commit (5)



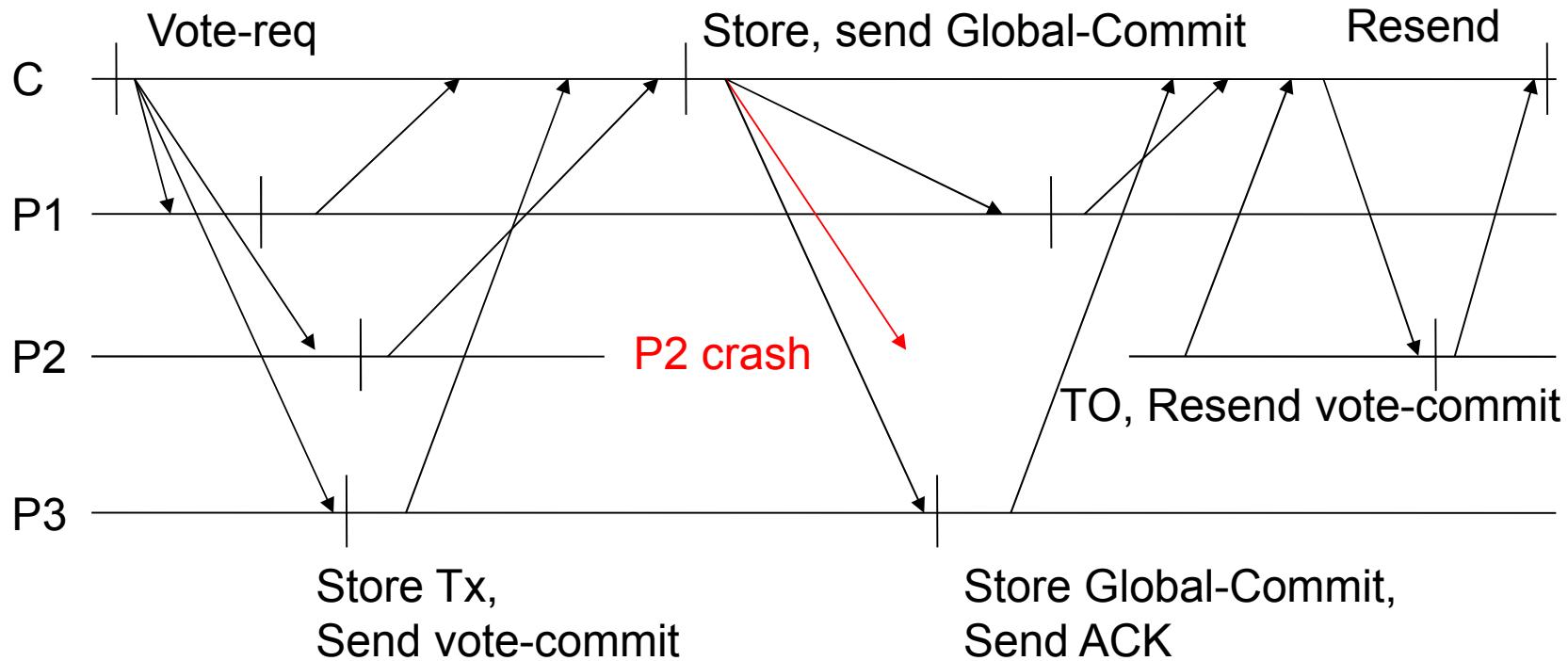
- Normal behavior of 2PC
- All participants vote to commit, ACK message lost.

Two-Phase Commit (6.1)



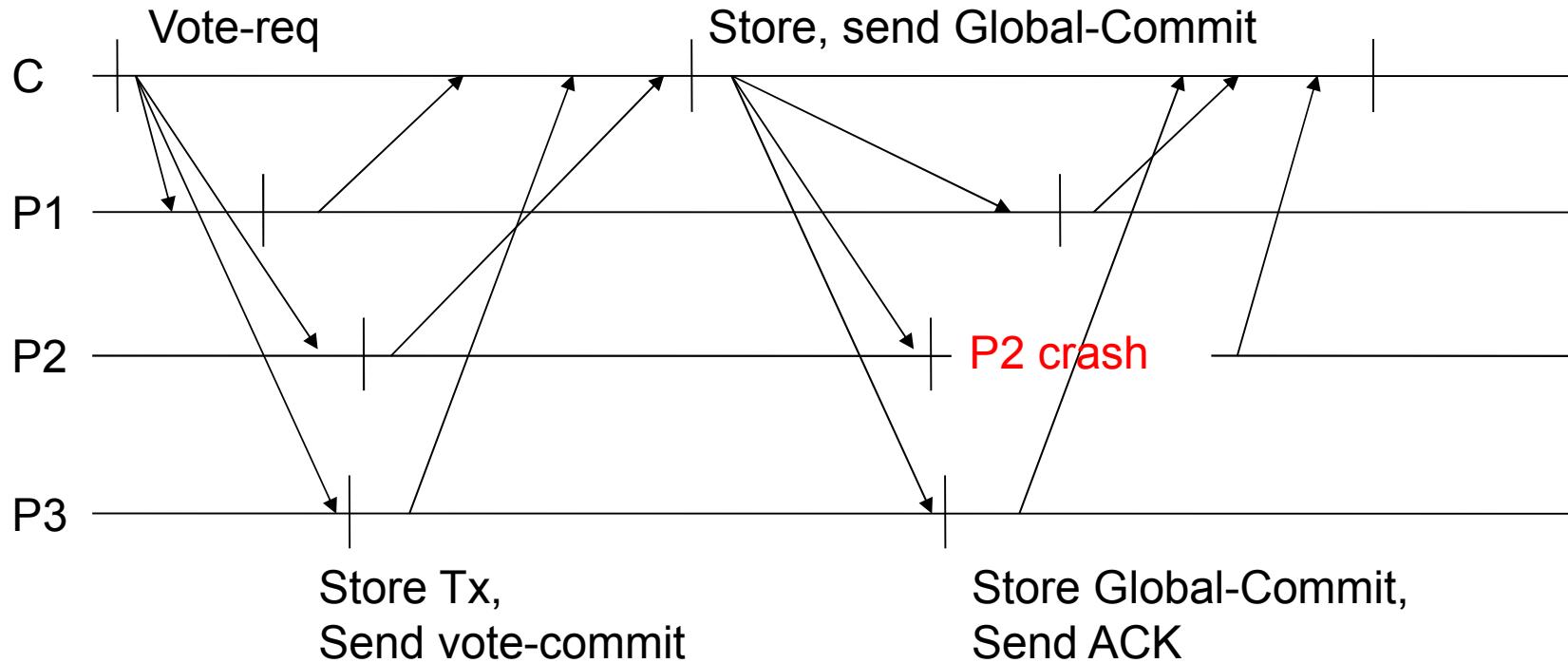
- Behavior of 2PC when Participant crashes...
- All participants vote to commit, no messages lost.

Two-Phase Commit (6.2)



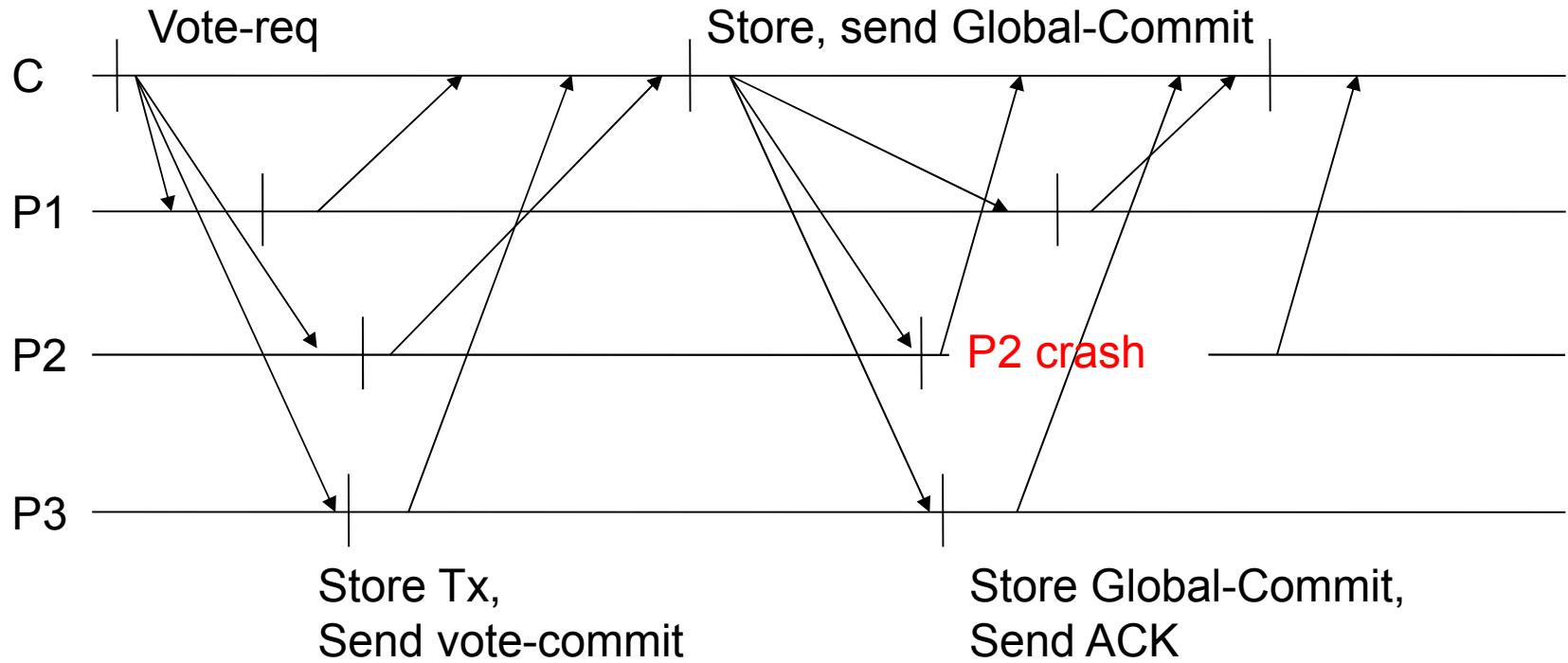
- Behavior of 2PC when Participant crashes
- All participants vote to commit, no messages lost.

Two-Phase Commit (7.1)



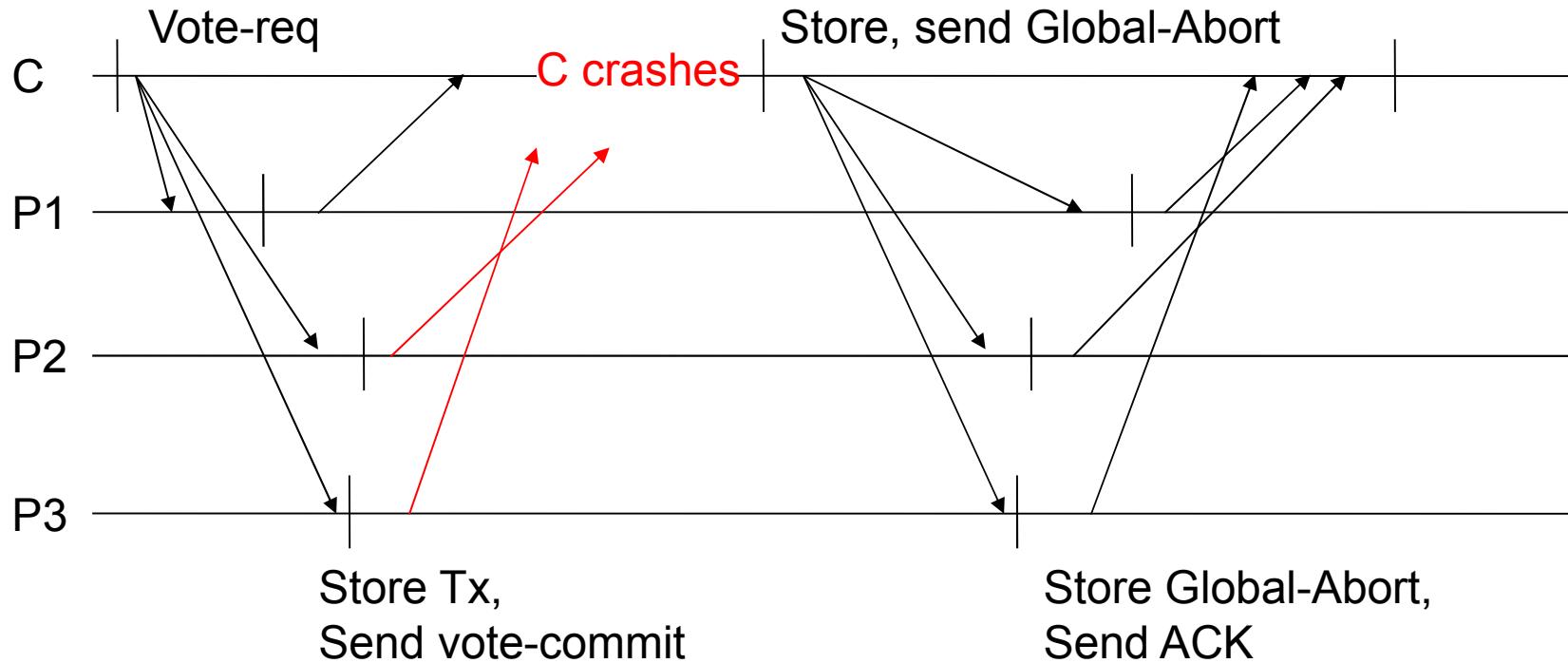
- Behavior of 2PC when Participant crashes
- All participants vote to commit, no messages lost.

Two-Phase Commit (7.2)



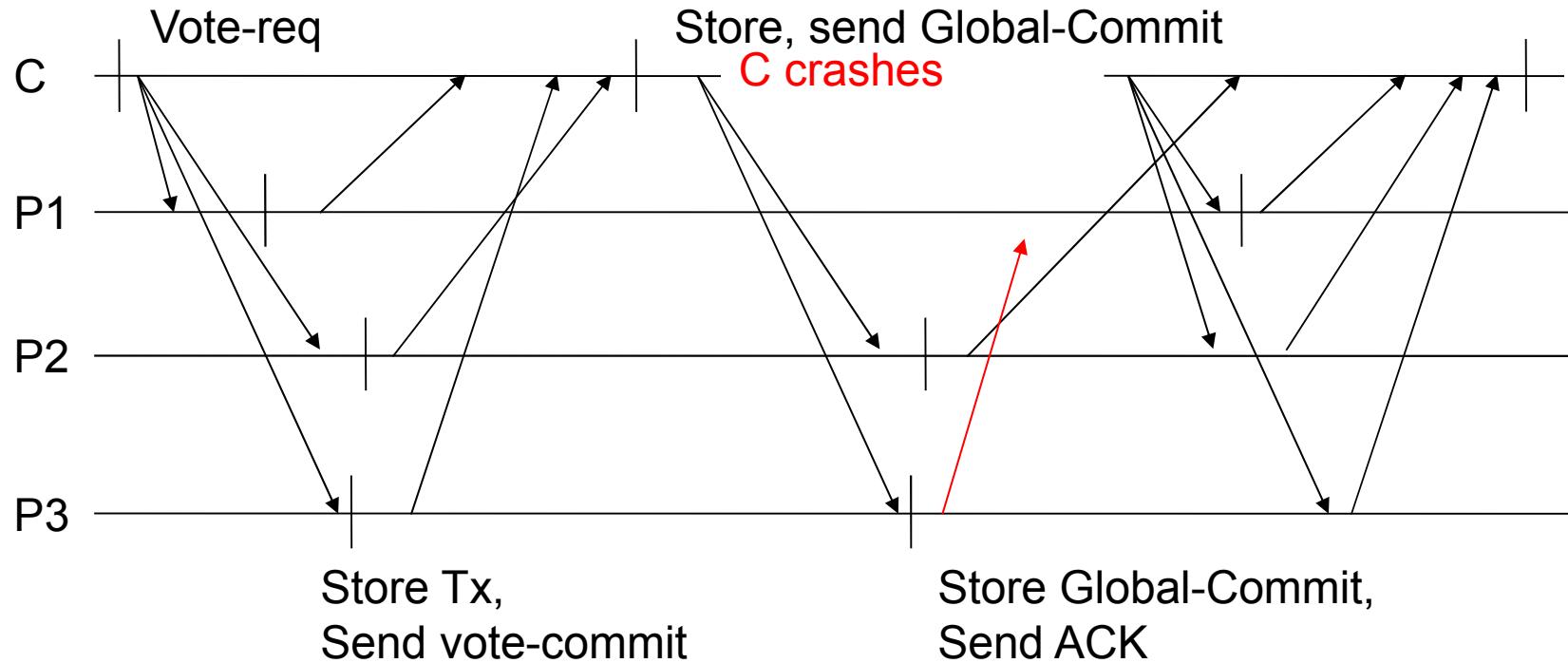
- Behavior of 2PC when Participant crashes.
- All participants vote to commit, no messages lost.

Two-Phase Commit (8)



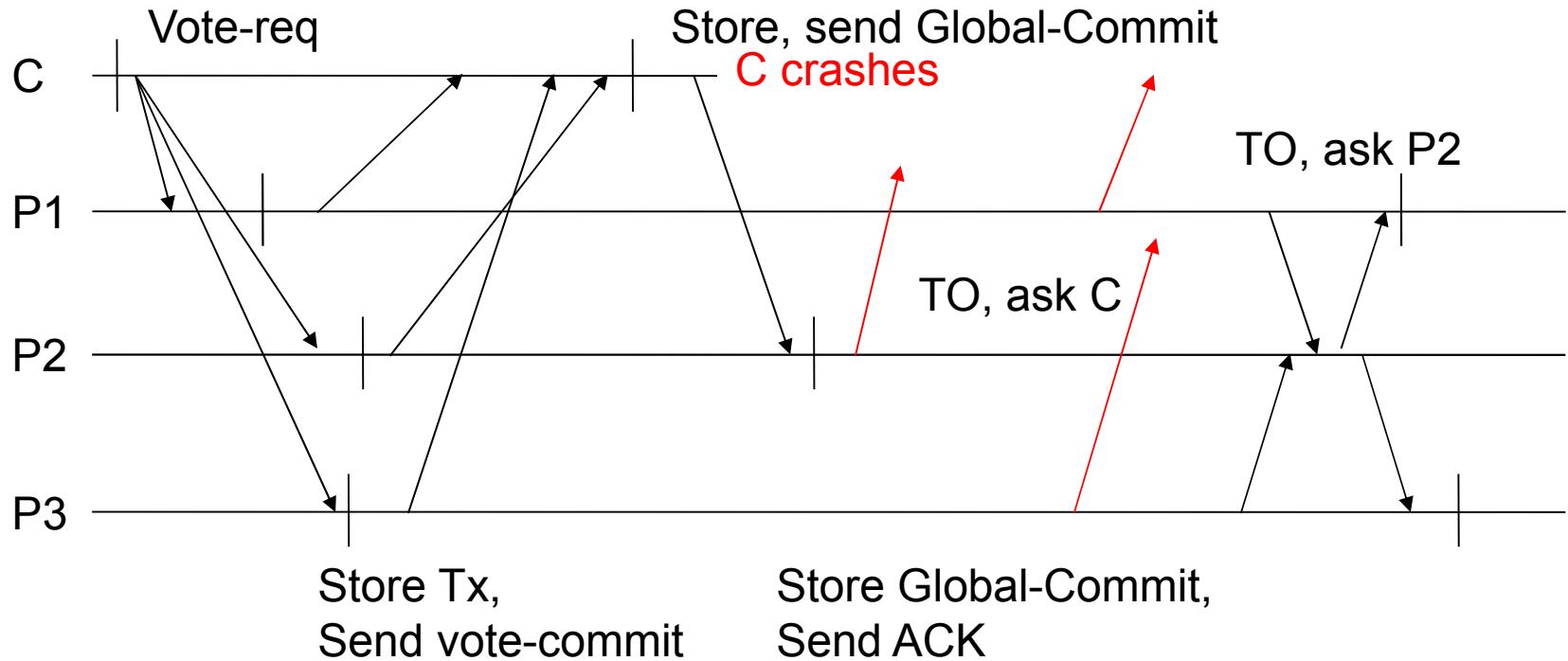
- Behavior of 2PC when Coordinator crashes
- All participants vote to commit, but C aborts.

Two-Phase Commit (9.1)



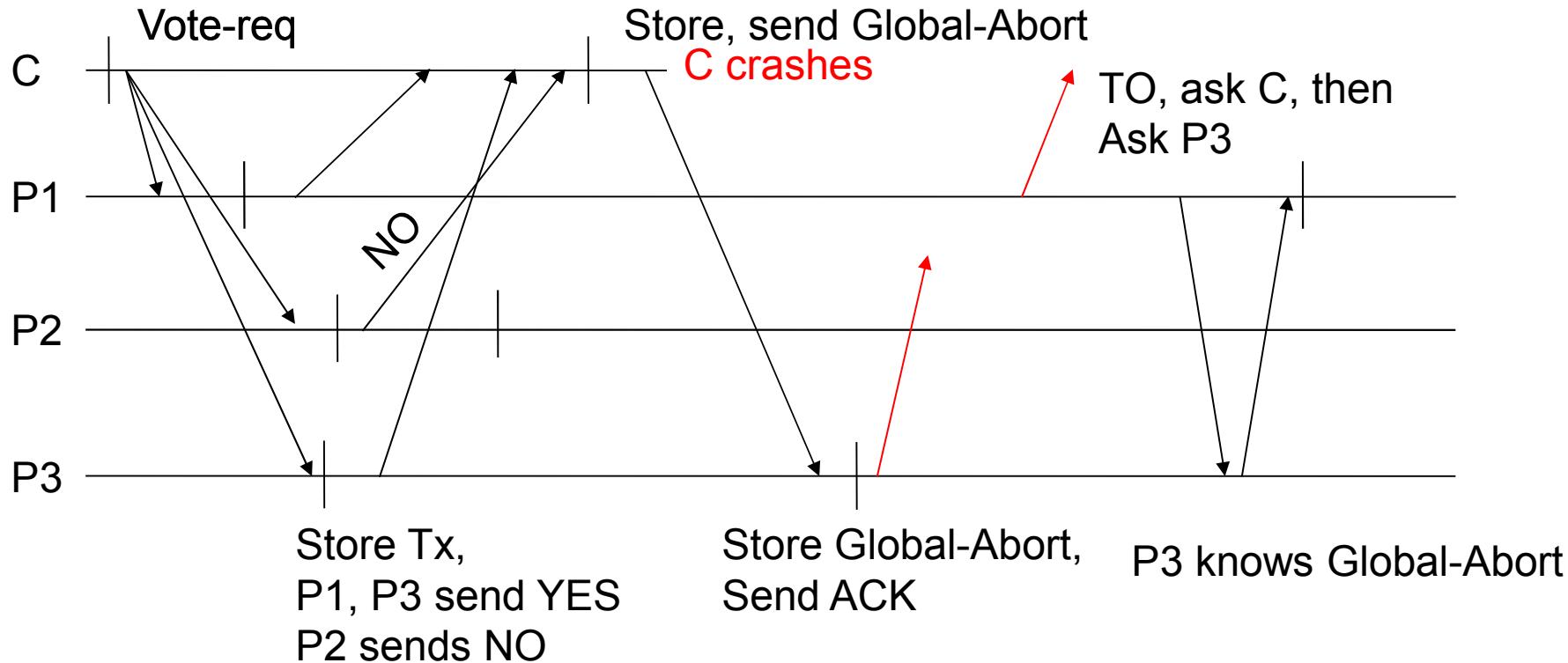
- Behavior of 2PC when Coordinator crashes
- All participants vote to commit, and C commits.

Two-Phase Commit (9.2)



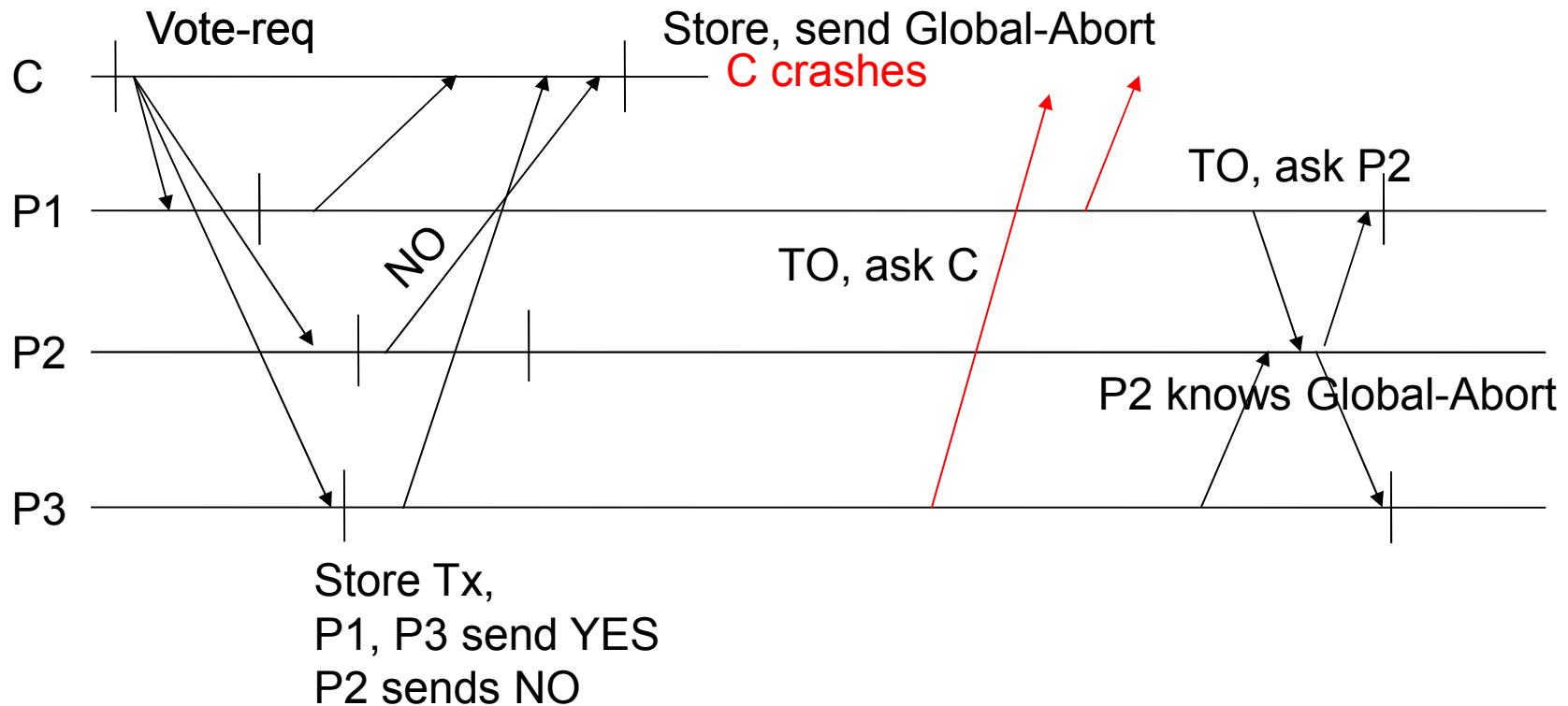
- Behavior of 2PC when Coordinator crashes
- All participants vote to commit, and C commits.
- At least one participant receives Global-Commit.

Two-Phase Commit (10.1)



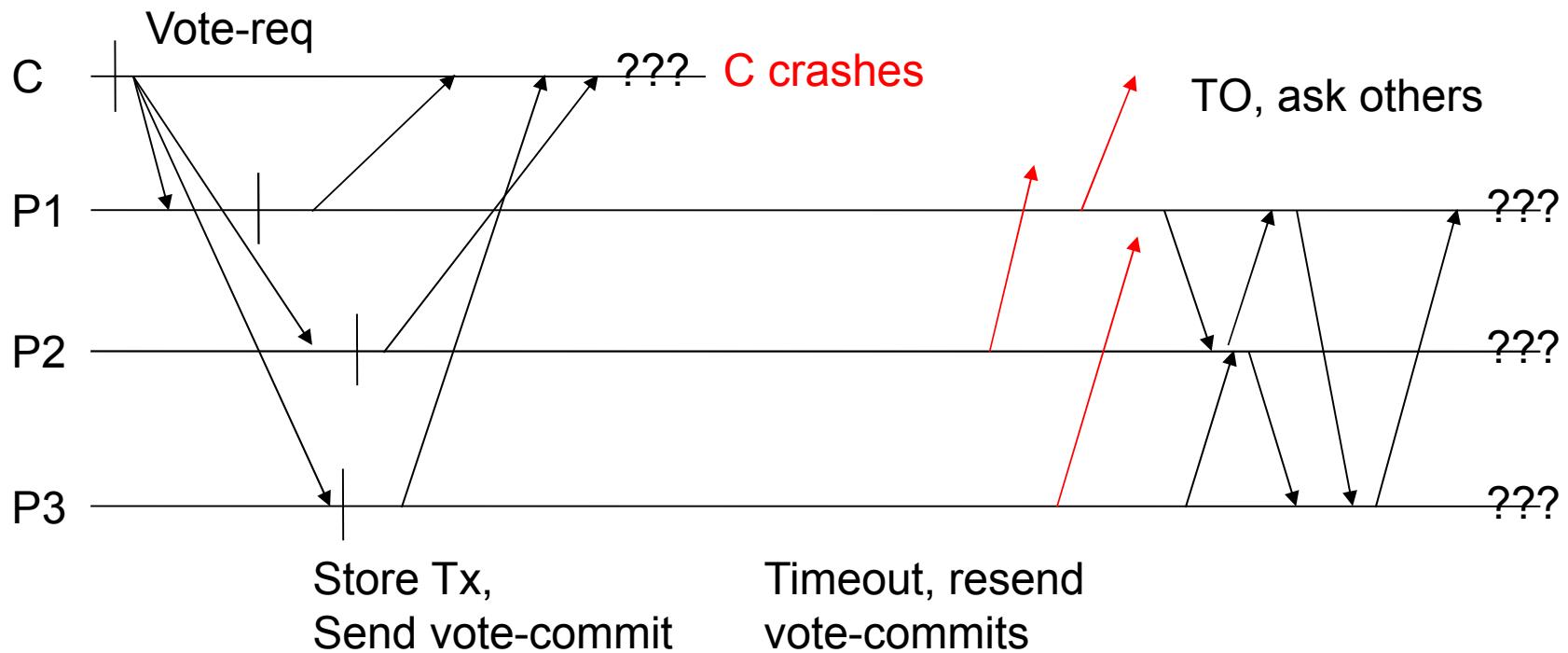
- Behavior of 2PC when Coordinator crashes
- Coordinator sends global-abort.
- Result known since Global-Abort message received.

Two-Phase Commit (10.2)



- Behavior of 2PC when Coordinator crashes
- One participant votes to abort, forces global-abort.
- Result known even if no Global-Abort message received.

Two-Phase Commit (11 final)



- Behavior of 2PC when Coordinator crashes
- All participants vote to commit, but none hear from C.

Three-Phase Commit (3PC)

- A problem with the two-phase commit protocol is that when the coordinator has crashed, participants may not be able to reach a final decision.
 - Participants may need to remain blocked until the coordinator recovers.
- Three-phase commit protocol (3PC) avoids blocking processes in the presence of fail-stop crashes.
 - 3PC it is not applied often in practice as the conditions under which 2PC blocks rarely occur

Three-Phase Commit (3PC)

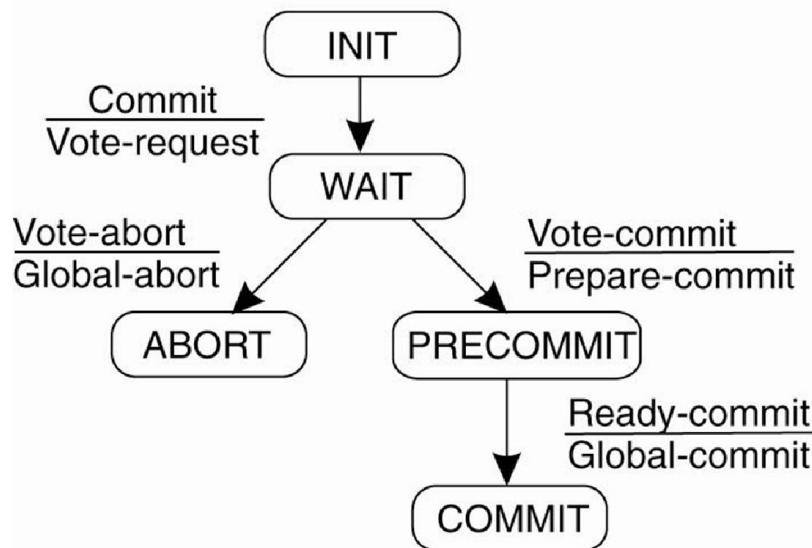
- The states of the coordinator and each participant satisfy the following two conditions:
 - There is no single state from which it is possible to make a transition directly to either a **COMMIT** or an **ABORT** state.
 - There is no state in which it is not possible to make a final decision, and from which a transition to a **COMMIT** state can be made.

Three-Phase Commit (3PC)

- The coordinator starts with sending a **VOTE_REQUEST** message to all participants, after which it waits for incoming responses.
- If any participant votes to abort the transaction, the final decision will be to abort as well, so the coordinator sends **GLOBAL_ABORT**.
- When the transaction can be committed, a **PREPARE_COMMIT** message is sent.
- Only after each participant has acknowledged it is now **prepared to commit**, will the coordinator send the final **GLOBAL_COMMIT** message by which the transaction is actually committed.

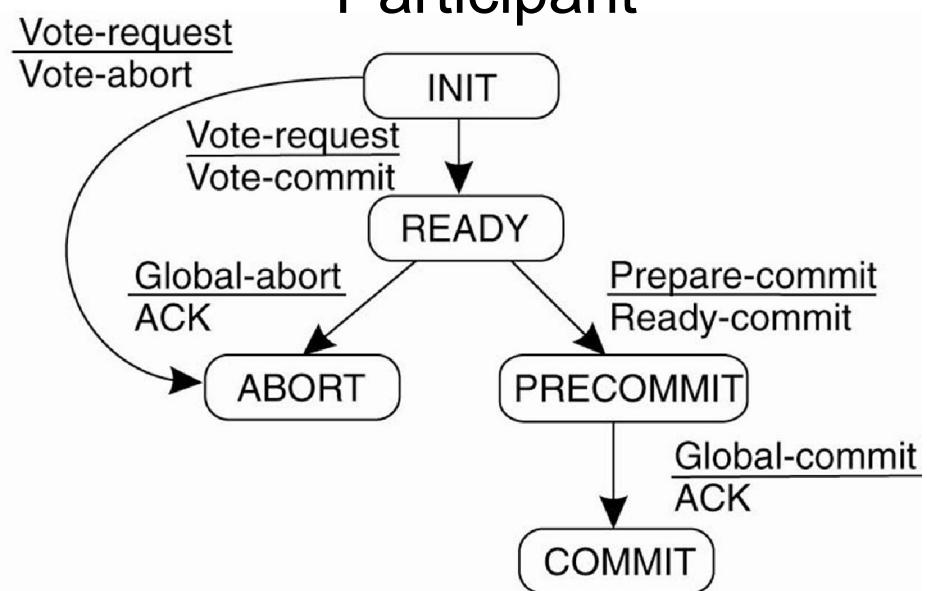
Finite State Machine for 3PC

Coordinator



(a)

Participant



(b)

- Observation: no node can be in **INIT** if any other node is in **PRECOMMIT**
- If participant stuck in **READY** or **PRECOMMIT**, contact others
 - **READY**: if any in **INIT** or **ABORT**, then **ABORT**
 - **PRECOMMIT**: if majority in **PRECOMMIT**, then **COMMIT** else **ABORT**

End

DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 9
Distributed
Web-Based Systems

Traditional Web-Based Systems

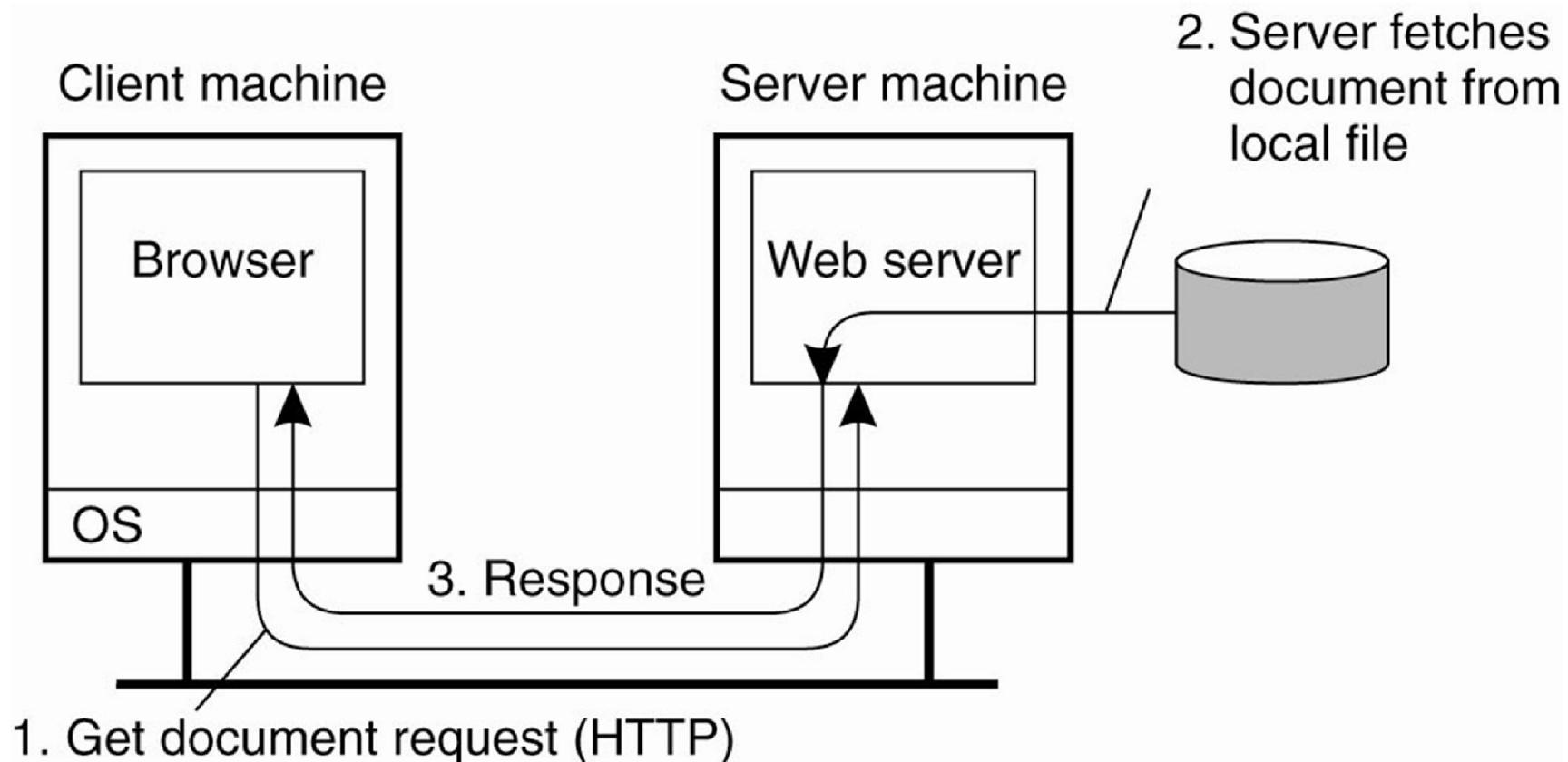


Figure 12-1. The overall organization of a traditional Web site.

Web Documents

Type	Subtype	Description
Text	Plain	Unformatted text
	HTML	Text including HTML markup commands
	XML	Text including XML markup commands
Image	GIF	Still image in GIF format
	JPEG	Still image in JPEG format
Audio	Basic	Audio, 8-bit PCM sampled at 8000 Hz
	Tone	A specific audible tone
Video	MPEG	Movie in MPEG format
	Pointer	Representation of a pointer device for presentations
Application	Octet-stream	An uninterpreted byte sequence
	Postscript	A printable document in Postscript
	PDF	A printable document in PDF
Multipart	Mixed	Independent parts in the specified order
	Parallel	Parts must be viewed simultaneously

Figure 12-2. Six top-level MIME types and some common subtypes.

Multitiered Architectures

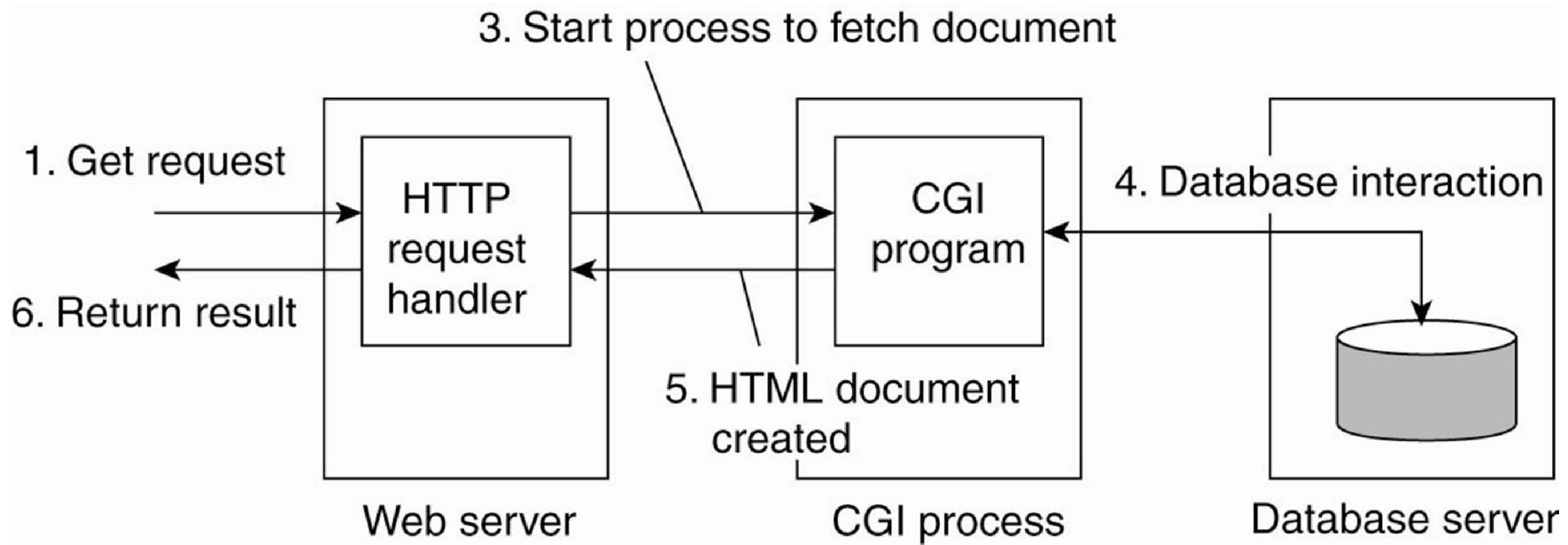


Figure 12-3. The principle of using server-side CGI programs.

Web Services Fundamentals

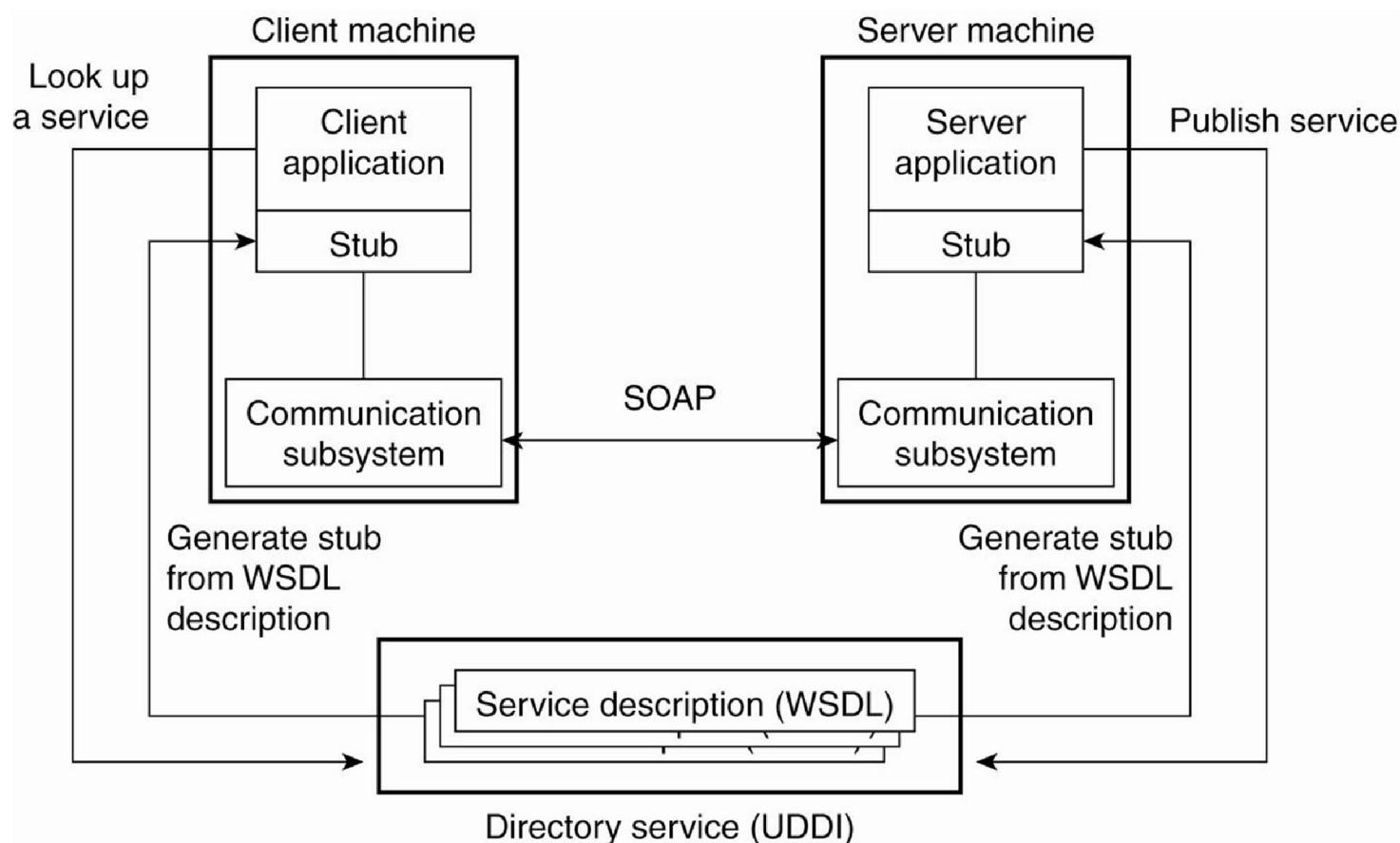


Figure 12-4. The principle of a Web service.

Processes – Clients (1)

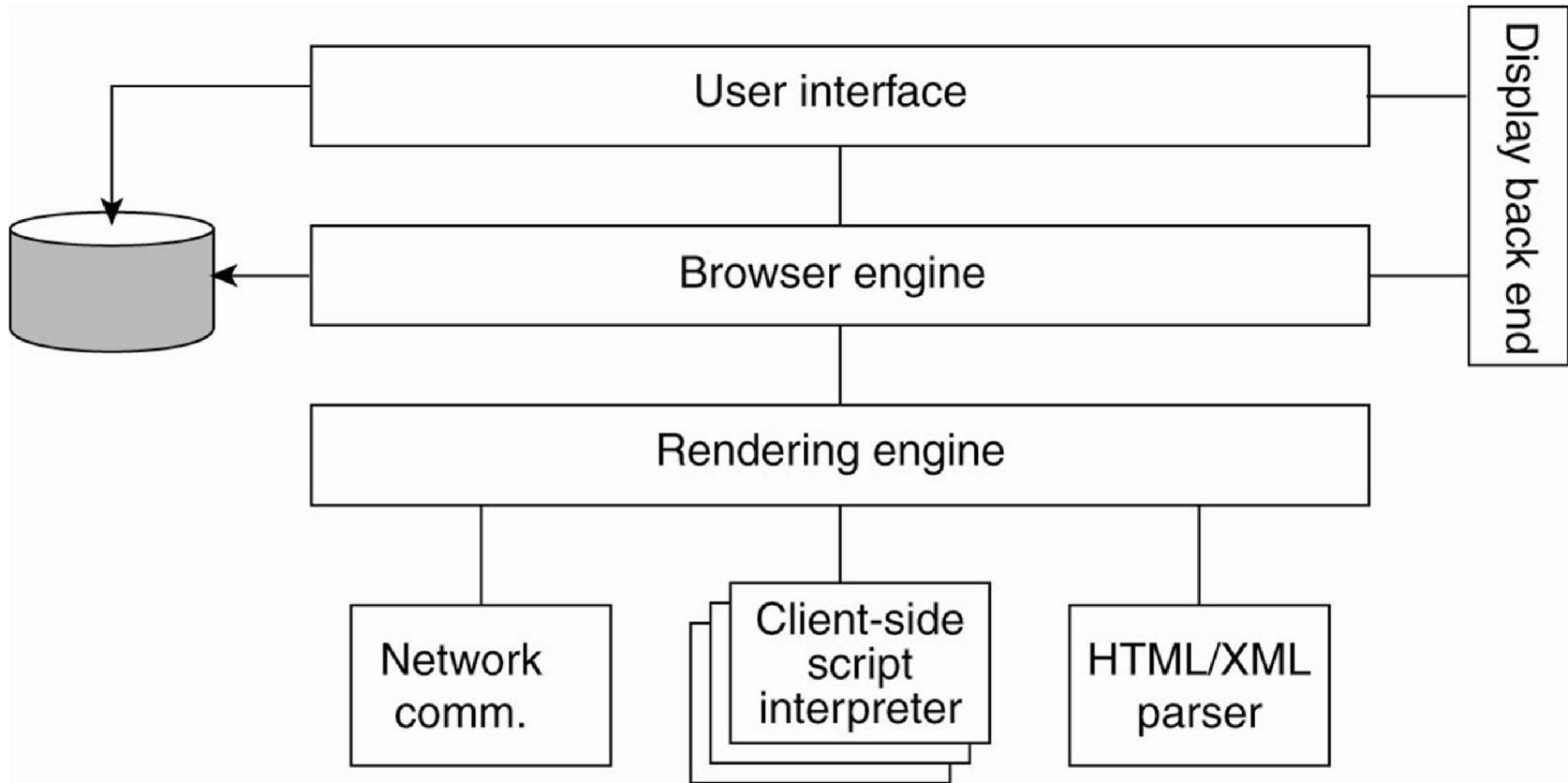


Figure 12-5. The logical components of a Web browser.

Processes – Clients (2)



Figure 12-6. Using a Web proxy when the browser does not speak FTP.

The Apache Web Server

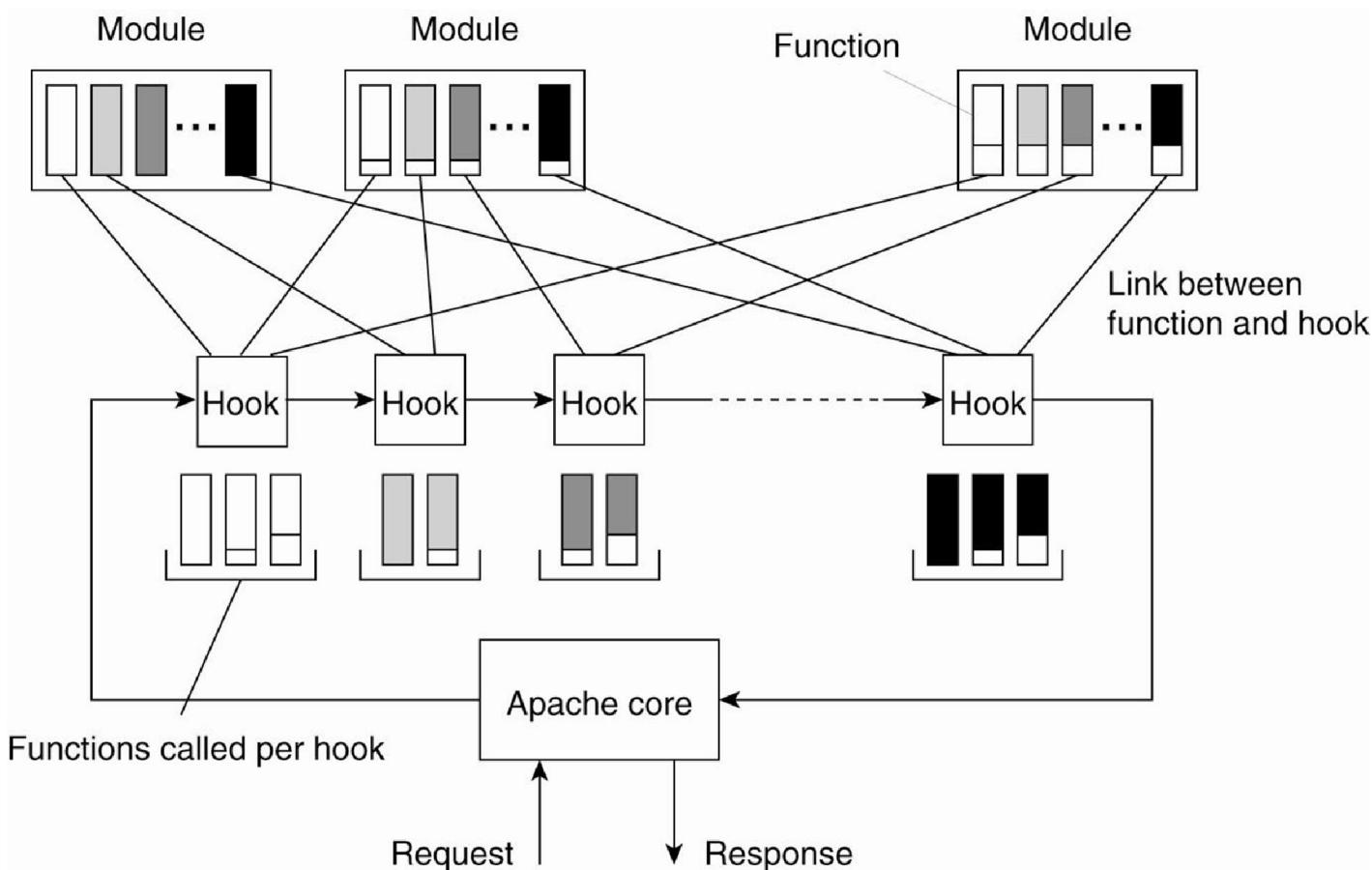


Figure 12-7. The general organization of the Apache Web server.

Web Server Clusters (1)

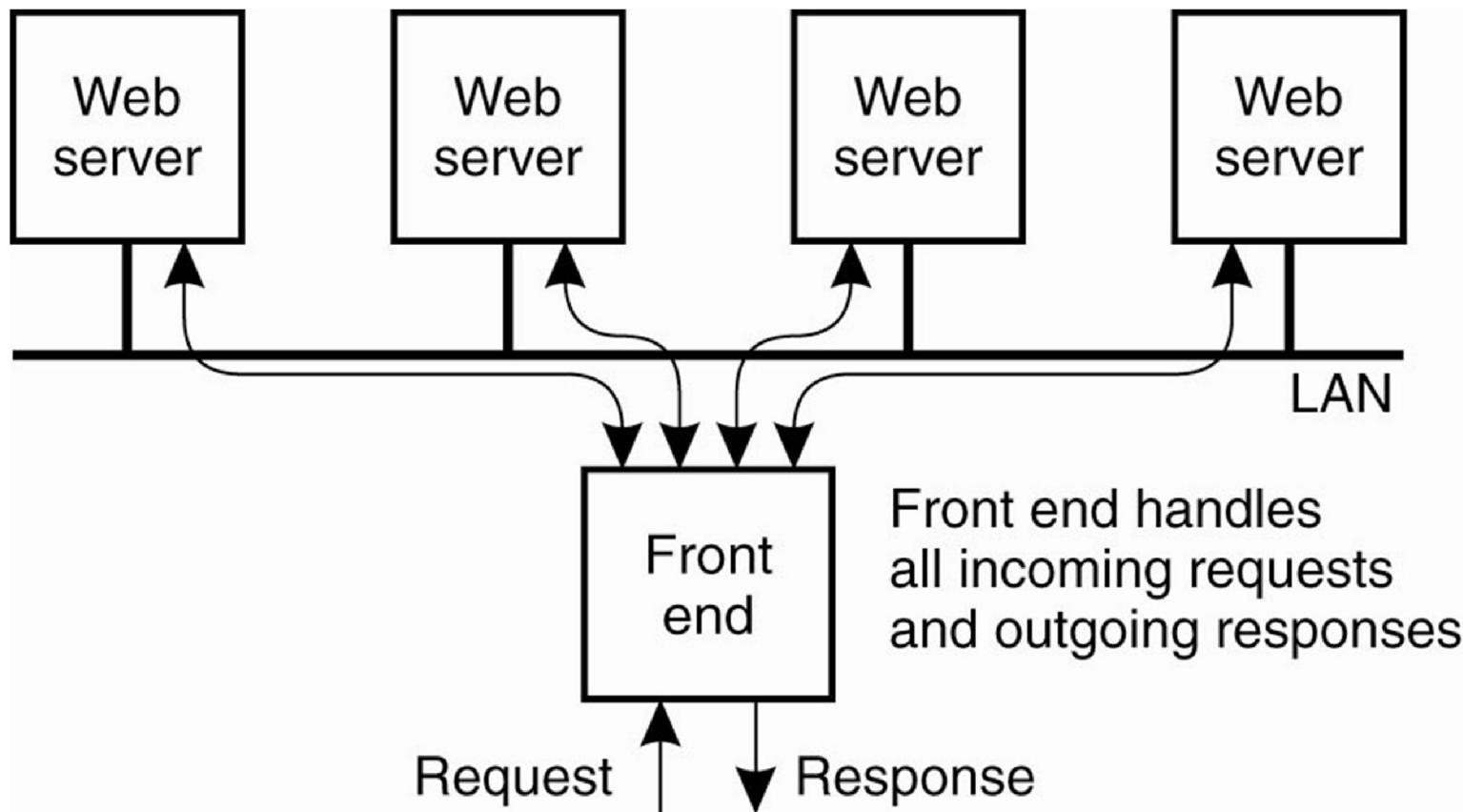


Figure 12-8. The principle of using a server cluster in combination with a front end to implement a Web service.

Web Server Clusters (2)

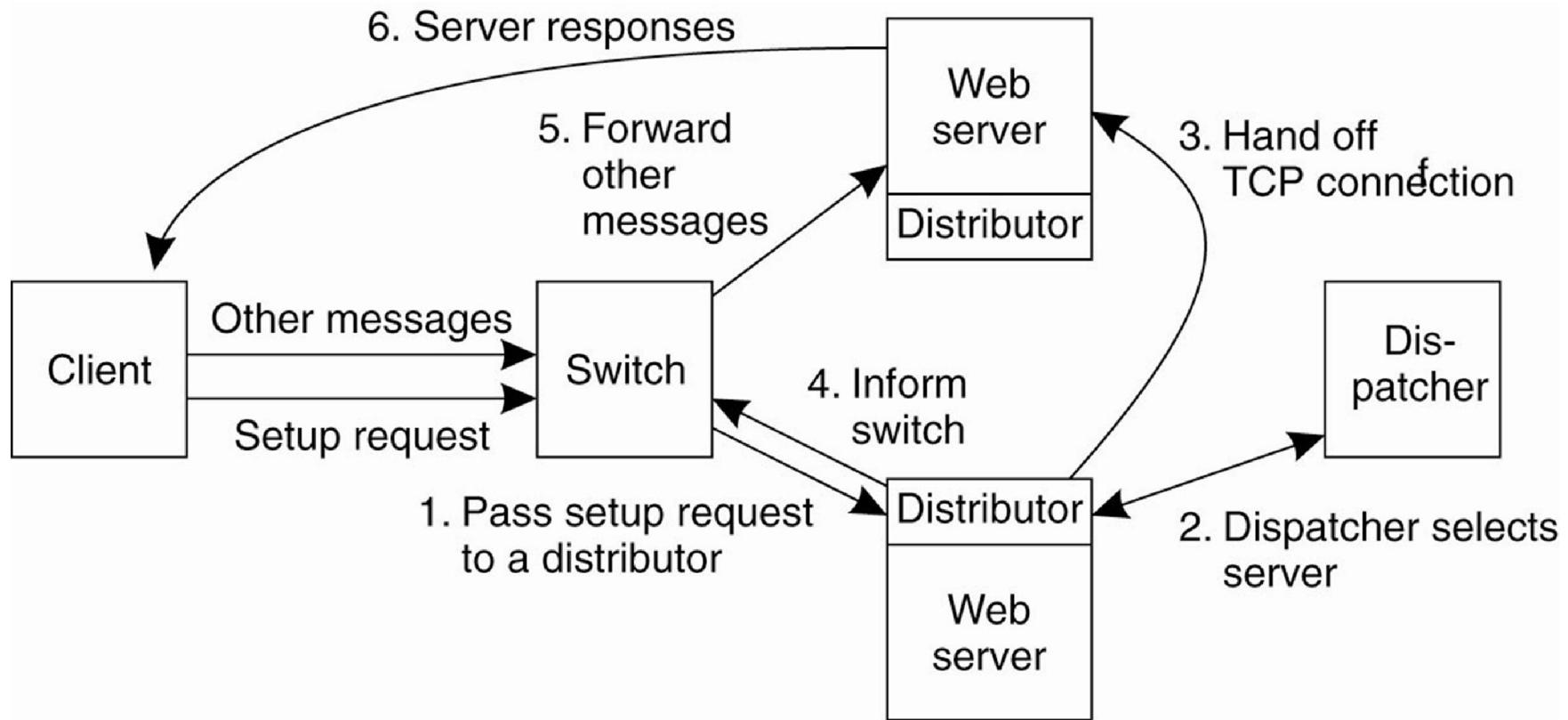


Figure 12-9. A scalable content-aware cluster of Web servers.

HTTP Connections (1)

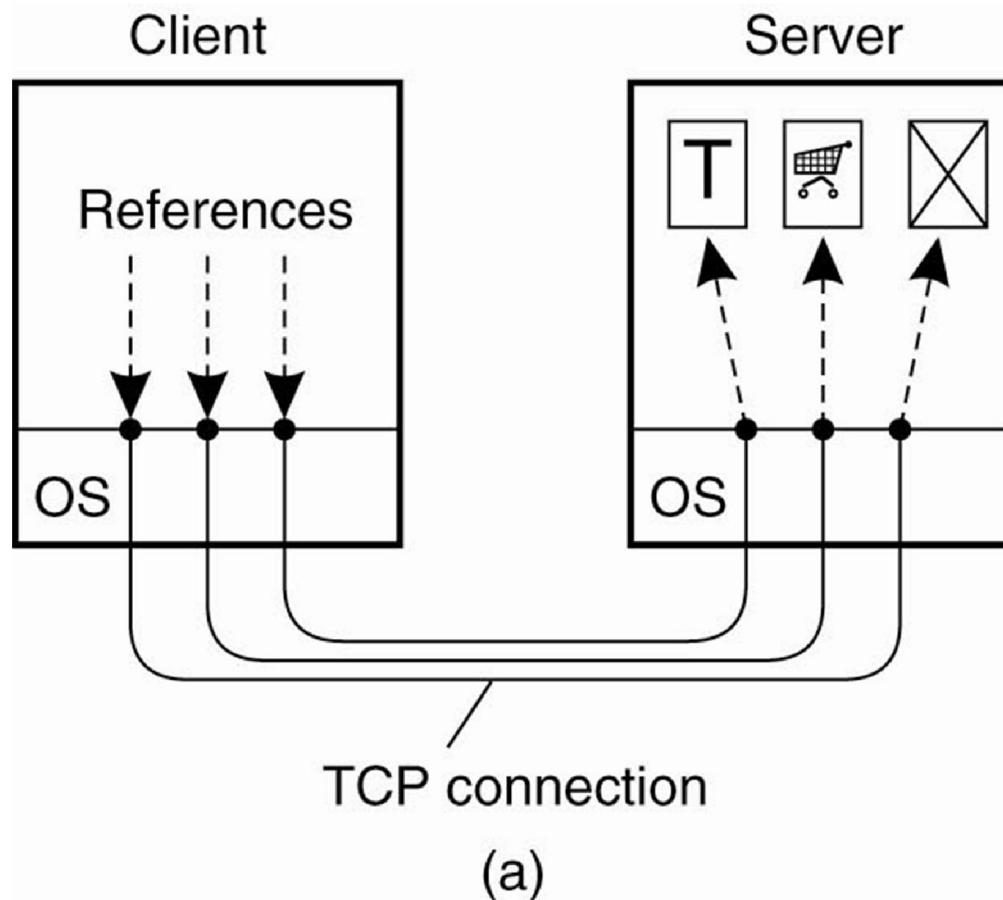


Figure 12-10. (a) Using nonpersistent connections.

HTTP Connections (2)

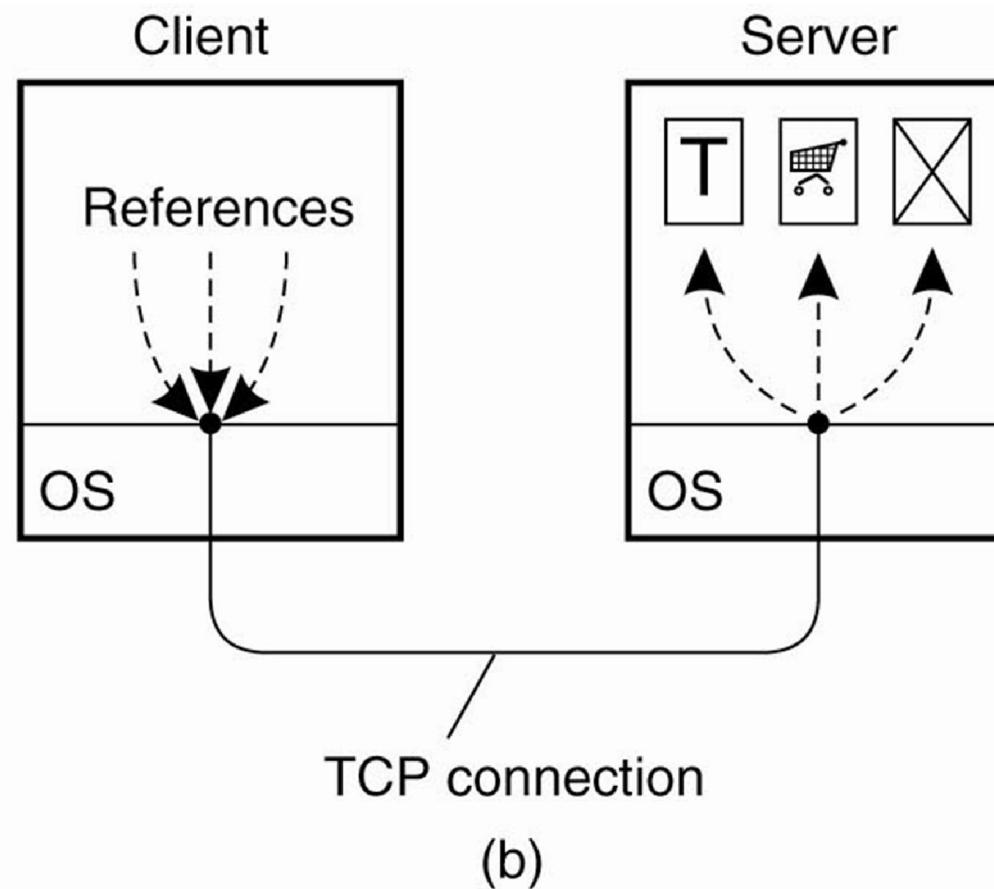


Figure 12-10. (b) Using persistent connections.

HTTP Methods

Operation	Description
Head	Request to return the header of a document
Get	Request to return a document to the client
Put	Request to store a document
Post	Provide data that are to be added to a document (collection)
Delete	Request to delete a document

Figure 12-11. Operations supported by HTTP.

HTTP Messages (1)

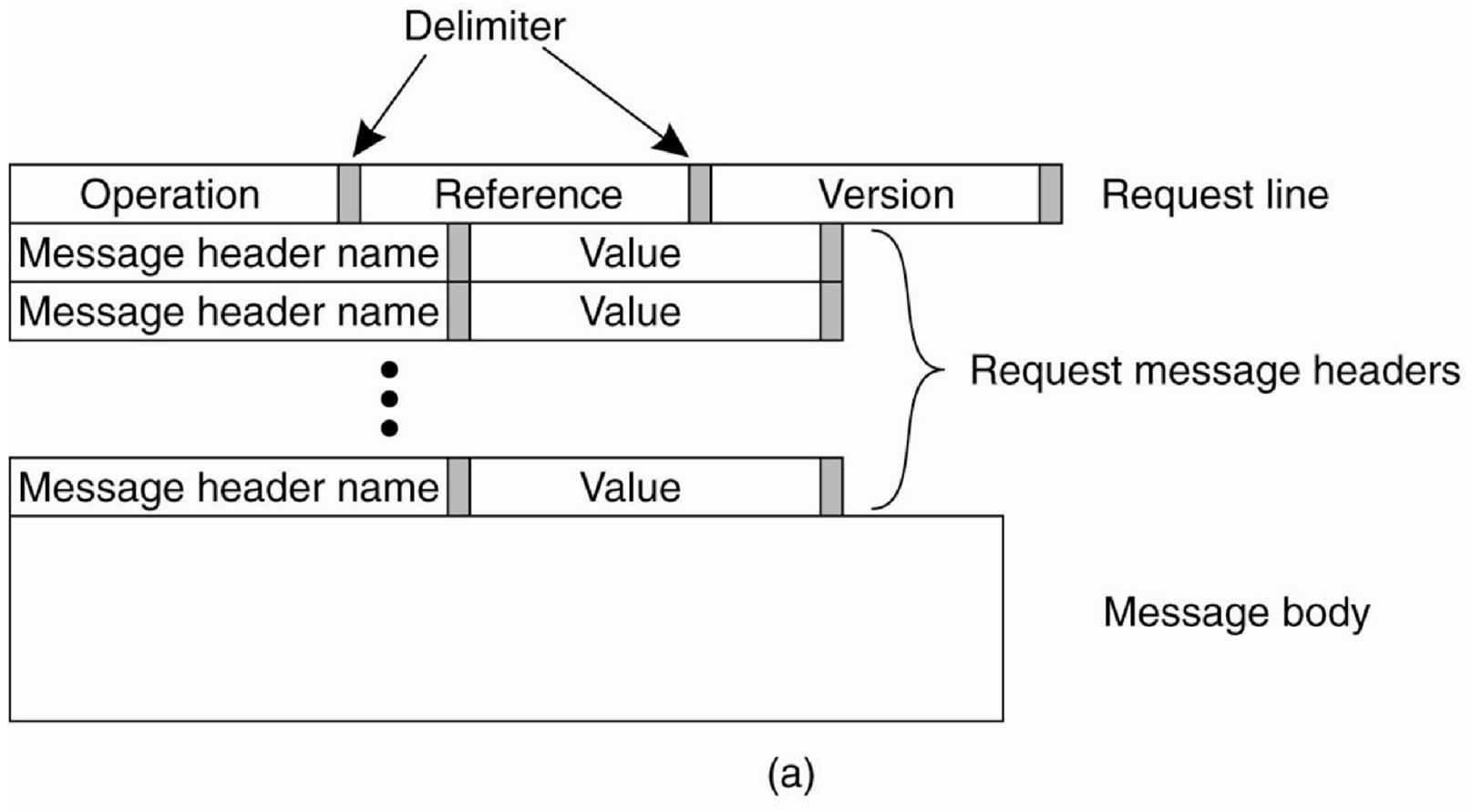


Figure 12-12. (a) HTTP request message.

HTTP Messages (2)

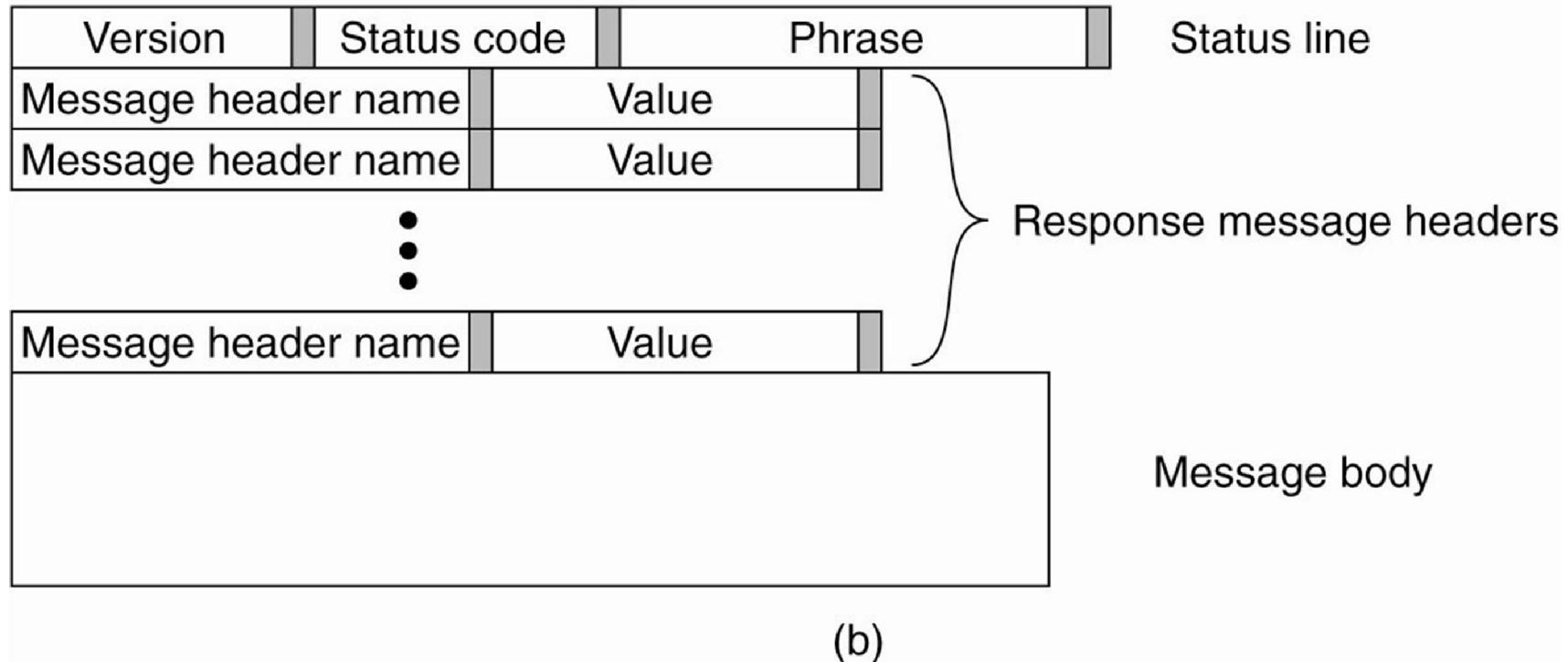


Figure 12-12. (b) HTTP response message.

HTTP Messages (3)

Header	Source	Contents
Accept	Client	The type of documents the client can handle
Accept-Charset	Client	The character sets are acceptable for the client
Accept-Encoding	Client	The document encodings the client can handle
Accept-Language	Client	The natural language the client can handle
Authorization	Client	A list of the client's credentials
WWW-Authenticate	Server	Security challenge the client should respond to
Date	Both	Date and time the message was sent
ETag	Server	The tags associated with the returned document
Expires	Server	The time for how long the response remains valid
From	Client	The client's e-mail address
Host	Client	The DNS name of the document's server

Figure 12-13. Some HTTP message headers.

HTTP Messages (4)

Header	Source	Contents
If-Match	Client	The tags the document should have
If-None-Match	Client	The tags the document should not have
If-Modified-Since	Client	Tells the server to return a document only if it has been modified since the specified time
If-Unmodified-Since	Client	Tells the server to return a document only if it has not been modified since the specified time
Last-Modified	Server	The time the returned document was last modified
Location	Server	A document reference to which the client should redirect its request
Referer	Client	Refers to client's most recently requested document
Upgrade	Both	The application protocol the sender wants to switch to
Warning	Both	Information about the status of the data in the message

Figure 12-13. Some HTTP message headers.

Simple Object Access Protocol

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Pick up Mary at school at 2pm</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

Figure 12-14. An example of an XML-based SOAP message.

Naming (1)

Scheme	Host name	Pathname
--------	-----------	----------

http :// www.cs.vu.nl /home/steen/mbox

(a)

Scheme	Host name	Port	Pathname
--------	-----------	------	----------

http :// www.cs.vu.nl : 80 /home/steen/mbox

(b)

Scheme	Host name	Port	Pathname
--------	-----------	------	----------

http :// 130.37.24.11 : 80 /home/steen/mbox

(c)

Figure 12-15. Often-used structures for URLs. (a) Using only a DNS name. (b) Combining a DNS name with a port number. (c) Combining an IP address with a port number.

Naming (2)

Name	Used for	Example
http	HTTP	http://www.cs.vu.nl:80/globe
mailto	E-mail	mailto:steen@cs.vu.nl
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	file:/edu/book/work/chp/11/11
data	Inline data	data:text/plain;charset=iso-8859-7,%e1%e2%e3
telnet	Remote login	telnet://flits.cs.vu.nl
tel	Telephone	tel:+31201234567
modem	Modem	modem:+31201234567;type=v32

Figure 12-16. Examples of URIs.

Web Proxy Caching

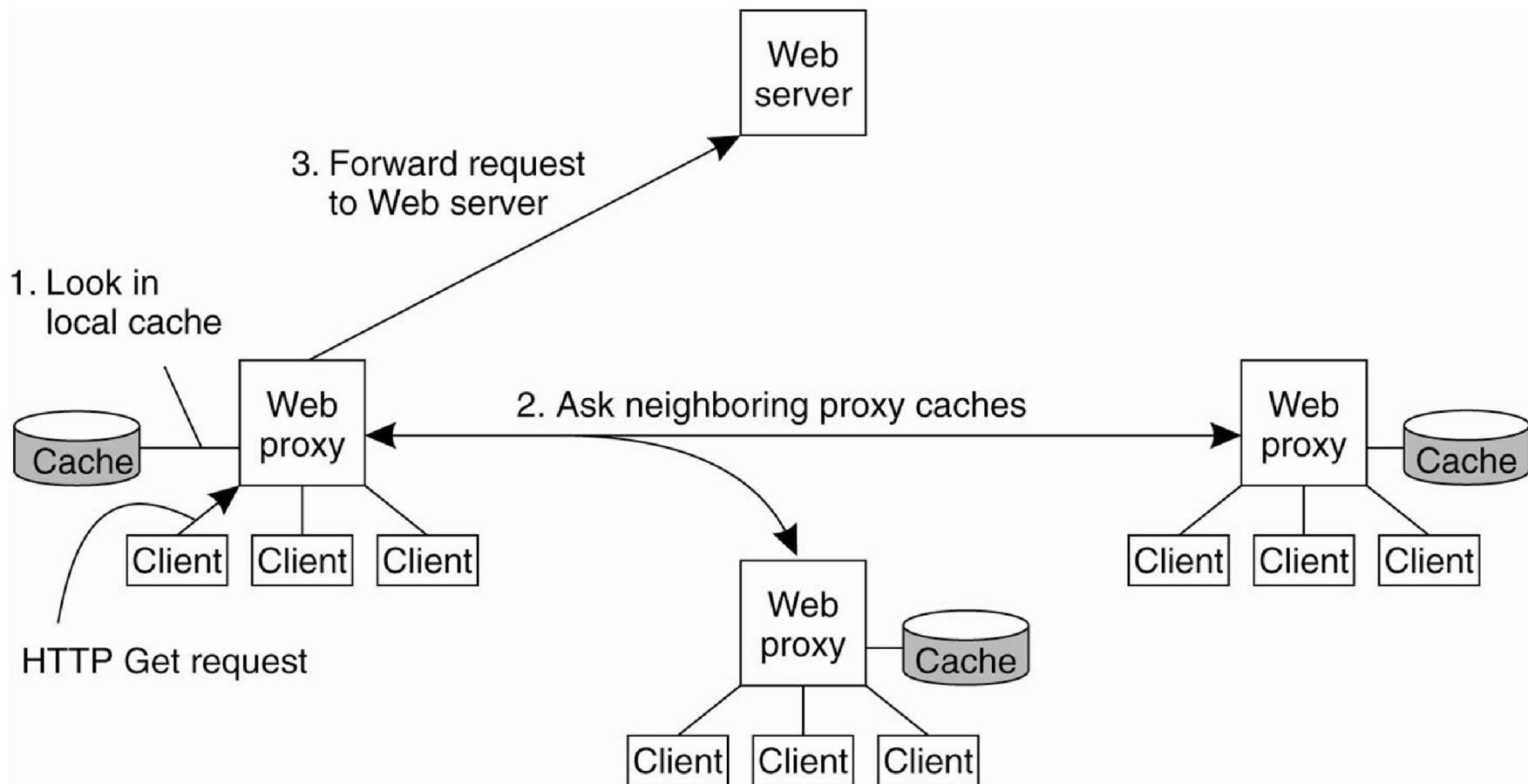


Figure 12-17. The principle of cooperative caching.

Replication for Web Hosting Systems

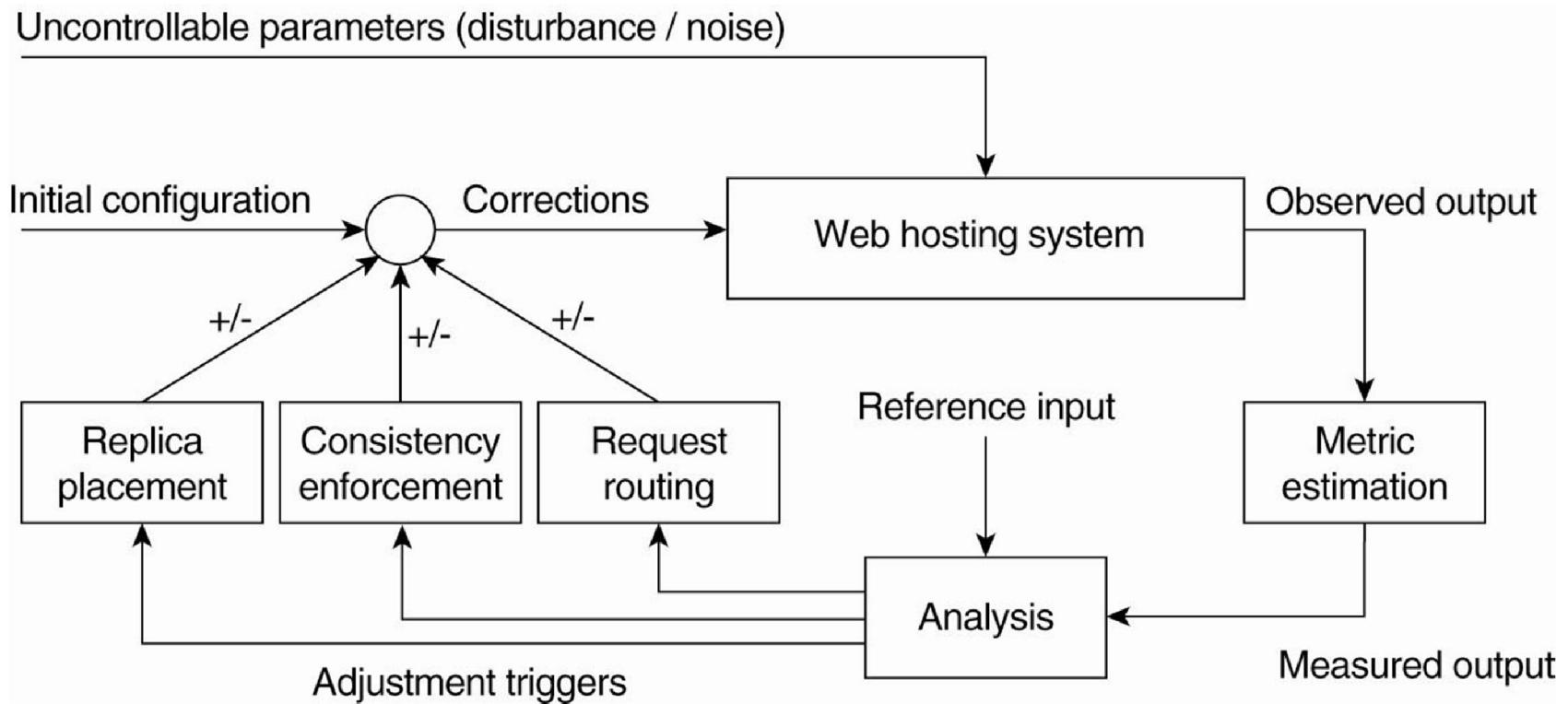


Figure 12-18. The general organization of a CDN as a feedback-control system (adapted from Sivasubramanian et al., 2004b).

Adaptation Triggering

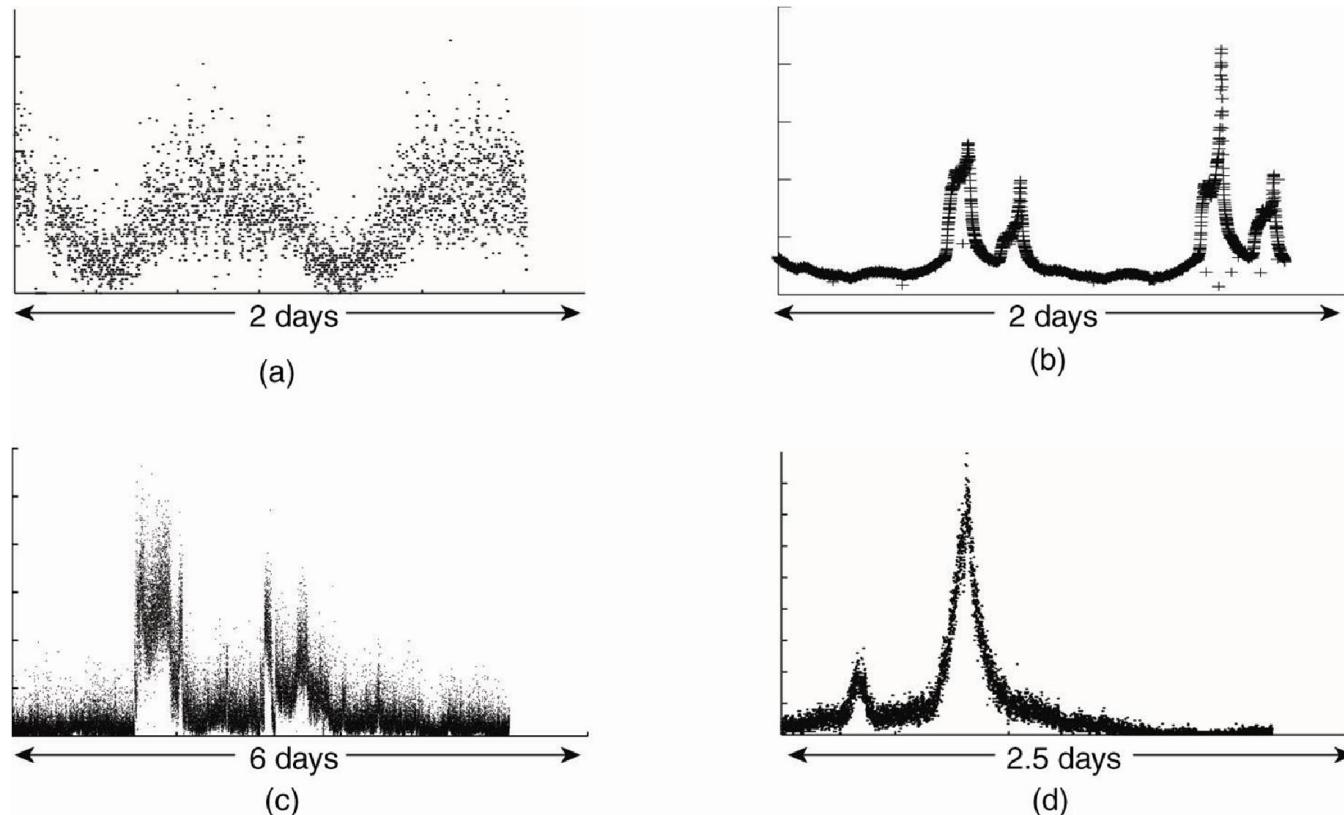


Figure 12-19. One normal and three different access patterns reflecting flashcrowd behavior
(adapted from Baryshnikov et al., 2005).

Adjustment Measures

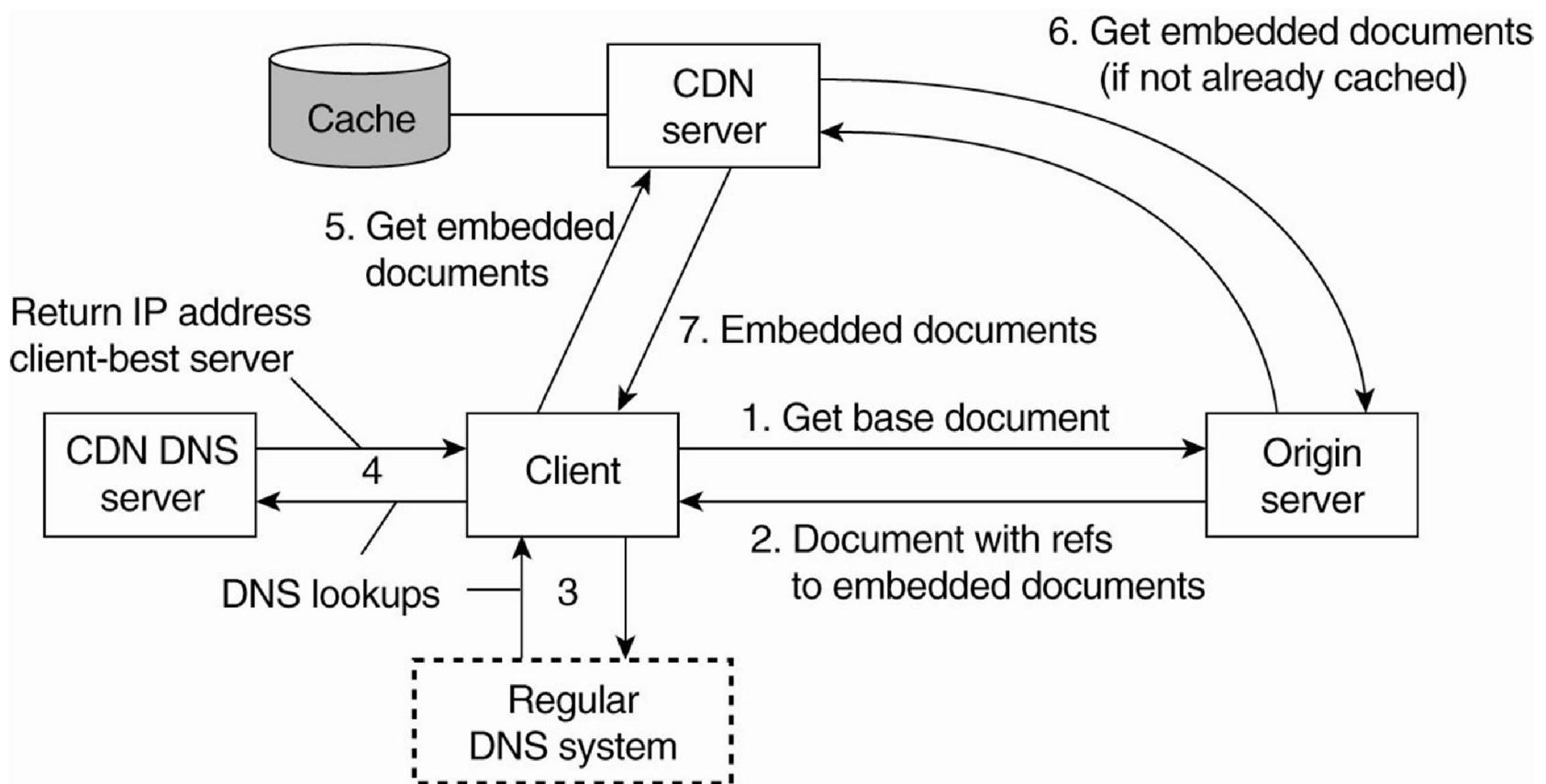


Figure 12-20. The principal working of the Akamai CDN.

Replication of Web Applications

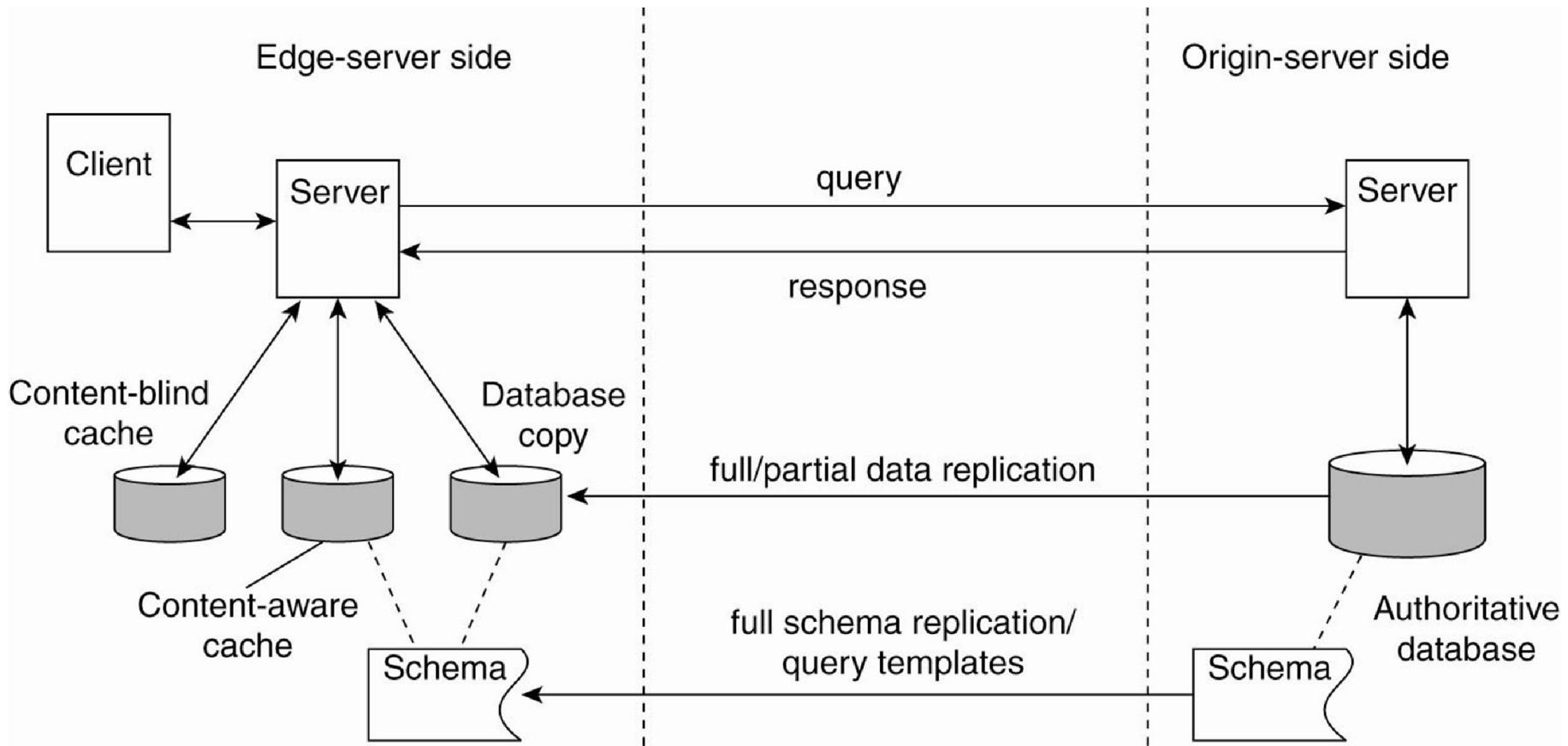


Figure 12-21. Alternatives for caching and replication with Web applications.

Security (1)

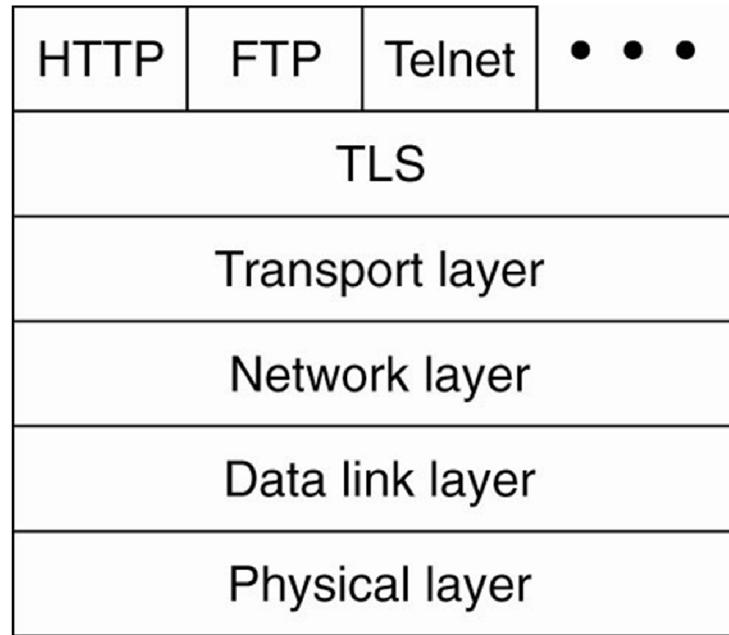


Figure 12-22. The position of TLS in the Internet protocol stack.

Security (2)

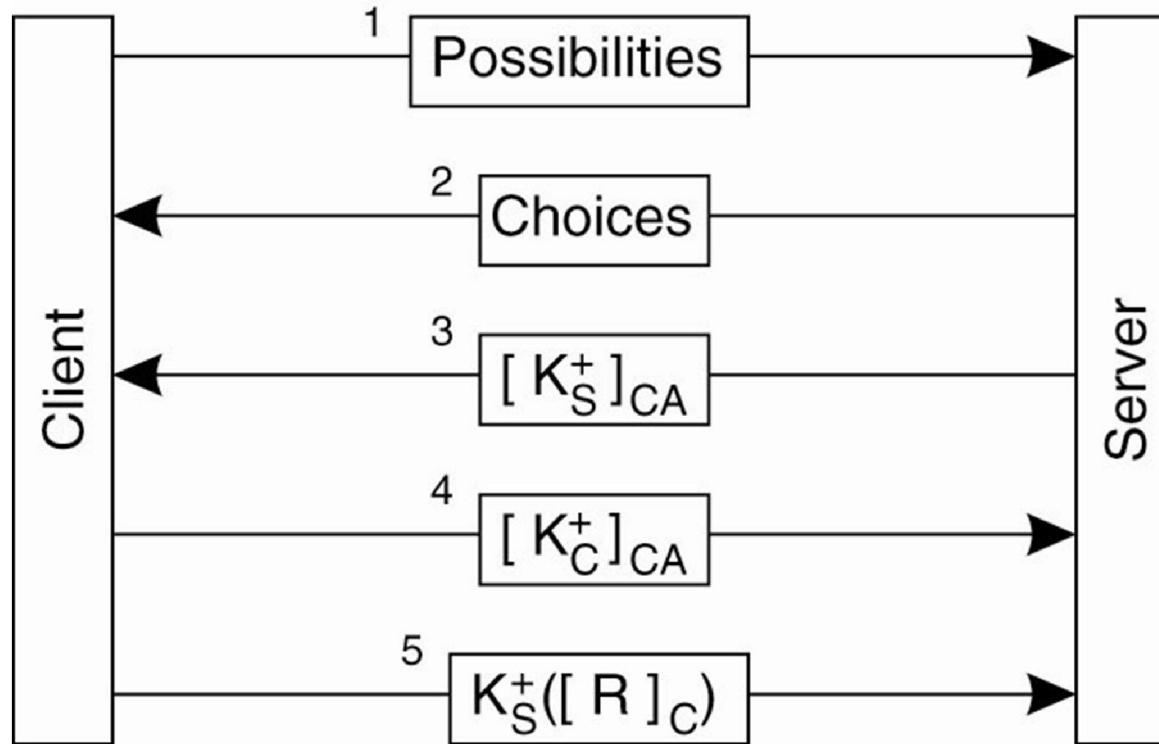


Figure 12-23. TLS with mutual authentication.