

CE4013/CZ4013 Distributed Systems

2021-2022 Semester 2

Instructor:

Tang Xueyan, asxytang@ntu.edu.sg,
N4-2a-31, 6790-5356

Textbook

Coulouris, Dollimore, Kindberg and Blair, "Distributed Systems: Concepts and Design", Addison-Wesley

4th edition

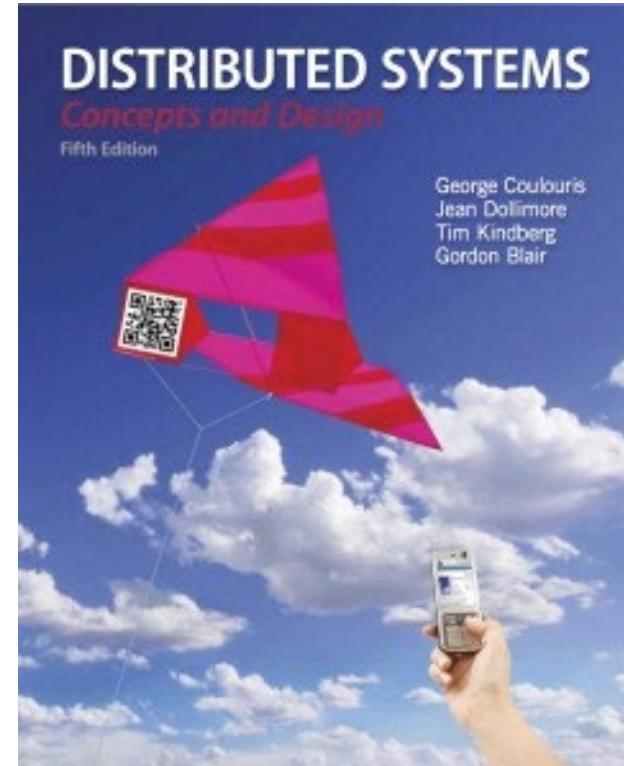


DISTRIBUTED SYSTEMS
CONCEPTS AND DESIGN

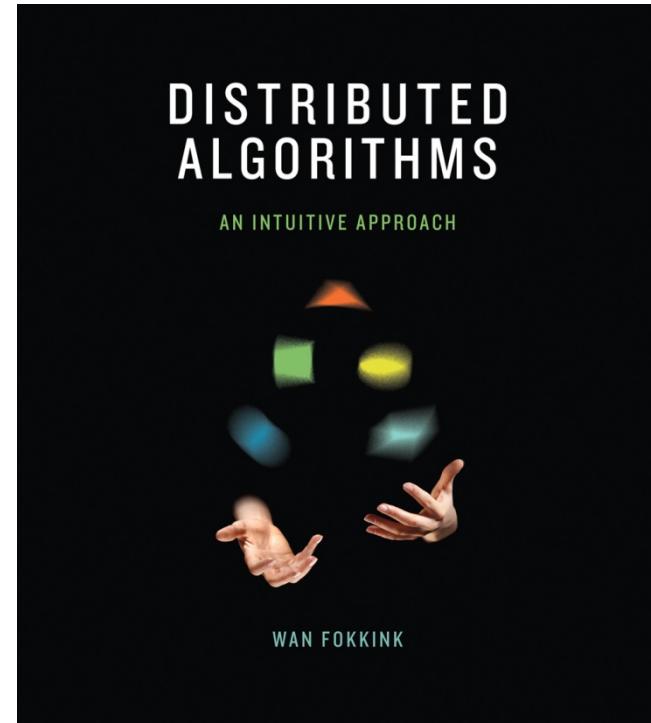
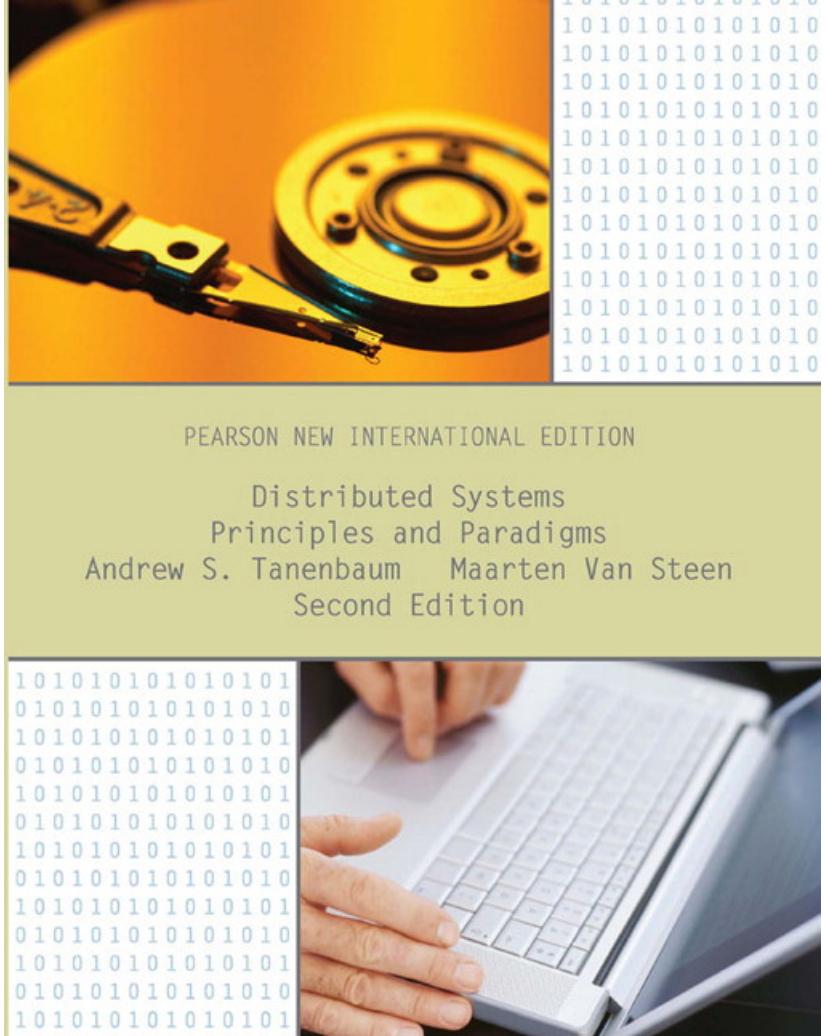
George Coulouris
Jean Dollimore
Tim Kindberg



5th edition



Reference Books



Approaches

- Lectures, tutorials, project and exam
- Roadmap
 - Foundation: Characterization of Distributed Systems, System Models, Interprocess Communication
 - Middleware: Distributed Objects & Remote Invocation, Name Services
 - System Infrastructure: Distributed File Systems, Peer-to-Peer Systems
 - Distributed Algorithms: Time and Global States, Coordination & Agreement (Case Study: Blockchain), Replication & Consistency

* Some figures in the lecture notes are from "Distributed Systems: Principles and Paradigms" (by Tanenbaum and Steen), Prentice Hall

1. Characterization of Distributed Systems & System Models

Outline

- What is a Distributed System (DS)?
- Fundamental characteristics of DS
- Main motivation of DS – resource sharing (the Web example)
- What are the issues and problems in DS?
- Architecture models
- Fundamental models
- Summary

What is a Distributed System?

- Human society evolves into a distributed system
- Structures/protocols that evolved, e.g.,
 - We live in manageable, autonomous units – families
 - We work in organizations; some members become specialized
 - To achieve different purposes, we cooperate and interact with each other in certain ways
 - Phone calls, sending emails, shouting in stock market etc.
 - We use clocks and landmarks to synchronize with each other (e.g. attending classes in lecture theatres)

What is a Distributed System?

- Computers/devices evolve into distributed systems
- Enabled by Internet, campus networks, home networks, corporate networks, mobile phone networks, ...
- Definition
 - A set of **networked computers**
 - that communicate and coordinate their actions only by **passing messages**
- Example
 - Internet: a very large distributed system, enabling users to make use of services like WWW, email & file transfer from everywhere

Outline

- What is a Distributed System (DS)?
- Fundamental characteristics of DS
- Main motivation of DS – resource sharing (the Web example)
- What are the issues and problems in DS?
- Architecture models
- Fundamental models
- Summary

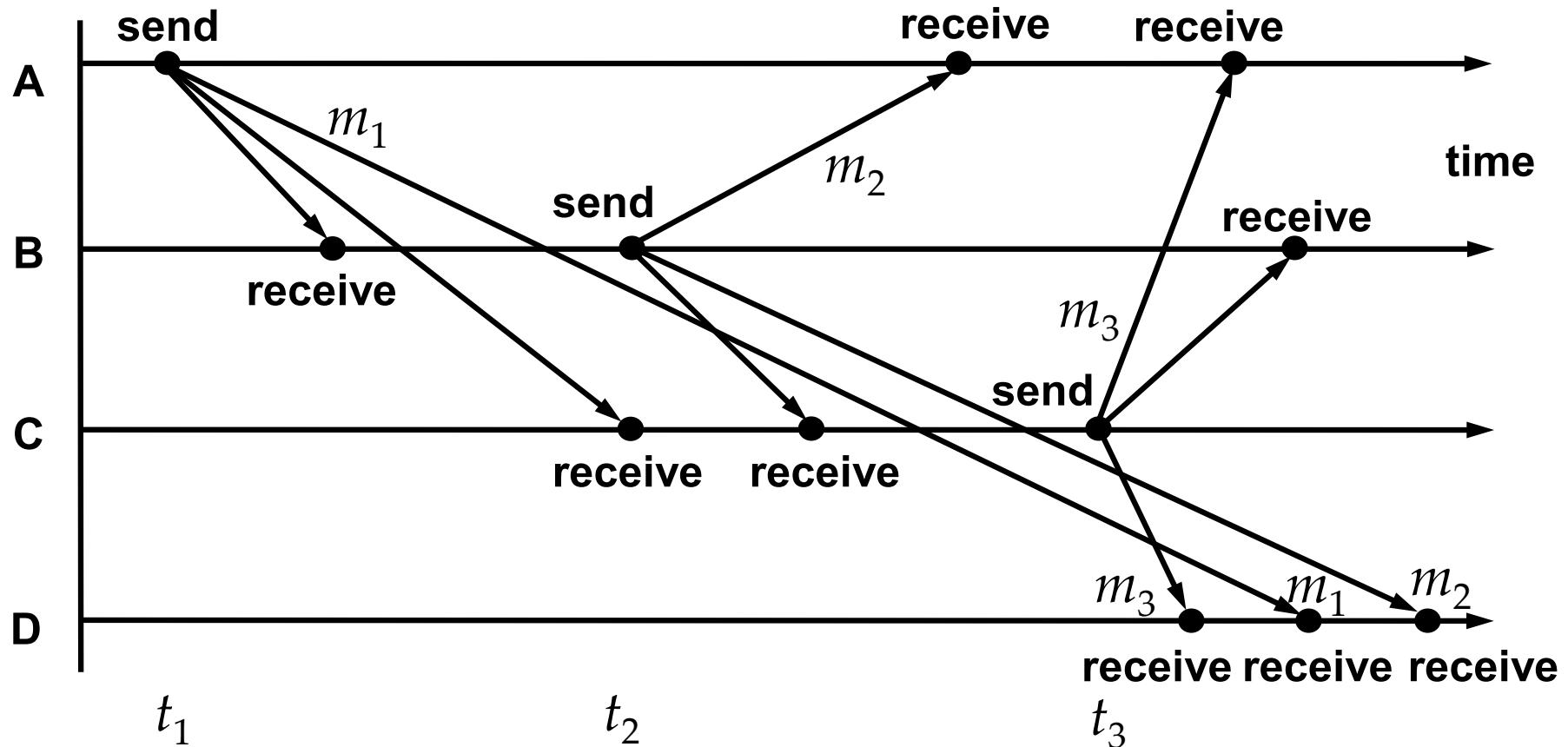
Fundamental Characteristics of DS

- Concurrency: concurrent program execution
 - Higher capacity → Possibly higher performance
 - System capacity can be increased by adding more resources (e.g., computers) to the network
 - Need to coordinate concurrently executing programs

Fundamental Characteristics of DS

- Loosely coupled:

- no global clock (difficult to synchronize perfectly, lack of certainty, difficult to design correct algorithms)



Fundamental Characteristics of DS

- Loosely coupled (cont'd)
 - no **global shared memory** (programs interact by passing messages → to reduce communication in design)
 - vs. **tightly coupled**: coordinate through global memory, synchronization through global clock (e.g., a multi-core processor)

Fundamental Characteristics of DS

- Independent failures: any computer (including its software/hardware components) and subnetwork can **fail at anytime**
 - Detection may be difficult or even impossible
 - However, distributed systems can be made **more fault-tolerant** than stand-alone systems if properly designed

Outline

- What is a Distributed System (DS)?
- Fundamental characteristics of DS
- Main motivation of DS – resource sharing (the Web example)
- What are the issues and problems in DS?
- Architecture models
- Fundamental models
- Summary

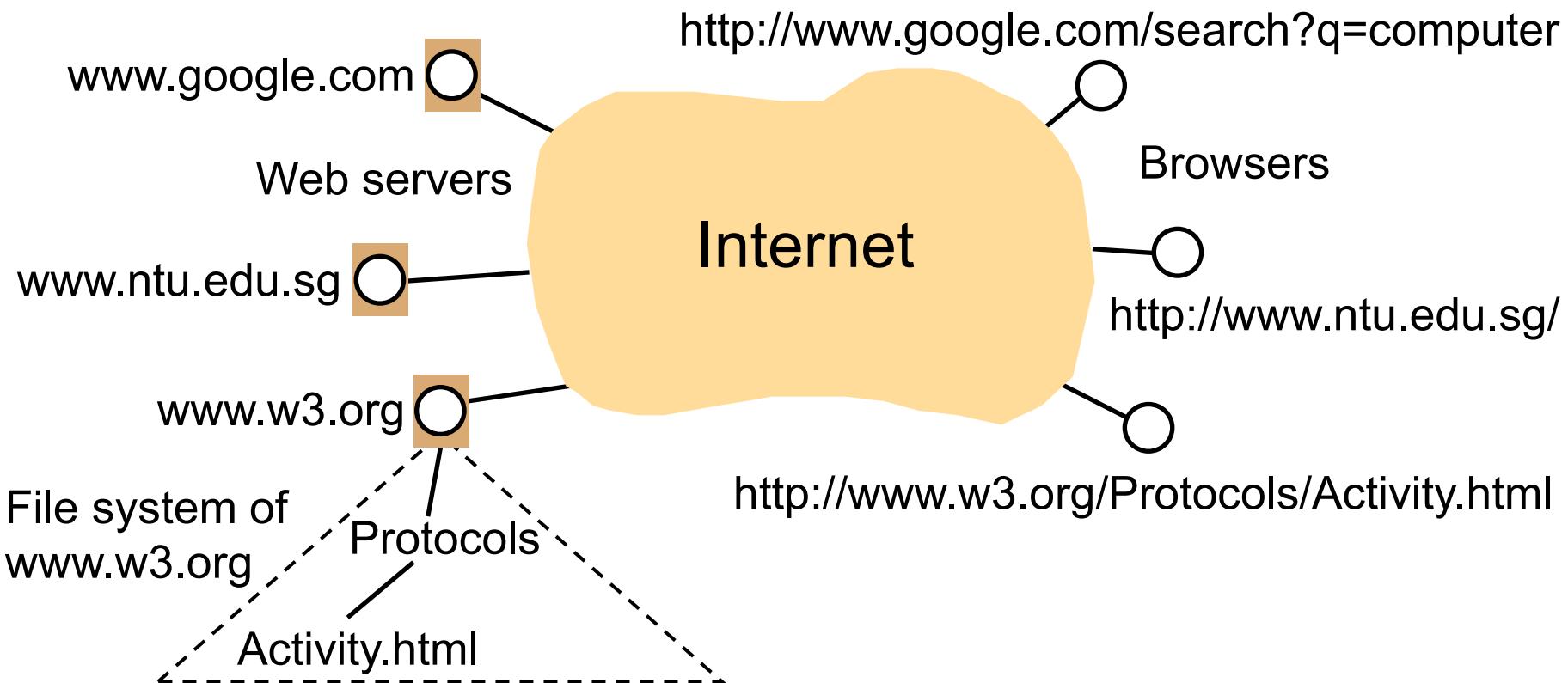
Distributed Services & Resource Sharing

- Resources may be in the form of
 - Information (e.g., text/picture/audio/video, news, email)
 - Hardware (e.g., computing power, storage, cameras, printers)
 - Software (e.g., system software, application programs)
- Benefits: save cost and man-power

Distributed Services & Resource Sharing

- How are resources shared? – via services
 - Service: a distinct part of a computer system **providing accesses to the managed resources**
 - e.g., read/write/delete operations on files
 - **Server:** the process managing resources (e.g., web server)
 - **Client:** the process requesting to access resources (e.g., web browser)
 - The server accepts requests from clients, performs services, and responds to clients
 - Requests and replies are sent in the form of messages
 - **Remote invocation:** a complete interaction between a client and a server, from the point when the client sends its request to the point when it receives the server's response

Example: The Web



Outline

- What is a Distributed System (DS)?
- Fundamental characteristics of DS
- Main motivation of DS – resource sharing (the Web example)
- **What are the issues and problems in DS?**
- Architecture models
- Fundamental models
- Summary

Challenges of Distributed Systems

- Heterogeneity – hardware and software components are not identical
 - Networks (Ethernet, ISDN connections, modem connections, wireless LAN, infra-red, bluetooth, etc.)
 - Masked by all computers using the Internet protocols to communicate with one another
 - Computer hardware (supercomputers, desktop PCs, laptops, tablet computers, wearable devices, etc.)
 - For example, two alternatives for byte orderings: big-endian and little-endian
 - How to exchange data between programs running on different hardware?

Challenges of Distributed Systems

- Heterogeneity (cont'd)
 - Operating systems (Windows, Unix, etc.)
 - Different application programming interfaces
 - e.g., different system calls for exchanging messages in UNIX and Windows
 - Programming languages
 - e.g., different representations for characters, arrays ...
 - How can programs written in one language communicate with those in another language?

Challenges of Distributed Systems

- Scalability: to remain effective when there is a significant increase in number of resources & users
 - Computers and web servers in the Internet (according to surveys by Internet Systems Consortium and Netcraft)

Date	Computers	Web Servers
July, 1993	1,776,000	130
July, 1995	6,642,000	23,500
July, 1999	56,218,000	6,598,697
July, 2002	162,128,493	37,235,470
July, 2006	439,286,364	88,166,395
July, 2010	768,913,036	205,714,253
July, 2015	1,033,836,245	849,602,745

Challenges of Distributed Systems

- Scalability (cont'd)
 - Approaches to handle increasing demand
 - Caching and replication of data
 - Deploy multiple servers
 - Avoid performance bottlenecks
 - Prefer decentralized design to centralized design
 - Prevent software resources from running out
 - e.g., length of Internet (IP) addresses

Challenges of Distributed Systems

■ Failure handling

- Detecting (e.g., checksums to detect corrupted messages)
- Masking (e.g., retransmit messages when they are lost)
- Tolerating (**use redundancy to achieve fault tolerance**, e.g., store the same files on two disks)
- Recovery (“roll back” when a server crashes, not to leave data in an inconsistent state)

Outline

- What is a Distributed System (DS)?
- Fundamental characteristics of DS
- Main motivation of DS – resource sharing (the Web example)
- What are the issues and problems in DS?
- **Architecture models**
- Fundamental models
- Summary

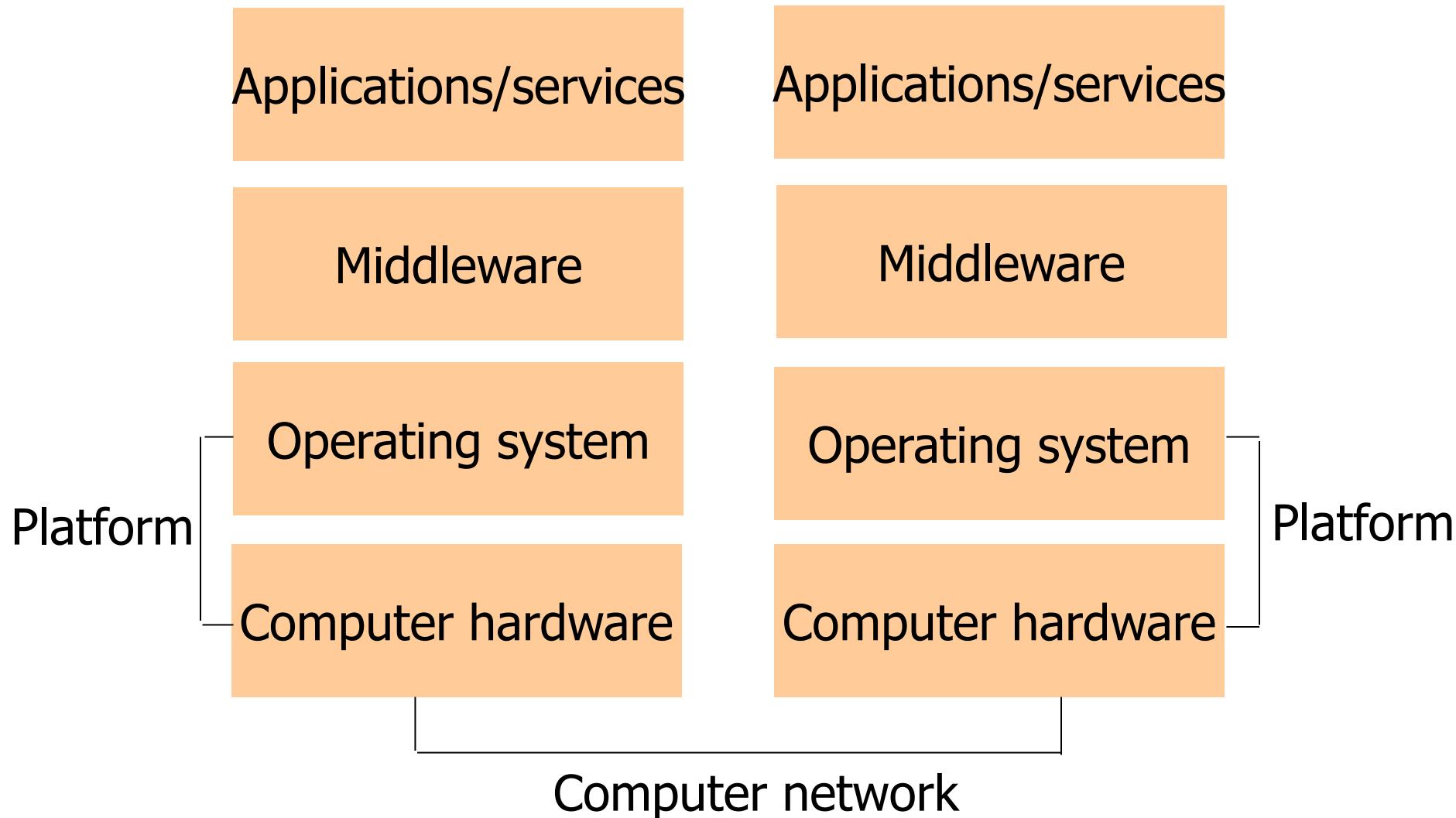
Models

- An abstract, simplified but consistent description
 - Example: Von Neumann's model for computers
- Architectural models:
 - Identify components; describe functions; placement of components; relationship between components (e.g., how do they interact with one another?)
- Fundamental models:
 - More formal description of **common and intrinsic properties** in architectural models
 - Deal with correctness, reliability and security
 - Make/verify exact claims by logical analysis and mathematical proof

Architectural Models

- Software and hardware layers ("vertical")
 - Structuring of system as layers/modules in terms of services offered and requested between processes located in the same or different computers
- System architectures ("horizontal")
 - Client-server model
 - Peer-to-peer model

Software & hardware service layers in DS



Platform

- Lowest two levels (hardware and OS) → **platform**
 - Bring the system's programming interface up to a level that facilitates communication and coordination between processes
 - e.g., Intel x86/Windows, Sun SPARC/SunOS

Middleware

- A software layer to mask heterogeneity and provide a convenient programming model
 - Heterogeneity in underlying network protocols, hardware, operating systems, programming languages
- Provide **generic services** to applications, e.g.,
 - Naming: e.g., add/search name-to-address mappings
 - Security: e.g., authentication
 - Persistent storage: e.g., store and retrieve objects
- Example:
 - Sun RPC, Java RMI
 - CORBA (Common Object Request Broker Architecture)

Architectural Models (Revisit)

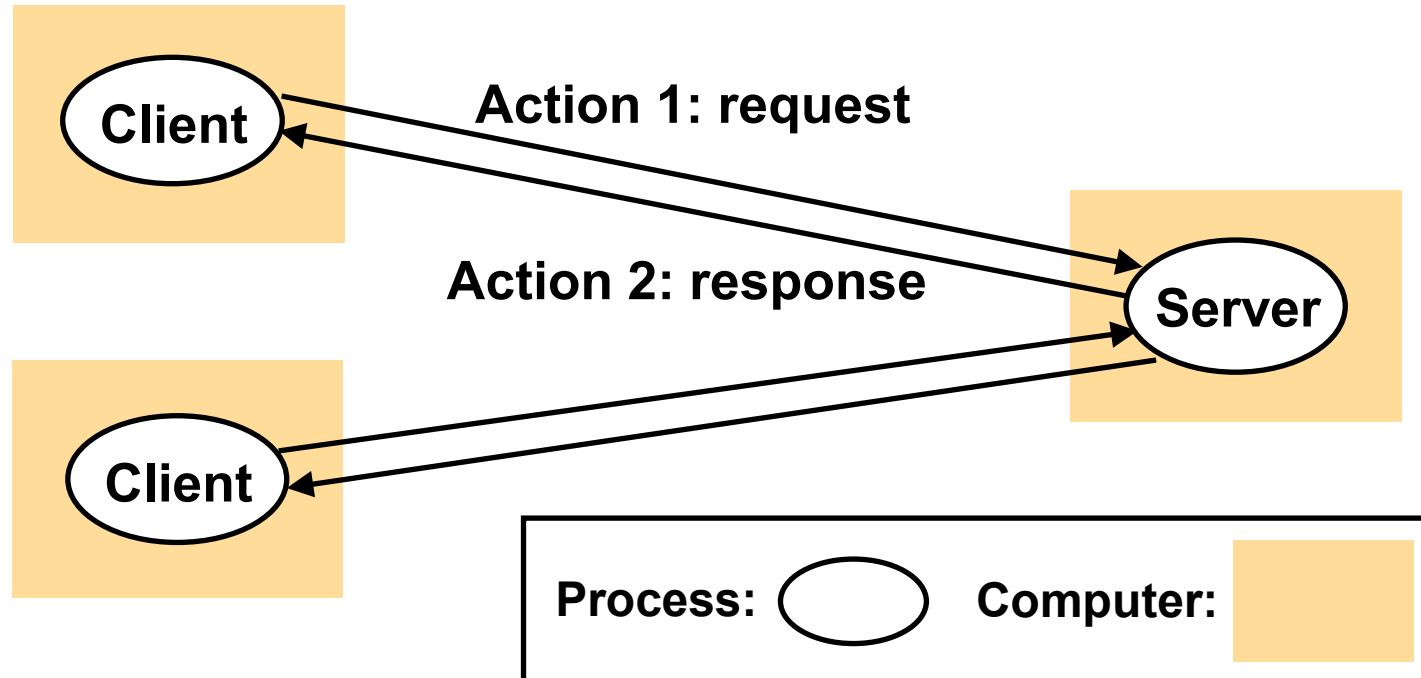
- Software and hardware layers ("vertical")
- System architectures ("horizontal")
 - Client-server model
 - Peer-to-peer model

Client-Server Model

- Server: processes that **manage resources to provide a service**, they accept requests from clients and respond
- Client: processes that **use/access the service**
- Client-server is a simple yet useful model for developing distributed systems

Simple Client-Server Model

- Simple request-response interactions



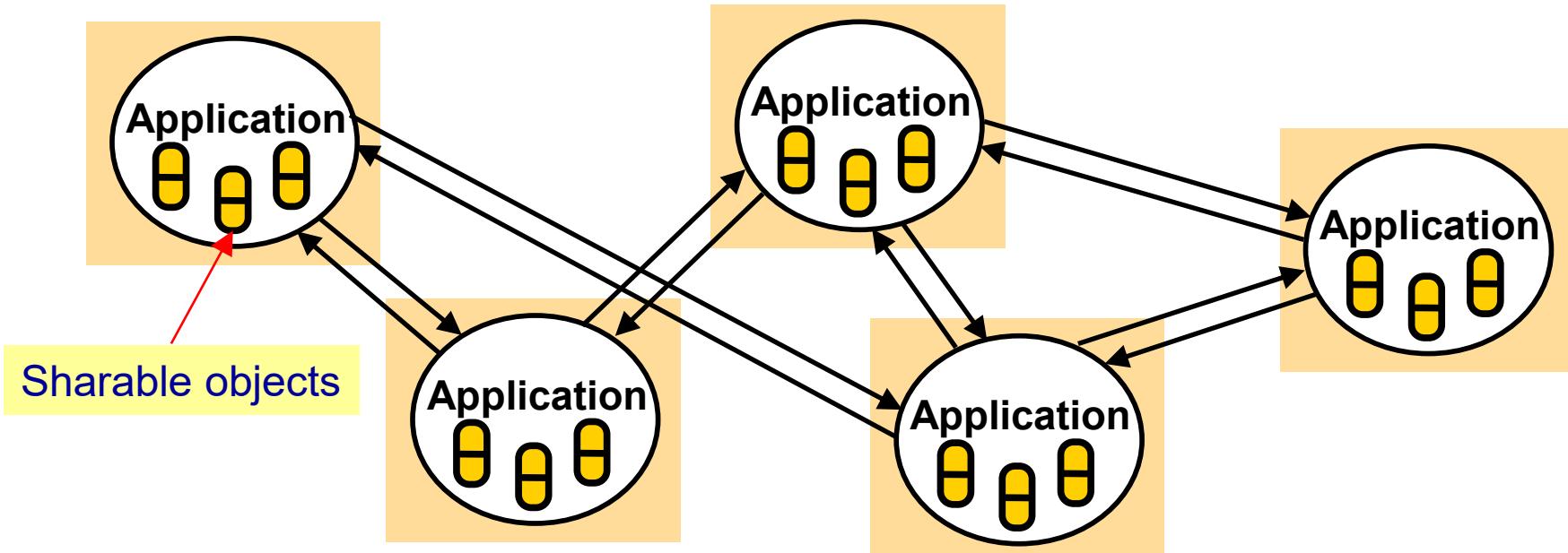
- N clients accessing the same service provided by a server: N-to-1

Variation of Client-Server Model

More generally:

- Services may be provided by multiple servers
- M servers working together for N clients
- Division of work among M servers:
 - partitioning: function (cooperative servers) or data
 - e.g., each server manages its own set of resources,
www.cnn.com, money.cnn.com, sports.cnn.com
 - replication: function (identical servers) or data
 - e.g., servers manage the same set of resources,
www.google.com, www.google.com.sg,
www.google.com.hk

Peer-to-Peer Model



- All processes play **similar roles**, they interact cooperatively as peers to perform distributed computation
- No distinction between clients and servers
- e.g., chatting, whiteboard, p2p file sharing
- Peer-to-peer architecture is substantially more complex than client-server architecture

Outline

- What is a Distributed System (DS)?
- Fundamental characteristics of DS
- Main motivation of DS – resource sharing (the Web example)
- What are the issues and problems in DS?
- Architecture models
- **Fundamental models**
- Summary

Fundamental Models

- Revisit: more formal description of **common and intrinsic properties** in architectural models
- Interaction Model
 - Assumptions about behaviors in terms of time
- Failure Model
 - Define and classify faults
 - Basis for designing systems to tolerate faults

Interaction Model

- **Synchronous distributed system:** assume upper/lower bounds on
 - processing time (of each step of a program)
 - transmission time (communication performance is often a limiting factor for the accuracy of coordination)
 - clock drift rate (computer clocks drift away from perfect time)
- Synchronous distributed systems can be built (e.g., reserve resources if requirements are known), but difficult
- Allow us to **infer properties** → see next slide

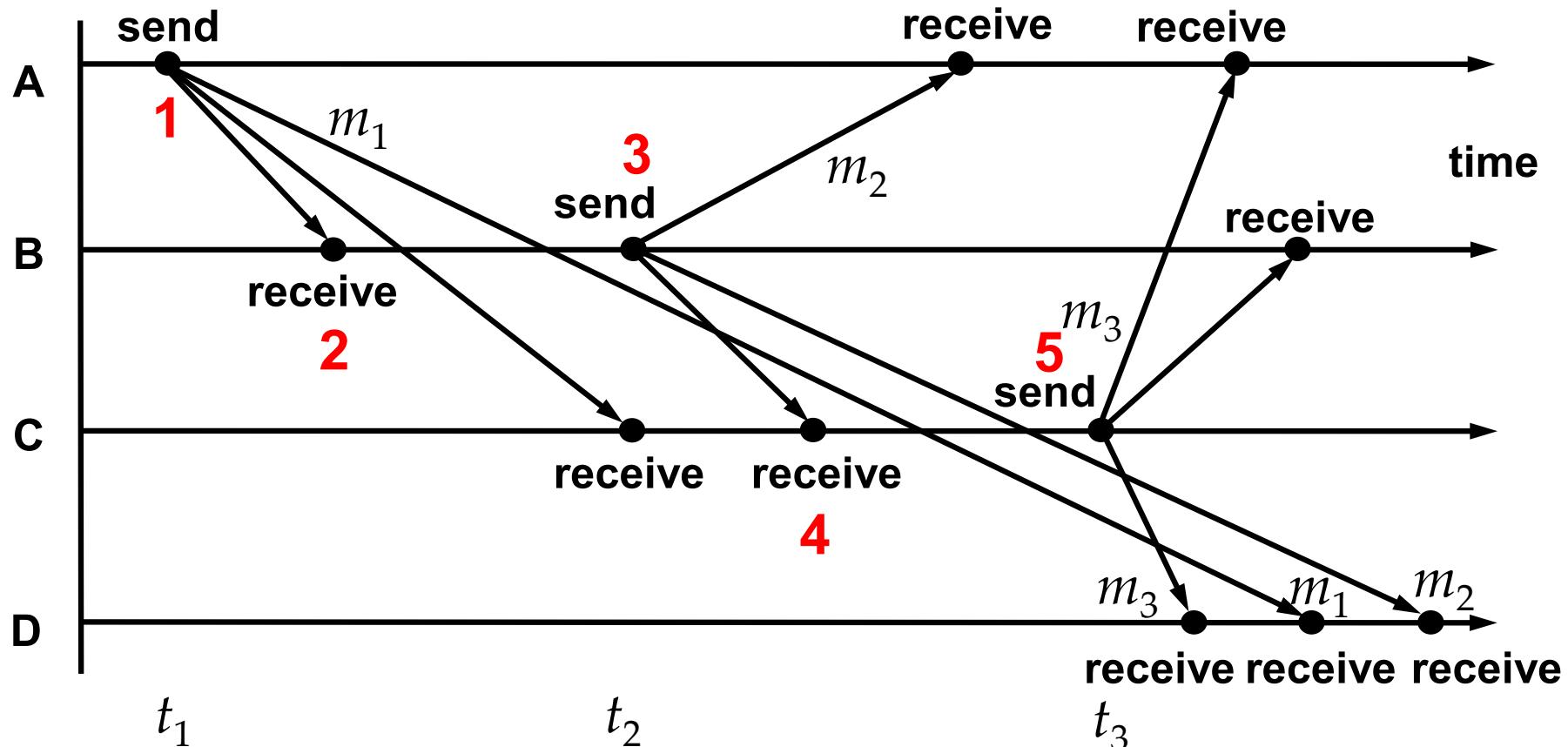
Interaction Model

- Allow us to **infer properties** such as:
 - Use timeouts to detect failures
 - Assume transmission time is in $[t_{\min}, t_{\max}]$, computer 1 sends a message to computer 2, then waits t_{\min} before starting a program, computer 2 starts a program on receiving the message, then the program on computer 2 must be within a lag of $t_{\max} - t_{\min}$ from that on computer 1

Interaction Model

- **Asynchronous distributed system:** assume no bounds on
 - processing time (of each step of a program)
 - transmission time
 - clock drift rate
- Asynchronous distributed systems exactly model the Internet due to sharing of processors and networks
- Does **NOT** allow us to infer properties such as:
 - Using timeouts to detect failures
- But does allow us to infer other useful properties
→ see next slide

Example: Ordering of Events



- For any message m , event $Send(m)$ must precede $Receive(m)$
- Replies are sent after receiving messages
- Therefore, it can be inferred that $Send(m_1)$ precedes $Send(m_3)$

Failure Model

- Define the ways in which failures may occur
- **Omission Failures:** a process or communication channel fails to perform actions it is supposed to do
- Process omission failure – **crash:** process halts and will not execute any further steps of its program
 - Detecting crashes is possible in synchronous systems, but is difficult in asynchronous systems
- Communication omission failure – a message sent by one process is not delivered to the other process

The Generals' Paradox

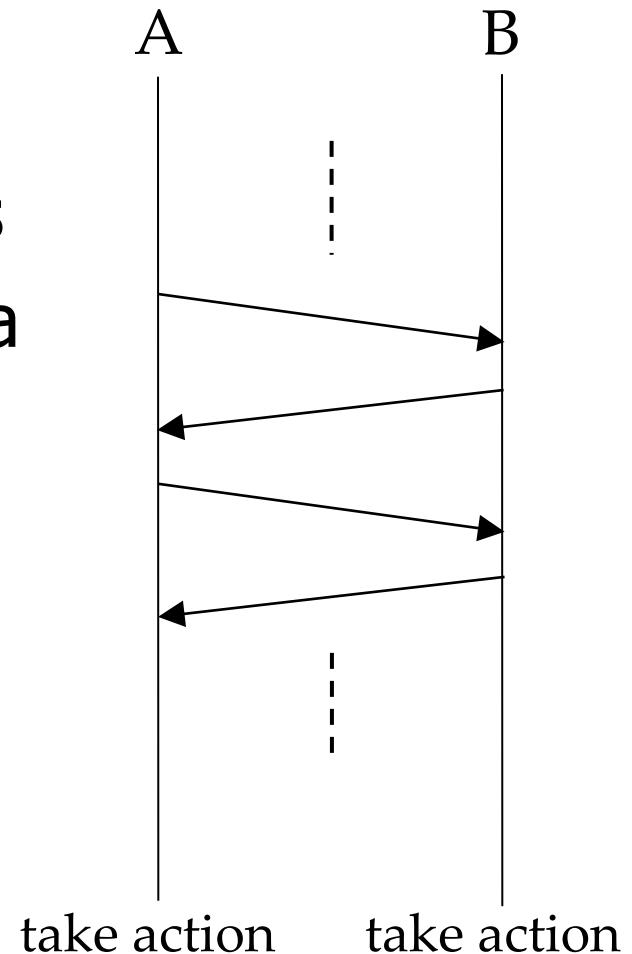
- Let's look at an example to illustrate the effects of failures on the design of a distributed system
- Problem
 - Two Pepperland army divisions: "Apple" & "Orange"
 - Their enemies "Blue Meanies" are located in between
 - Suppose "Blue Meanies" can capture the messages sent between "Apple" and "Orange" and prevent them from arriving at the destination
 - Can we design a protocol for "Apple" and "Orange" to consistently decide to both charge at the "Blue Meanies" or both surrender?
- It is impossible to design such a protocol

Formal Definition of the Problem

- Two processes A and B communicate by passing messages on a bidirectional channel
- Neither process can fail
- But there exist communication omission failures
 - A message sent from either process may be lost in the channel before reaching the other process
- Each process wants to select and take one of two actions x and y
- Can we devise a protocol for both processes to agree and take the same action?
- This problem has no solution – how to prove it?

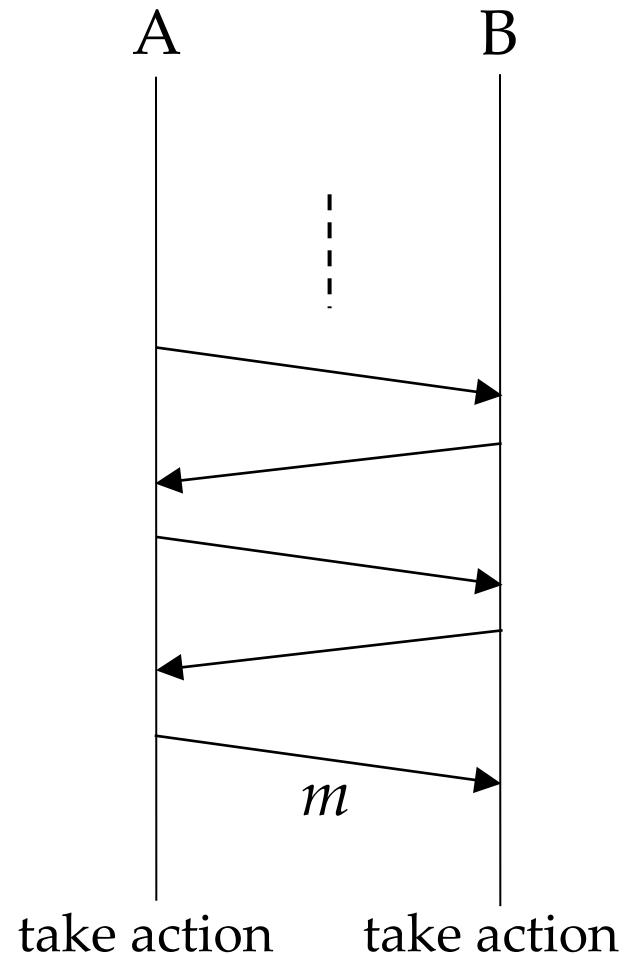
Proof

- Any protocol is equivalent to one in which A and B exchange messages in rounds
- Our approach: if there exists a protocol to solve the problem
→ derive a contradiction



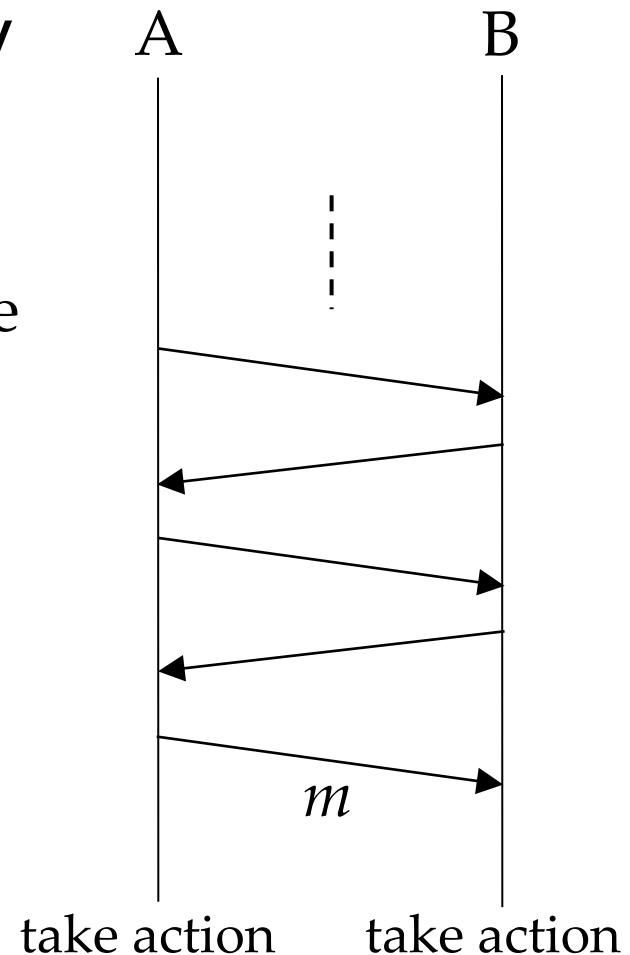
Proof

- Among all solutions, consider the protocol that solves the problem using the fewest message exchanges
 - No protocol solving the problem uses fewer message exchanges
- Without loss of generality, assume in this protocol, the last message m is sent by A



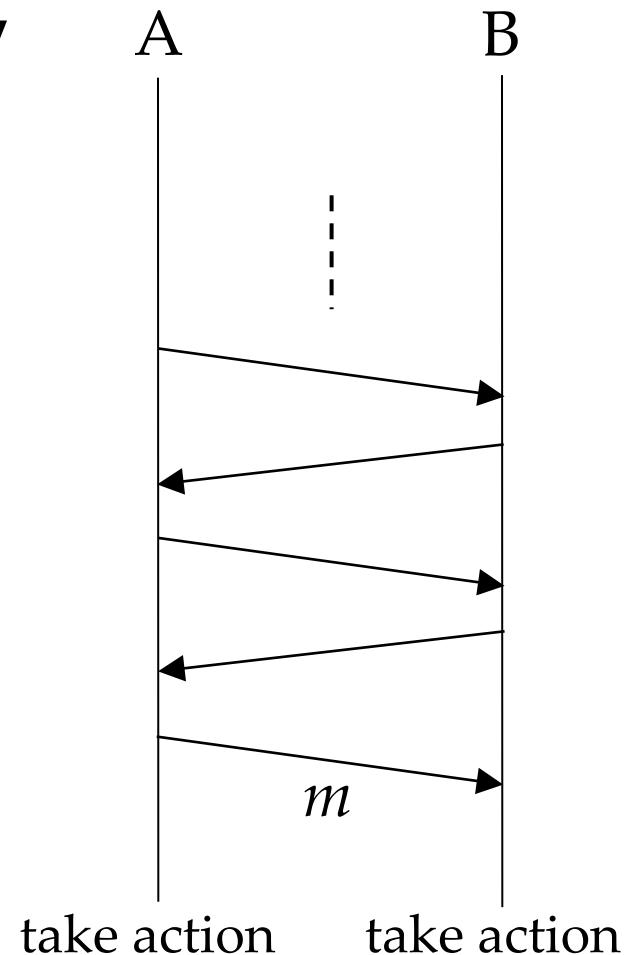
Proof

- The action ultimately taken by A cannot depend on whether m is received by B
 - communication-omission failure
→ m might be lost
 - m is the last message → B does not send any message after m
→ the receipt of m can never be learned by A
- So, A's choice of x or y does not depend on m



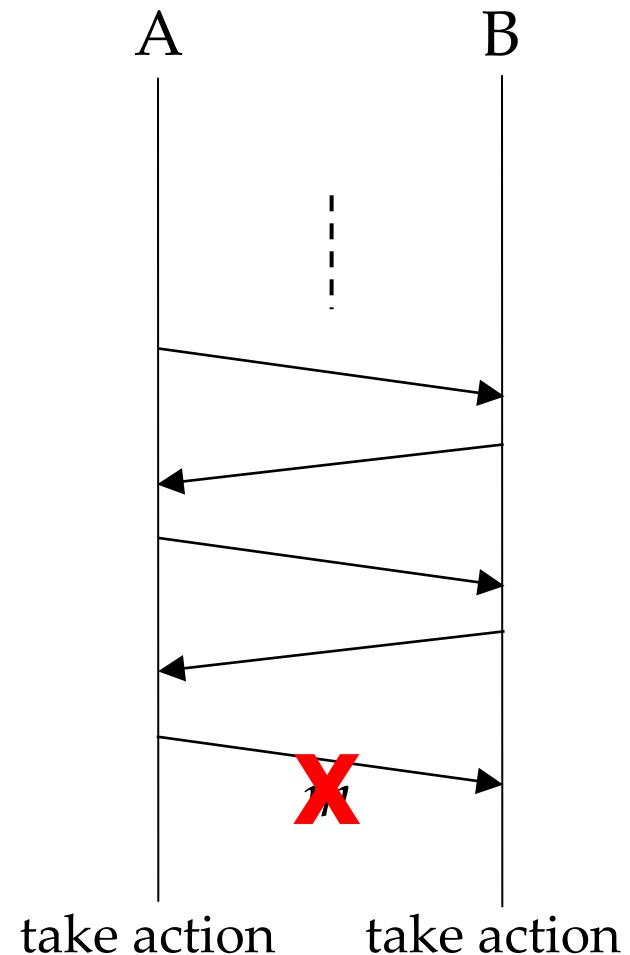
Proof

- Similarly, the action ultimately taken by B cannot depend on whether m is received by B
 - correct solution \rightarrow A and B always take the same action
 - Since A's choice of action does not depend on m , B must make the same choice of action regardless of whether m is lost or not
- So, B's choice of x or y does not depend on m



Proof

- The actions chosen by A and B do not depend on m
- Therefore, m is unnecessary
→ we can discard it
- Now, we get a new protocol with one fewer message exchange → contradiction



Failure Model

- **Arbitrary (Byzantine) Failures**
 - Do not simply fail but cause more trouble
 - Arbitrarily omit intended processing steps and take **unintended** processing steps
 - Processes, e.g.,
 - A process may set wrong values to its data items
 - It may return a wrong result in response to an invocation
 - Communication, e.g.,
 - Corrupted messages may arrive at the destination
 - A message may be delivered more than once

Summary

- Distributed systems
 - A set of networked computers
 - communicate only by passing messages
- Fundamental characteristics of distributed systems: concurrency, loosely coupled, independent failures
- Resource sharing is the main motivation for constructing distributed systems
 - Typically, resources are managed by servers and accessed by clients

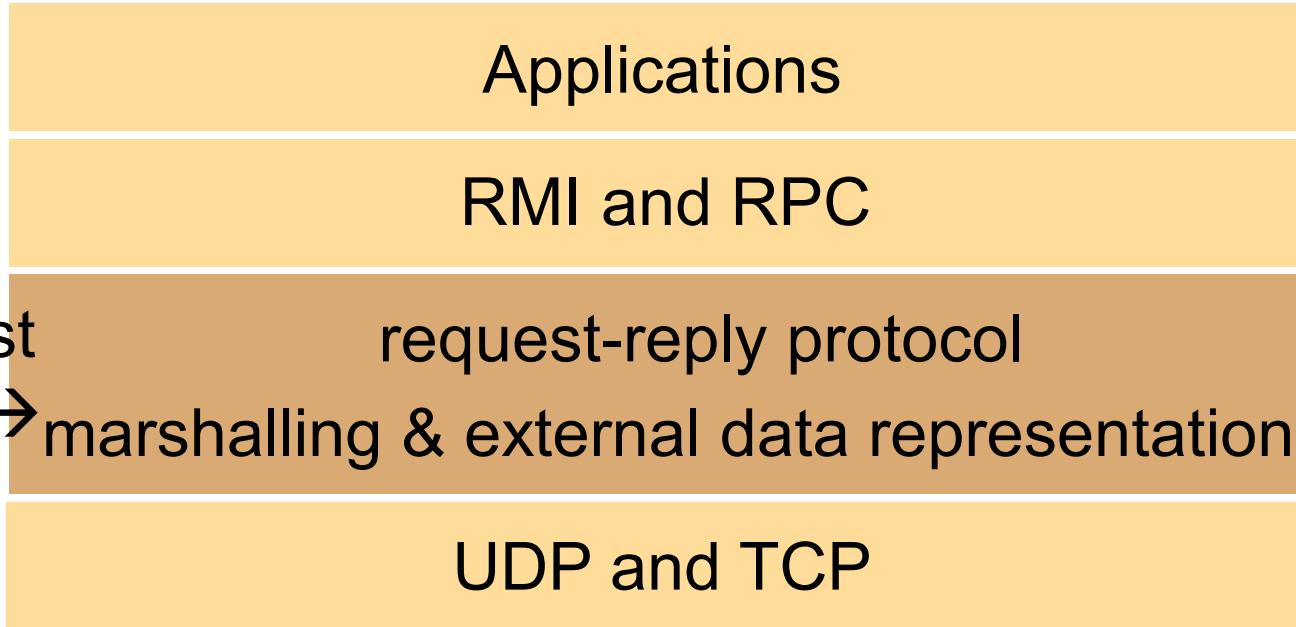
Summary

- Architectural models:
 - client-server & peer-to-peer
- Interaction models: assumptions about behaviors in terms of time
 - synchronous/asynchronous distributed systems
- Failure models: faults of processes & communication channels
 - omission and byzantine failures

2. Interprocess Communication

Middleware Layers

Let's first look at → request-reply protocol
marshalling & external data representation



Outline

- External Data Representation & Marshalling
- Client-Server Communication
- Summary

Marshalling and Unmarshalling

- Irrespective of the communication form used, data are transmitted as a sequence of bytes
- Data structures (e.g., array, record, list, tree) must be **flattened** (converted to a sequence of bytes) before transmission and rebuilt on arrival
- Client and server may have different representations of the same data type
 - Big-endian ordering of integer: most significant byte comes first
 - Little-endian ordering of integer: least significant byte comes first
 - $1934 = 078E$ (big endian ordering) $\rightarrow 078E = 36359$ (little endian ordering)

Marshalling and Unmarshalling

- External data representation
 - An agreed standard for representation of data structures and primitive values
- For interprocess communication
 - Define a standard form suitable for transmission
 - **Marshalling:** convert data items into the form suitable for transmission (at the source, structured and primitive data items → external data representation)
 - **Unmarshalling:** disassemble a message on arrival and restore data items (at the destination, external data representation → structured and primitive data items)
- Examples
 - CORBA's Common Data Representation (CDR)
 - Java's object serialization

CORBA's Common Data Representation

- CORBA: Common Object Request Broker Architecture (it is a middleware)
- CORBA's Common Data Representation (CDR) can represent all primitive and constructed data types that can be used as arguments and results of remote method invocations in CORBA

CORBA's Common Data Representation

- 15 primitive types: short (16-bit), long (32-bit), unsigned short, unsigned long, float (32-bit), double (64-bit), char, boolean, octet,
- Big-/little-endian: transmit in the sender's ordering and the recipient translates if necessary
- Floating-point: IEEE standard (sign + exponent + fractional part)
- Characters: an agreed code set
- Each primitive value is placed at an index in the sequence of bytes according to its size
 - A primitive value of size n bytes is appended to the sequence at an index that is a multiple of n

CORBA's Common Data Representation

- Constructed types:

Type	Representation
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order
array	array elements in order (no length specified because it is fixed)
struct	in order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Example CORBA CDR Message

CDR (flattened) form of a Person struct with value: {"Smith", "London", 1934}

index in sequence of bytes	← 4 bytes →
0 – 3	5
4 – 7	“Smit”
8 – 11	“h_”
12 – 15	6
16 – 19	“Lond”
20 – 23	“on_”
24 – 27	1934

```
struct Person{  
    string name;  
    string place;  
    unsigned long year;};
```

length of string

padded to flush on word boundary

length of string (started at index 12 instead of 9)

unsigned long (started at index 24 instead of 22)

CORBA's Common Data Representation

- Types of data items are **not** given in CDR form
- It is assumed that the sender and recipient **have common knowledge** of the order and types of the data items in a message
- For example, for RMI or RPC, each method invocation passes arguments of particular types in predefined order, and the result is a value of a particular type
- Support a variety of programming languages

Java Object Serialization

- For use by Java only
- In Java, both objects and primitive data values may be passed as arguments and results of method invocations
- **Serialization (synonym of marshalling in Java):** the activity of flattening an object or a connected set of objects into a serialized form suitable for storing on disk or transmitting in a message
- **Deserialization (synonym of unmarshalling in Java):** restoring an object or a set of objects from their serialized form

Java Object Serialization

- Assumption: the process doing deserialization has **no prior knowledge** of the object types in the serialized form, so some information about the class of each object needs to be included in the serialized form

Example Java Serialized Form

Java serialized form of a Person instance
with value: {"Smith", "London", 1934}

```
public class Person  
implements Serializable{  
    private String name;  
    private String place;  
    private int year;};
```

Person	8 byte version number	h0	
3 (# of fields)	int year	java.lang.String name	java.lang.String place
1934	5 Smith	6 London	h1

class name and
version number

number, type and name of
instance variables

values of instance variables

h0 is a class handle and h1 is an object handle
(usage to be discussed in the next example)

Java Object Serialization

- Format: class information, types and names of instance variables, values of instance variables
- If the instance variables belong to new classes, their class information must also be written out, followed by the types and names of their instance variables
- This recursive procedure continues until all necessary classes have been written out
- Each class is given a handle and no class is written more than once – the handles are written instead where necessary
 - Handles are references within the serialized form

Example Java Serialized Form

Java serialized form of a Couple instance with value:
{{“Smith”, “London”, 1934},
{“Jones”, “Paris”, 1945}}

```
class Couple implements Serializable{  
    private Person one;  
    private Person two;  
    public Couple(Person a, Person b) {  
        one = a; two = b;};}
```

Couple	8 byte version number	h0	
2	Person one	Person two	
Person	8 byte version number	h1	
3	int year	java.lang.String name	java.lang.String place
1934	5 Smith	6 London	h2
h1	1945	5 Jones	5 Paris
	h3	h4	

class name, version number
number, type and name of
instance variables

serialize instance variable
“one” of Couple

values of instance variables
serialize instance variable
“two” of Couple
values of instance variables

Summary of External Data Representations

- Need **standards** to facilitate communication and development of applications
- Standards cover a wide range: primitive data and structured data
- CORBA and its predecessors choose to marshal data for use by recipients that have **prior knowledge** of the types of its components
- Java object serialization includes **full information** about the types of its contents, allowing the recipient to reconstruct it purely from the contents

Outline

- External Data Representation & Marshalling
- Client-Server Communication
- Summary

Request-Reply Message Structure

messageType	int (0=Request, 1=Reply)
requestId	int
application-specific request contents or reply contents	

- Unique request message identifier:
requestId (incremented each time a request is sent by the client process)
+ an identifier for the client process (e.g., Internet address + port number)

Interprocess Communication

RMI and RPC

We now look at → request-reply protocol
marshalling & external data representation

UDP and TCP

Middleware layers

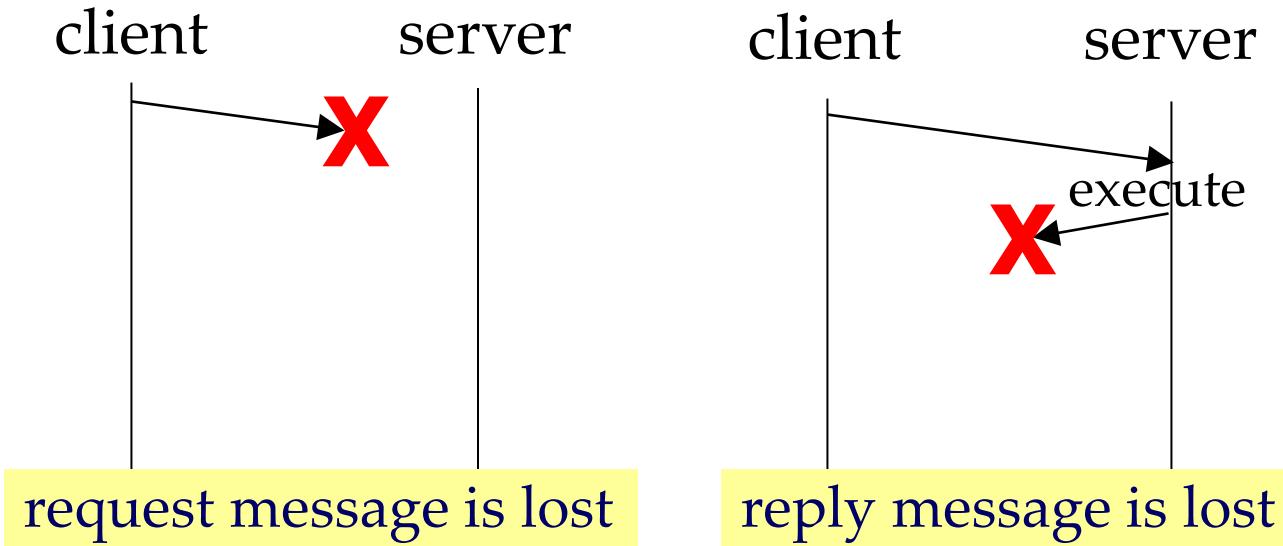
- We perceive a network supporting TCP and UDP
- Reliability
 - **Validity:** the message reaches the destination (the message in the outgoing message buffer of the sender is eventually delivered to the incoming message buffer of the receiver)
 - **Integrity:** the message received is identical to the one sent, and no message is delivered more than once

Request-Reply Protocol over UDP

- A message sent by UDP is transmitted without acknowledgements or retries
 - If a failure occurs, the message may not arrive
 - Messages larger than maximum allowable size must be fragmented for transmission
- UDP is not a reliable communication service
 - **Integrity:** use checksums to detect and reject corrupt packets
 - **Validity:** messages may be dropped occasionally (communication omission failures)
- Applications: any application that requires quick response (e.g., DNS requests), can assume reliability (e.g., on LAN), or handles faults itself

Request-Reply Protocol over UDP

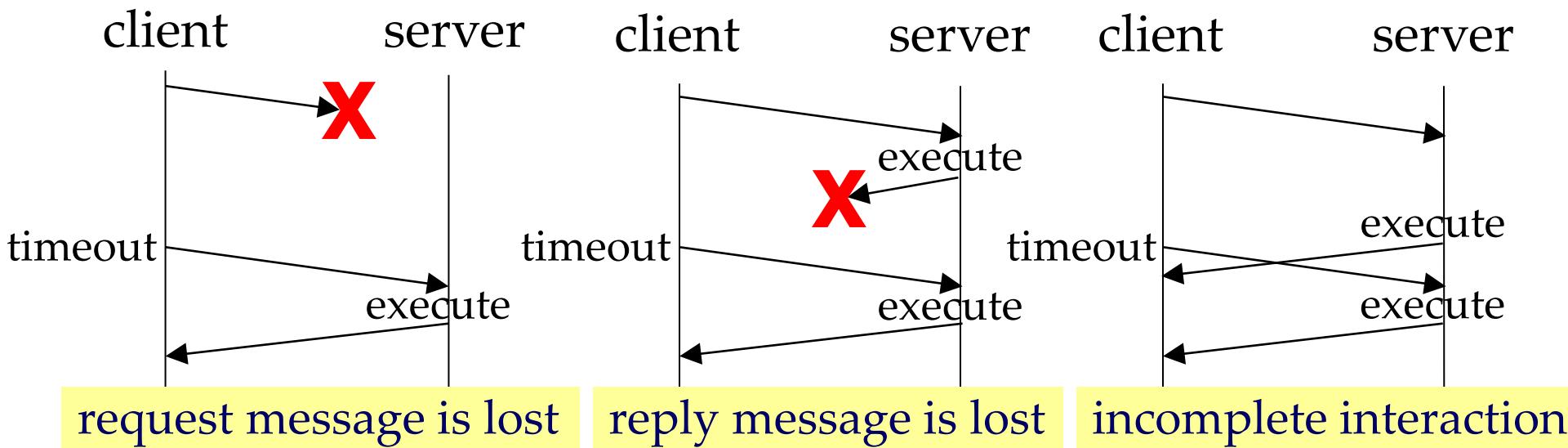
- UDP suffers from communication omission failures



- Goal: to build **reliable** request-reply protocols over UDP
- Other possible failures: processes may have crash and Byzantine failures (these are common to request-reply interactions over both UDP and TCP, will be discussed in later chapters)

Building Reliable Request-Reply Protocols over UDP

- Client uses a **timeout** when waiting for server's reply and sends the request repeatedly after a timeout



- Solve the problem of lost request message
- For lost reply message, operation is executed more than once
- Introduce a new problem of incomplete interaction, where the operation is also executed more than once

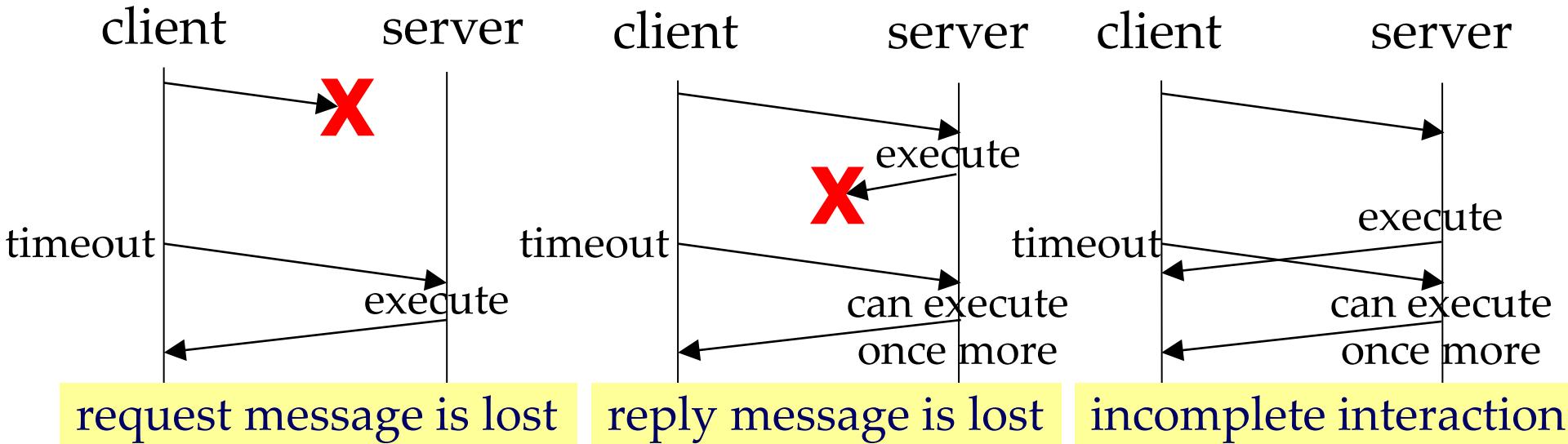
Building Reliable Request-Reply Protocols over UDP

- Is repeated execution harmful?
 - **Idempotent operations:** those can be performed repeatedly with the same effect as if performed exactly once
 - e.g., adding an element to a set
$$\{1,2\} \cup \{3\} = \{1,2,3\}; \{1,2,3\} \cup \{3\} = \{1,2,3\}; \dots$$
 - e.g., checking the balance of a bank account
 - **Non-idempotent operations:** those if performed repeatedly have different effects from if performed exactly once
 - e.g., increasing a variable by 3
$$2 + 3 = 5; 5 + 3 = 8; 8 + 3 = 11; \dots$$
 - e.g., depositing money into a bank account

Building Reliable Request-Reply Protocols over UDP

■ For idempotent operations

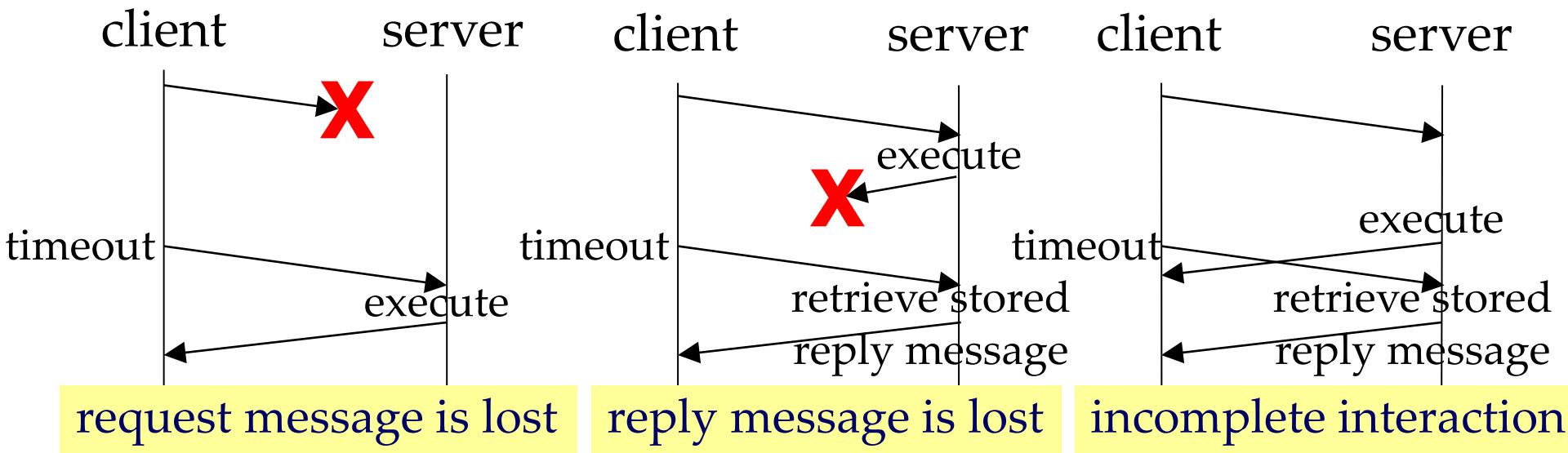
- No need to avoid server executing the operations more than once for the same request
- Execute operations and reply as usual for all requests



Building Reliable Request-Reply Protocols over UDP

For non-idempotent operations

- Server identifies messages from the same client with same requestId and **filter out duplicates** to avoid server executing an operation more than once for the same request
- If an incoming request is seen for the first time, execute operation, reply as usual, and store the reply message (results) in a history
- If an incoming request is a duplicate request, retrieve stored reply message from history and retransmit it to the client



Request-Reply Protocol over TCP

- TCP offers an abstraction of “reliable stream”
 - Connection-oriented: a connection must be setup before any data are transferred
 - Acknowledgements and retries
 - Transmit without worrying about message size, large messages get segmented in transmission transparently

Request-Reply Protocol over TCP

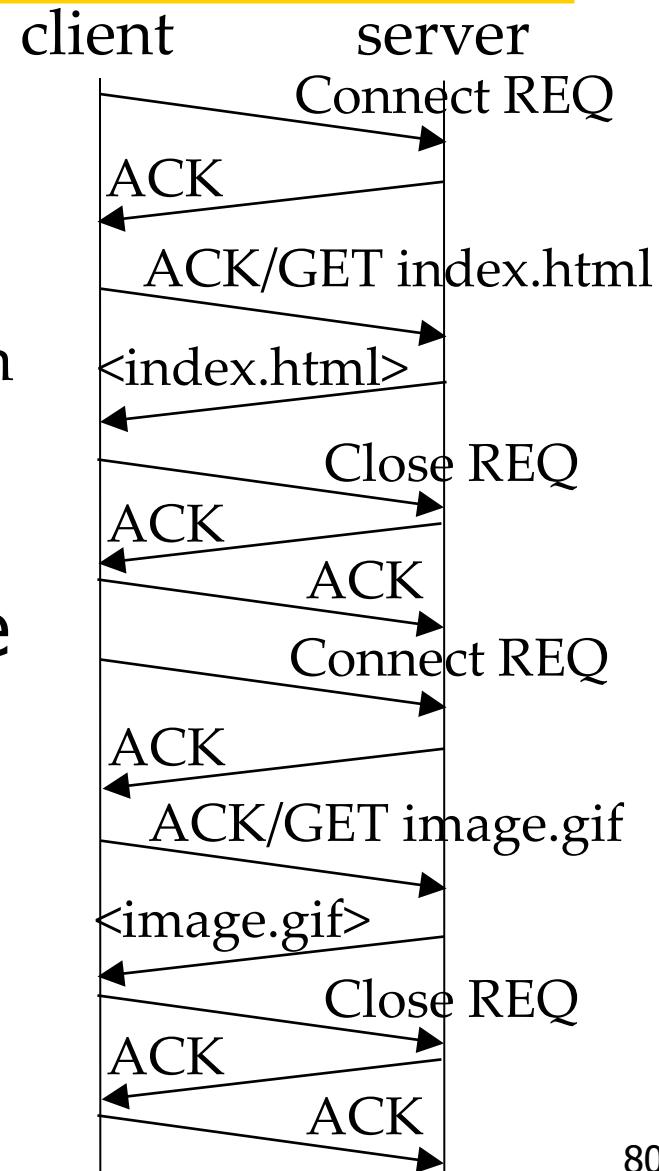
- TCP is a reliable communication service
 - **Integrity:** use checksums to detect and reject corrupt packets; use sequence numbers to detect and reject duplicate packets
 - **Validity:** successfully received packets are acknowledged; use timeouts and retransmissions to deal with lost packets
 - TCP ensures request and reply messages are delivered reliably
 - no need to deal with retransmission of messages, filtering of duplicates and maintaining histories in request-reply protocols over TCP

Request-Reply Protocol over TCP

- Overhead
 - Store state information at source and destination (e.g., to detect duplicate packets)
 - Transmission of extra messages (e.g., connect request, acknowledgement)
 - Additional latency (e.g., due to connection setup)
- Applications: any application that does not wish to handle missed/duplicated data or message sizes
 - Telnet, FTP, SMTP, HTTP,
- Our goal: to **reduce overhead** of request-reply protocols over TCP

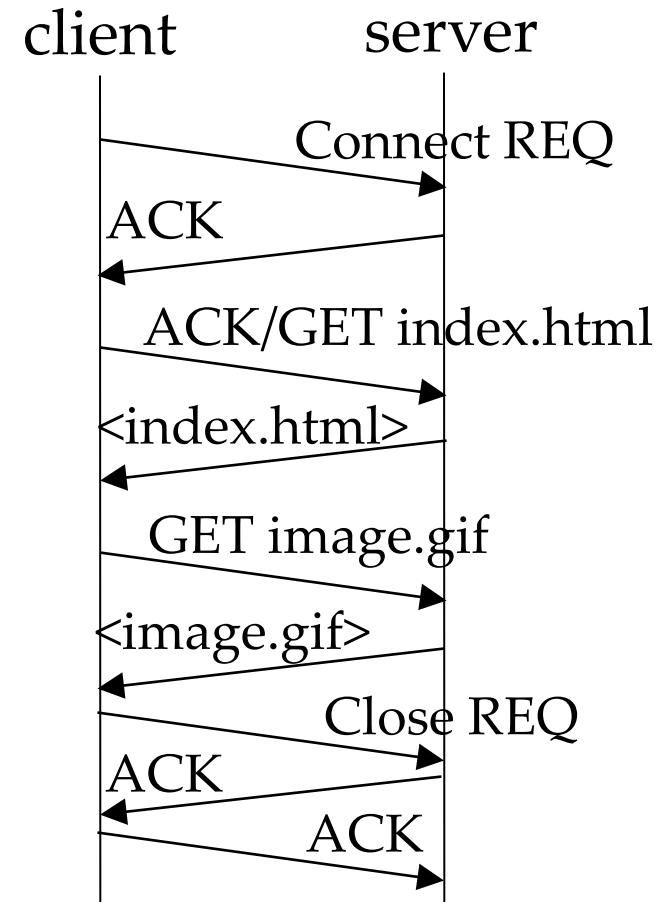
Reducing Overhead of Request-Reply Protocols over TCP

- Earlier version of HTTP sets up a new TCP connection for each HTTP request
 - Connection is closed on completion of the request
- Multiple pairs of requests and replies in a short duration can be sent over the same connection
→ see next slide



Reducing Overhead of Request-Reply Protocols over TCP

- Recent version of HTTP uses **persistent connections** that allow a client to reuse a single TCP connection to send multiple HTTP requests
 - Connection is kept alive on completion of a request
 - Advantage: amortizing the overhead of connection establishment over a group of requests
 - Advantage: avoiding multiple TCP slow-starts



Summary

Learned about:

- External representations & marshalling (CDR and Java Serialization)
- Request-reply protocols for client-server communication

These are basis for building distributed systems and applications



3. Distributed Objects & Remote Invocation

Middleware Layers

Let's now
look at →
We have
learned →

Applications

RMI and RPC

request-reply protocol

marshalling & external data representation

UDP and TCP

Middleware
layers

- We now look at programming models for distributed applications which are composed of cooperating programs running in several different processes

Outline

- Object-based model: remote method invocation (RMI)
 - Distributed object model
 - Architecture of RMI
 - An example of Java RMI
- Summary

Object Model

- An object-oriented program consists of a collection of interacting objects
 - Each object encapsulates a set of data (called the state) and methods (the operations on those data)
 - They communicate by **invoking methods** (passing arguments and receiving results)
 - What do we need to invoke a method?
 - **An object reference, a method name, necessary arguments**
 - Object references can be passed as arguments and returned as results of methods

Object Model

- An object-oriented program consists of a collection of interacting objects (cont'd)
 - An **interface** defines the signatures of the methods (arguments, return values and exceptions) that can be accessed from other modules
 - Definitions of the methods only
 - No implementation (code) of the methods
 - Implementation of the methods may be changed without changing the interface
 - Exceptions: errors and unexpected conditions

An Example of Object Model

- **GraphicalObject class**

```
public class GraphicalObject {  
    public String type;  
    public Rectangle enclosing;  
    public Color line;  
    public Color fill;  
    public boolean isFilled;  
  
    // constructors  
    public GraphicalObject() {}  
    public GraphicalObject(...) {...}  
  
    public void print() {...}  
}
```

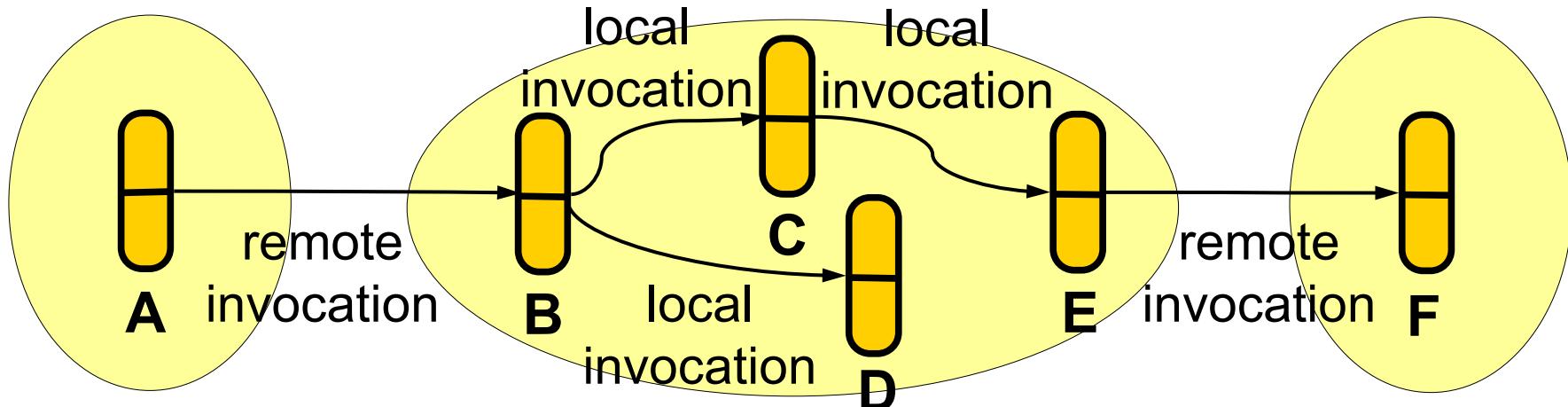
An Example of Object Model

- Another class accessing an object of GraphicalObject class

```
public class AnotherClass{  
    public static void main() {  
        try {  
            GraphicalObject g = new GraphicalObject(...);  
            System.out.println("Created graphical object");  
            g.print();  
        } catch(Exception e) { System.out.println("main: " +  
            e.getMessage()); }  
    }  
}
```

Distributed Object Model

- A natural extension of object model – distribute objects into different processes/computers
 - Objects are managed by servers and their methods are invoked by clients using remote method invocation (**client-server arch.**)
 - **Remote method invocation** – between objects in different processes
 - **Local method invocation** – between objects in the same process
 - Remote objects: objects that can receive remote invocations



Distributed Object Model

- What do we need to invoke a method?
 - Object reference → remote object reference: to uniquely identify an object throughout a distributed system
 - An example remote object reference

Internet address	port number	object number
------------------	-------------	---------------

incremented each time a new object
is created in the server process

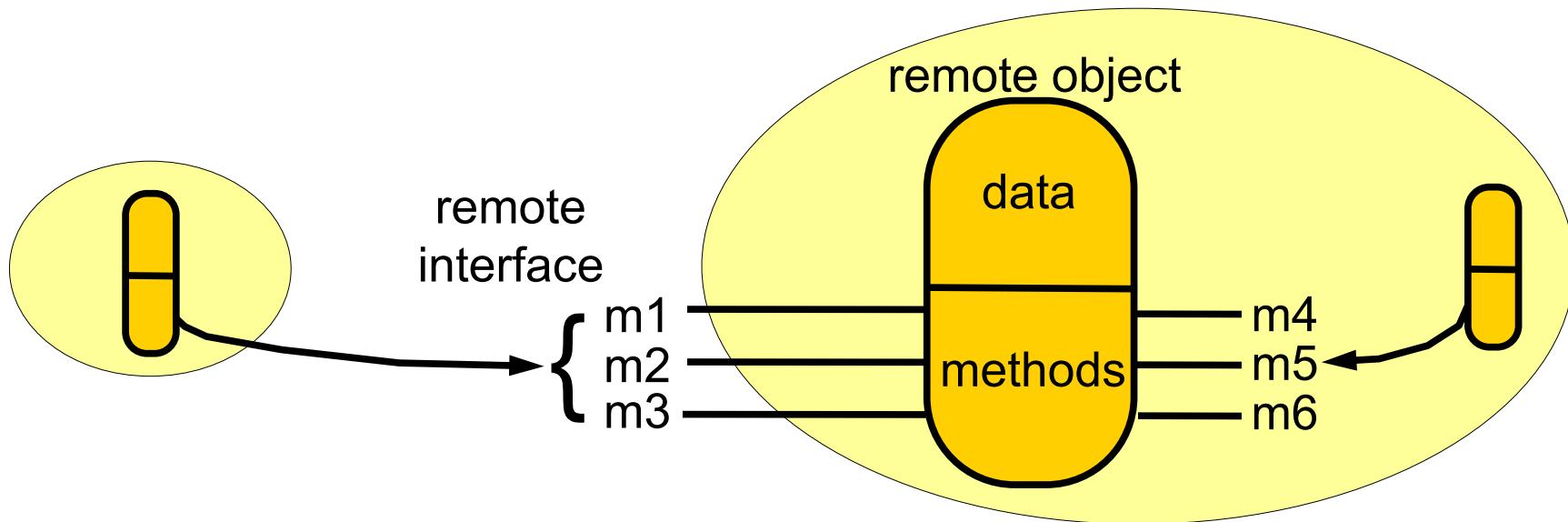
- A remote object reference is created for an object by the server when the object is created
- The remote object reference must be obtained by the client in order to access the object
- Remote object references can also be passed as arguments and results of remote method invocations
- Method name and necessary arguments

Distributed Object Model

- Interface → remote interface
 - Each remote object has a **remote interface** specifying the methods that can be invoked remotely (i.e., by objects in other processes)
 - Objects in other processes can invoke only the methods in the remote interface
 - Local objects can invoke methods in the remote interface as well as other methods implemented by a remote object
 - Can be defined by
 - Java interface (will be discussed in this chapter)
 - CORBA Interface Definition Language (IDL)

Distributed Object Model

- Interface → remote interface (cont'd)



- Remote method invocation raises exceptions due to distribution (e.g., timeouts caused by server crashes or lost messages) as well as those raised during the execution of the method invoked

Invocation Semantics

- Local method invocations are always executed exactly once
- However, this is not true for remote method invocation due to various types of failures
- Revisit: techniques to handle failures
 - Retransmit request message
 - Filter duplicate requests
 - Re-execute method or retransmit stored reply message (results)
- Combinations of these techniques lead to a variety of possible semantics for remote method invocation

Invocation Semantics

- Maybe invocation semantics

- No retransmission of request message
- If the client receives a result, the method has been executed
- If no result is received, the client cannot tell whether a remote method has been executed or not at all
 - Communication omission failure (if request message is lost, the method would not have been executed; if reply message is lost, the method would have been executed)
 - Process omission failure (if server crashes, the method may or may not have been executed)
- Useful only for applications where occasional failed invocations are acceptable

Invocation Semantics

- At-least-once invocation semantics
 - Retransmit request message, but no filtering of duplicate requests, re-execute method upon request
 - If the client receives a result, the method has been executed at least once
 - Arbitrary failures for non-idempotent operations
 - If no result is received, the method may have been executed more than once, once or not at all
 - Request message loss and reply message loss
 - Acceptable if all methods in remote interfaces are idempotent operations
 - In Sun RPC, the invocation semantics is at-least-once

Invocation Semantics

- At-most-once invocation semantics
 - Retransmit request message, filter duplicate requests, retransmit stored reply message upon duplicate request
 - If the client receives a result, the method has been executed exactly once
 - If no result is received, the method may have been executed either once or not at all
 - Request message loss and reply message loss
 - In both Java RMI and CORBA, the invocation semantics is at-most-once, but CORBA allows maybe semantics for methods that do not return results

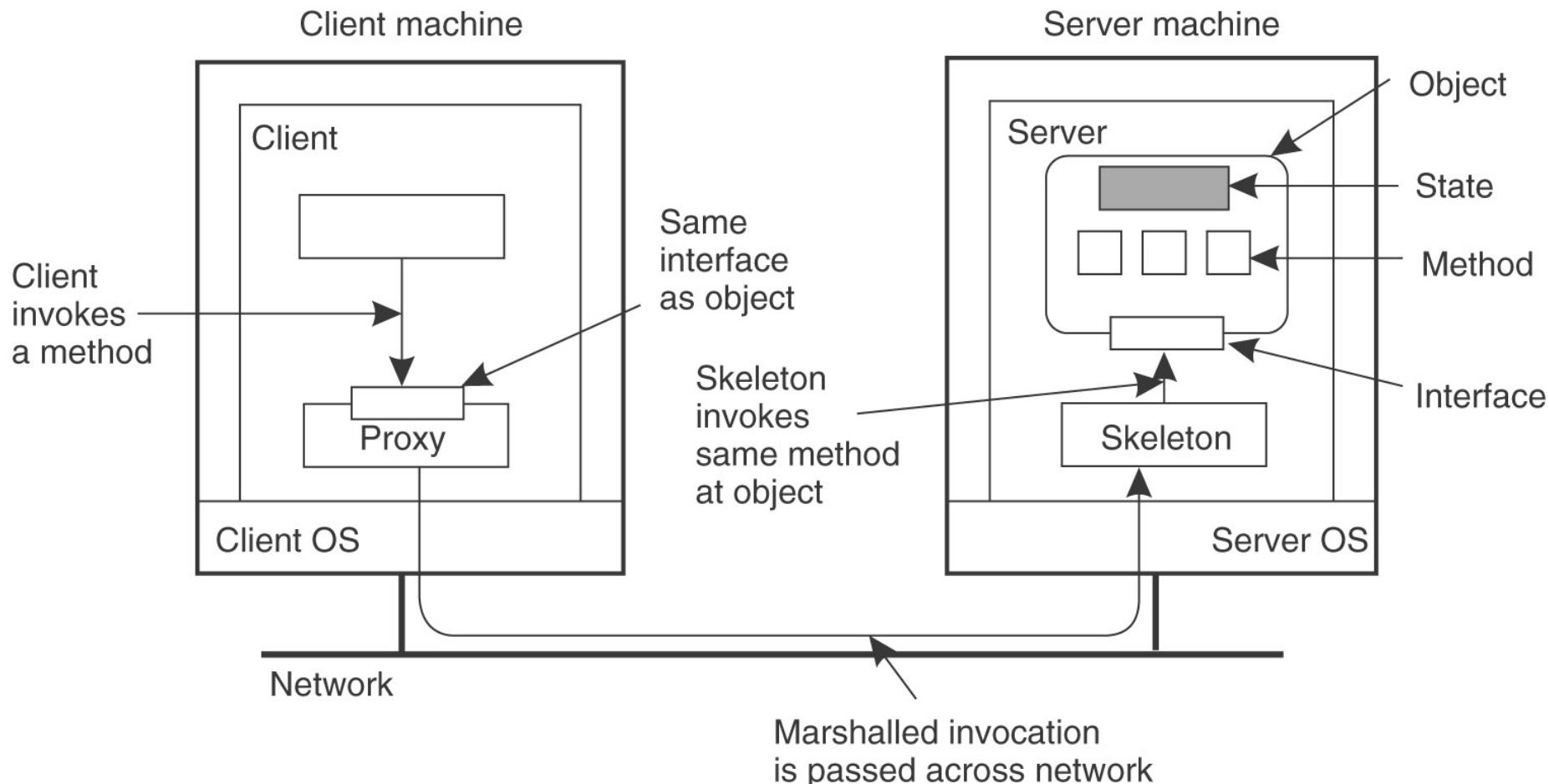
Summary of Invocation Semantics

Fault tolerance measures			Invocation semantics
Retransmit request message	Duplicate filtering	Re-execute method or retransmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute method	At-least-once
Yes	Yes	Retransmit reply	At-most-once

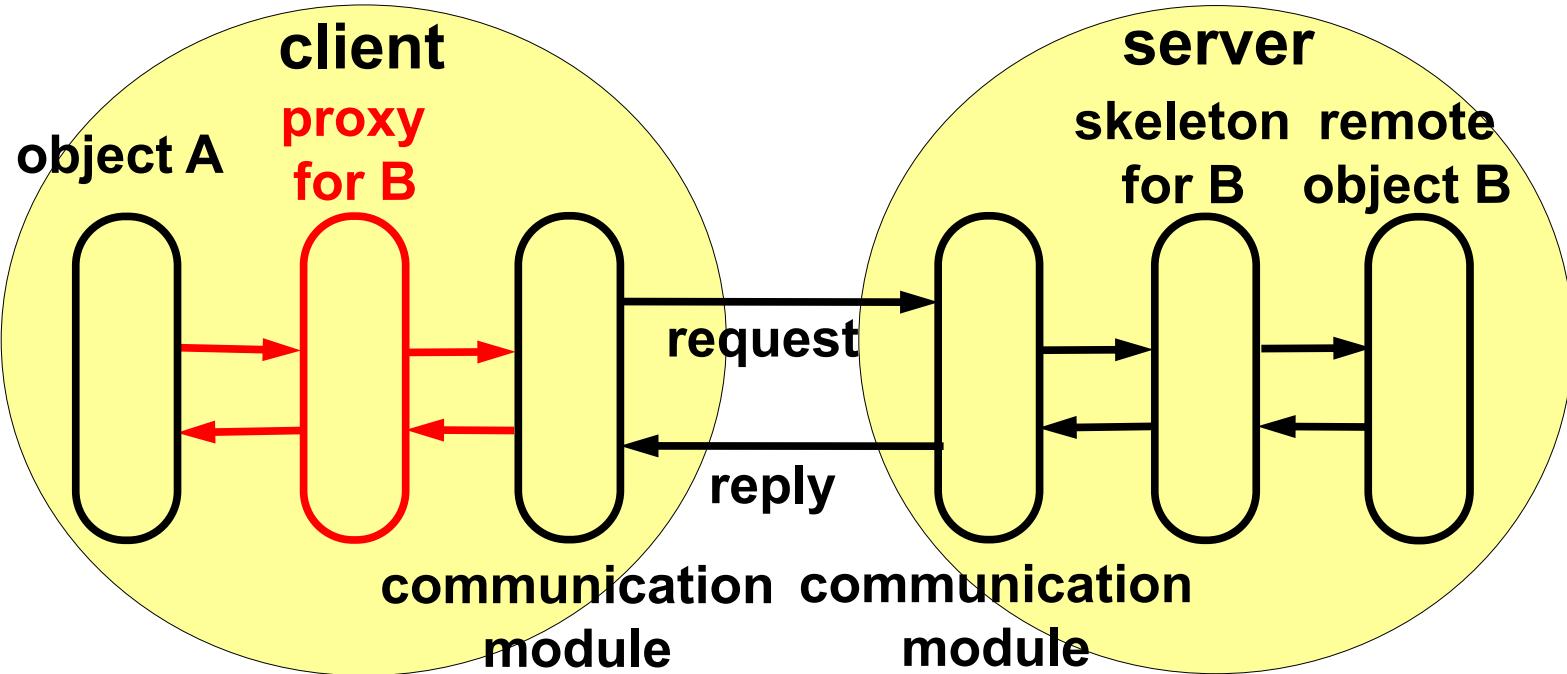
Outline

- Object-based model: remote method invocation (RMI)
 - Distributed object model
 - Architecture of RMI
 - An example of Java RMI
- Summary

How Does RMI Work?

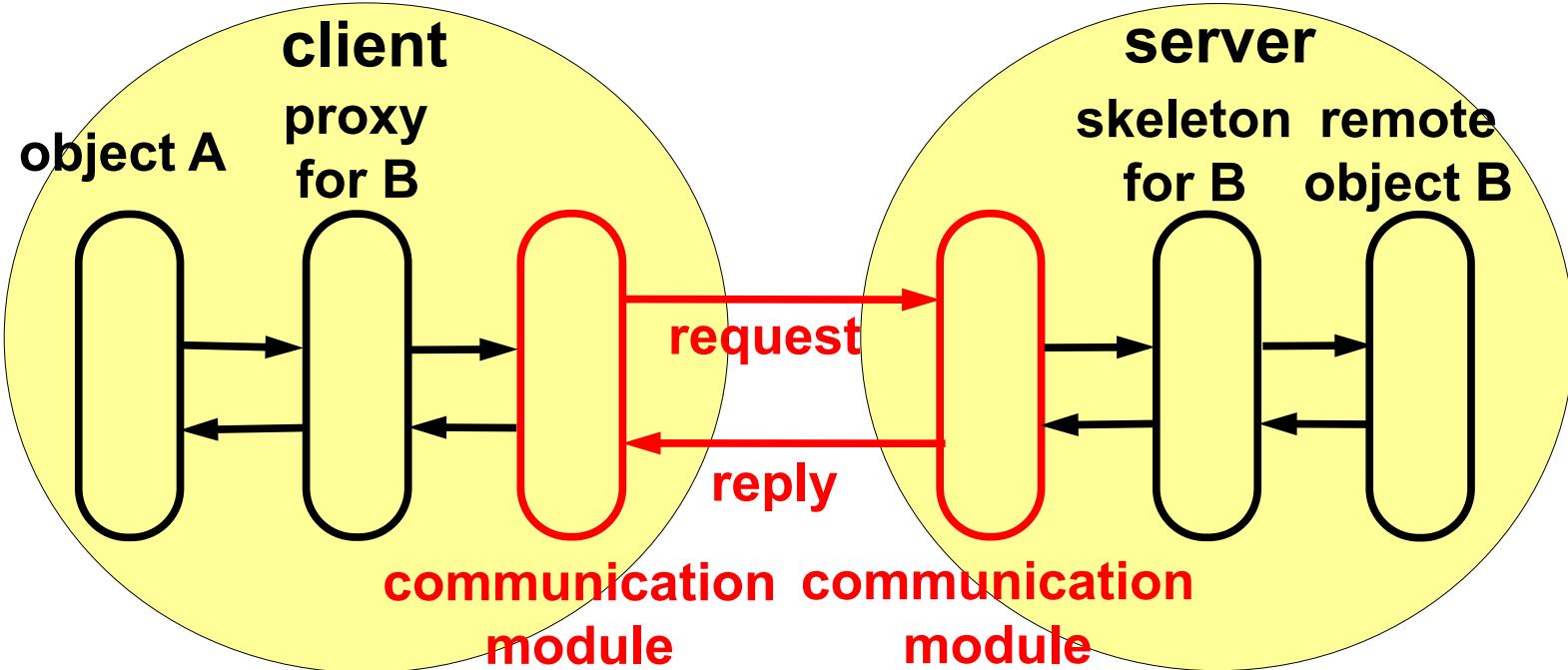


Architecture of RMI



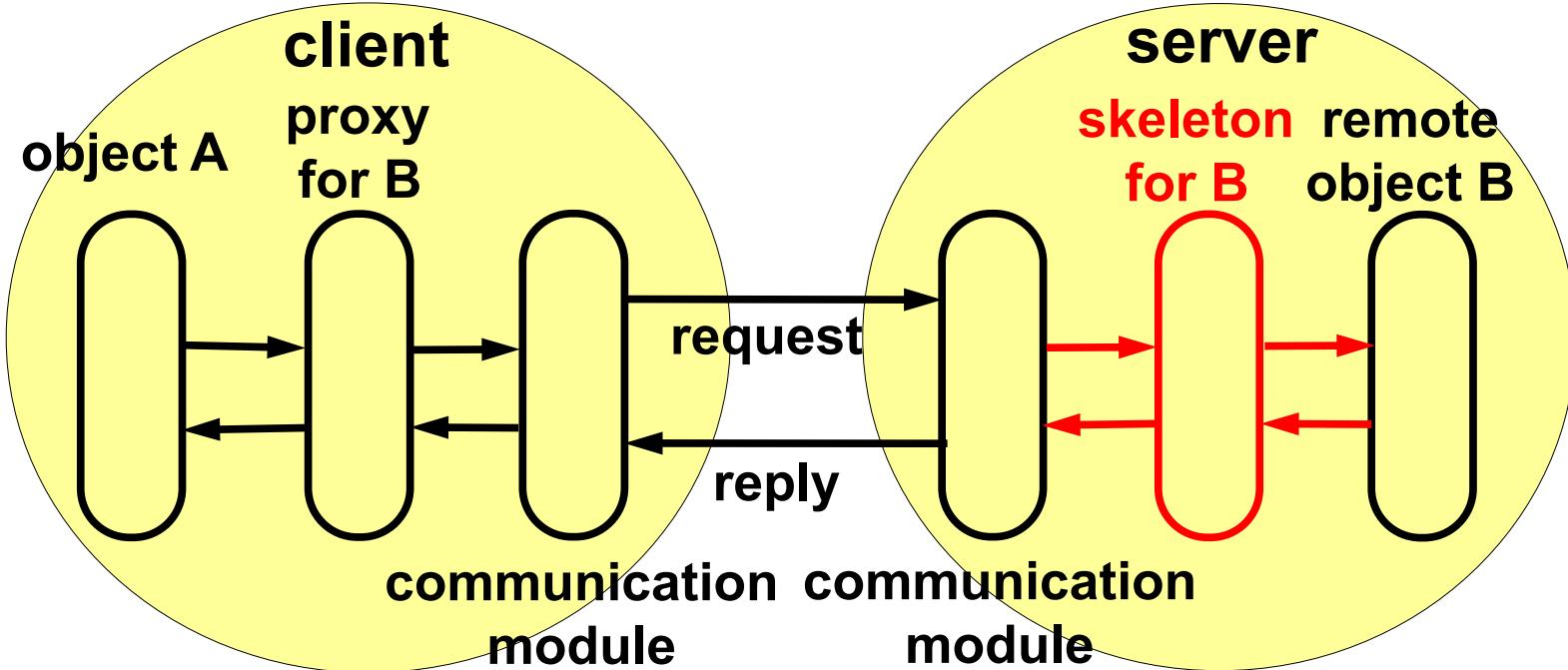
- One proxy for each class of remote objects for which a client holds a remote object reference
- The proxy implements the methods in remote interface, makes remote method invocation transparent to clients
- It marshals arguments and unmarshals results

Architecture of RMI



- Communication modules transmit request and reply messages between client and server
- An example message structure: messageType (request/reply) + requestId + remoteObjectReference + methodId + arguments

Architecture of RMI

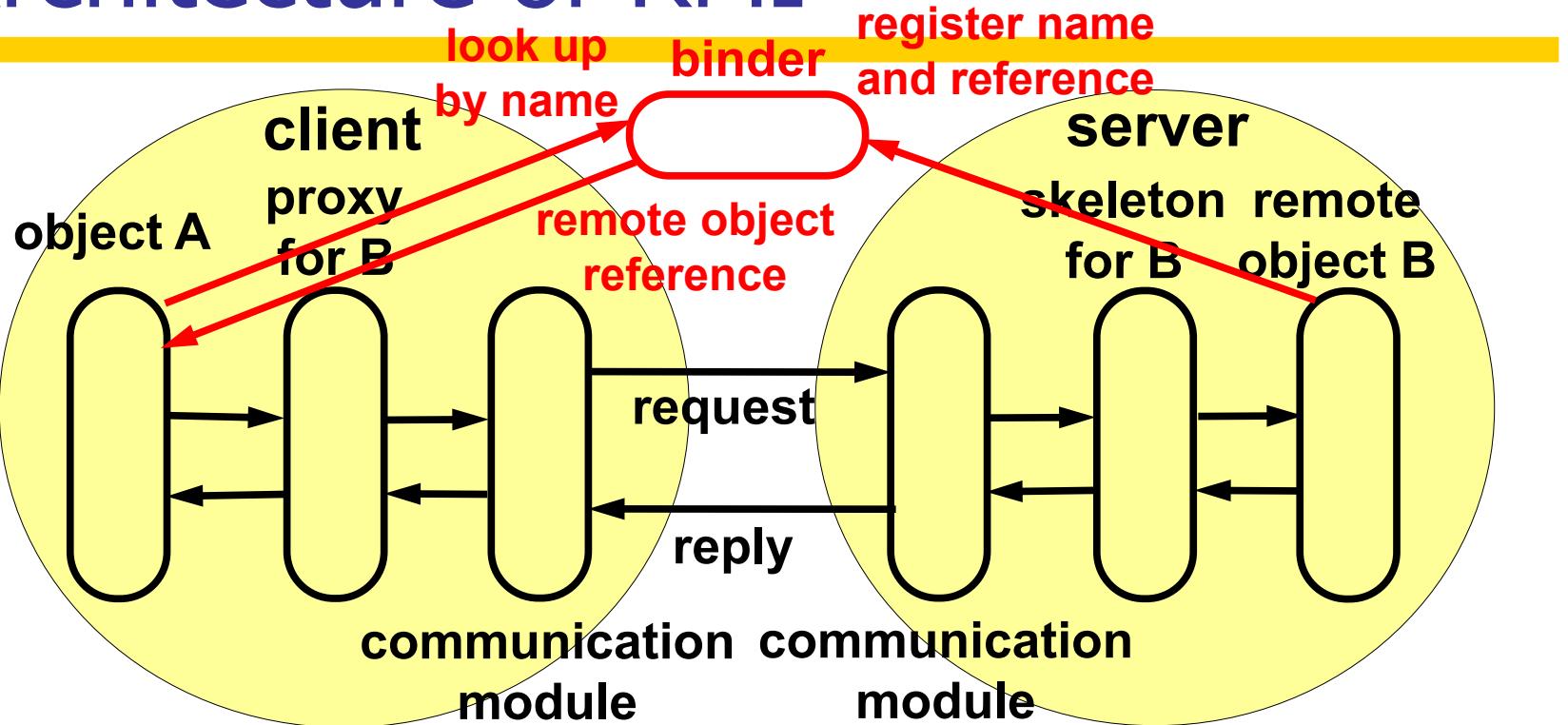


- One skeleton for each class of remote objects
- The skeleton implements the methods in remote interface
- It unmarshals arguments, invokes the corresponding method in the remote object, and marshals results

Architecture of RMI

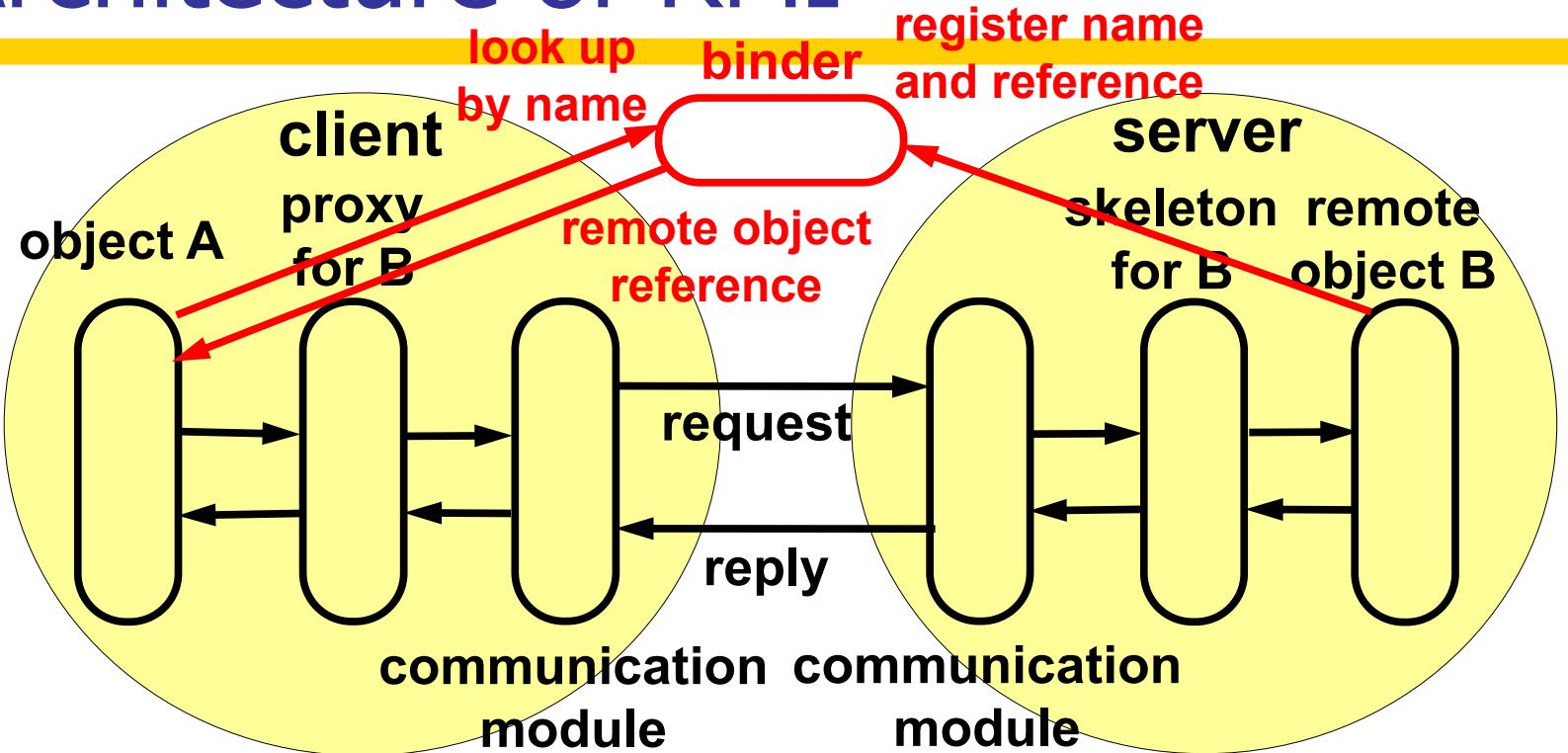
- Proxy and skeleton are automatically generated by an interface compiler
 - In Java, remote interfaces are defined as Java interfaces → use Java RMI compiler
 - In CORBA, remote interfaces are defined in CORBA IDL (Interface Definition Language) → use interface compiler
- Server-side: skeleton and the real implementation of methods
- Client-side: proxy

Architecture of RMI



- How does the client obtain a remote object reference?
 - Remote object references may not be available at the time of programming
 - Object names in the text form are more convenient for programmers

Architecture of RMI



- A binder is a name service that maintains mappings from object names to remote object references
- Used by servers to register their remote objects by name and by clients to look up remote object references

Outline

- Object-based model: remote method invocation (RMI)
 - Distributed object model
 - Architecture of RMI
 - An example of Java RMI
- Summary

Java RMI

- Overview – steps for developing Java RMI
 - Design remote interfaces (identify remote objects, their roles, what do they take in, what do they produce)
 - Implementation
 - **Servant:** do the actual work – implement the methods in the remote interface
 - **Server:** do the easy job – create remote objects and register them in RMIClientRegistry (binder)
 - **Client:** look up remote objects and access them
 - Compile remote interface (generating proxies, skeletons etc.), compile source codes
 - Start server, followed by clients

An Example of Java RMI

- Write a Java RMI application where the server maintains pre-recorded information (population, temperature, etc.) about cities and the clients query the information through remote method invocation.

An Example of Java RMI

■ Step 1: Design the remote interface

```
import java.rmi.*;  
public interface City extends Remote {  
    int getPopulation(String cityName) throws  
        RemoteException;
```

Remote interfaces are defined by
extending the *Remote* interface

```
    int getTemperature(String cityName) throws  
        RemoteException;
```

arguments – inputs

Methods must throw *RemoteException*

result – output

■ Main differences from regular/local interfaces

- Remote extension
- RemoteException for each method signature

An Example of Java RMI

- Step 2: Design the servant class to implement the methods specified in the remote interface

```
import java.rmi.*;  
import java.rmi.server.*;  
  
public class CityImpl extends UnicastRemoteObject  
    implements City {  
  
    public CityImpl() throws RemoteException {  
        super(); //invoke the base class's constructor  
    }  
}
```

Classes of remote objects are defined by extending the *RemoteObject* class; *UnicastRemoteObject* is a subclass of *RemoteObject* that defines a non-replicated remote object whose references are valid only while the server process is alive

An Example of Java RMI

```
public int getPopulation(String cityName) throws
    RemoteException {
    if (cityName.equals("Toronto")) { return 10; }
    else if (cityName.equals("Ottawa")) { return 2; }
    else { return 100; }
}

public int getTemperature(String cityName) throws
    RemoteException {
    if (cityName.equals("Toronto")) { return 20; }
    else if (cityName.equals("Ottawa")) { return 30; }
    else { return 10; }
}
```

Java RMI

- Overview – steps for developing Java RMI (revisit)
 - Design remote interfaces (identify remote objects, their roles, what do they take in, what do they produce)
 - Implementation
 - **Servant:** do the actual work – implement the methods in the remote interface
 - **Server:** do the easy job – create remote objects and register them in RMIClientRegistry (binder)
 - **Client:** look up remote objects and access them
 - Compile remote interface (generating proxies, skeletons etc.), compile source codes
 - Start server, followed by clients

Java RMI

■ RMIServer – the binder for Java RMI

- It maintains a table mapping text, URL-style names to remote object references
- An instance of RMIServer must run on every server computer that hosts remote objects

void rebind (String name, Remote obj) throws RemoteException

Used by a server to register the identifier of a remote object by name.

void bind (String name, Remote obj) throws RemoteException

Used alternatively by a server to register a remote object by name, but if the name is already bound to a remote object reference, an exception is thrown.

void unbind (String name, Remote obj) throws RemoteException

This method removes a binding.

Remote lookup(String name) throws RemoteException

For clients to look up a remote object by name. A remote object reference is returned.

String [] list() throws RemoteException

This method returns an array of Strings containing the names bound in the registry.

Java RMI

- An RMI registry may be shared by all servers of different remote objects on the same host (default port number 1099)
 - In this case, the RMI registry maintains the name-to-reference mappings of all remote objects on the host
- Alternatively, an individual server may create and use its own registry if desired
 - In this case, multiple RMI registries may run at different port numbers on the host, each maintaining the name-to-reference mappings of a separate list of remote objects

An Example of Java RMI

- Step 3: Design the server class to create remote objects (instances of servant class) and register them in RMIClient

```
import java.rmi.*;  
import java.rmi.server.*;  
public class CityServer {  
    public static void main(String args[]){  
        try{  
            CityImpl aCityImpl = new CityImpl();  
            Naming.rebind("rmi://.../City", aCityImpl);  
        } catch (Exception e) { ... }  
    }  
}
```

 Name of remote object:
rmi://hostName:port/ObjectName

An Example of Java RMI

- Step 4: Design the client class to look up remote objects and access them

```
import java.rmi.*;  
public class CityClient {  
    public static void main(String args[]) {  
        try {  
            City aCityServer = (City) Naming.lookup("rmi://.../City");  
            int pop = aCityServer.getPopulation("Toronto");  
            System.out.println(pop);  
        } catch (RemoteException e) { ... }  
        catch (Exception e) { ... }  
    }  
}
```

Name of remote object:
rmi://hostName:port/ObjectName

RMI-related exception

Application-specific exception

Java RMI

- Overview – steps for developing Java RMI (revisit)
 - Design remote interfaces (identify remote objects, their roles, what do they take in, what do they produce)
 - Implementation
 - **Servant:** do the actual work – implement the methods in the remote interface
 - **Server:** do the easy job – create remote objects and register them in RMIClientRegistry (binder)
 - **Client:** look up remote objects and access them
 - Compile remote interface (generating proxies, skeletons etc.), compile source codes
 - Start server, followed by clients

An Example of Java RMI

- At the server side, to compile
 - javac City.java
 - javac CityImpl.java
 - javac CityServer.java
- To generate proxies and skeletons
 - rmic CityImpl
 - This produces CityImpl_Skel.class and CityImpl_Stub.class
- To start an RMI registry at system prompt
 - rmiregistry <port number>
- To start the server
 - java CityServer

An Example of Java RMI

- At the client-side, to obtain from the server
 - A copy of City.java
 - A copy of CityImpl_Stub.class
- To compile
 - javac City.java
 - javac CityClient.java
- To start the client
 - java CityClient

- This is a simple RMI example, more complicated examples will be given in the tutorial

Java RMI

- In the previous example, the client invokes the methods at the server to retrieve information, he can do it from time to time to check whether there is any update to the information (polling)
- Is there any alternative?
- **Callback:** the server informs the client when the information is updated
- Polling vs. Callback
 - Ask your mom every 5 minutes “Is dinner ready?” (polling)
 - Please let me know when dinner is ready (callback)

Java RMI

■ How to implement callback?

- Client creates a remote object that implements an interface containing a method for server to call (**callback object**)

```
public interface Callback extends Remote {
```

```
    void cbMethod(...) throws RemoteException; }
```

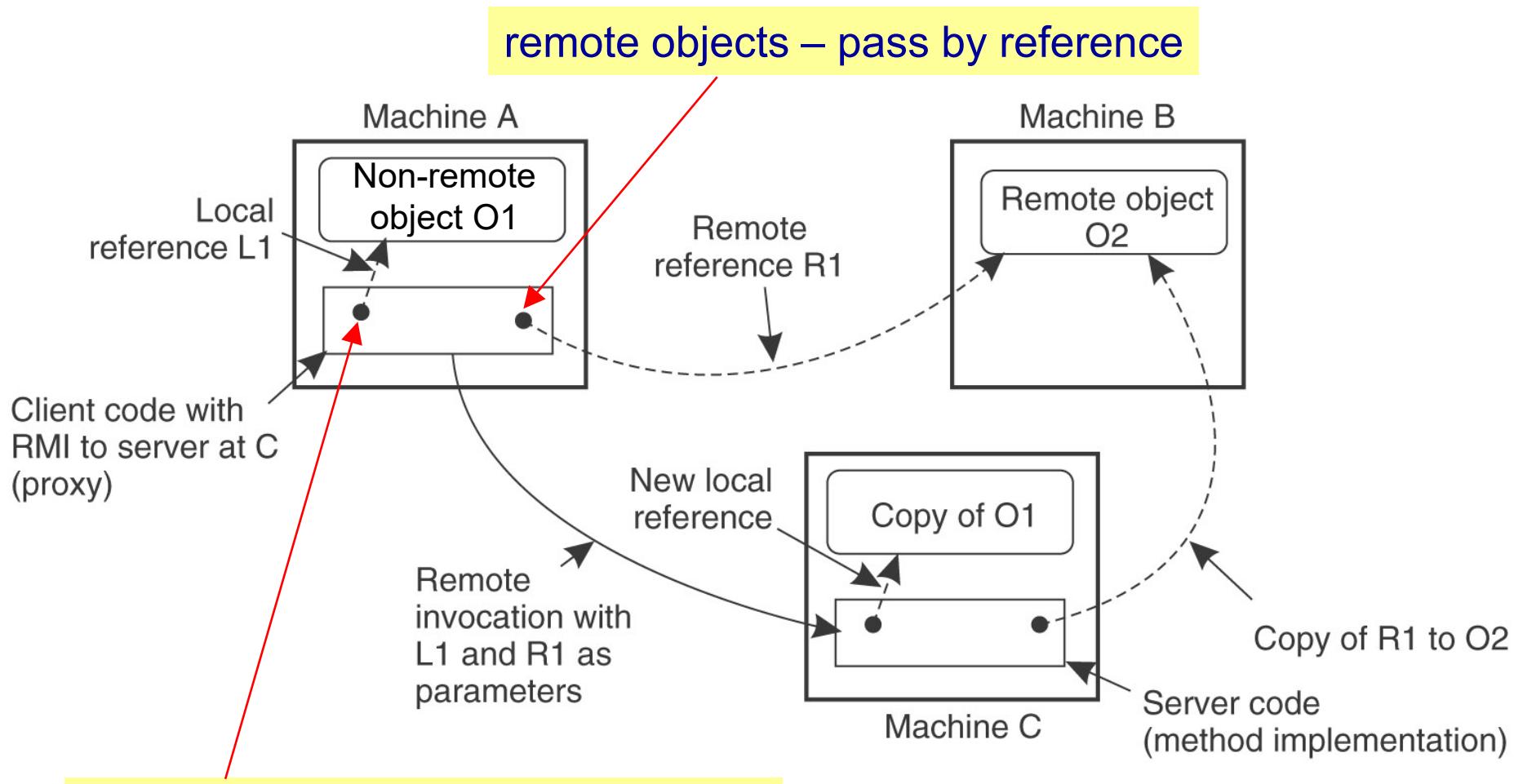
- Server provides an operation for interested clients to inform it of the remote object references of their callback objects and records them in a list

```
void register(Callback cbObject) throws RemoteException;
```

```
void deregister(Callback cbObject) throws RemoteException;
```

- When an event of interest occurs, server invokes the method in callback objects

Parameter Passing Semantics in RMI



Summary

- Remote Method Invocation (RMI)
 - Remote object
 - Remote method invocation
 - Remote object reference
 - Remote interface
 - Invocation semantics
 - Java RMI architecture
 - How to write Java remote interfaces?
 - How to write Java RMI programs?
- The counterpart in procedure-based programming model is Remote Procedure Call (RPC)
 - RPC is very similar to remote method invocation – a client process calls a procedure in a server process

4. Distributed File Systems

Outline

- **Introduction**
- Sun Network File System
- Andrew and Coda File Systems
- Summary

File System

- Allow multiple clients to share access to files
 - Files are stored on disks (persistent storage)
- Provide services (API) to programmers, hiding them from details of storage media
 - Naming and organization
 - /usr/students/bob/hello.txt
 - Storage and retrieval
 - count = read(filedes, buffer, n)
 - count = write(filedes, buffer, n)
 - Sharing and protection (access control)
 - read/write/execute, owner/group/universe, e.g.,
rwxrw-r--

Elements of A File System

- Files contain data and attributes
- Directories are a special type of files
 - Provide mappings from text names to internal file identifiers
 - Serve as folders containing other files
 - Form hierarchical structures

A typical attribute record structure

File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

updated by system:

updated by owner:

Distributed File System

- Support file accesses throughout an intranet
 - Enable users to store and access remote files exactly as they do local files
- New stuff in distributed file system
 - Distributed naming and location of files
 - Component to deal with client-server communication

Distributed File System Requirements

- Transparency
 - **Access transparency:** clients should use the same interface for accesses to local and remote files
 - **Location transparency:** clients should see a uniform file name space
- File replication & consistency maintenance
 - A file may be replicated at different locations
 - To improve performance and enhance scalability
 - To enhance fault tolerance
 - → consistency maintenance

Distributed File System Requirements

- Fault tolerance: stateless vs. stateful servers
 - Stateful server: keep track of client requests
 - Server remembers client's previous operations, the requests are **inter-dependent**
 - E.g., UNIX creates a file descriptor after a client opens a file and keeps track of current position in the file, so client requests are like “read 5 more blocks” (non-idempotent)
 - Implications: heavier demand on server and harder to set up or restore on crashes
 - When server crashes, it must restore the states or notify all clients after restart
 - When client crashes, server must detect it and discard all states about the client (not easy in distributed systems)

Some UNIX File System Operations

filedes = open(name, mode)

Opens an existing file with the given *name*.

filedes = creat(name, mode)

Creates a new file with the given *name*.

Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.

status = close(filedes)

Closes the open file *filedes*.

count = read(filedes, buffer, n)

Transfers *n* bytes from the file referenced by *filedes* to *buffer*.

count = write(filedes, buffer, n)

Transfers *n* bytes to the file referenced by *filedes* from *buffer*.

Both operations deliver the number of bytes actually transferred and advance the read-write pointer.

pos = lseek(filedes, offset, whence)

Moves the read-write pointer to offset (relative or absolute, depending on *whence*).

Distributed File System Requirements

- Fault tolerance: stateless vs. stateful servers (cont'd)
 - Stateless server: client requests are not tracked
 - Server does not remember client's previous operations, each request is independent of other requests
 - Each request must be self-contained, e.g., "read 5 blocks from block 10" (idempotent)
 - Implications: easier to set up and restore, less burden on server, heavier demand on network
 - When server crashes, it needs not recover any previous state; client may simply retransmit requests until it gets a reply
 - When client crashes, client can restart and rebuild its own state, no action is needed by server
 - So, for distributed file systems, stateless services are preferred

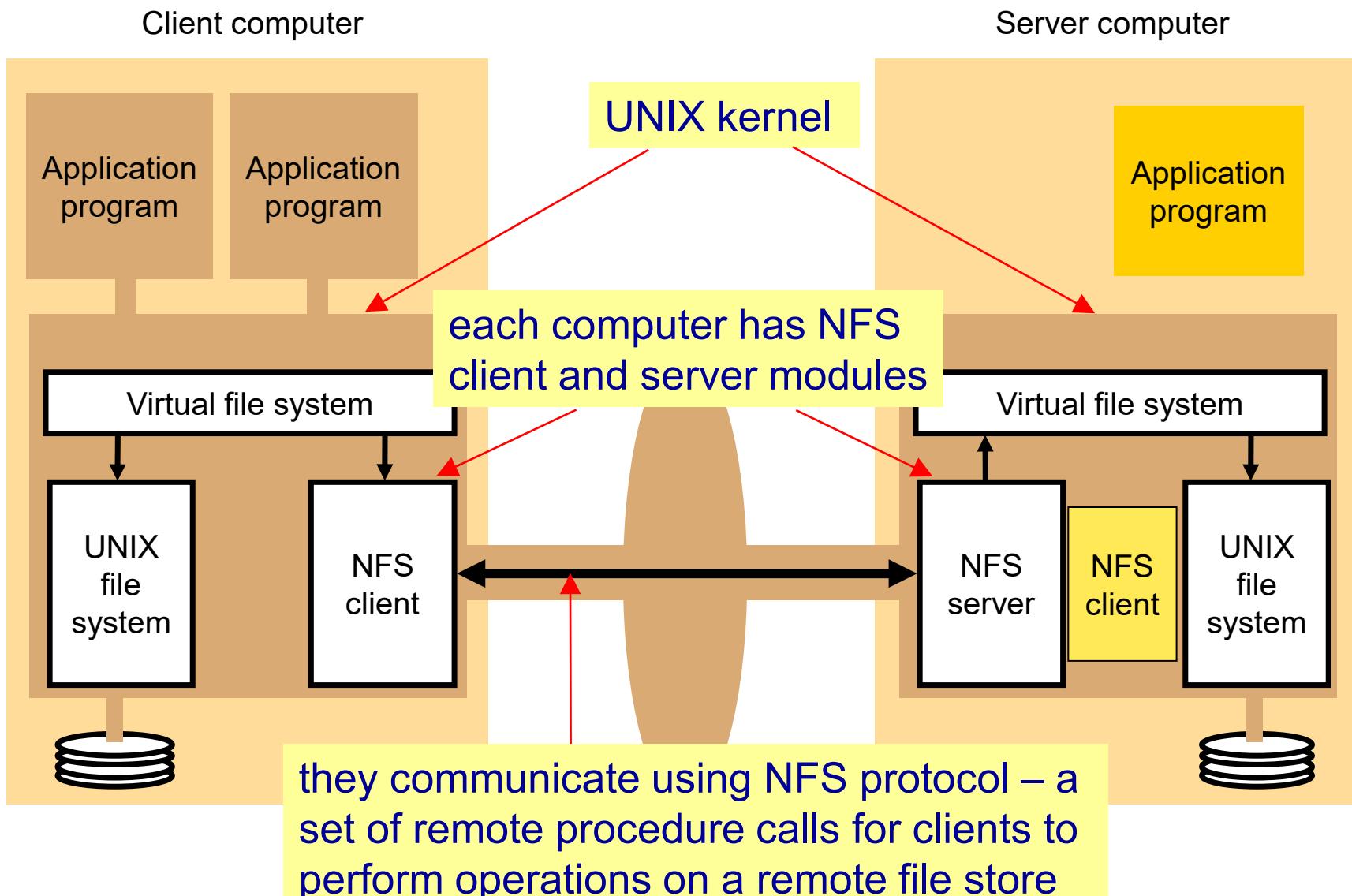
Outline

- Introduction
- Sun Network File System
 - Architecture
 - Client-Server Communication
 - Client Caching
- Andrew and Coda File Systems
- Summary

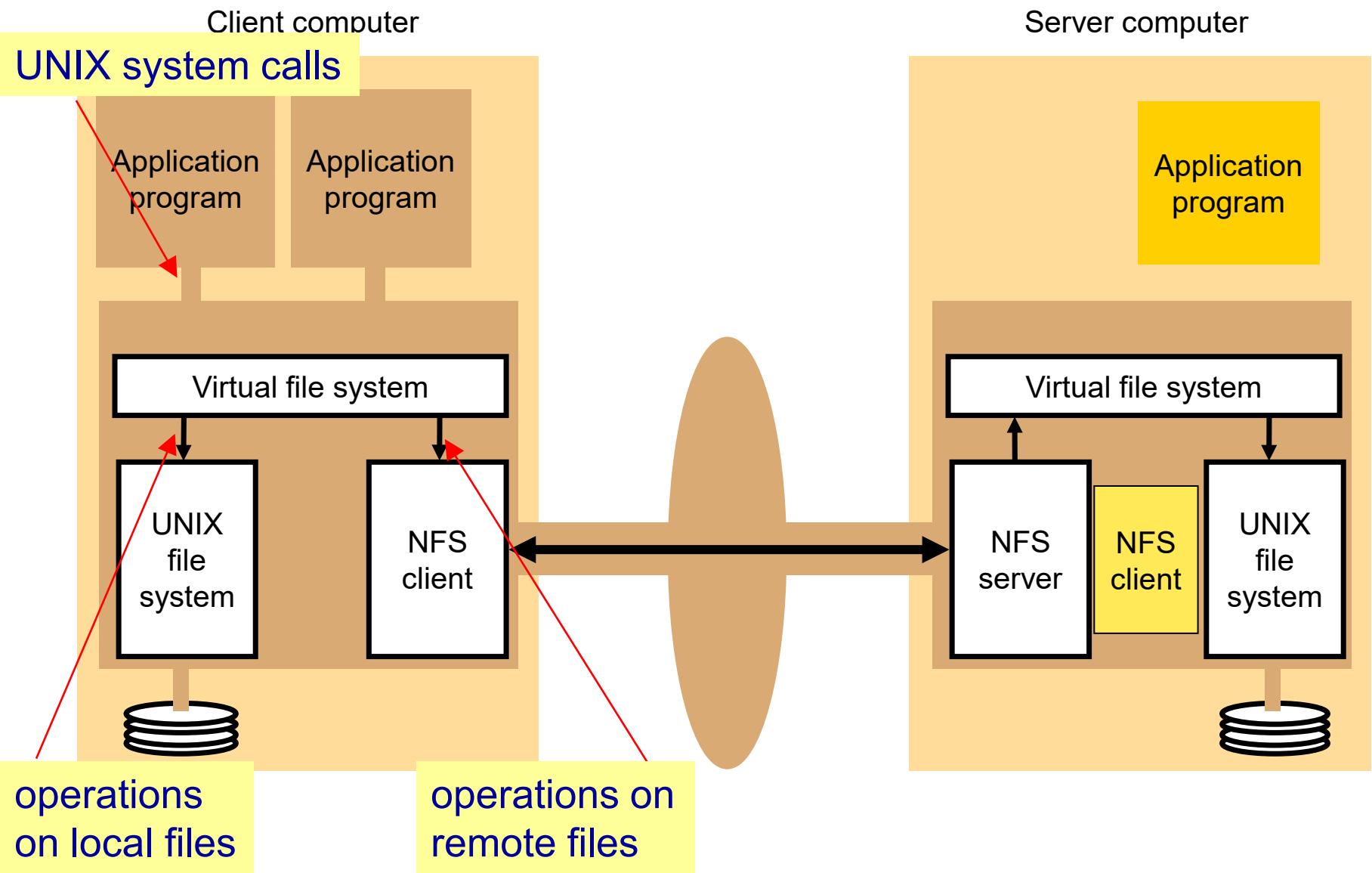
Sun Network File System

- Client-server relationship is symmetric
 - Each computer in an NFS network can act as both a client (accessing files on other computers) and a server (exporting some of its files)
 - Common practice: some dedicated servers + clients
- NFS design is operating-system independent
 - Client and server implementations exist for almost all known operating systems and platforms (Windows 95, Windows NT, MacOS, Linux, ...)
 - We shall describe the implementation on UNIX

Sun Network File System



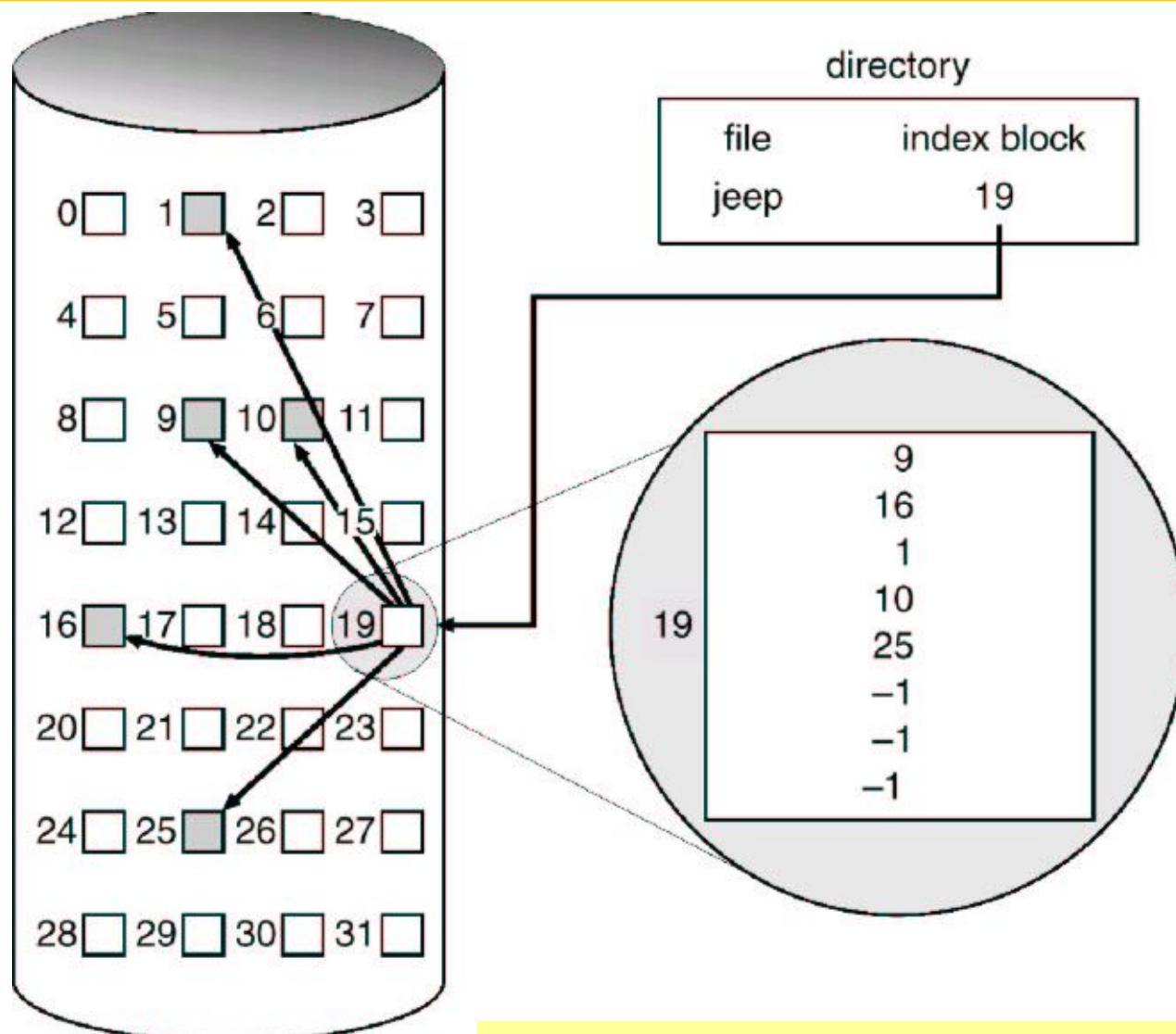
Sun Network File System



Sun Network File System

- Files are accessed by means of file identifiers
 - A file identifier is a reference to a file within a file system
 - Independent of the file name
 - Created by the server hosting the file system
 - Unique with respect to all files in the system
 - File identifiers are called **file handles** in NFS
 - In NFS, a file handle includes two parts:
 - A filesystem identifier (a unique number allocated to each file system when it is created)
 - An identifier that uniquely identifies a file within a file system (e.g., i-node number of the file in UNIX)

UNIX File System (Revisit)



Outline

- Introduction
- Sun Network File System
 - Architecture
 - Client-Server Communication
 - Client Caching
- Andrew and Coda File Systems
- Summary

NFS Server Interface

These operations are used by NFS client in RPC, they are not used directly by user programs, they are stateless & idempotent

lookup(dirfh, name) → fh, attr	Returns file handle and attributes for the file name in the directory dirfh.
create(dirfh, name, attr) → newfh, attr	Creates a new file name in directory dirfh with attributes attr and returns the new file handle and attributes.
remove(dirfh, name) → status	Removes file name from directory dirfh.
getattr(fh) → attr	Returns file attributes of file fh. (Similar to the UNIX stat system call.)
setattr(fh, attr) → attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) → attr, data	Returns up to count bytes of data from a file starting at offset. Also returns the latest attributes of the file.
write(fh, offset, count, data) → attr	Writes count bytes of data to a file starting at offset. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) → status	Changes the name of file name in directory dirfh to toname in directory to todirfh
link(newdirfh, newname, dirfh, name) → status	Creates an entry newname in the directory newdirfh which refers to file name in the directory dirfh.

NFS Server Interface

These operations are used by NFS client in RPC, they are not used directly by user programs, they are stateless & idempotent

symlink(newdirfh, newname, string) → status	Creates an entry newname in the directory newdirfh of type symbolic link with the value string. The server does not interpret the string but makes a symbolic link file to hold it.
readlink(fh) → string	Returns the string that is associated with the symbolic link file identified by fh.
mkdir(dirfh, name, attr) → newfh, attr	Creates a new directory name with attributes attr and returns the new file handle and attributes.
rmdir(dirfh, name) → status	Removes the empty directory name from the parent directory dirfh. Fails if the directory is not empty.
readdir(dirfh, cookie, count) → entries	Returns up to count bytes of directory entries from the directory dirfh. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a cookie. The cookie is used in subsequent readdir calls to start reading from the following entry. If the value of cookie is 0, reads from the first entry in the directory.
statfs(fh) → fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file fh.

NFS Server Interface

- The operations are used by NFS client in RPC, they are not used directly by user programs
- The operations are **stateless and idempotent**
 - No open and close operations, no read-write pointer
 - Each RPC request from a client contains all information needed (including the file handle) to process the request
 - Idempotent → allow at-least-once invocation semantics

NFS Client

- Supply an interface suitable for use by conventional application programs
- The interface emulates UNIX file system semantics
 - How to translate UNIX file system calls to the RPC operations in NFS server interface?
- Transfer blocks of files to and from NFS servers

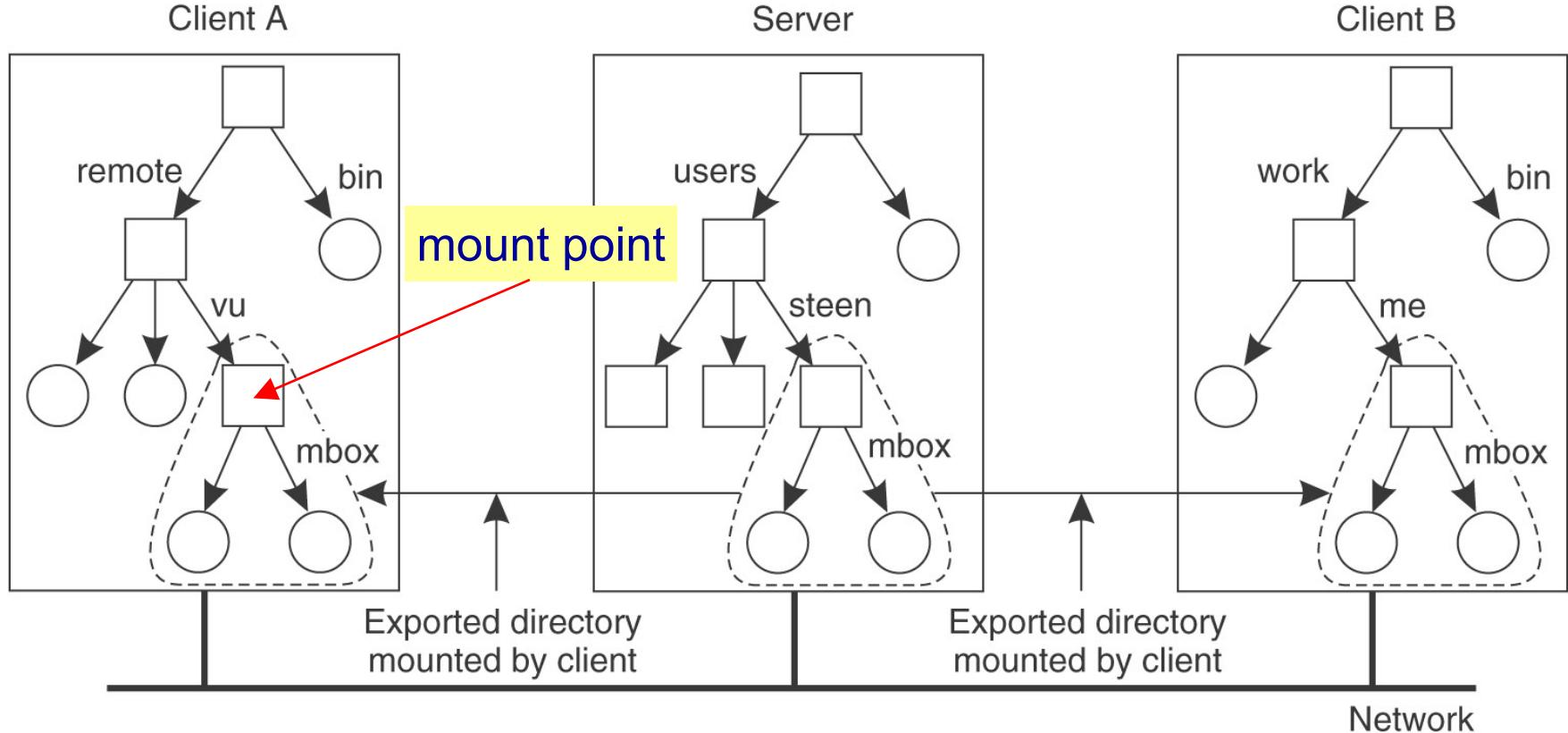
Mount Service

- How does a client obtain file handles for a remote file system?
 - File systems have to be **exported** by the server that holds them and be **mounted** by a client before they can be accessed by the client
- At the server
 - A mount service process runs at user level on each server
 - A file (/etc/exports) contains the names of local file systems available for remote mounting and which hosts are permitted to mount them (access list)

Mount Service

- At the client
 - Use a modified UNIX mount command to request mounting, specifying **remote host name, pathname of a directory in remote file system, local name to be mounted**
 - mount command communicates with mount service process using a mount protocol which is an RPC protocol
 - Request passes remote directory pathname to server
 - Reply returns the file handle of the directory as result
 - A table of mounted file systems is maintained at the client
 - NFS does not enforce a single network-wide file name space – clients can assign different local names to the same remote directory

Example of Mount Service



file named /remote/vu/mbox at client A is named /work/me/mbox at client B
remote files may have different pathnames on different clients

Mount Service

- Pathname translation

- Pathnames are parsed, and their translation is performed in **an iterative manner (step-by-step)** by clients – a single part of a pathname is looked up in each step
- The lookup operation in the NFS server interface looks for a single part of a pathname in a given directory and returns the corresponding file handle and file attributes
- File handle returned in the previous step is used as a parameter in the next lookup step

Mount Service

■ Pathname translation

- If the root directory of a remote server is mounted, access to /bin/draw/readme on the server requires three lookup requests:
 - $\text{lookup}(\text{rootfh}, \text{"bin"}) \rightarrow \text{binfh}$
 - $\text{lookup}(\text{binfh}, \text{"draw"}) \rightarrow \text{drawfh}$
 - $\text{lookup}(\text{drawfh}, \text{"readme"}) \rightarrow \text{readmefh}$
- How to improve efficiency? – caching
 - To take advantage of reference locality to files and directories
 - For example, if /bin/draw/install is accessed soon after /bin/draw/readme on the same server, only one lookup request to the server is needed

Outline

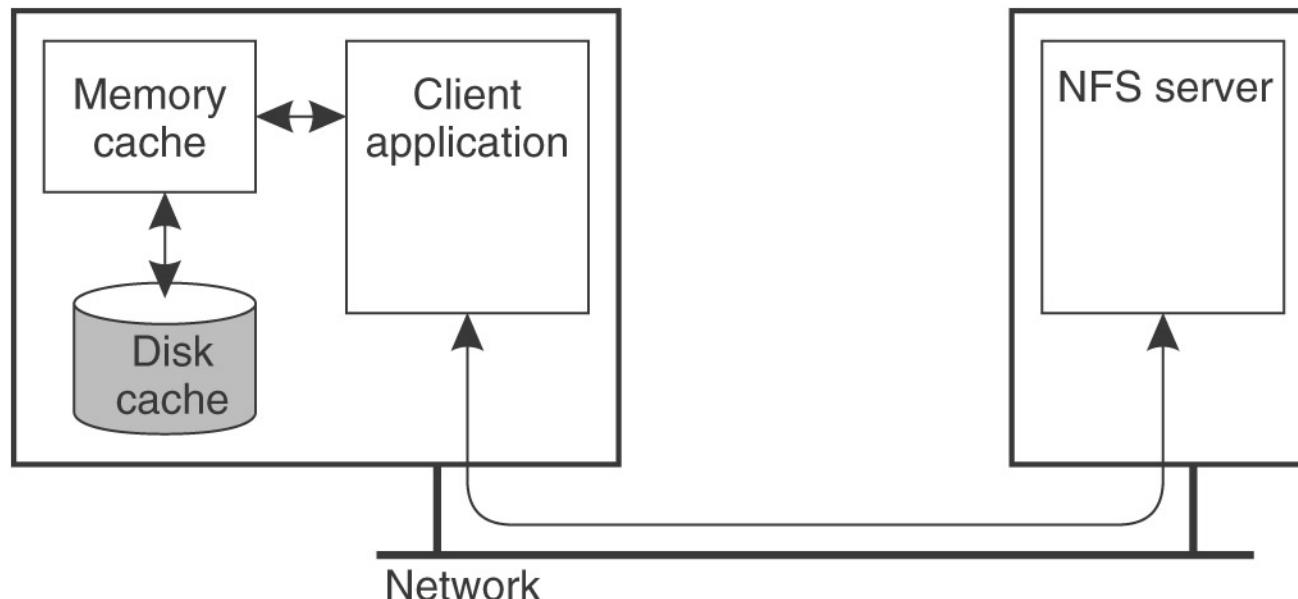
- Introduction
- Sun Network File System
 - Architecture
 - Client-Server Communication
 - Client Caching
- Andrew and Coda File Systems
- Summary

Caching in File Systems

- In the conventional (local) UNIX system, file data read from disk can be cached in the main memory
 - A later read/write request can be satisfied without another disk access
- In NFS, the servers cache data in main memory just like local file systems

Client Caching in NFS

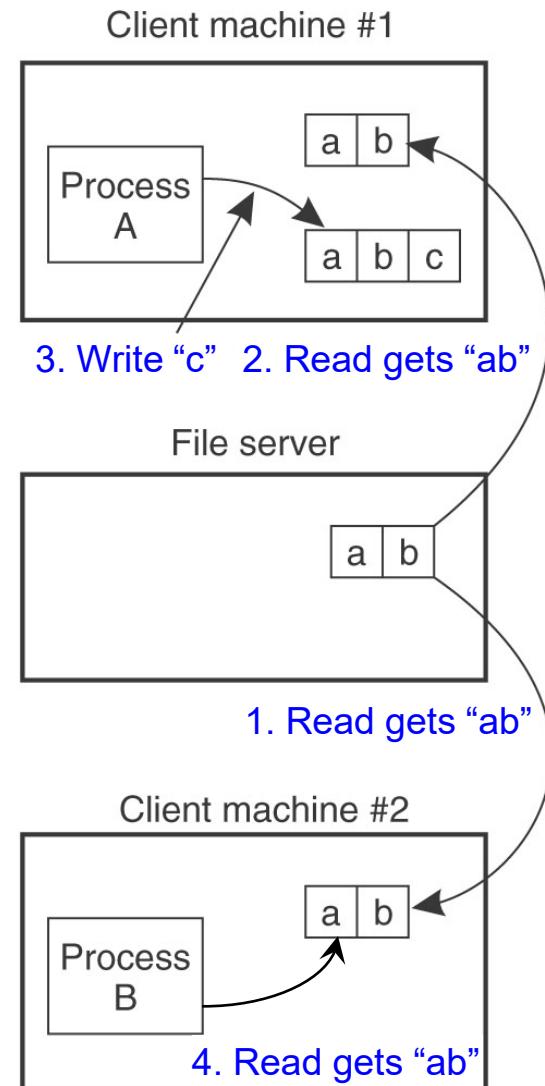
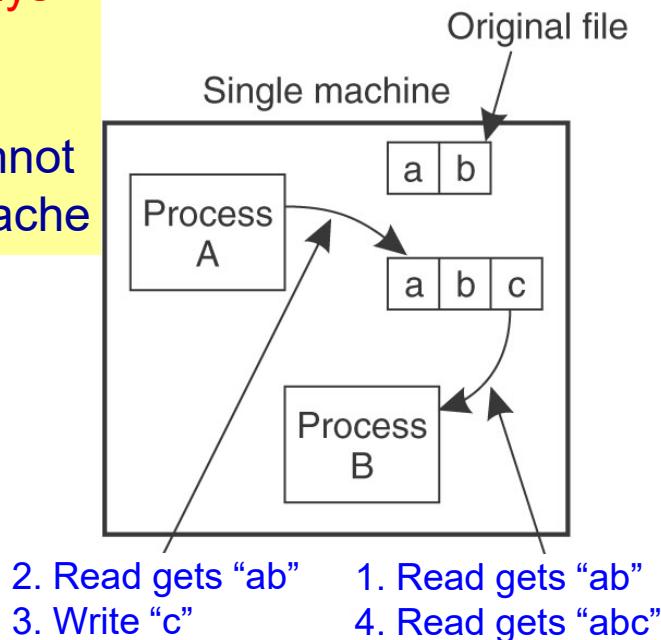
- NFS client also caches file data (i.e., results of read, write etc.)
 - Objective: to reduce communication with server
 - Granularity: block
 - Mainly cache in main memory, may also cache on disk



Client Caching in NFS

- Comparison of caching in local systems and in distributed systems

In local systems,
cache is **always**
up-to-date
because file
accesses cannot
bypass the cache



Client Caching in NFS

- Writes by a client do not result in immediate updating of cached copies of the same file in other clients
- Potentially different versions of files at different clients
- How to maintain consistency? – Clients are responsible for **polling** the server to check the currency of cached data they hold

Client Caching in NFS

- Each cache entry is tagged with two timestamps
 - T_c : the time when the cache entry was last validated
 - $T_{mclient}$: the time when the cached file was last modified at server
 - Entry is considered valid when $T - T_c < t$, where t is a freshness interval (typically 3 to 60 sec) and T is current time
- Perform validity check whenever a cache entry is used
 - Evaluate whether $T - T_c < t$ (this does not need access to server)
 - If $T - T_c < t$, no need to evaluate further, read data from cache
 - If $T - T_c \geq t$, issue getattr call to server to obtain $T_{mserver}$
 - If $T_{mclient} = T_{mserver}$, the cache entry is valid (the data have not been modified at the server) and T_c is updated to current time
 - If $T_{mclient} < T_{mserver}$, the cache entry is invalidated, and a new request is sent to server for updated data, T_c is also updated

Comparison of Consistency Maintenance

- Local file system: **one-copy update semantics** – when a client modifies a file, all other clients see the updates immediately
- Distributed file system: difficult to achieve one-copy update semantics due to inevitable network delay in update propagation
 - NFS provides a close **approximation** to one-copy semantics and meets the needs of many applications

Summary of NFS

- Access transparency is achieved
 - Applications use the same file operations for both local and remote files
- Location transparency? Not really
 - Different clients can mount the same server directory to different local directories (no common view of shared files)

Summary of NFS

- Stateless servers and idempotent operations
 - No need for servers to recover any state after resuming execution from failure
 - A client failure has no effect on any server that it may be accessing since servers do not hold states on behalf of their clients
- Use a timeout mechanism to provide a close approximation to one-copy update semantics

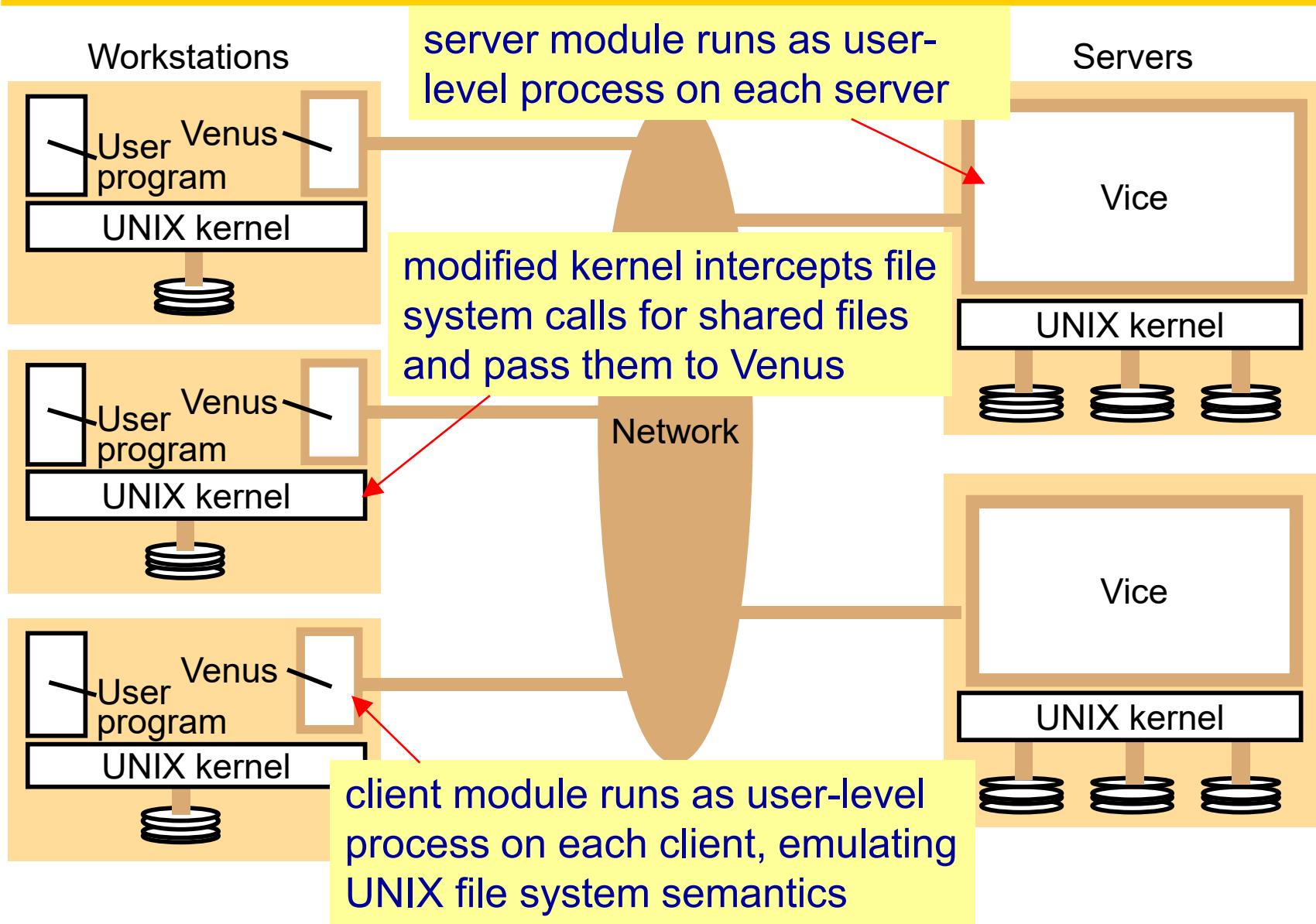
Outline

- Introduction
- Sun Network File System
- Andrew and Coda File Systems
 - AFS Architecture
 - Whole-File Serving and Caching
 - Coda File System
- Summary

Andrew File System

- Scalability as outstanding feature
 - Designed to perform well with large number of active users
- AFS nodes are partitioned into two groups
 - One group: dedicated file servers (usually small in number)
 - Another group: a large number of clients
 - Servers and clients all run UNIX
- Similar to NFS: access to AFS files is via normal UNIX file system operations

Andrew File System: Architecture



Andrew File System

- Similar to NFS
 - Venus clients and vice servers communicate using RPC
 - User programs use conventional UNIX pathnames to refer to files
 - Venus first translates the pathnames into file identifiers using a step-by-step lookup from the file directories held in Vice servers
 - Files are then accessed by file identifiers

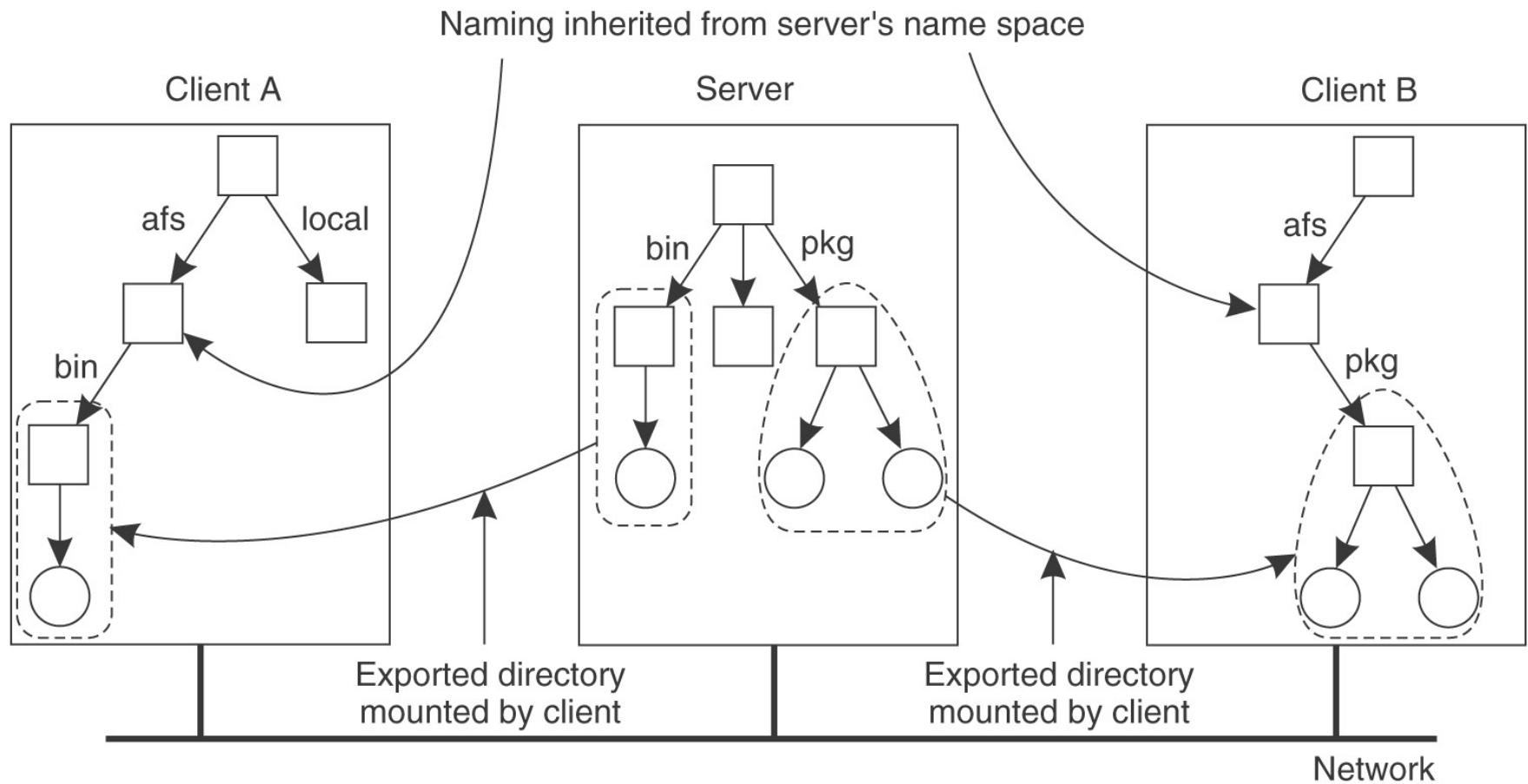
AFS: Shared File Name Space

- Shared file name space
 - Vice servers maintain a globally shared file name space
 - Venus clients follow the structure of shared name space
- Clients have access to shared name space by means of a special local subdirectory (e.g., /afs)
 - Whenever a client accesses a file in this subdirectory, Venus ensures that the appropriate part of the shared name space is mounted locally
 - When mounting, Venus ensures that the naming graph rooted at this subdirectory is always a subgraph of the complete shared name space maintained by Vice servers
 - Shared files have the same pathname on different clients

AFS: Shared File Name Space

local files at the client side: stored on local disks and handled as normal UNIX files

shared files: stored on servers and copies of them are cached on local disks



AFS: Volume

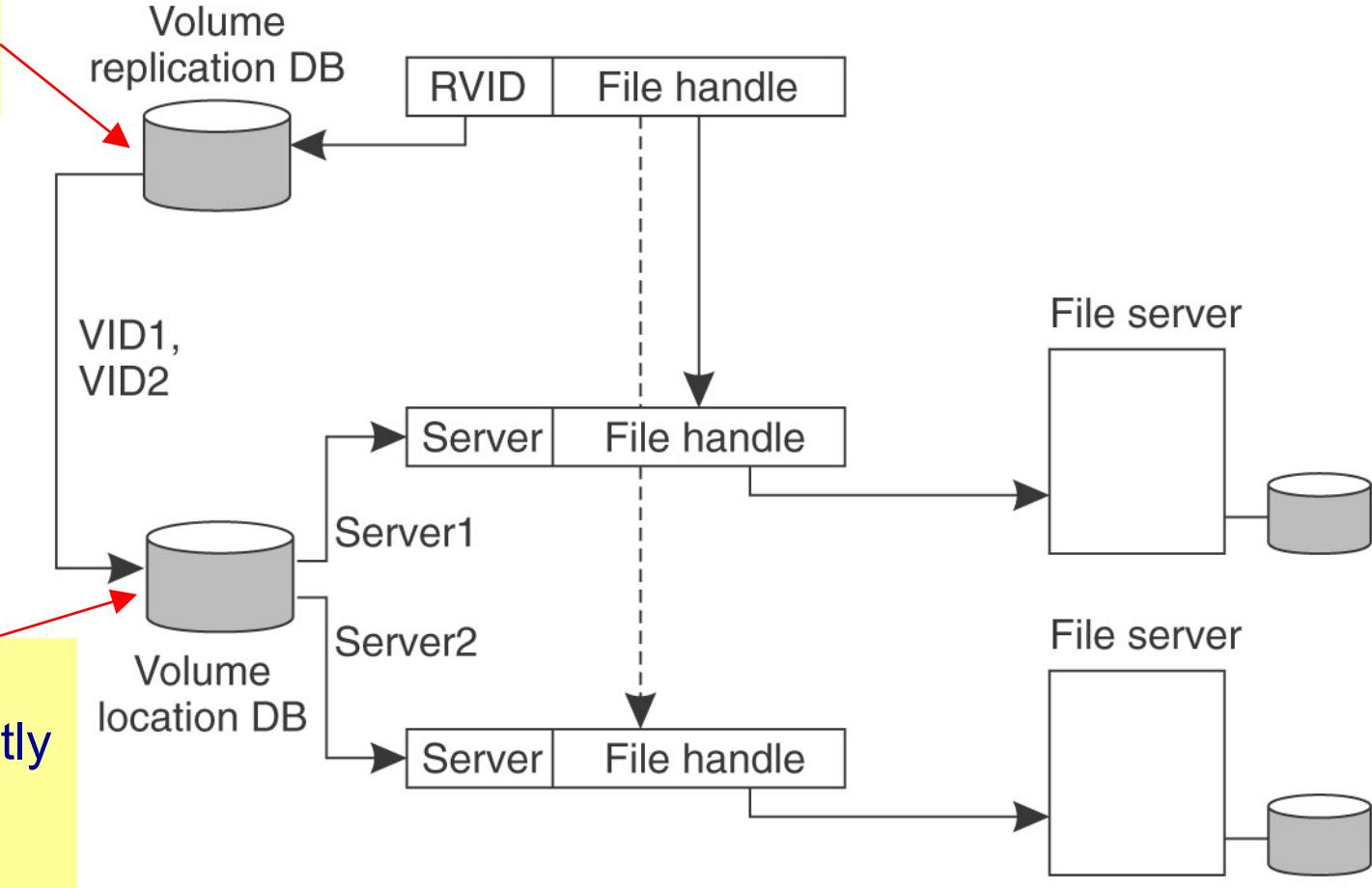
- Shared files are grouped into **volumes** for ease of replication, location and movement
 - Basic unit of mounting and server-side replication
 - Volumes are generally smaller than UNIX file systems (e.g., each user's personal files, system binaries, documentation and library code)
 - Each volume corresponds to a partial subtree in the shared name space maintained by Vice servers

AFS: Server-Side Replication

- Server-side replication
 - Each file is contained in exactly one (logical) volume
 - A volume may be replicated across several servers → **a logical volume may have several physical volumes**
 - Each logical volume is associated with a Replicated Volume Identifier (RVID) which is location and replication-independent
 - Each physical volume has its own Volume Identifier (VID) which is location-independent
 - Each shared file is identified by a unique file identifier
 - RVID + file handle (uniquely identifying a file within a volume)

AFS: Location

Map each RVID to a list of VIDs



- The databases are fully replicated at each server

AFS: Server-Side Replication

- Consistency in server-side replication
 - Read-One, Write-All
 - When a client wants to read a file, it contacts one of the physical volumes to which the file belongs
 - When a client closes a session on an updated file, it transfers the updates in parallel to all physical volumes to which the file belongs

Outline

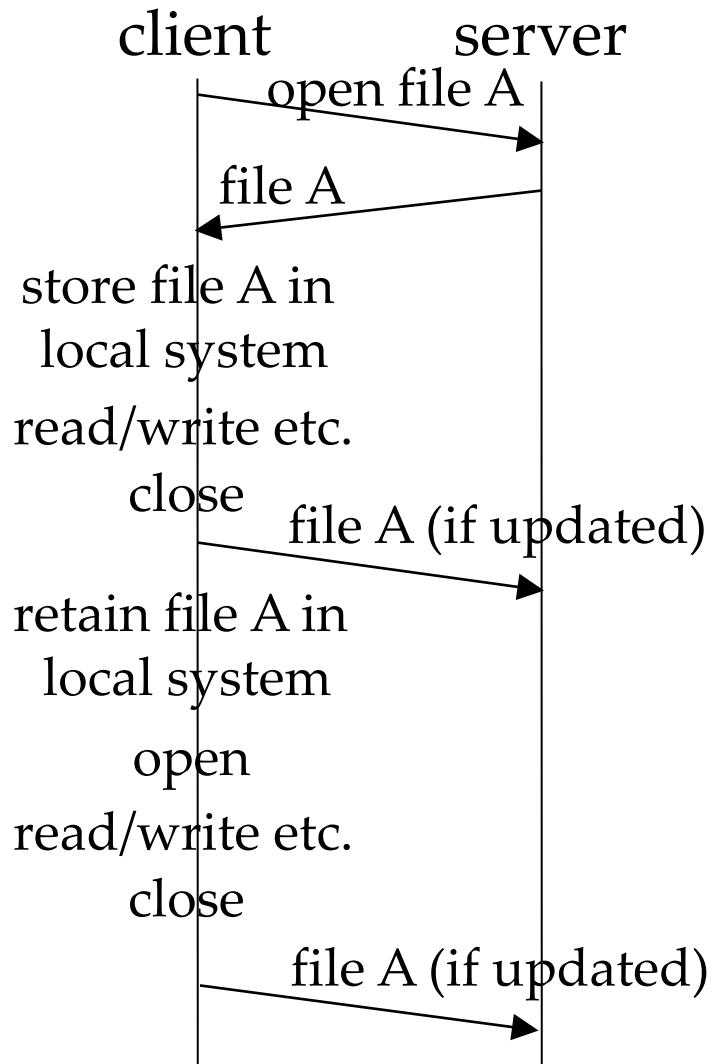
- Introduction
- Sun Network File System
- Andrew and Coda File Systems
 - AFS Architecture
 - Whole-File Serving and Caching
 - Coda File System
- Summary

Whole-File Serving and Caching

- Observations of typical UNIX workloads
 - Read operations are much more common than writes
 - Files are usually accessed in entirety; sequential access is common; random access is rare
 - Most files are read and written by only one user
 - Files are referenced in bursts (temporal locality)
- AFS strategy: caching of whole files at clients
 - **Whole-file serving:** entire files are transmitted to clients by AFS servers
 - **Whole-file caching:** files transferred to a client are stored in a cache on local disk to satisfy future requests

Whole-File Serving and Caching

- How does it work?
 - On opening a file, **the entire file** is transmitted to the client
 - All reads and writes are performed **on the local copy**
 - On closing the file, the client transmits the updates to the server & **retains the local copy**
- Local cache has large capacity (> working set of user)
 - Client is largely independent of servers once a working set of the user has been cached



Scalability vs. Update Semantics

- **One-copy update semantics:** when a client modifies a file, all other clients see the updates immediately
 - Enforcing this strict semantics in a distributed system causes excessive network traffic and performance degradation
 - To be more scalable → relax on update semantics
- **Session update semantics:** all other clients are able to see a modified file only after the file is closed by the client that modified it
 - Reducing client-server communication during the session
 - Justified by the observations of typical UNIX workloads
 - AFS uses session update semantics

Cache Consistency of AFS

- **Callback mechanism**

- Main idea: for each file, the server keeps track of which clients have a copy of the file and guarantees to notify these clients when any of them modifies the file
- **Callback promise** – token issued by Vice server to Venus client (one callback promise per file)
- A callback promise has two states: **valid and cancelled**
 - Valid: the cached copy of the file is fresh
 - Cancelled: the cached copy of the file is out-of-date

Cache Consistency of AFS

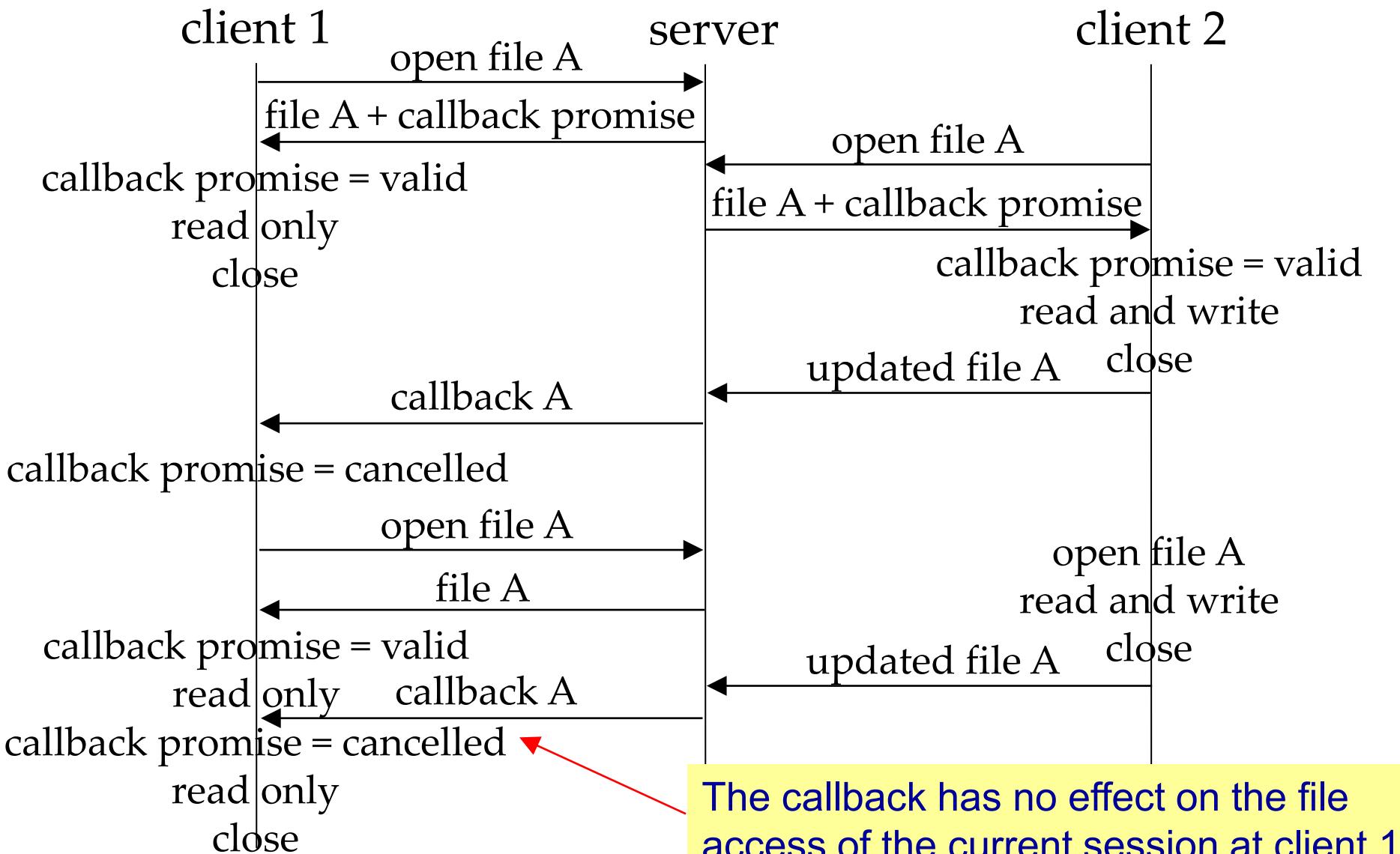
- Callback mechanism
 - When a client wants to open a file
 - If the file is not cached, fetch a copy of the file + callback promise from the server (initial state is valid)
 - If the file is cached and its callback promise is cancelled, fetch a fresh copy of the file from the server + set callback promise to valid
 - If the file is cached and its callback promise is valid, the cached copy is opened without contacting server
 - When a client closes a file
 - If the file was modified during the session, the updates are sent to the server

Cache Consistency of AFS

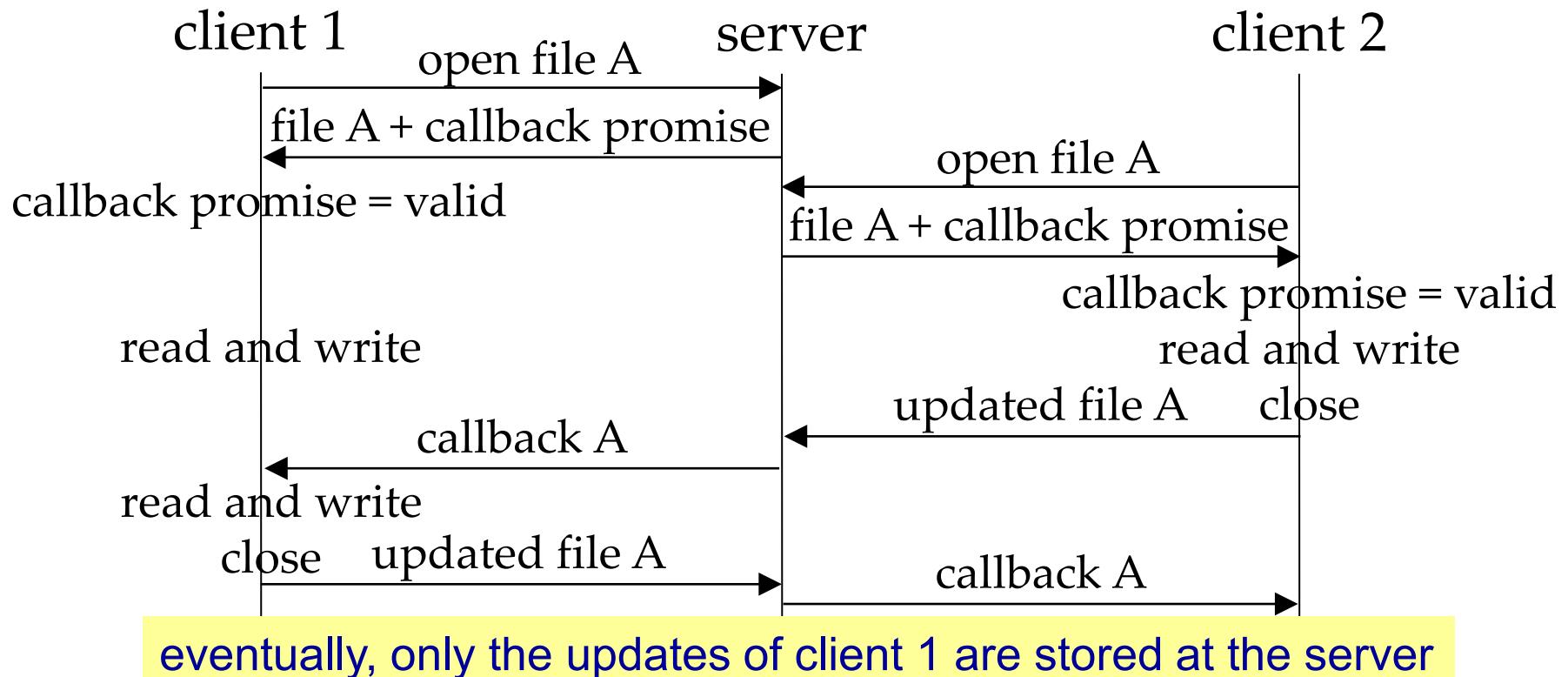
- Callback mechanism

- When the server receives the updates of a file from a client,
the server notifies all other clients holding valid callback promises for the updated file by sending callbacks
 - To do so, for each file, the server maintains a list of clients to which callback promises have been issued and are valid
 - When the server sends a fresh file copy to a client, the client is inserted into the list for the file
 - A callback is a remote procedure call from Vice to Venus
 - On receiving a callback from the server, the client sets the callback promise to cancelled
 - After sending callbacks, the server removes the clients from the list for the relevant file

Cache Consistency of AFS



Cache Consistency of AFS



- If multiple clients write to a file concurrently, all updates are silently lost except those of the last client closing the file
- Clients must implement concurrency control independently if they require it

Summary of AFS

- Access transparency is achieved
- A high degree of location transparency
- Servers are not stateless
- Relaxed consistency – session update semantics
- Scalable – callback mechanism reduces client-server communication

Outline

- Introduction
- Sun Network File System
- Andrew and Coda File Systems
 - AFS Architecture
 - Whole-File Serving and Caching
 - Coda File System
- Summary

Coda File System

- A descendant of Andrew File System
- High availability as the main goal of design
 - Allow a client to **continue operation despite being disconnected from a server**
 - Due to network failures and server failures, some or all file servers may be temporarily unavailable
 - A mobile client may deliberately disconnect from the file service

Coda File System

- A client is disconnected with respect to a volume if it cannot access any server having a copy of the volume
 - In most file systems (e.g., NFS), a client is not allowed to proceed unless it can contact at least one server
 - In Coda, client-side whole-file caching → a client resorts to using its locally cached copy of the files when disconnected from servers and reconcile later when the connection is established again

Coda File System

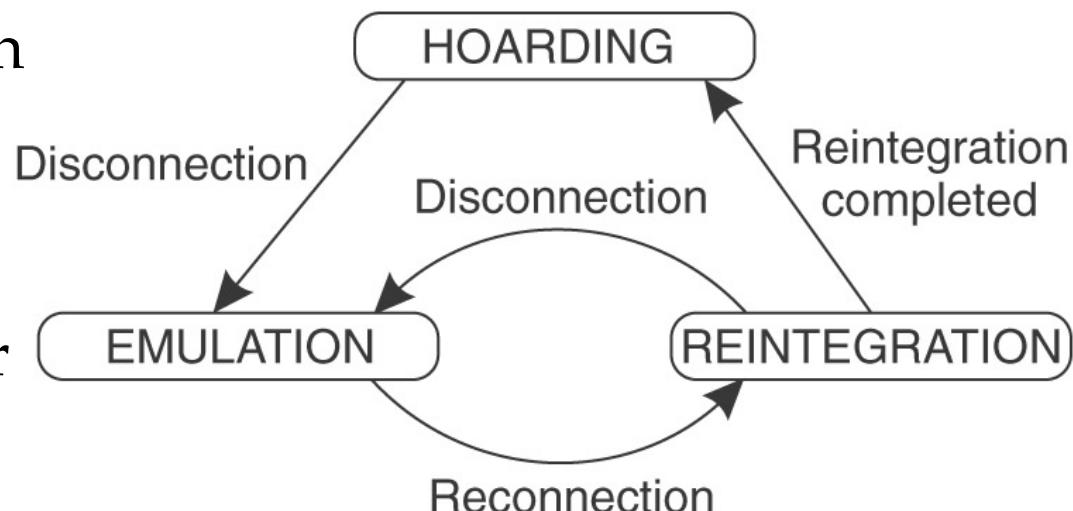
- Main issue to make disconnected operation a success – ensure a client's cache contains those files that will be accessed during disconnection
- **Hoarding** – filling the cache in advance with the useful files

Hoarding

- Ask user to specify useful files in a hoard database
- Compute **a priority** for each file by combining the hoard database with info on recent file accesses
- Fetch files in priority to **reach equilibrium**
 - All cached files have higher priorities than uncached files
 - Cache is full or all uncached files have zero priority
 - Cached files are up-to-date
- File priority may change over time, and cached files may be replaced
- → periodically reorganize the cache to maintain equilibrium (**hoard walk**)

Coda File System

- 3 states of a Coda client with respect to a volume
 - **Hoarder** – when connected to the server(s), the client issues file requests to the server(s) to perform its work and attempts to keep its cache filled with useful data
 - **Emulation** – when disconnected from the server(s), all file requests are directly serviced using the locally cached copy of the files
 - **Reintegration** – when reconnection occurs, the client enters this state to transfer updates to the server



Coda File System

- When the connection is established again, updates made to files during disconnection are transferred to the server
 - Conflicts occur if two or more users open the same file for writing during disconnection
 - In case of conflicts, try **automatic conflict resolution** (in many cases, conflict resolution can be automated in an application-dependent way)
 - If automatic conflict resolution fails, **manual intervention** – users will have to assist manually
 - Conflicts are rare because most files are written by only one user at a time

Summary

Learned about:

- Stateless servers are preferred to stateful servers
- Use client caching to improve performance
 - Granularity: block of file or entire file
- Tradeoff between scalability & update semantics
 - One-copy update semantics (approximated by timeout mechanism)
 - Session update semantics (implemented by callback mechanism)

5. Peer-to-Peer File Sharing Systems

The objective of this lecture is not to encourage you to use P2P file sharing services. CITS does not allow configuration of computers to provide P2P file sharing services. Do not download or upload unlicensed copyrighted materials.

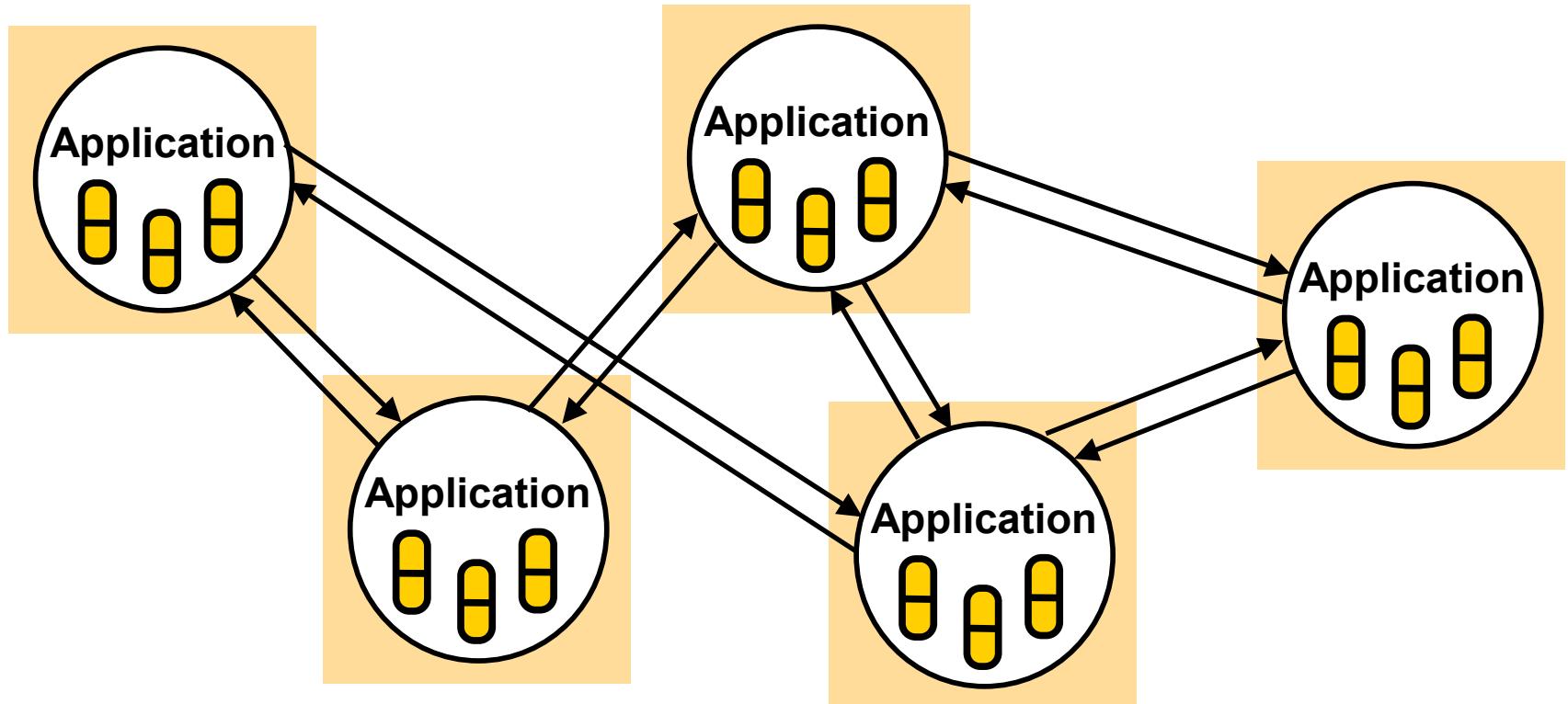
Acknowledgement: A number of slides in this lecture are from Professors Keith W. Ross and Dan Rubenstein

CE4013/CZ4013 Distributed Systems
2021-2022 Semester 2

Outline

- **Introduction**
- Unstructured P2P File Sharing
- Structured DHT Systems
- Summary

Peer-to-Peer Model (Revisit)



- All processes play **similar roles**, they interact cooperatively as peers to perform distributed computation
- No distinction between clients and servers

Peer-to-Peer Systems

- Exploit resources at the edges of the Internet
 - Storage and content
 - CPU cycles
- Every node provides some service that helps other nodes in the network to get service
- Resources at edge have intermittent connectivity
 - They are added and removed from time to time

Peer-to-Peer Systems

- P2P file sharing
 - Napster, Gnutella, KaZaA, eDonkey, BitTorrent, etc.
- P2P communication
 - Instant messaging
 - Voice-over-IP: Skype
- Distributed hash tables (DHT) & their applications
 - Chord, CAN, Pastry, Tapestry

P2P File Sharing

- Alice runs P2P file sharing software on her notebook computer
 - Intermittently connect to the Internet and get new IP address for each connection
 - Register her contents in P2P system
 - Alice asks for “Star Wars”
 - Software displays other peers that have copies of “Star Wars”
 - Alice chooses one of the peers – Bob
 - File is copied from Bob’s PC to Alice’s notebook (P2P)
 - While Alice is downloading from Bob, other users may be downloading other contents from Alice

P2P File Sharing Software

- Allow Alice to open up a directory in her file system
 - Anyone can retrieve a file from the directory
 - Like a web server
- Allow Alice to copy files from other users' open directories
 - Like a web client
- Allow users to search nodes for contents based on keyword matches
 - Like google

P2P File Sharing – Killer Deployments

- Napster
 - Proof of concept
- Gnutella
 - Open source
- KaZaA/FastTrack
 - More KaZaA traffic than Web traffic
- eDonkey/Overnet
 - Appear to use a DHT
- BitTorrent

Outline

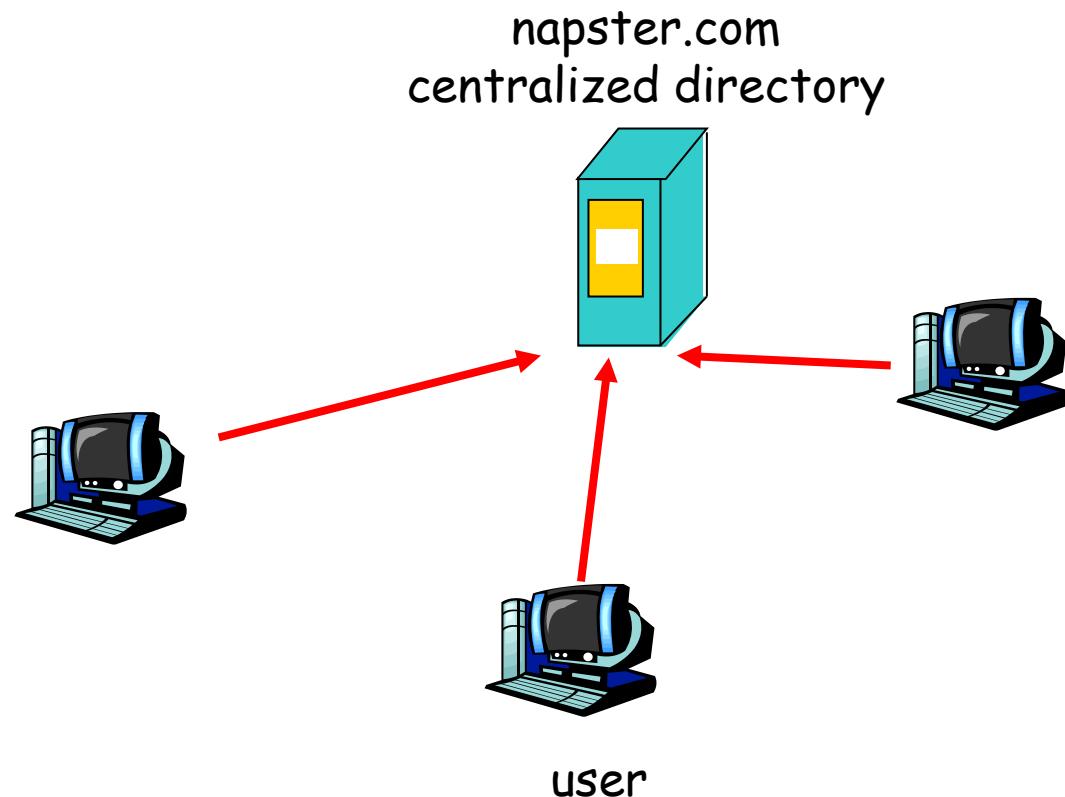
- Introduction
- Unstructured P2P File Sharing
 - Napster
 - Gnutella
 - KaZaA
- Structured DHT Systems
- Summary

Napster

- Mid 1999
 - Paradigm shift – searching and sharing files over the Internet
 - Instructive for what it gets right
-
- Centralized directory server
 - centralized search: simple
 - Distributed download: P2P

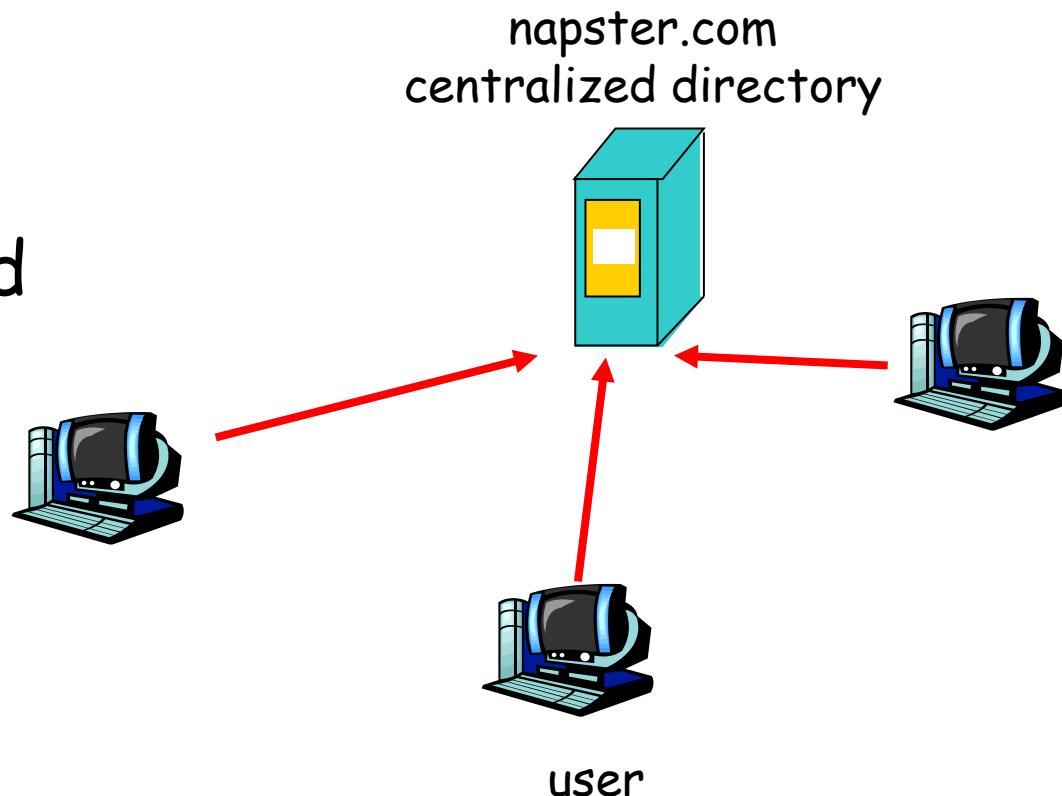
Napster: How does it work?

1. Connect to Napster server



Napster: How does it work?

2. File list and IP address are uploaded

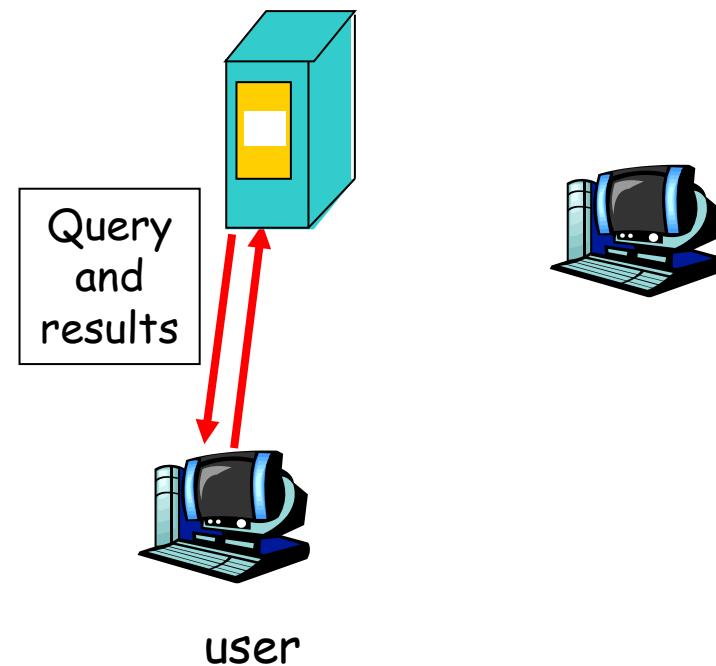


Napster: How does it work?

3. User requests search at server by supplying keywords



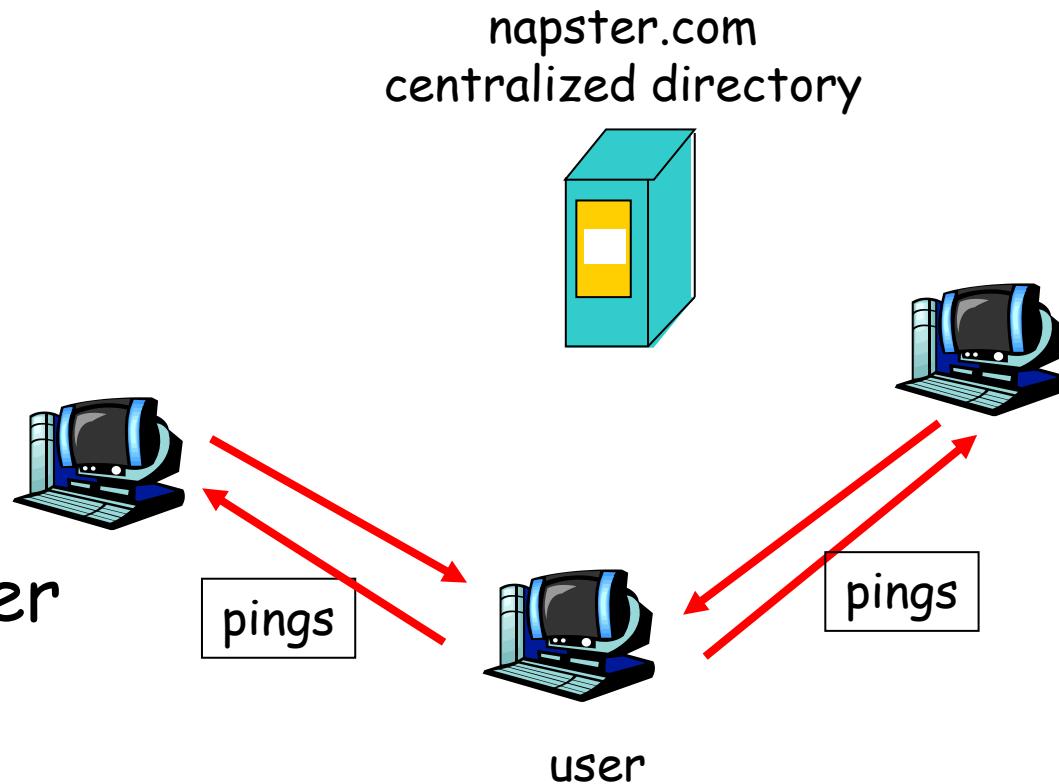
napster.com
centralized directory



Napster: How does it work?

4. User pings hosts that have the wanted file

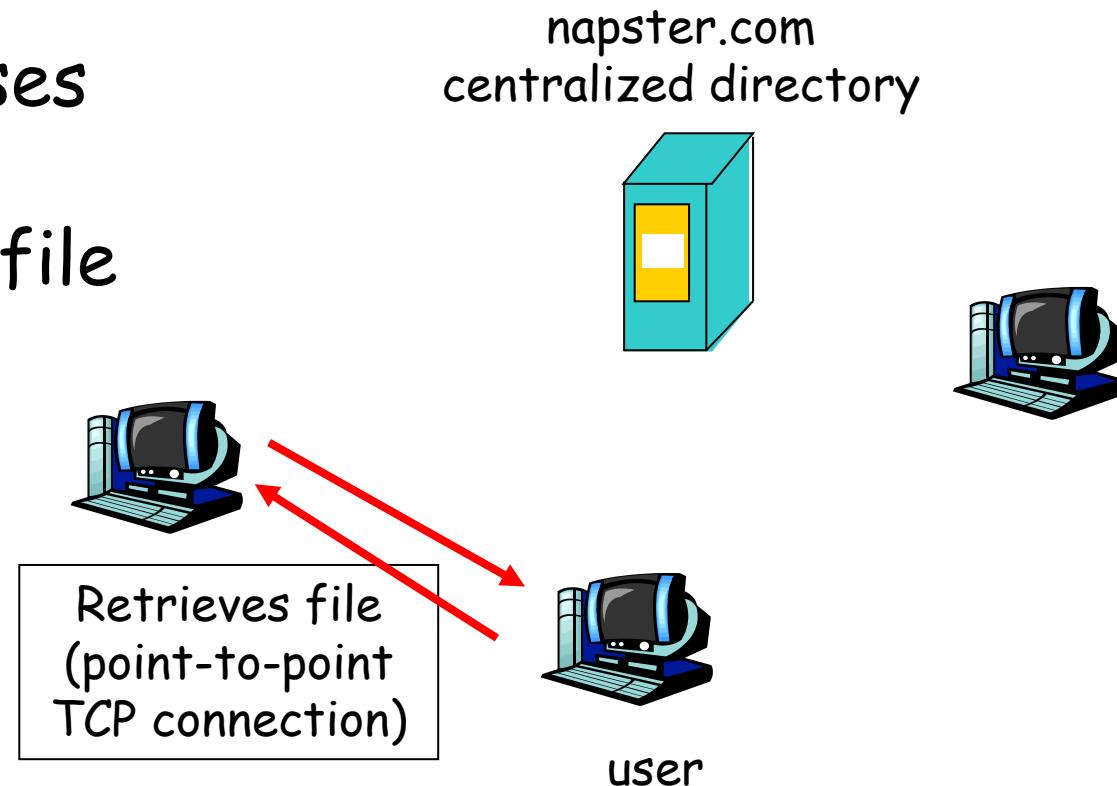
Looks for
best transfer
rate for
downloading



Napster: How does it work?

5. User chooses server and downloads file

Napster's centralized server farm had difficult time keeping up with traffic



A principle in designing distributed systems:
centralized design usually does not scale well,
it also presents a single point of failure

Gnutella

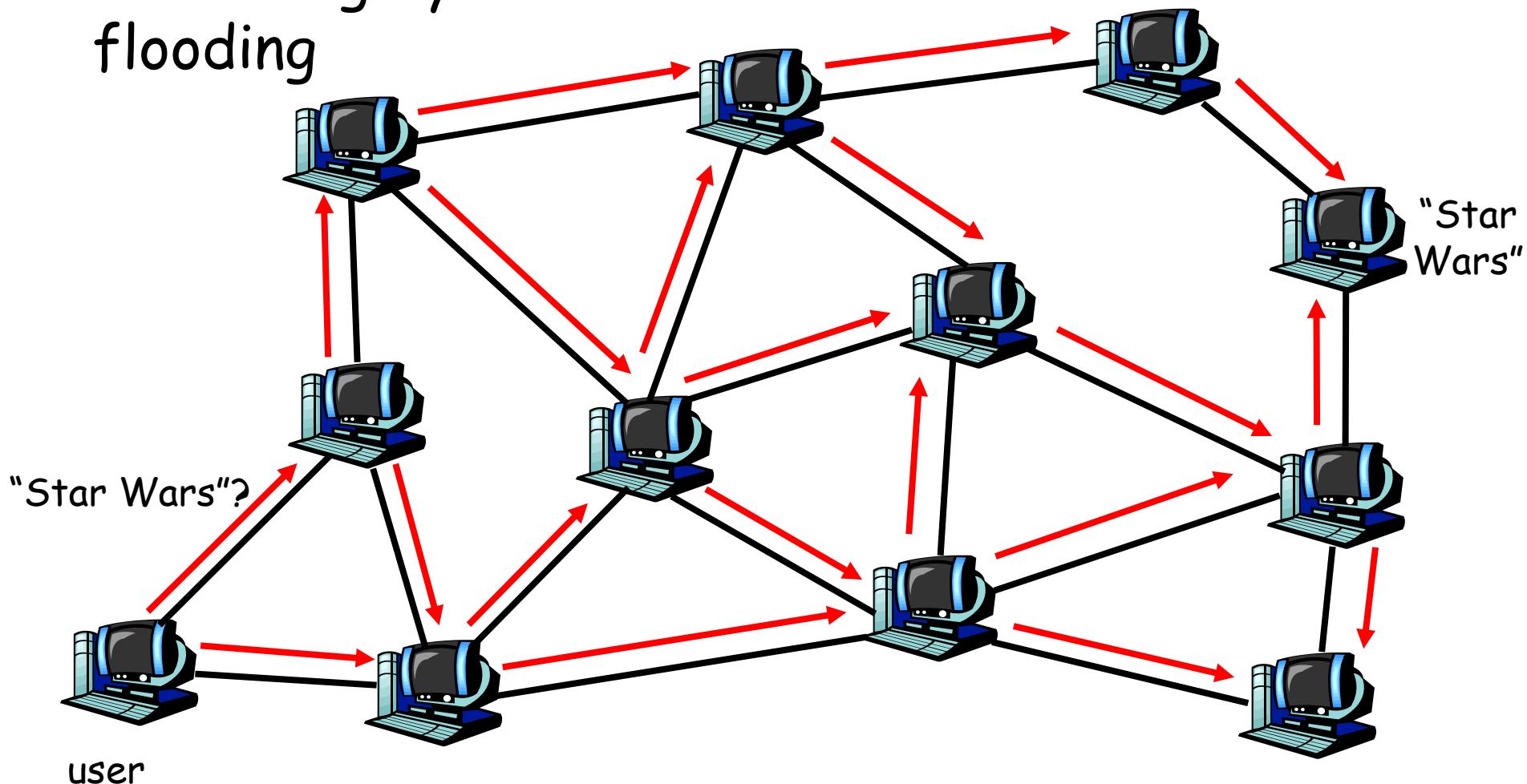
- Late 1999
- Open source
- Focus: **decentralized method** of searching for files
 - No central directory server
 - Each node has a set of “neighbors” to form an overlay network (the node knows its neighbors’ IP addresses)
- Each application instance serves to:
 - Store selected files
 - Route queries from and to its neighboring peers
 - Respond to queries if files are stored locally
 - Serve files

Gnutella: How does it work?

- Searching by **flooding**:
 - If you don't have the file you want, query your neighbors
 - If they don't have it, they contact their neighbors
 - Like breadth-first traversal
 - Reverse path forwarding for responses (not files)

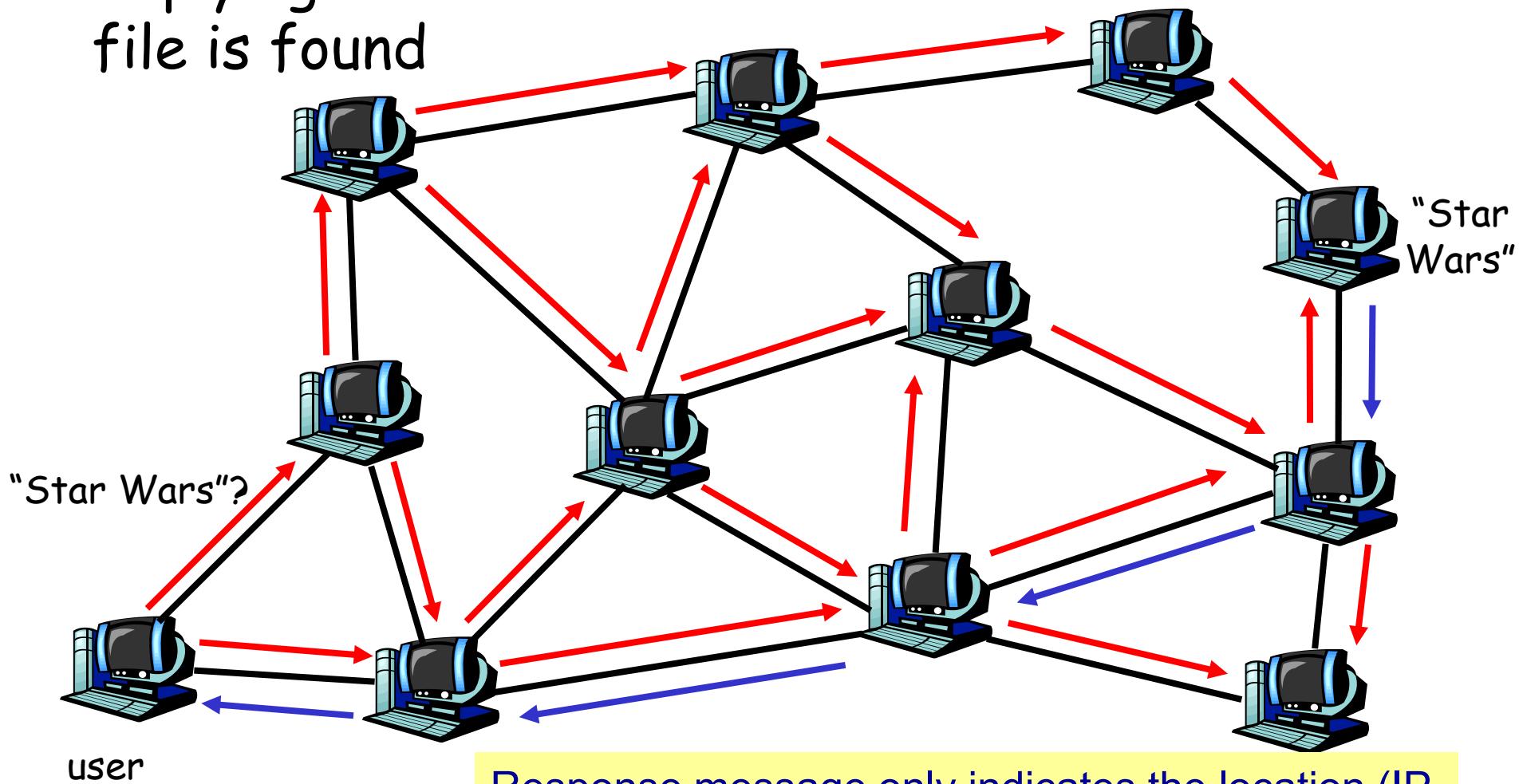
Gnutella: How does it work?

1. Searching by flooding



Gnutella: How does it work?

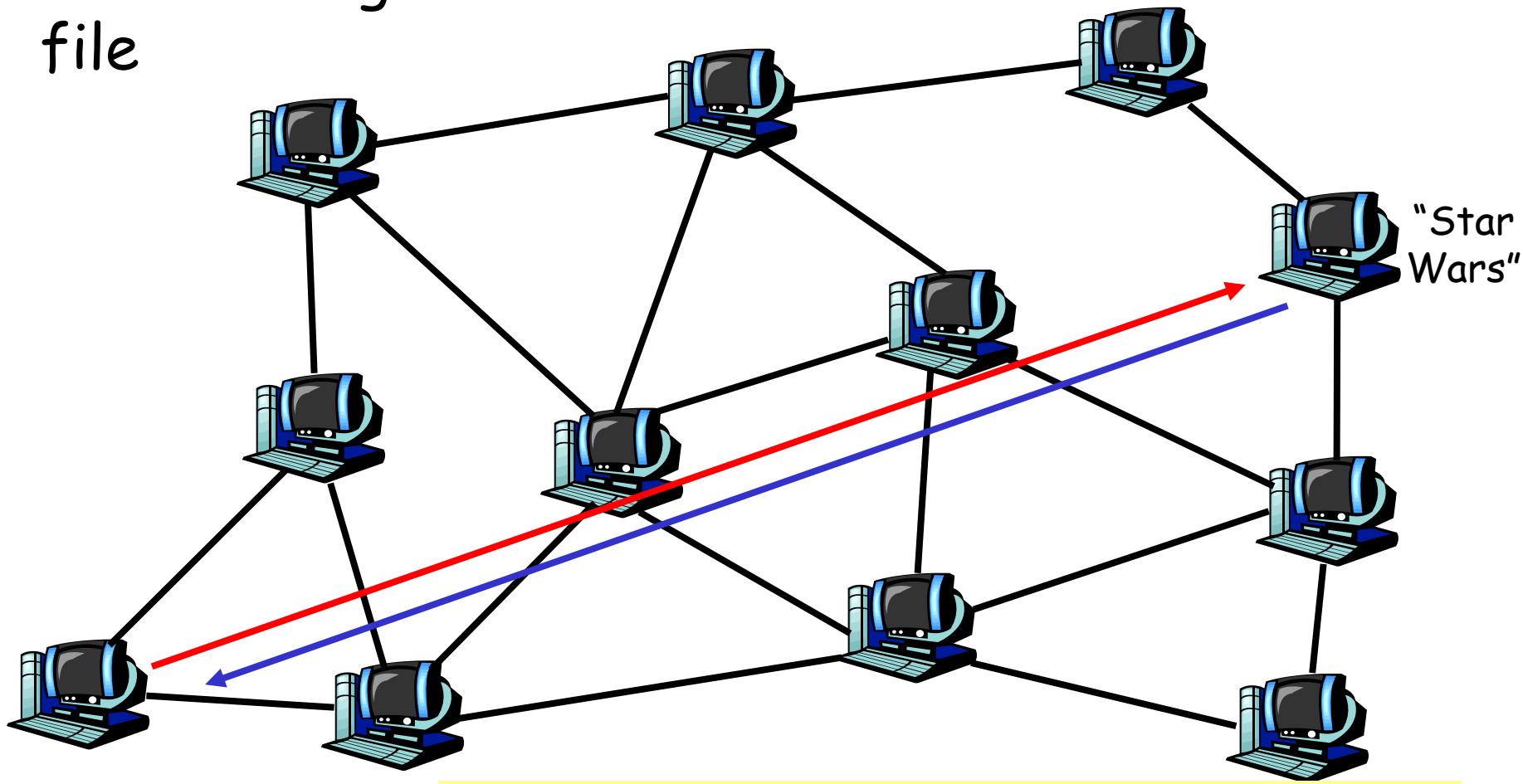
2. Replying that file is found



Response message only indicates the location (IP address) of the file, but does not carry the file itself

Gnutella: How does it work?

3. Downloading file



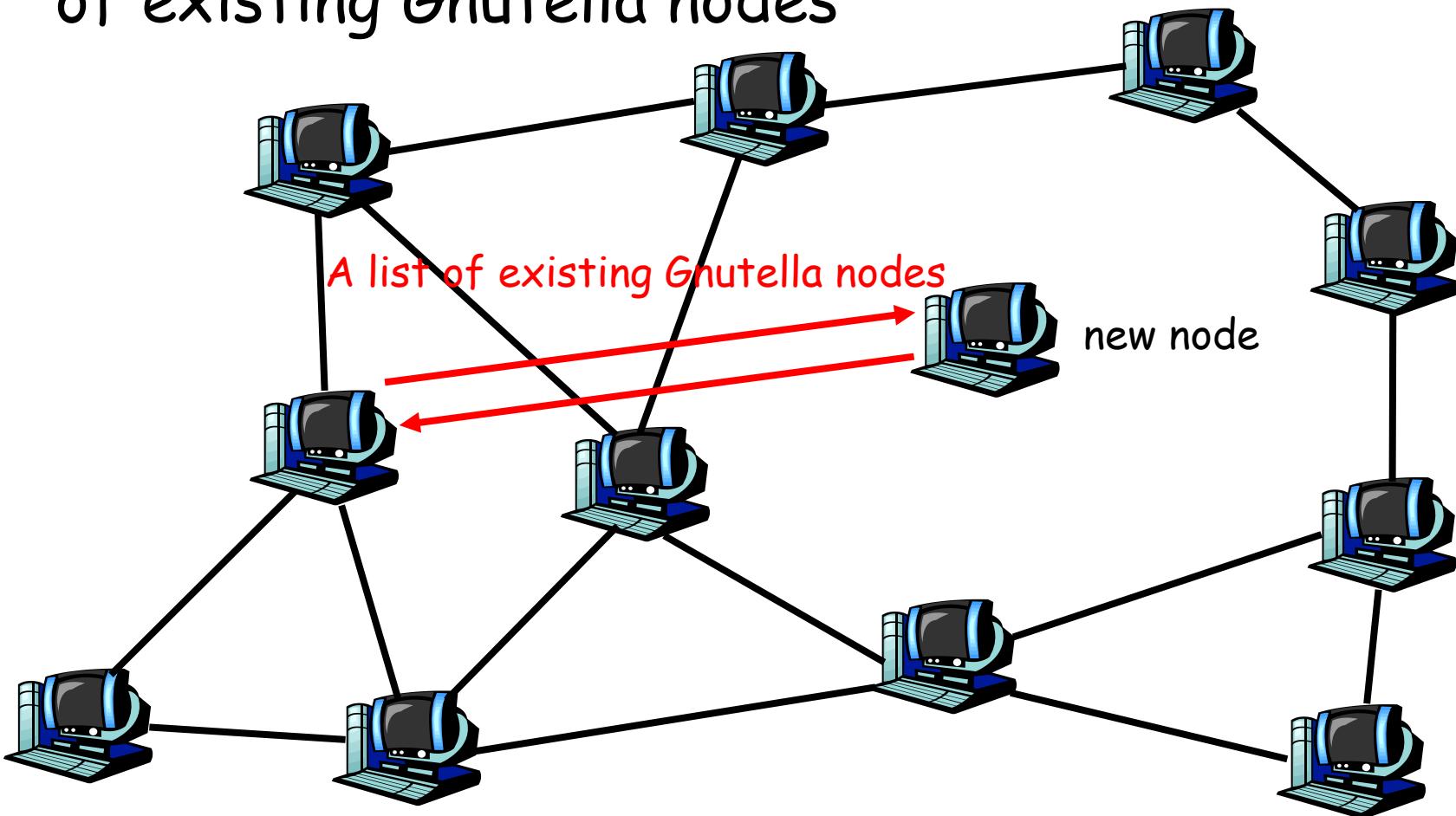
Actual downloading is by point-to-point TCP, the file is not relayed by intermediate nodes

Gnutella: How does it work?

- Limiting the number of hops to travel
 - User attaches a TTL field to his query message and initializes it with **the maximum allowable hop count**
 - For nodes receiving the query
 - If $\text{TTL} > 0$, decrease TTL by 1 and forward the query to neighbors
 - If $\text{TTL} = 0$, stop forwarding and do nothing
 - Can also limit the number of neighbors to forward the query at each hop
- Expanded-ring TTL search
 - Try $\text{TTL} = 1$ first
 - If object not found, try $\text{TTL} = 2$
 - If still not found, try $\text{TTL} = 3, \dots$ (to some max TTL)

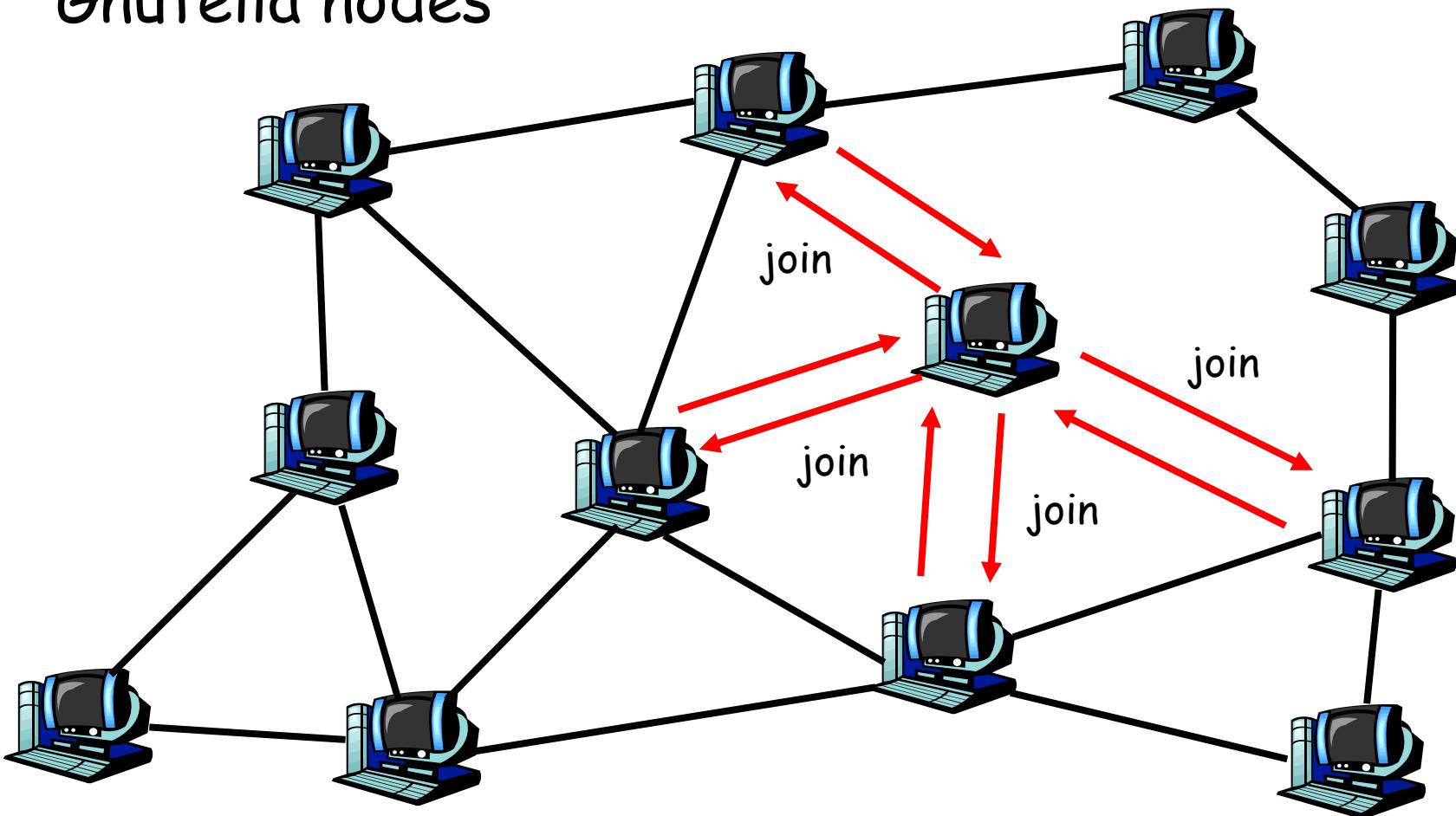
Gnutella: Connection

1. Use bootstrap node to get IP addresses of existing Gnutella nodes



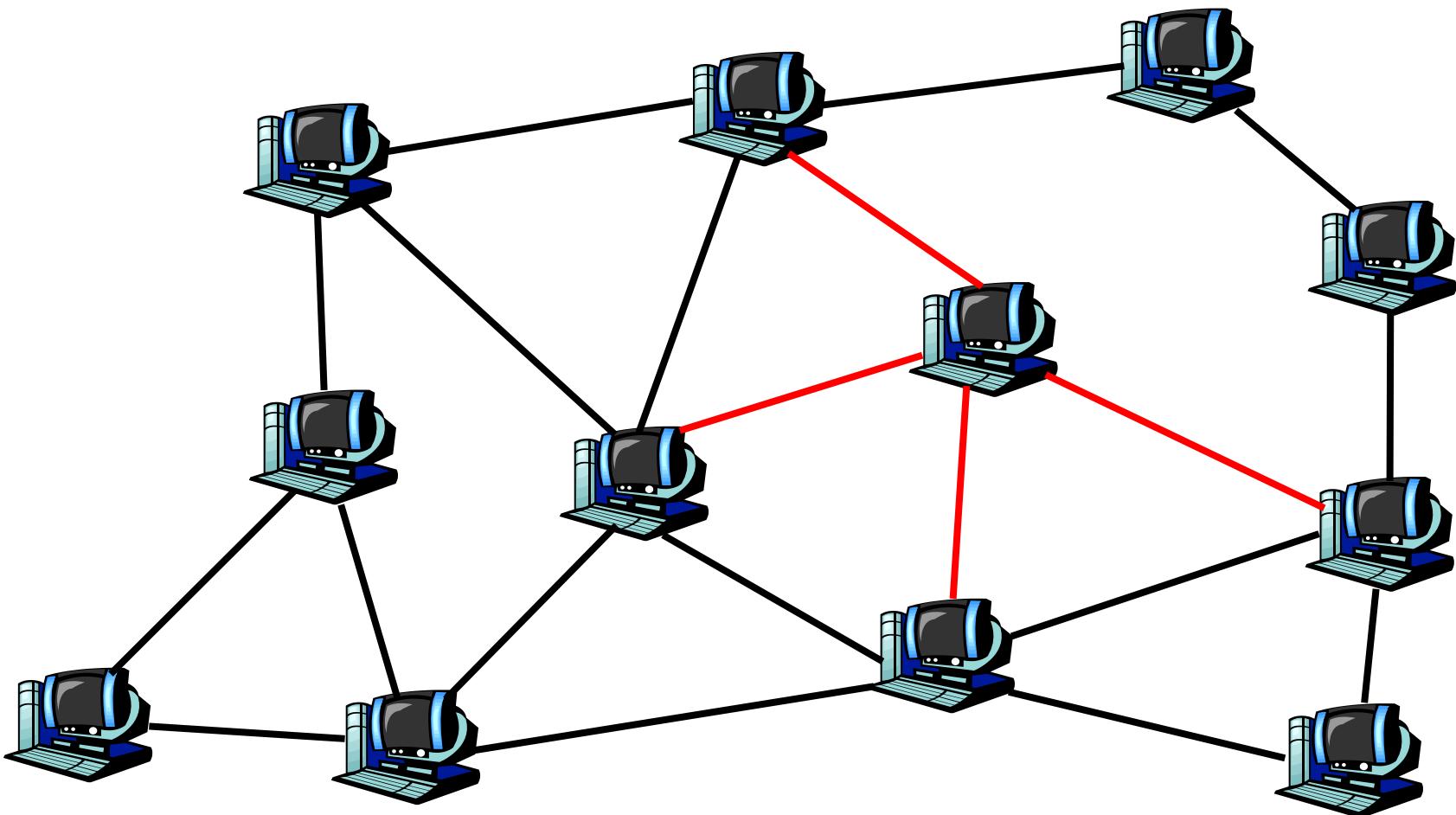
Gnutella: Connection

2. Send join messages to some existing Gnutella nodes



Gnutella: Connection

3. Neighboring relations established



Gnutella: Issues

- How much traffic does one query generate?
 - Flooding is not a scalable design
 - A single query can generate huge amount of traffic
- What is the success rate of finding wanted files?
 - Search can fail even if the wanted file exists in the P2P system
- Downloads may not complete?
- To fix these problems: hierarchy, queue management, parallel download, ...

KaZaA

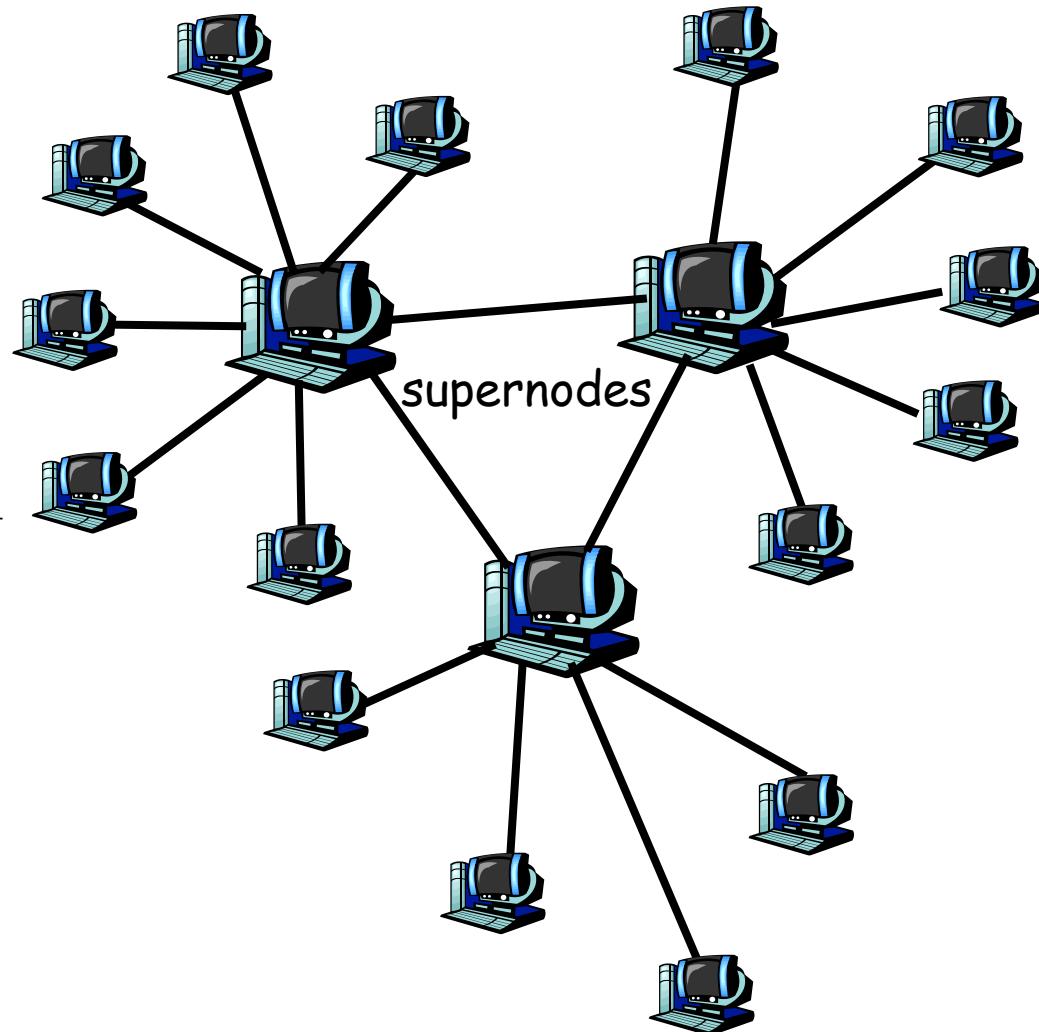
- More than 3 million peers sharing over 3,000 terabytes of contents
- More popular than Napster ever was
- KaZaA provides powerful file search and transfer service **without** server infrastructure
- Queue management at server and client
 - Frequent uploaders can get priority in server queue

KaZaA

- Software
 - Proprietary
 - Control data encrypted
 - Everything in HTTP request and response messages
- Architecture
 - Hierarchical
 - Cross between Napster and Gnutella

KaZaA: Architecture

- Each peer is either a supernode or is assigned to a supernode
 - Each supernode has about 100–150 children
 - Roughly 30,000 supernodes
- Each supernode has TCP connections with 30-50 supernodes
 - 0.1% connectivity



KaZaA: Architecture

- Nodes that have more connection bandwidth and are more available are designated as supernodes
 - Exploit heterogeneity
- Each supernode acts as a mini-Napster hub, tracking the contents and IP addresses of its children
- Does a KaZaA supernode track only the contents of its children, or does it also track the contents under its neighboring supernodes?
 - Testing indicates only children

KaZaA: How does it work? – Metadata

- When an ordinary node connects to a supernode, it uploads its metadata
- For each file
 - File name, file size
 - ContentHash (like a digest): used to identify copies of the same file in the system
 - File descriptors: used for keyword matches in query processing

KaZaA: How does it work? – Connection

- List of potential supernodes included within software download
- New peer goes through the list until it finds an operational supernode
 - Connects, obtains more up-to-date list of supernodes
 - Supernodes on the list are “close” to the new peer
 - New peer then pings 5 supernodes on the list and connects to one
- If supernode goes down, node obtains updated list and chooses new supernode

KaZaA: How does it work? – Query

- Node first sends keyword query to supernode
 - Users can configure “up to x ” responses to search
 - Supernode responds with matches
 - If x matches found, done
- Otherwise, supernode forwards query to subset of supernodes
 - If total of x matches found, done
- Otherwise, query is further forwarded
 - Probably by original supernode rather than recursively
- Query results contain the ContentHash of the wanted file and a list of nodes holding the file

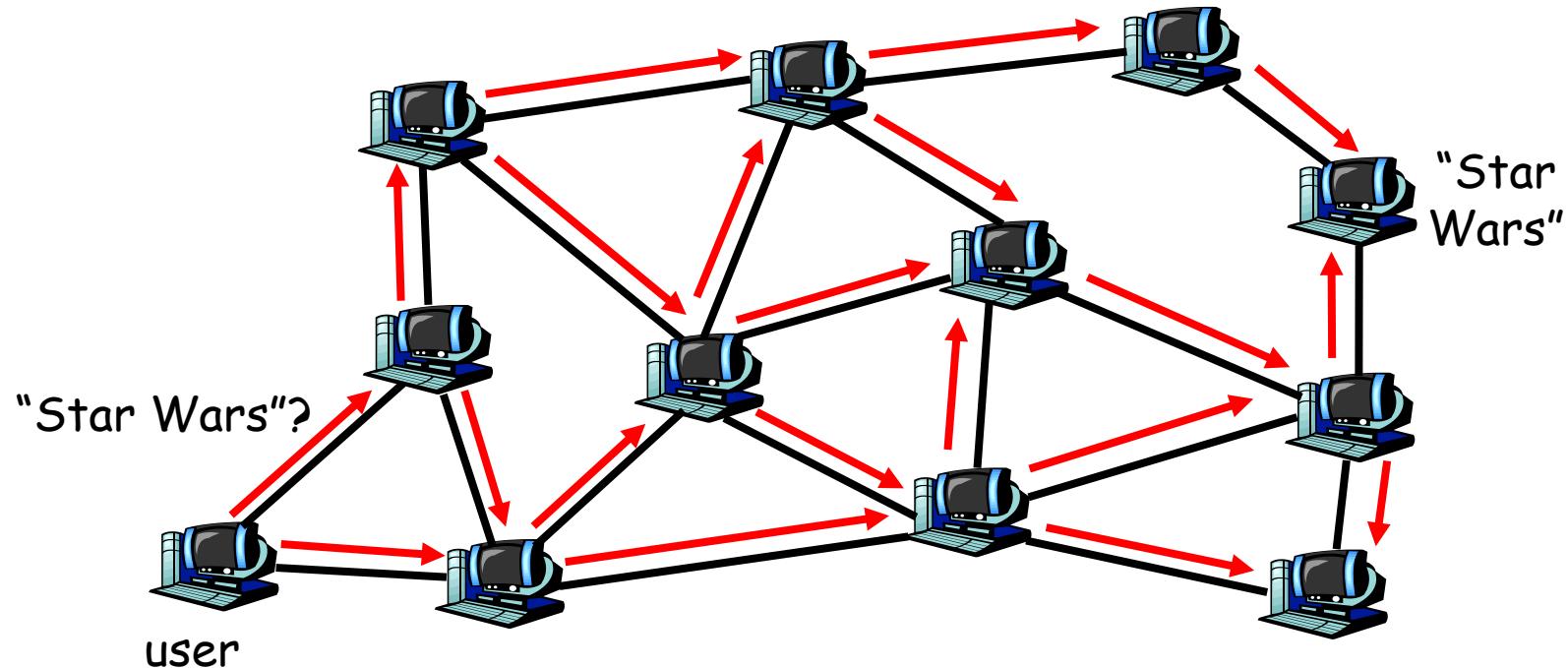
KaZaA: How does it work? – Download

- If file is found in multiple nodes, user can select parallel downloading (optional)
 - HTTP byte-range header used to request different portions of the file from different nodes
 - Identical copies identified by ContentHash
- Automatic recovery
 - Client peer automatically switches to new download server peer when the current server peer becomes unavailable (i.e., it stops sending file so that the download partially completes)
 - ContentHash is used to search for new copy of file

Wrap-up: Unstructured P2P

- Unstructured P2P searches are
 - Simple to build
 - Simple to understand algorithmically
- Little concrete is known about their performance
 - New node randomly chooses existing nodes as neighbors when it joins the system
 - No coupling between nodes and file locations
 - → random search
 - What is the expected overhead of a search?

Search by Flooding (Revisit)



- If K out of N nodes have copy of the wanted file, expected search cost is at least $N/(2K)$
- Need many copies (e.g., through caching and replication) to keep search overhead small

Outline

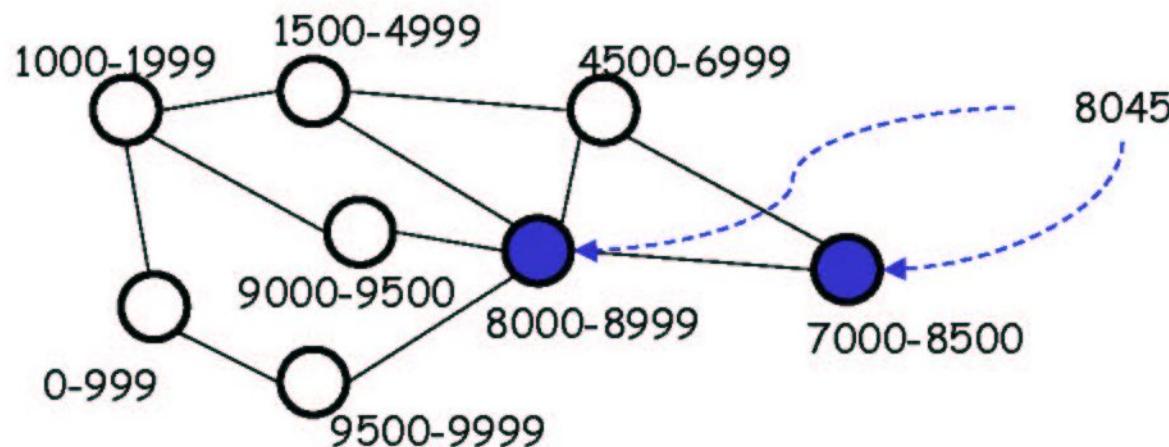
- Introduction
- Unstructured P2P File Sharing
- Structured DHT Systems
 - DHT Services (Distributed Hash Table)
 - Consistent Hashing
 - Chord
- Summary

Directed Searches

- Idea: **tight coupling between nodes and file locations**
 - Assign particular nodes to hold particular contents (or pointers to it, like an information booth)
 - → want to distribute responsibilities among existing nodes in the P2P system
 - When a node requests that content, go to the node that is supposed to have or know about it
 - Arrange neighbor relationship between nodes in a restrictive structure to facilitate query routing

DHT Step 1: The Hash

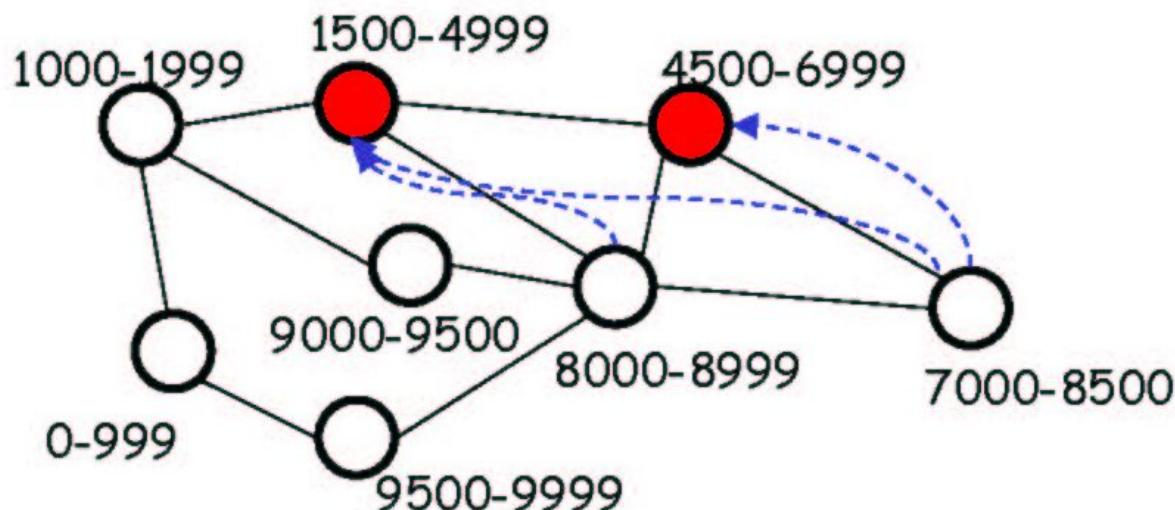
- Introduce a hash function to map files (i.e., the keys of files) to unique identifiers
 - e.g., $h(\text{"Star Wars"}) \rightarrow 8045$
- Distribute the range space of the hash function among all nodes in the network



- Each node is **responsible for** the files that hash within its range, it must “know about” these files

DHT Step 1: The Hash

- Two alternatives of “knowing about files”
 - Node can physically store the files that hash within its range – **direct**
 - Pointer-based: node stores the locations of the files that hash within its range (i.e., IP address of the node holding each file) – **indirect** (in our lecture, let’s assume this one)



DHT Step 2: Routing

- Basic operation to search for a file: **given the key of the file, find out the location of the file**
 - Query (which may be issued by any node in the system) must be **efficiently** routed to the node whose range covers the file
 - This should be implemented in a **fully distributed** manner (no centralized point that bottlenecks throughput or acts as a single point of failure)
 - To do so, each node must maintain information of the range spaces covered by some other nodes (called **routing information** or **distributed hash tables**)
 - Complexity of routing information must **scale** with system growth

DHT Step 2: Routing

- DHT mechanism should gracefully handle nodes joining/leaving the P2P system
 - Need bootstrap mechanism to connect new nodes into the existing DHT infrastructure
 - Need to repartition range space over existing nodes
 - Distribute knowledge and responsibility to joining nodes
 - Redistribute knowledge and responsibility from leaving nodes
 - Need to reorganize/update the routing information maintained at each node

Consistent Hashing

- Choose the range space of hash function as **a circle of identifiers** from 0 to $2^m - 1$ (identifier circle)
- Hash each file key to an identifier (**key identifier**)
 - E.g., use SHA or MD5
- Assign each node an identifier (**node identifier**)
 - Can also be done by hashing, e.g., on IP address or MAC address
- m must be large enough to make the probability of two nodes or two keys hashing to the same identifier negligible

Consistent Hashing

- Each node is assigned the range space from the identifier of its counter-clockwise neighbor node on the identifier circle (called **predecessor node**) to its own identifier
- Each node is responsible for the files that hash within its range
 - In other words, each file is assigned to the first node whose identifier is equal to or follows the key identifier of the file clockwise on the identifier circle
 - The location of the file (i.e., IP address of the node holding the file) is stored at the node

Example of Consistent Hashing

Node N1 is responsible for file K60

K60

N1

file K60 is at node X

N56

N8

Node N14 is responsible for file K10

K10

N14

file K10 is at node Y

Node 48

N51

N48

identifier circle
($m = 6$)

N42

N21

N38

K30

file K24 is at node Z
file K30 is at node W

N32

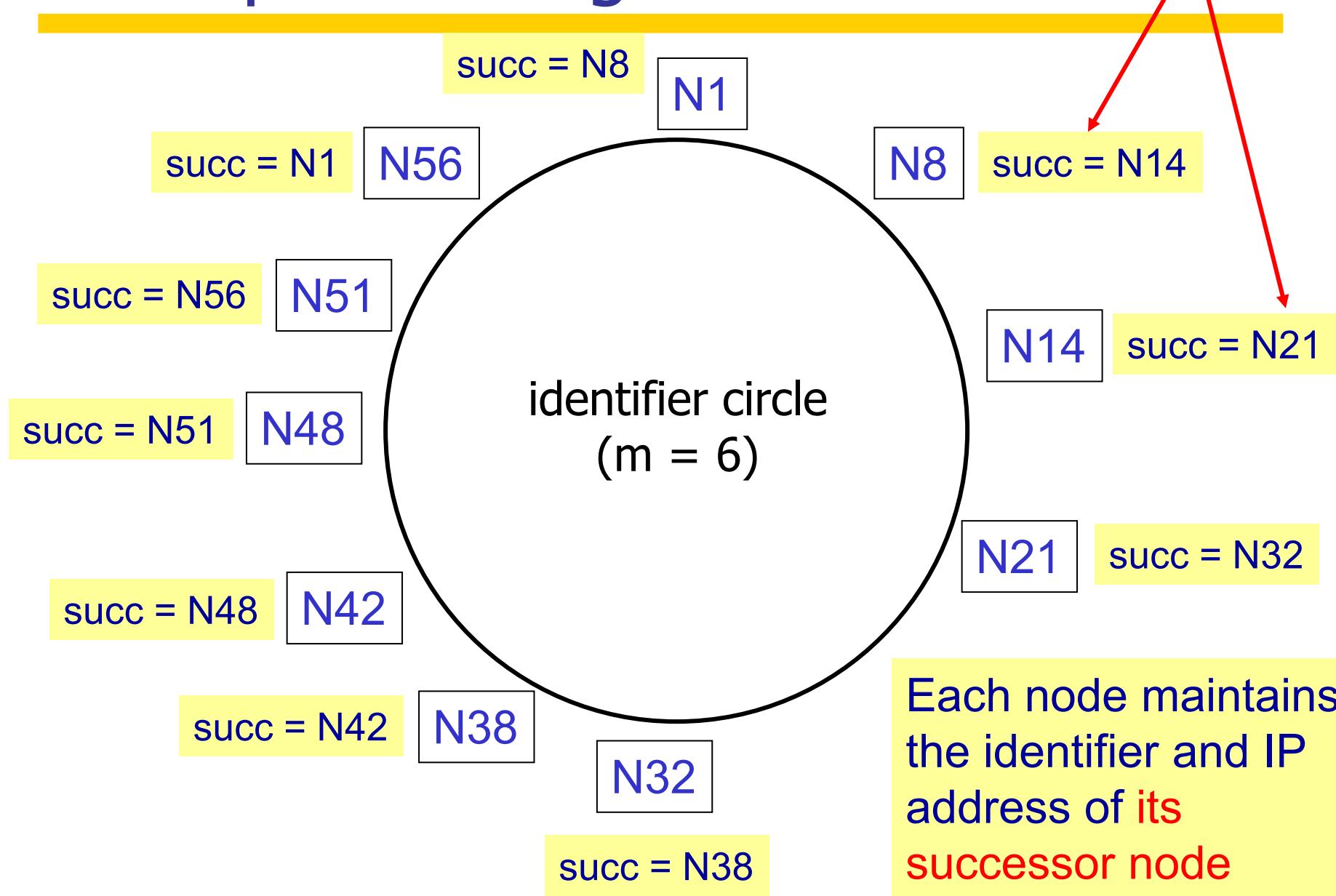
K24

Key 24

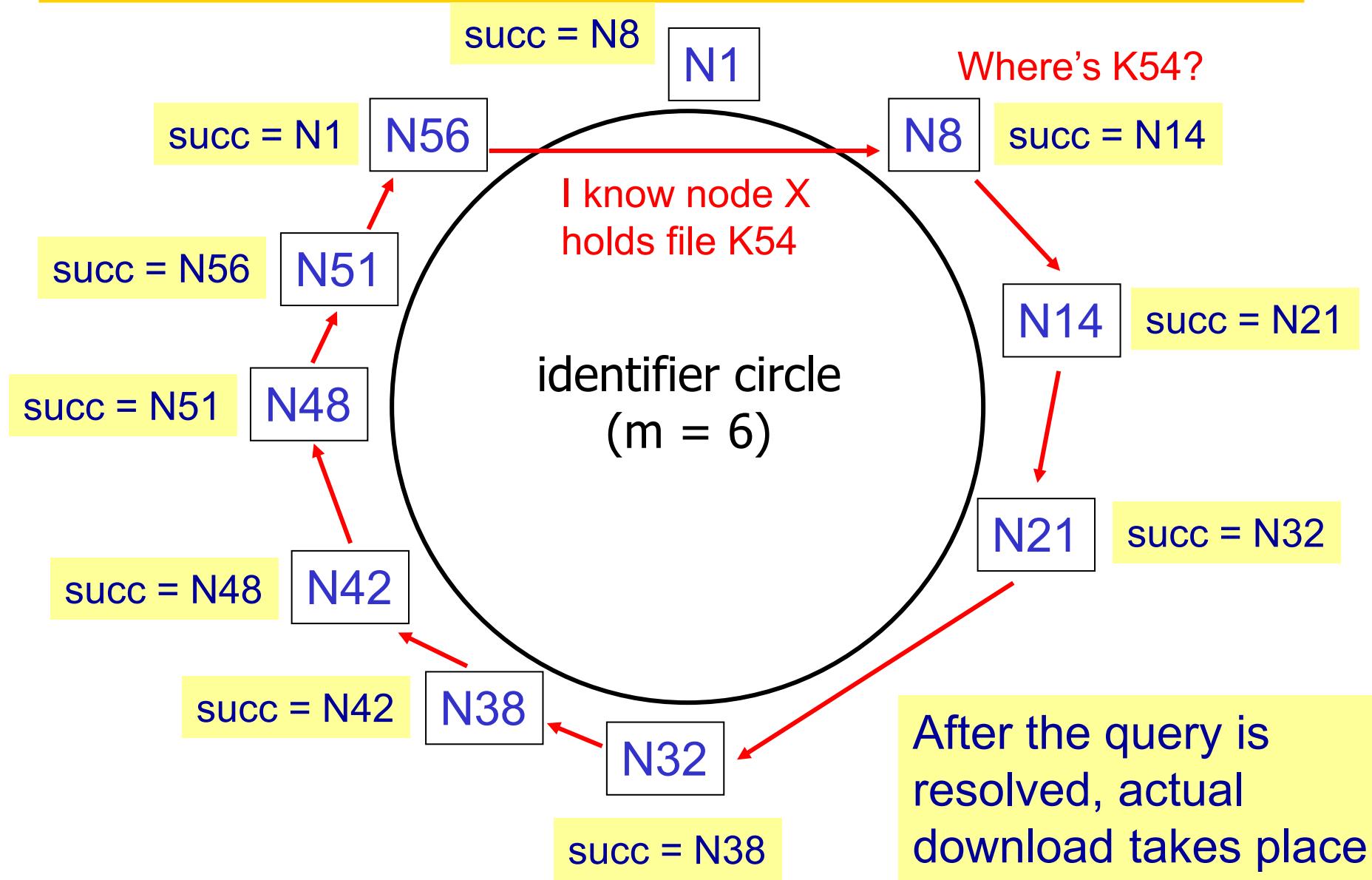
Node N32 is responsible for files K24 and K30

A Simple Routing Scheme

For simplicity, we omit the IP addresses



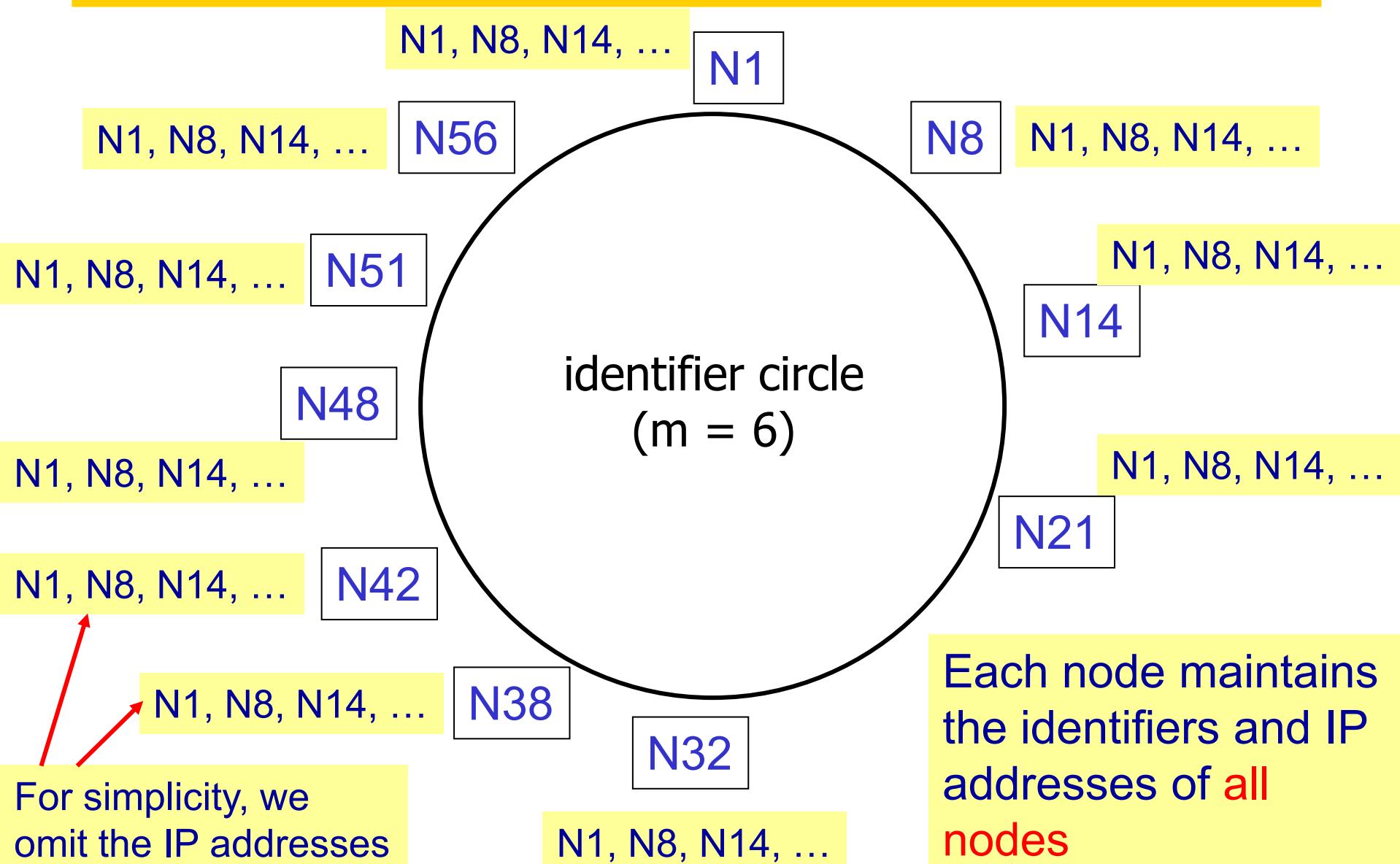
A Simple Routing Scheme



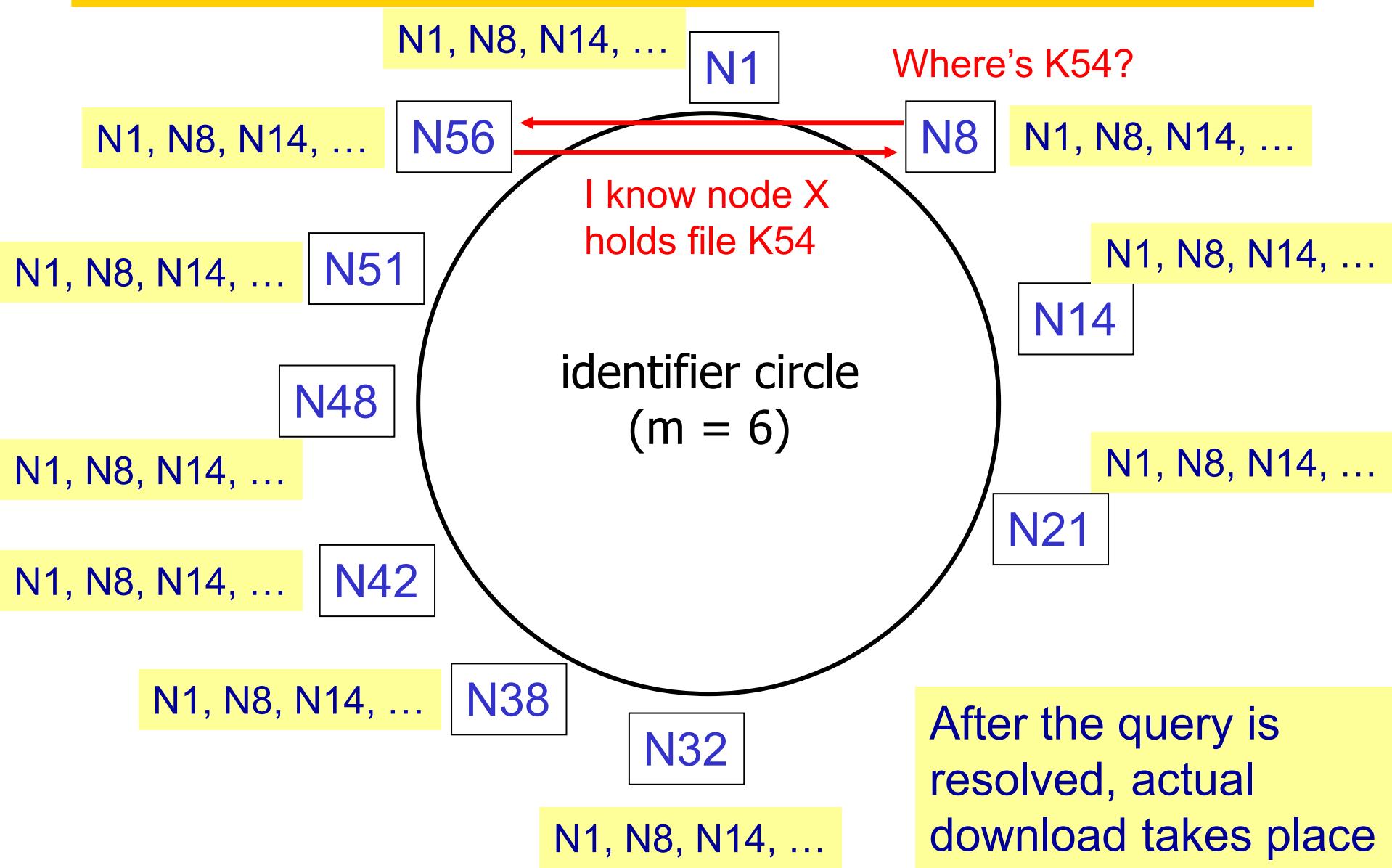
A Simple Routing Scheme

- Each node maintains the identifier and IP address of its **successor node** (clockwise neighbor node on the identifier circle)
- Advantage: **low complexity of routing information** maintained at each node – $O(1)$
 - Cheap to update routing information when nodes join and leave
- Disadvantage: **high complexity of query routing** – $O(N)$ messages to resolve a query
 - N is the number of nodes in the system

Another Simple Routing Scheme



Another Simple Routing Scheme



Another Simple Routing Scheme

- Each node maintains the identifiers and IP addresses of all nodes
- Advantage: **low complexity of query routing** – $O(1)$ messages to resolve a query
- Disadvantage: **high complexity of routing information** maintained at each node – $O(N)$
 - Expensive to update routing information when nodes join and leave

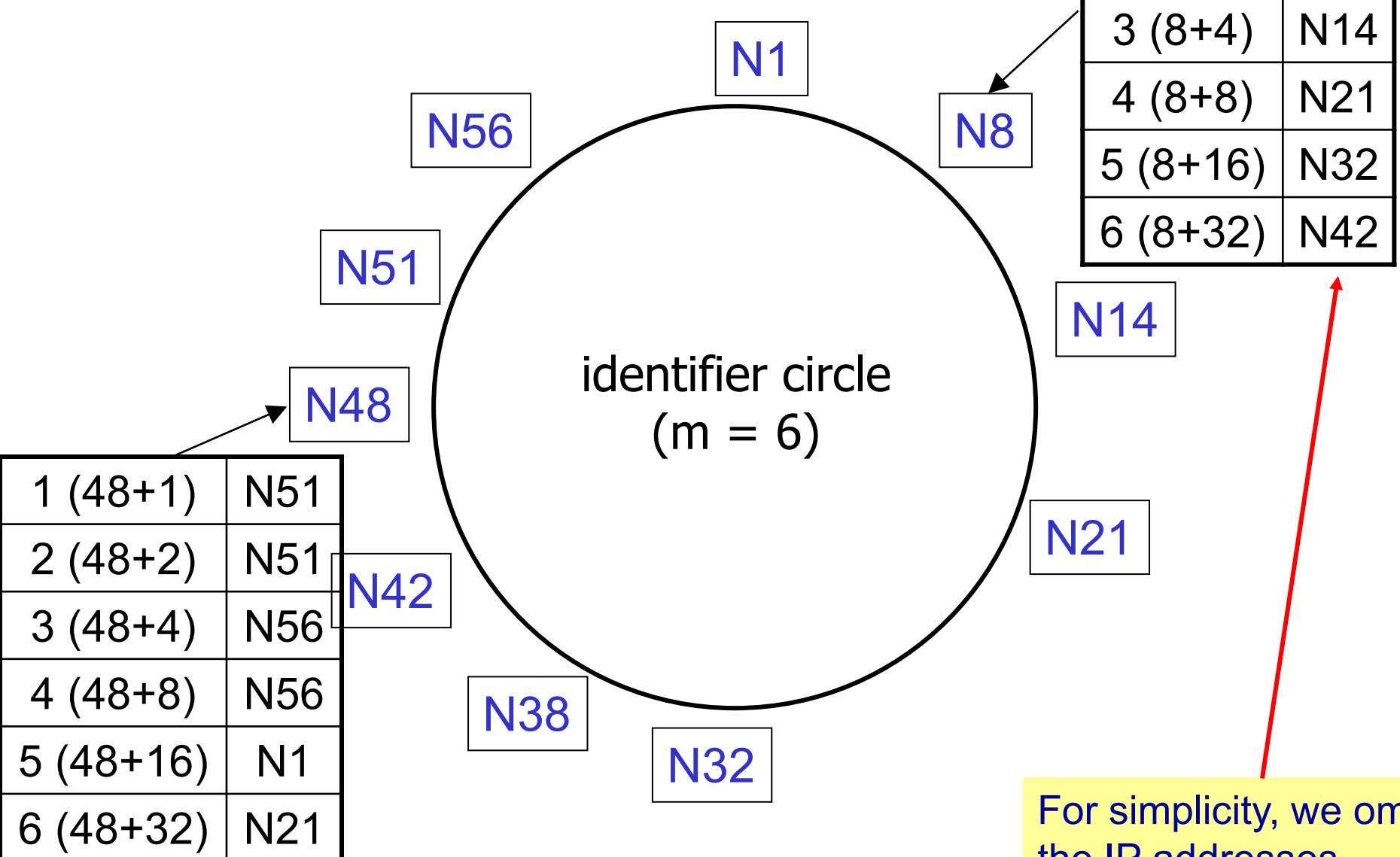
Can we do better?

Yes, Chord routing → $O(\log N)$ routing information per node and $O(\log N)$ messages per query resolution

Chord Routing – Finger Table

- Each node maintains the identifiers and IP addresses of **some nodes** in a **finger table**
- The i -th entry ($1 \leq i \leq m$) in the finger table at node n contains the identifier and IP address of the first node that succeeds n by **at least 2^{i-1}** on the identifier circle
 - Each entry is called a finger
- Note that the first entry contains the identifier and IP address of the successor node

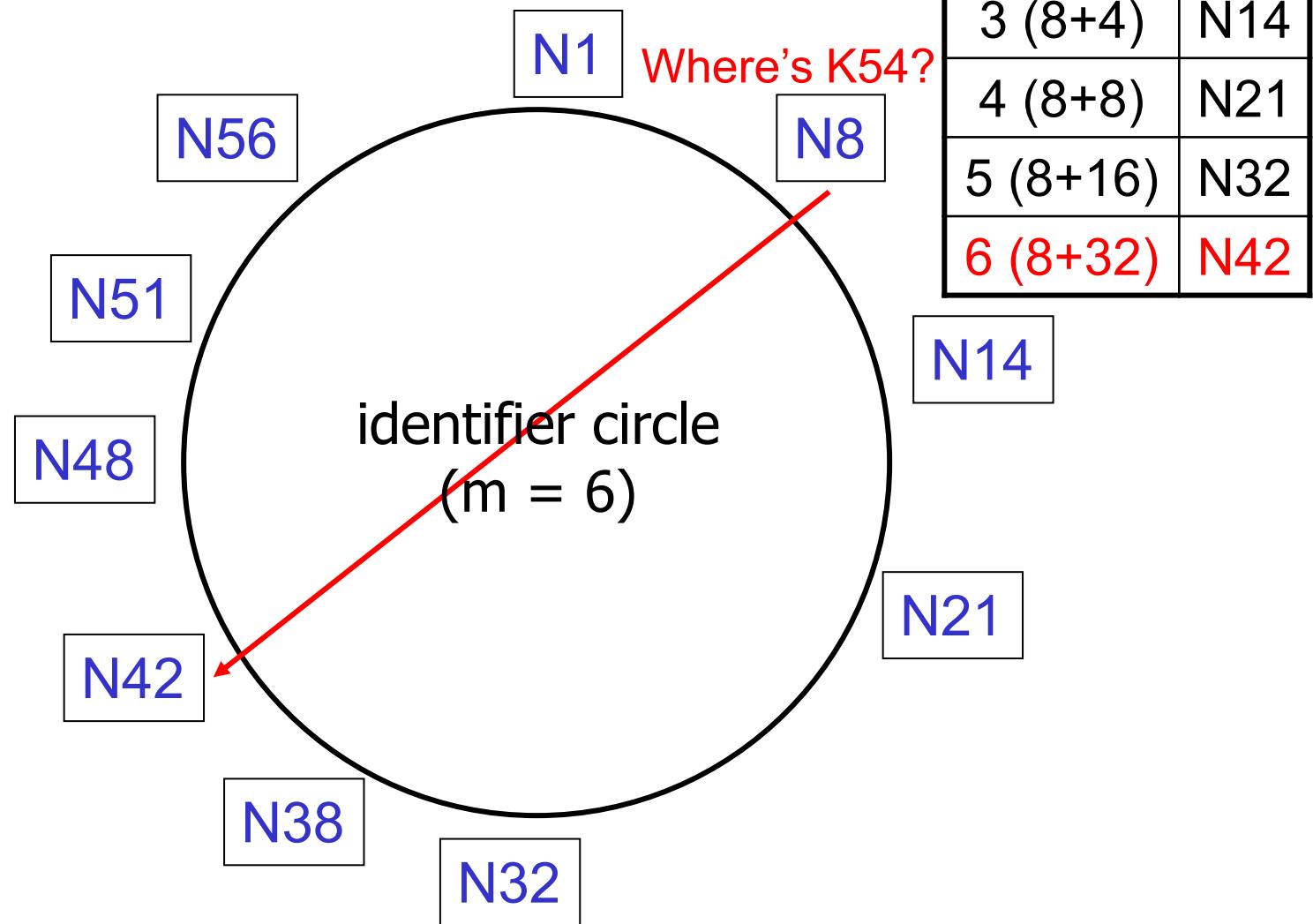
Example of Finger Table



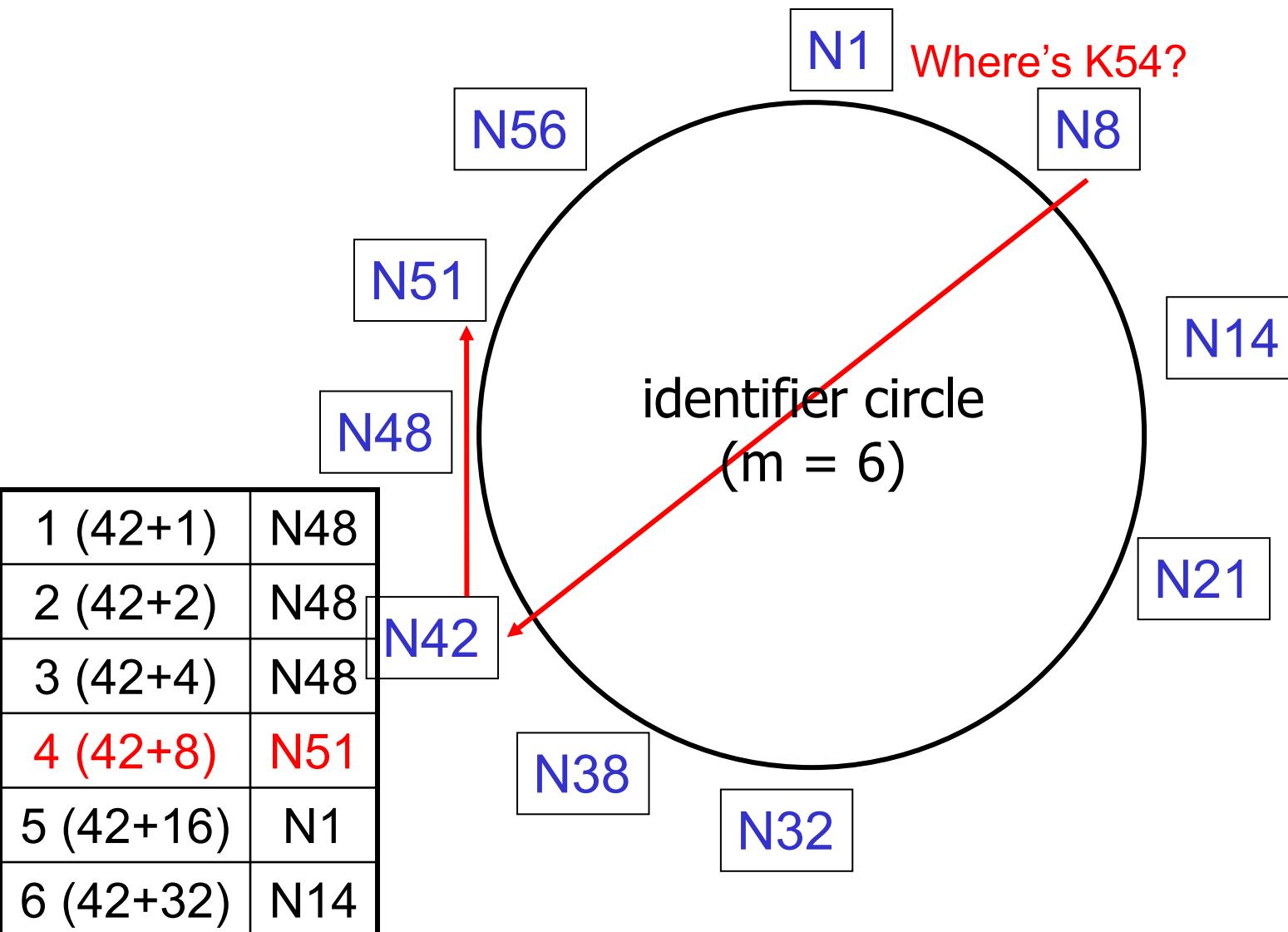
Chord Routing – Routing

- Each node stores information about only a small number of other nodes
- Each node knows more about nodes closely following it on the identifier circle than about nodes further away
- A node's finger table generally does not contain enough information to directly determine the successor of an arbitrary key k
- In routing, each node n sends a query for key k to the node in $(\lfloor \log_2(k - n) \rfloor + 1)$ -th entry of finger table

Example of Chord Routing

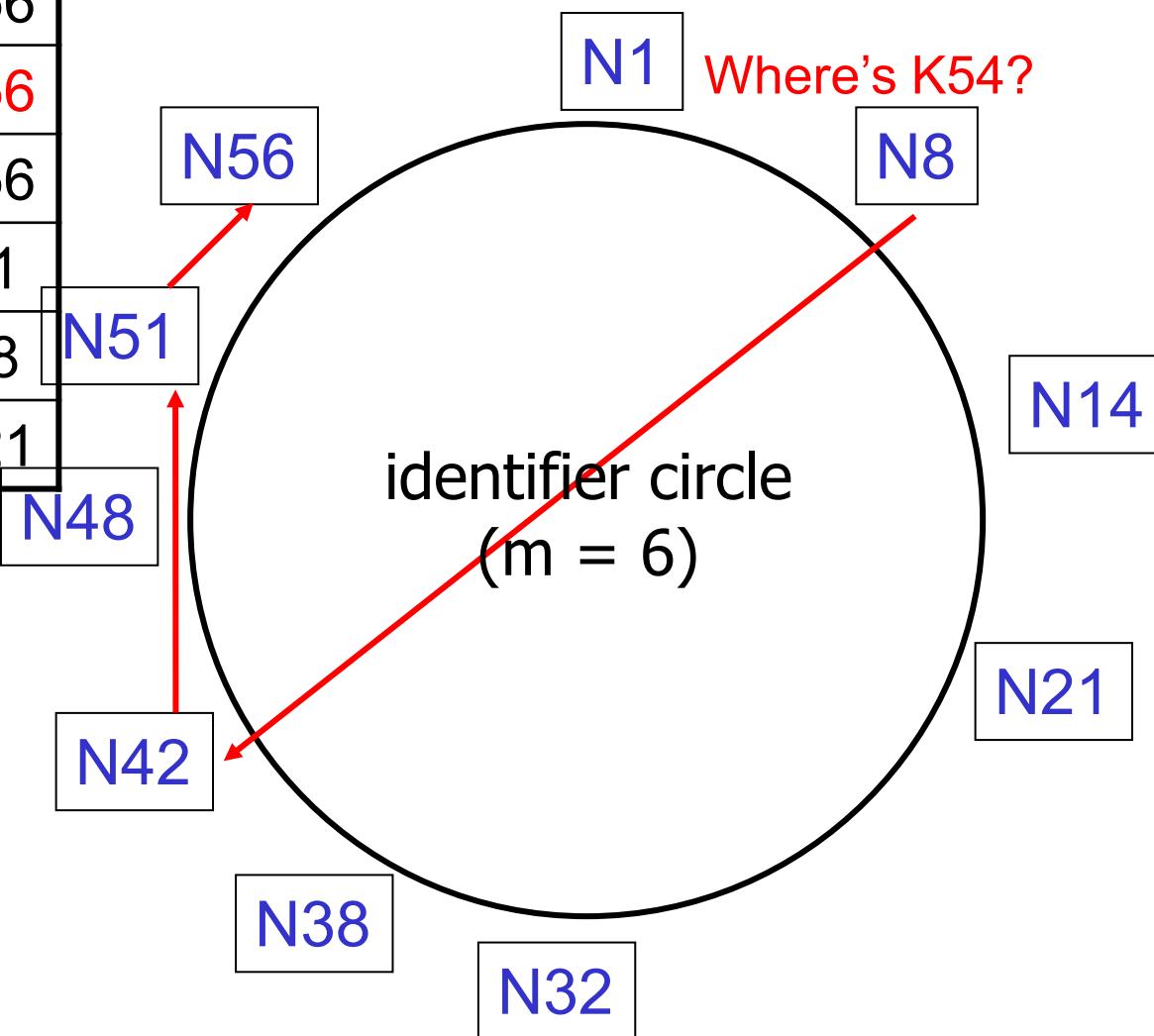


Example of Chord Routing

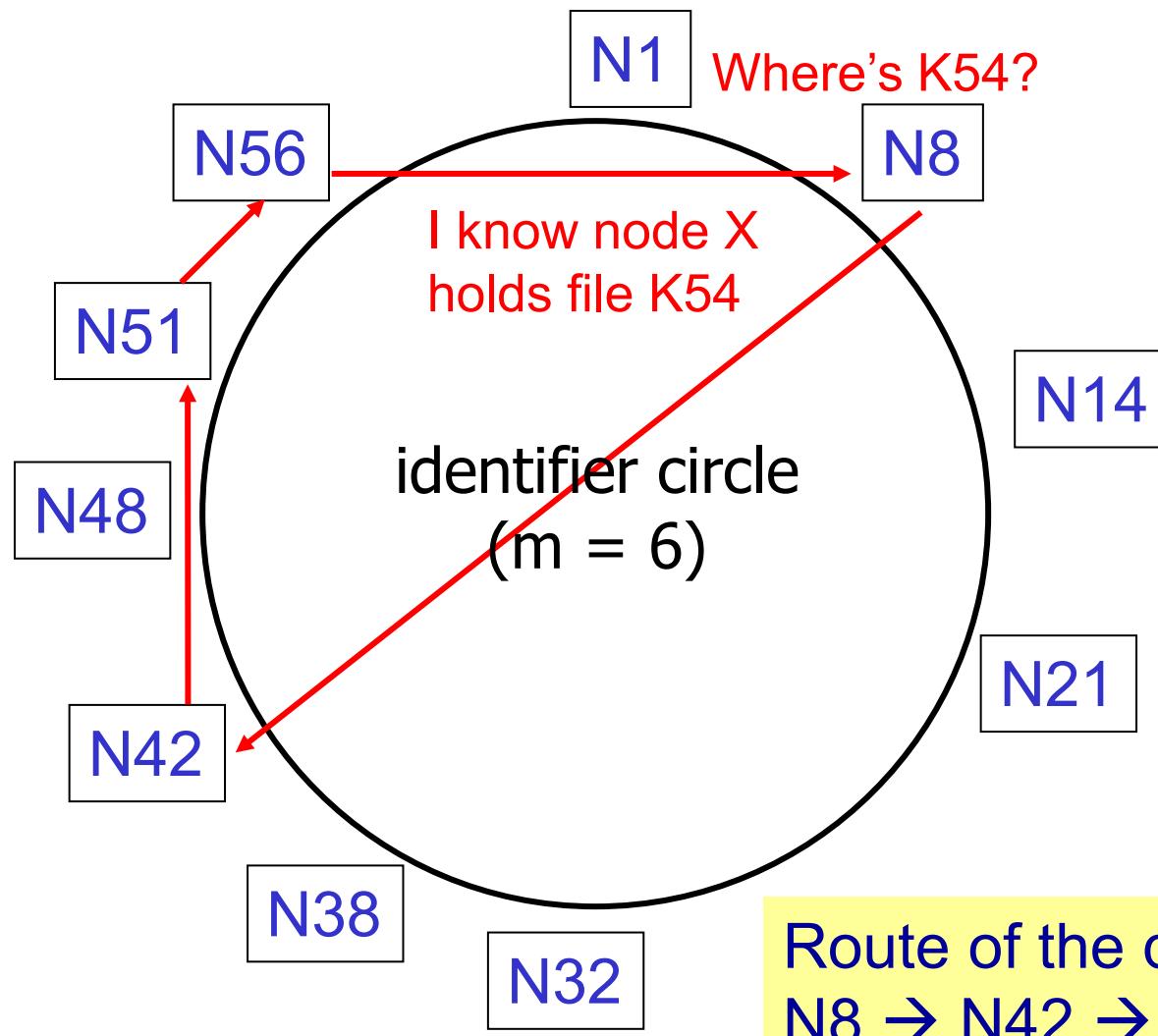


Example of Chord Routing

1 (51+1)	N56
2 (51+2)	N56
3 (51+4)	N56
4 (51+8)	N1
5 (51+16)	N8
6 (51+32)	N21



Example of Chord Routing



Chord Routing – Theoretical Results

- Complexity of routing information maintained at each node – $O(\log N)$
- Complexity of query routing – $O(\log N)$ messages to resolve a query
 - Each node forwards a query **at least halfway** along the remaining distance to the node that is responsible for the target file
- The fingers other than the first one (which records the successor node) are not essential for correctness of routing
- A query can always be routed correctly as long as each node knows its correct successor node

Summary

- What are peer-to-peer systems?
- Unstructured P2P file sharing (no coupling between nodes and file location information)
 - Centralized directory service
 - Search by flooding
 - Hierarchical architecture
- Structured P2P file sharing (tight coupling between nodes and file location information)
 - Distributed hash table (e.g., Chord)
 - There are many more types of distributed hash tables other than Chord

6. Name Services

Outline

- Names and Name Services
- Domain Name System
- Summary

Names and Addresses

- Resources (processes, objects, ...) need to identify each other to enable them to interact
- **Names** identify resources in a **location-independent** fashion
 - People: Alice, Bob, ...
 - Web sites: www.ntu.edu.sg, www.yahoo.com, ...
 - Remote objects: [//Jean.torriano.net/ShapeList](http://Jean.torriano.net/ShapeList), ...
 - Files: `/etc/passwd`, `/usr/joe/test.c`, ...

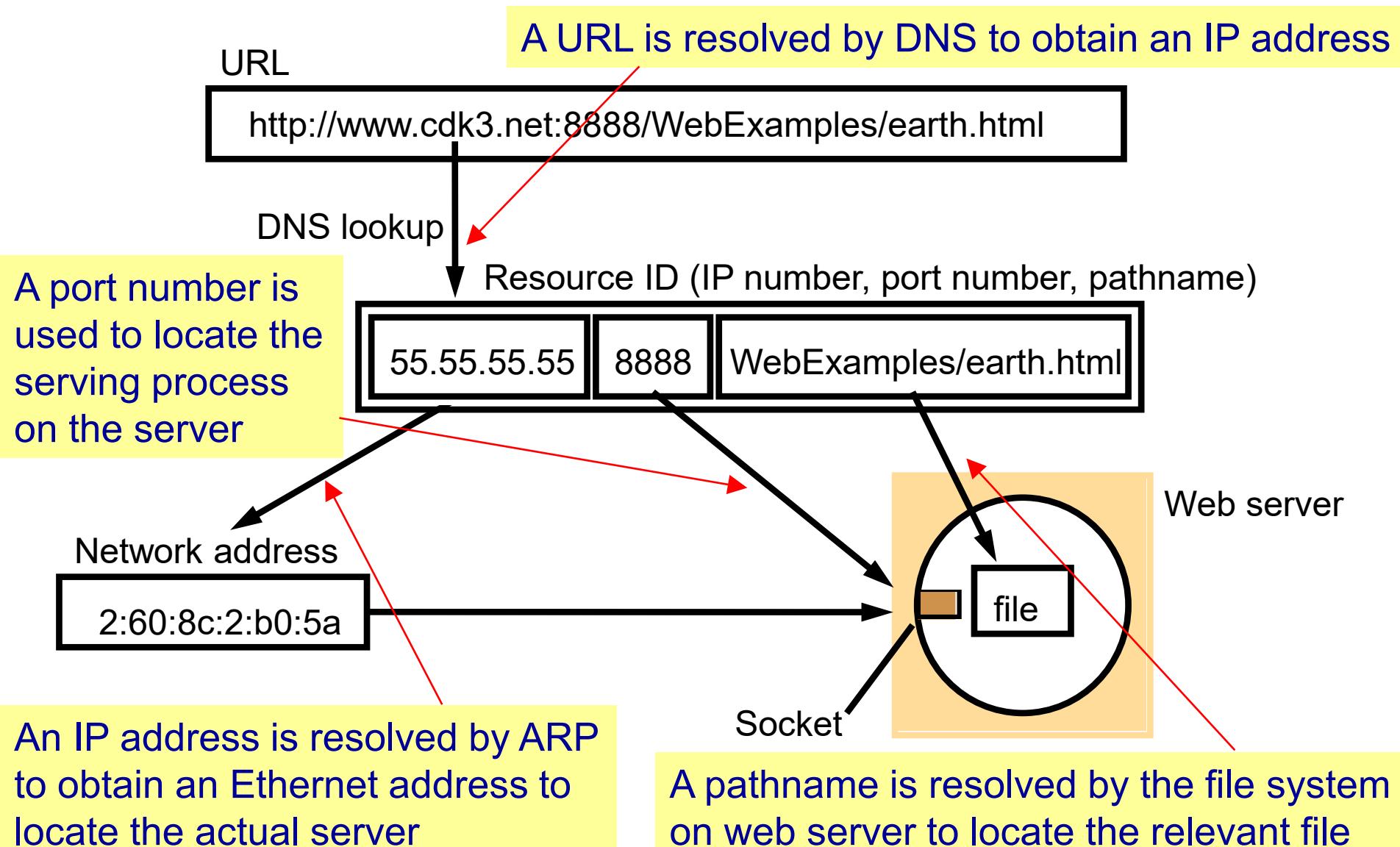
Names and Addresses

- **Addresses** identify resources in a **location-dependent** fashion
 - People: home address, office address, ...
 - Web sites: IP address, MAC address
 - Remote objects: remote object references
(e.g., IP address + port number + object number)
 - Files: file handles (e.g., filesystem identifier + i-node number of file)

Name Services

- **Name service:** translation between names and addresses
 - Domain Name System (DNS) translates between domain names and IP addresses
 - Java RMICRegistry translates between the names of remote objects and their remote object references
 - **Name resolution:** conversion process from names to addresses
 - Name resolution may involve more than one translation (see the example on next slide)

Example of Name Resolution



Name Services

- More generally
 - Address is just one of many attributes of resources
 - Examples of other attributes
 - People: age, job, ...
 - Files: last modification time, size, ...
 - Name service: translation between names and attributes

Main Requirements of Name Services

- Scalability: to handle an arbitrary number of names and to serve an arbitrary number of administrative organizations over long lifetime
 - All web sites in the Internet
 - E-mail addresses of all computer users around the world
- Reliability: name services should not fail frequently
 - high availability
 - Most other systems depend on name services, they cannot work when the name service is broken
- Performance: name resolution should not introduce much delay – short response time

Name Space

- **Name space:** the collection of all valid names
 - Name space requires a syntactic definition
 - E.g., “two” cannot be an IP address, “\$.&.^.#” cannot be the domain name of a host
- Name space can have a hierarchical structure
 - Each part of a name is resolved relative to a separate context
 - The same name may be used with different meanings in different contexts
 - Example of hierarchical name space: UNIX file system (e.g., /usr/joe/test.c, /usr/john/test.c)

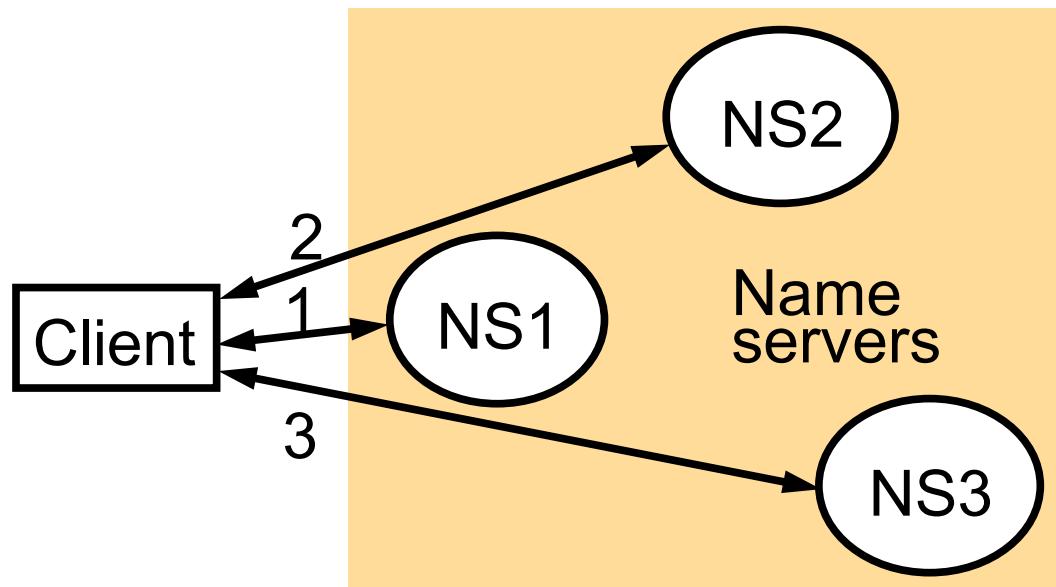
Name Space

- Name space can also have a flat structure
 - A flat set of numeric or symbolic identifiers
 - Hierarchical name space has better scalability than flat name space
- What is the name space used in DNS?
 - Hierarchical structure
 - A domain name consists of one or more strings (called name components or labels) separated by a dot “.”
 - E.g., piano.cais.ntu.edu.sg (this is a computer), cais.ntu.edu.sg, ntu.edu.sg (they are domains)

Name Resolution Process

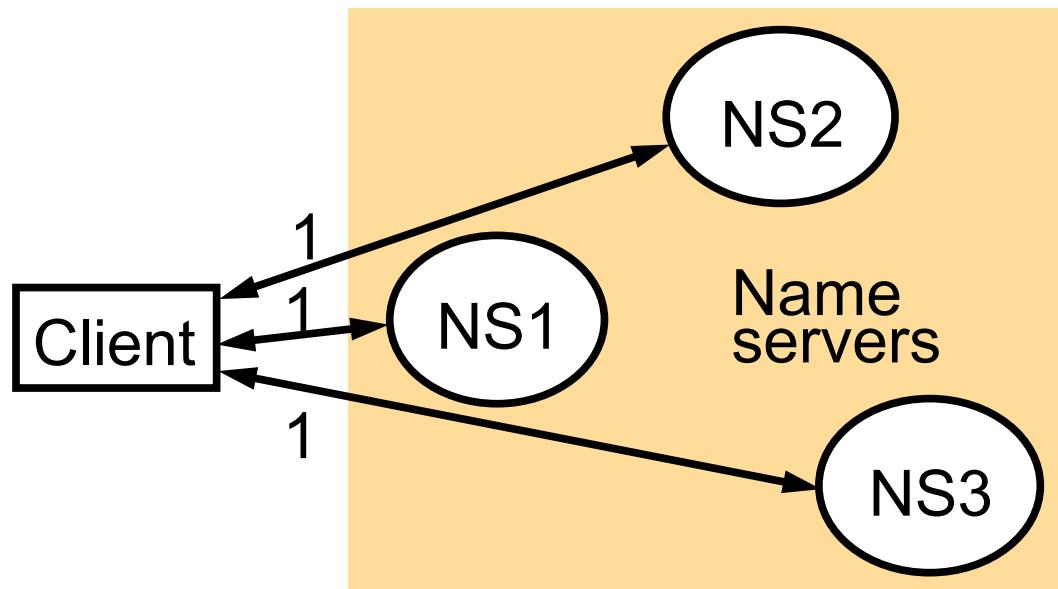
- Mappings between names and addresses of resources are maintained by name servers
 - Binding: associating a name with a resource
 - Unbound names do not correspond to any resource
- Physically, the resolution process may involve more than one name server (**navigation**)
 - Reason: the mapping may not be available at every name server; hierarchical name space
 - Iterative client-controlled navigation
 - Non-recursive server-controlled navigation
 - Recursive server-controlled navigation

Iterative Client-Controlled Navigation



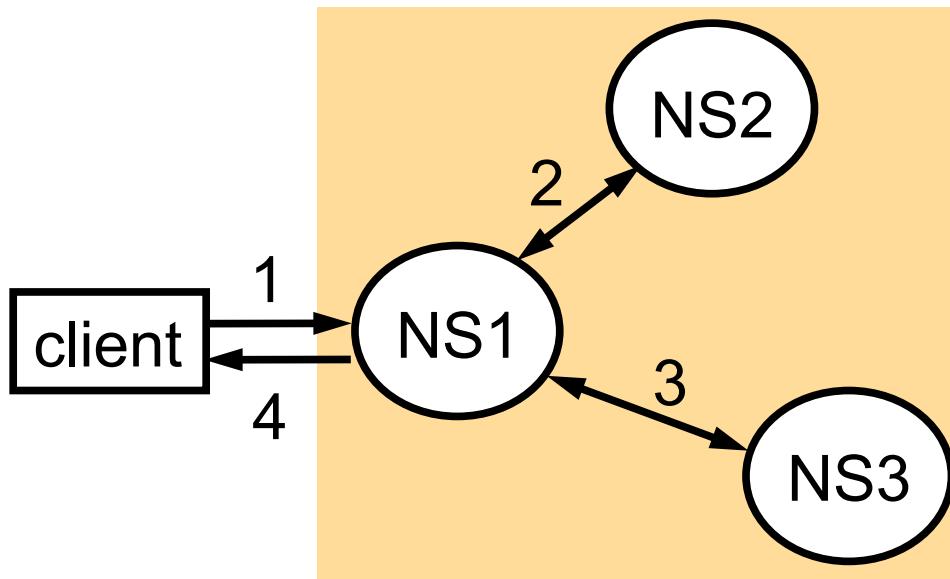
- A client iteratively contacts a group of name servers to resolve a name
 - Client presents the name to NS1 first, if NS1 has the name mapping, it returns results immediately, otherwise, NS1 suggests the client to contact NS2,
 - Disadvantage: potentially high communication cost if name servers are located far away

Iterative Client-Controlled Navigation



- A variation: multicast navigation
 - Client multicasts the name to a group of name servers
 - Only the server holding the name mapping responds
 - Advantage: possibly faster response
 - Disadvantage: possibly more network traffic

Non-Recursive Server-Controlled Navigation

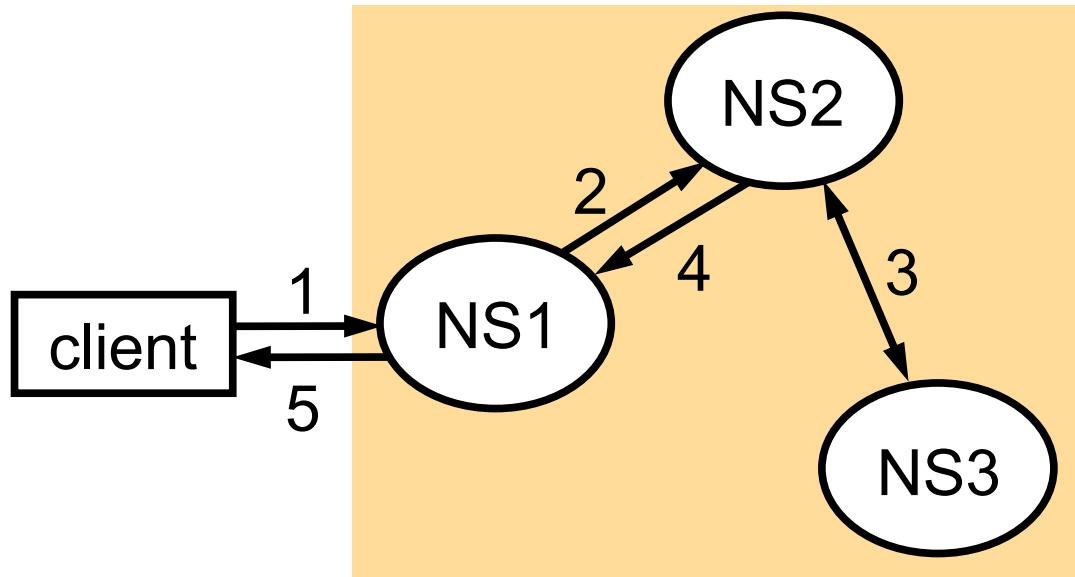


- A name server coordinates the name resolution process on behalf of a client and finally passes the result back to the client
- This name server communicates by multicast or iteratively with other name servers like a client

Navigation

- Potential problems with client-controlled and non-recursive server-controlled navigations
 - In the previous examples, the client or NS1 would be able to find out whether the mapping is maintained by NS2 or NS3
 - When a name service spans distinct administrative domains, the clients and name servers in one domain may need to be prohibited from discovering the distribution of naming data across name servers in another domain

Recursive Server-Controlled Navigation



- If a name server does not have the name mapping, it contacts another name server, this procedure continues recursively until the name is resolved
 - Advantage: caching results is more effective
 - Disadvantage: put higher performance demand on each name server, may tie up server threads and delay other requests

Outline

- Names and Name Services
- Domain Name System
 - Now let's look at a real example
- Summary

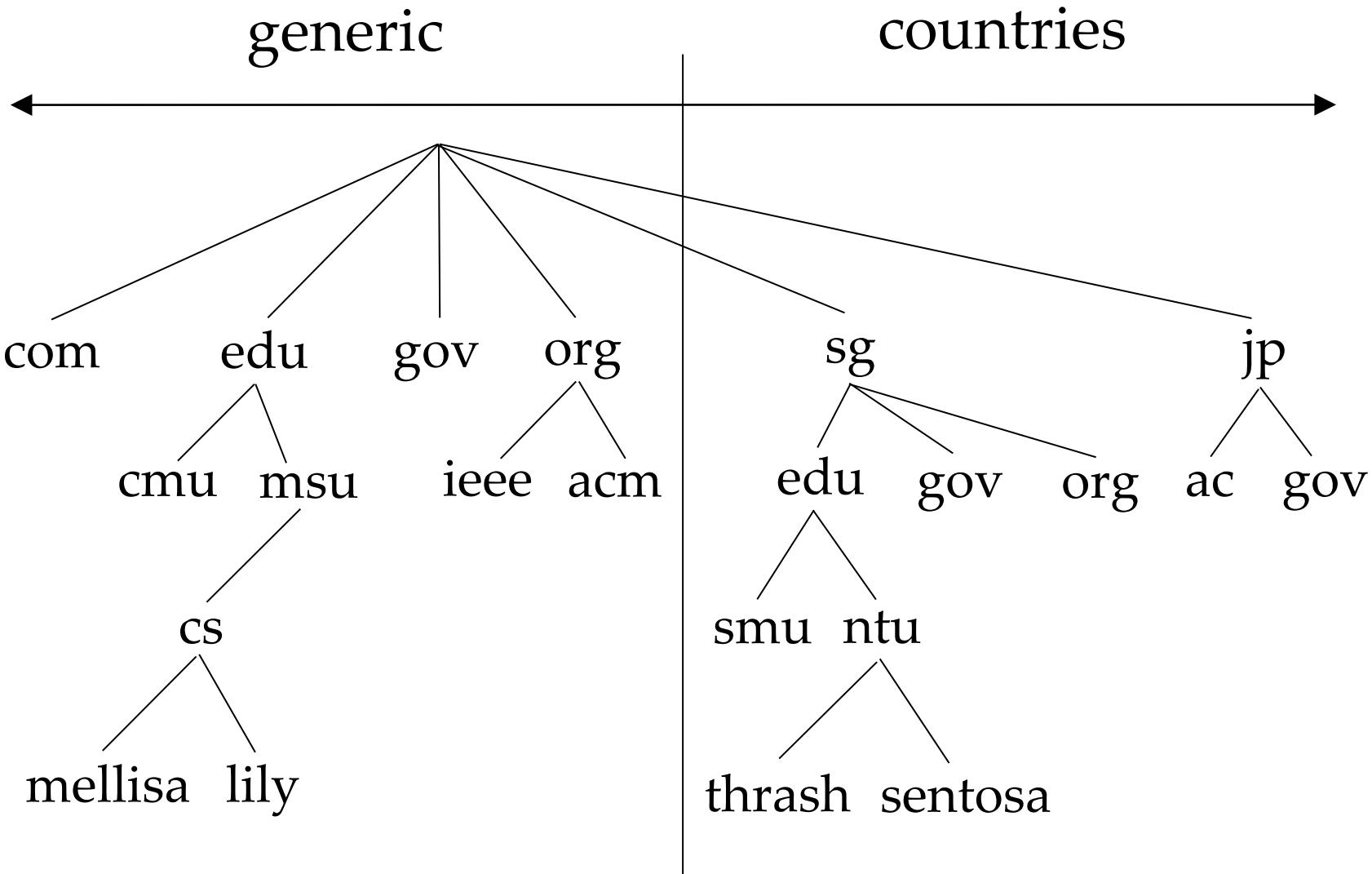
Domain Name System – Overview

- How does DNS meet the requirements of naming services?
 - Scalability: hierarchical name space; name space is partitioned and managed by a large group of name servers
 - Reliability: use replication
 - Performance: use caching

Hierarchical Name Space

- DNS manages the names of computers and domains
- Domains are associated with organizations
 - Domains form a tree-like hierarchical structure which reflects the structure of organizations
- Each domain manages their own names
 - NTU gets its name from whoever is in charge of edu.sg
 - Afterwards, it is free to assign hostnames within NTU – without going back to edu.sg administration

Hierarchical Name Space



Name Space Partitioning

- In the original Internet naming system, all hostnames and addresses were held in a single central master file which was downloaded by FTP to all computers that required them
 - It did not scale to large numbers of computers
 - Thus, it was replaced by the Domain Name System
- DNS uses a decentralized design
 - No single server is responsible for all names
 - The entire name space is partitioned among a large group of name servers
 - Each server holds authoritative mappings for the names in one or more domains

DNS Name Servers

Root name servers hold mappings for the name servers of top-level domains

ns1.nic.uk
(uk)

co.uk
ac.uk

a.root-servers.net
(root)

uk
purdue.edu
yahoo.com

Root name servers are also authoritative name servers for generic top-level domains, but not for country domains

ns.purdue.edu
(purdue.edu)

*.purdue.edu

ns0.ja.net
(ac.uk)

ic.ac.uk
qmw.ac.uk

dcs.qmw.ac.uk
*.qmw.ac.uk

*.dcs.qmw.ac.uk

*.ic.ac.uk

alpha.qmw.ac.uk
(qmw.ac.uk)

dns0.dcs.qmw.ac.uk
(dcs.qmw.ac.uk)

dns0-doc.ic.ac.uk
(ic.ac.uk)

Name server name

The domain a name server is responsible for

DNS Name Servers

- Information held in DNS name servers
 - Mappings between names and IP addresses for the local domain less any sub-domains administered by other name servers
 - Names & addresses of all name servers in local domain
 - Names & addresses of name servers in its sub-domains
 - Names & addresses of root name servers
- Each mapping contains domain name + time-to-live + class + type + value
 - For domain names in the Internet, the class is IN
 - To query, clients send names to the name server which responds with the values according to the mapping

DNS Name Servers

domain name	time-to-live	class	type	value
	1D (1 day)	IN	NS	dns0
	1D	IN	NS	dns1
	1D	IN	MX	1 mail1.qmw.ac.uk
	1D	IN	MX	2 mail2.qmw.ac.uk
www	1D	IN	CNAME	copper
copper	1D	IN	A	138.37.88.248

- The above table shows the information held in name server of dcs.qmw.ac.uk
 - A-type record maps a computer name to its IP address

DNS Name Servers

domain name	time-to-live	class	type	value
	1D (1 day)	IN	NS	dns0
	1D	IN	NS	dns1
	1D	IN	MX	1 mail1.qmw.ac.uk
	1D	IN	MX	2 mail2.qmw.ac.uk
www	1D	IN	CNAME	copper
copper	1D	IN	A	138.37.88.248

- DNS allows multiple names to be mapped to the same IP address
 - A canonical name (e.g., copper) and its aliases (e.g., www)
 - CNAME-type record maps an alias to its canonical name
 - Advantage of aliases: transparency

DNS Name Servers

domain name	time-to-live	class	type	value
	1D (1 day)	IN	NS	dns0
	1D	IN	NS	dns1
	1D	IN	MX	1 mail1.qmw.ac.uk
	1D	IN	MX	2 mail2.qmw.ac.uk
www	1D	IN	CNAME	copper
copper	1D	IN	A	138.37.88.248

- NS-type record specifies authoritative name servers
 - In the above examples, NS-type records refer to the name servers for the local domain

DNS Name Servers

domain name	time-to-live	class	type	value
dcs	1D	IN	NS	dns0.dcs
dns0.dcs	1D	IN	A	138.37.88.249
dcs	1D	IN	NS	dns1.dcs
dns1.dcs	1D	IN	A	138.37.94.248

- NS-type record specifies authoritative name servers
 - The above table shows the information held in the name server of qmw.ac.uk, where NS-type records refer to the name servers for its sub-domains
 - Each NS-record is accompanied by an A-type record

DNS Queries

- **Host name resolution:** to resolve host names into IP addresses
 - When you type a URL in your web browser
 - When you download files from an FTP server
 - When you login to a remote host using telnet
- **Mail host location:** to resolve domain names into IP addresses of mail hosts (which handle all emails sent to this domain)
 - Email address: john@yahoo.com → domain name: yahoo.com
 - Email applications transparently select this service by sending “mail” type queries to name servers

DNS Name Servers

domain name	time-to-live	class	type	value
	1D (1 day)	IN	NS	dns0
	1D	IN	NS	dns1
	1D	IN	MX	1 mail1.qmw.ac.uk
	1D	IN	MX	2 mail2.qmw.ac.uk
www	1D	IN	CNAME	copper
copper	1D	IN	A	138.37.88.248

- MX-type record gives preferences and domain names of mail hosts
 - Assume one mail service per domain, so no need to include names explicitly
 - If multiple domain names are returned, they are tried in the order of preference value

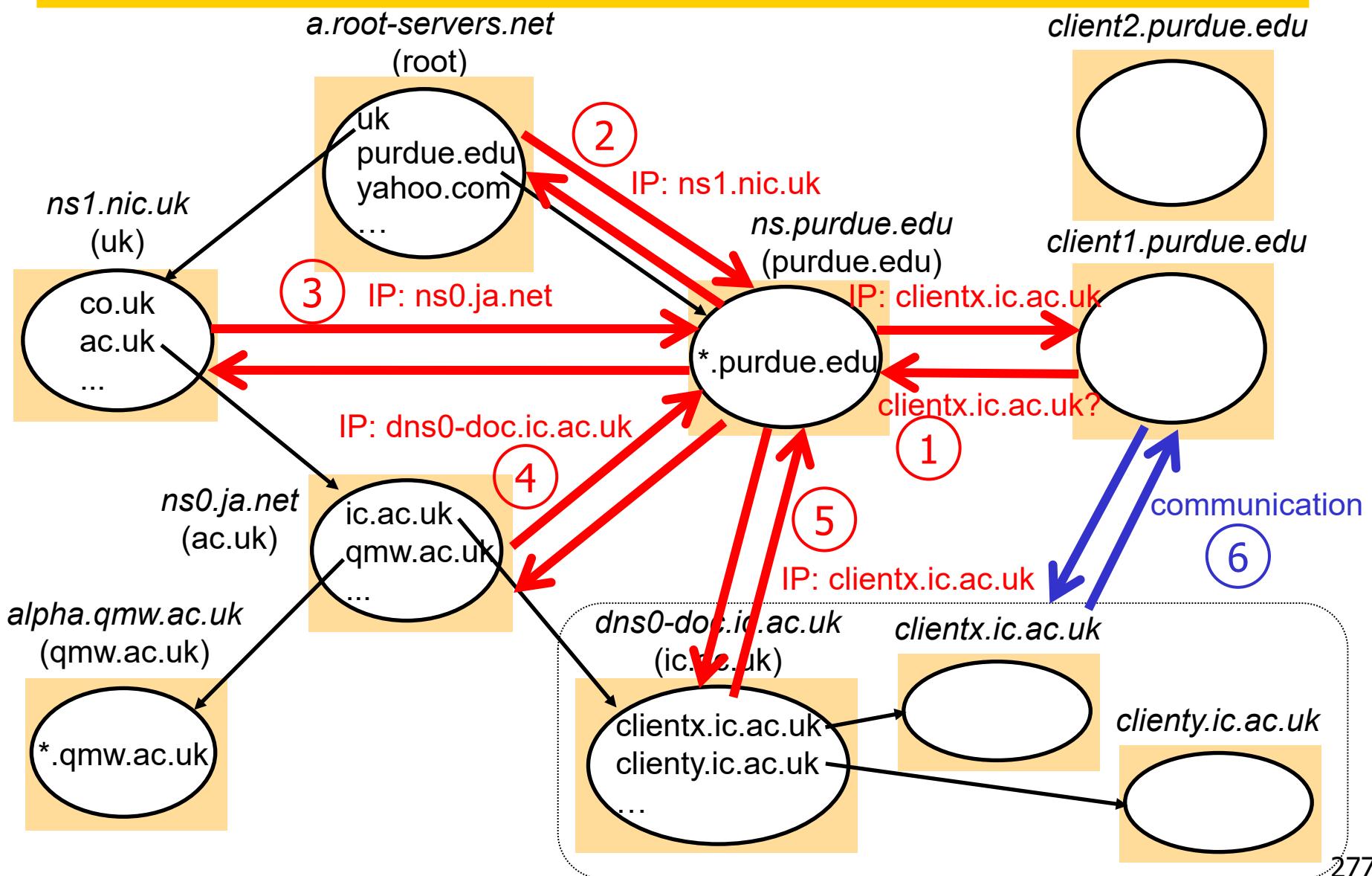
Replication for Reliability

- DNS uses **replication** to improve reliability
- Authoritative mappings of each domain must be held by at least two name servers
 - One is called **the primary server**, which reads mappings directly from a local master file
 - The remaining are called **secondary servers**, which download mappings from a primary server
 - Secondary servers periodically check with the primary server to see whether their stored mappings are up-to-date, this is typically done once or twice a day
 - Root domain has about a dozen secondary servers to share the heavy workload

DNS Name Resolution Process

- DNS uses a simple request-reply protocol between clients and name servers based on UDP
 - Name server implements the BIND (Berkeley Internet Name Domain) software – the most widely used implementation of DNS protocols
 - Name servers use a well-known port number
 - A DNS client is called a **resolver**
- DNS primarily uses a combination of iterative client-controlled navigation and non-recursive server-controlled navigation
 - A client contacts the first server on a list of servers it knows of, and progresses to other servers if the first server is not available
 - Then non-recursive server-controlled navigation is used

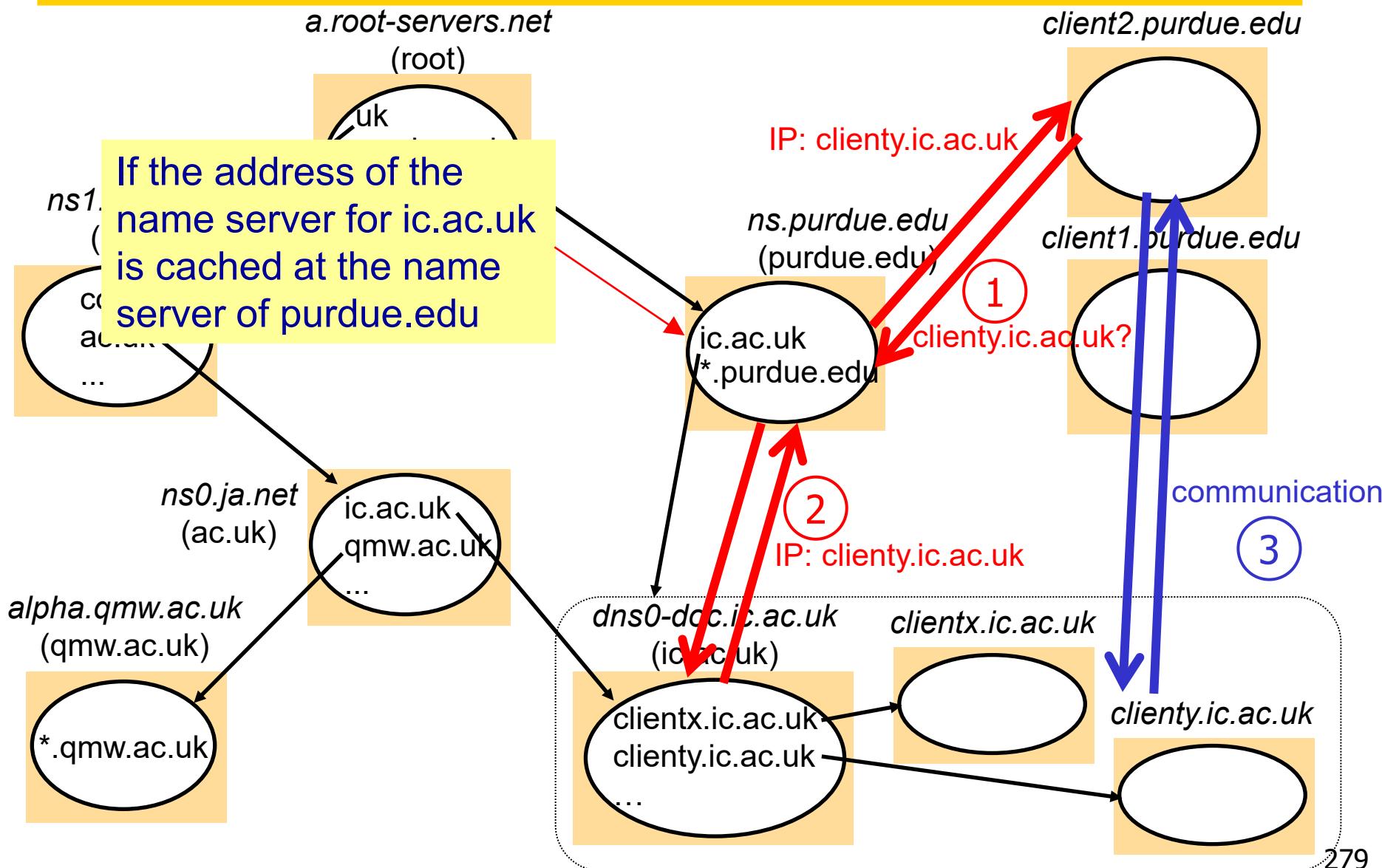
DNS Name Resolution Process



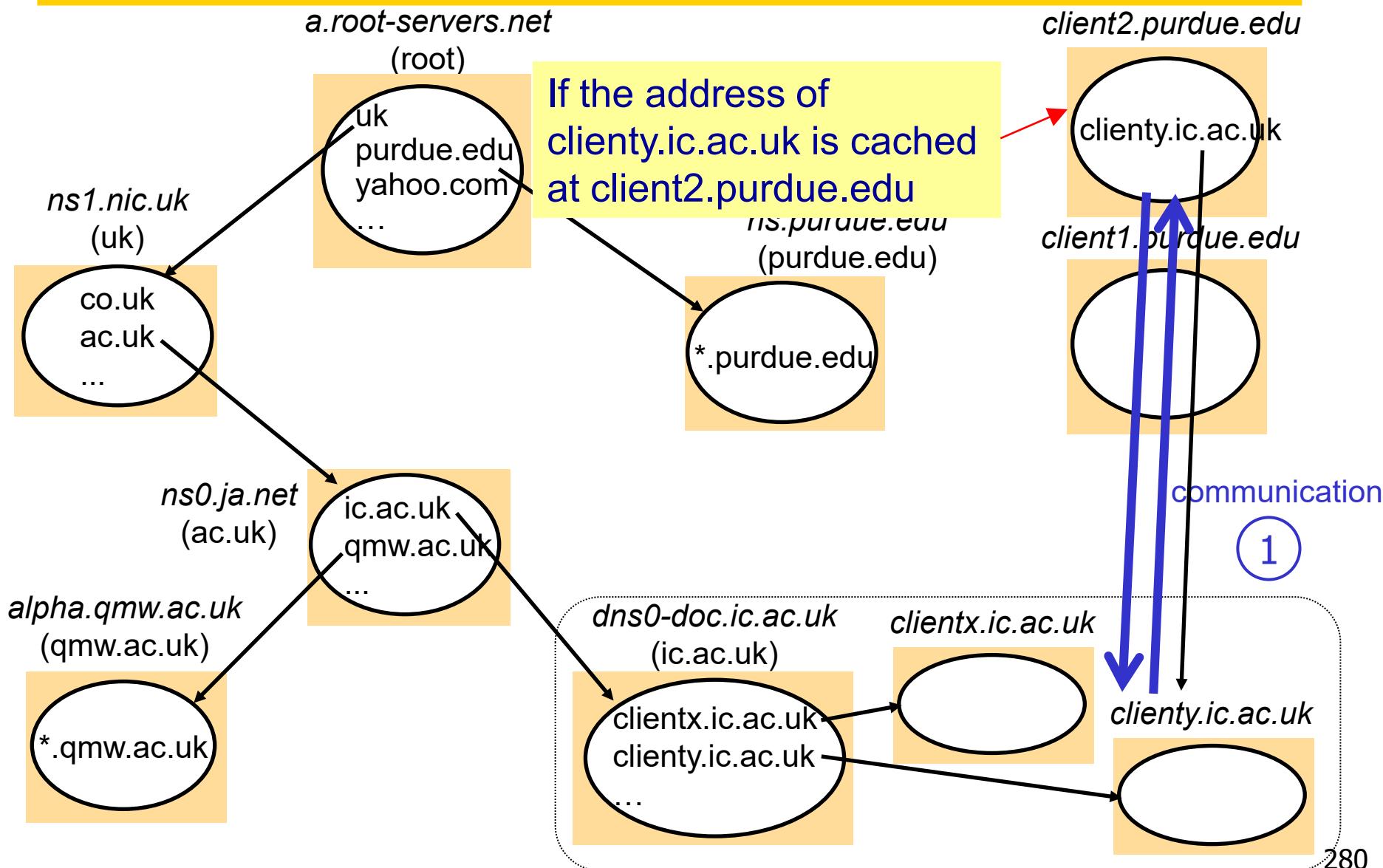
Caching for Performance Improvement

- DNS uses **caching** to improve performance
- Results of name resolution may be cached by clients and name servers
- Clients and name servers always consult their caches before sending queries to other servers for name resolution
- Advantages
 - Reduce response time
 - Save network traffic
 - Alleviate workload of high-level name servers

Name Resolution Process with Caching



Name Resolution Process with Caching



Caching for Performance Improvement

- Caching introduces the problem of **consistency**
 - Query answers returned from cached mappings are called **non-authoritative answers**
 - Cached mappings and hence non-authoritative answers may be out-of-date
- Consistency problem can be addressed by **timeouts**
 - Cached mappings are associated with timeout periods (**time-to-live values**) beyond which they expire
 - Stale answers may still be provided before timeout
 - **Longer time-to-live value → weaker consistency**
 - We can afford to set a long timeout period (e.g., one to seven days) because names and addresses generally do not change often

Summary

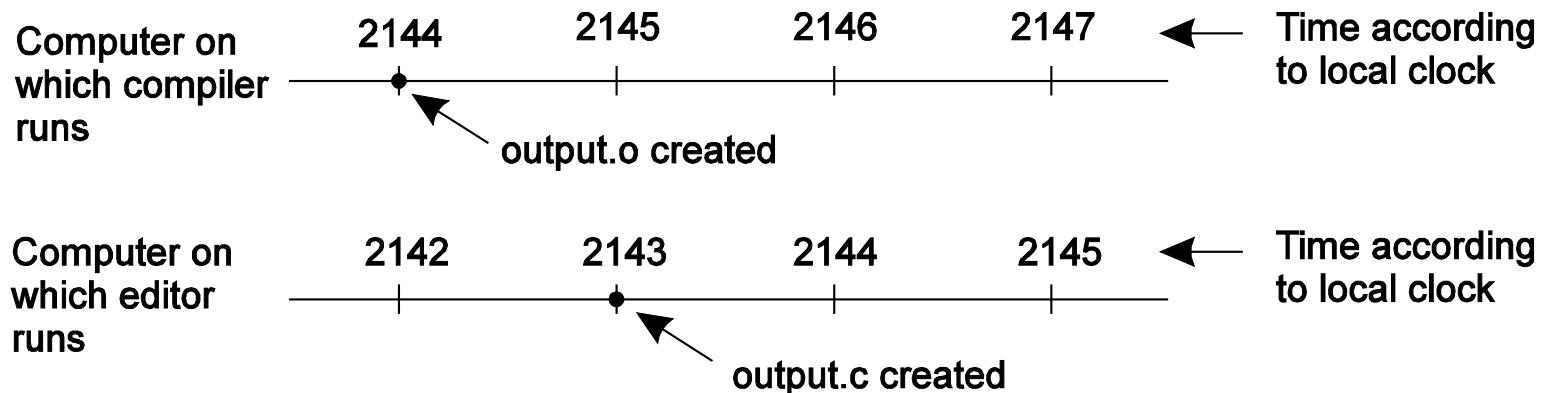
Learned about:

- **Names and Name Services**
 - Name service: names → attributes (in particular, addresses)
 - Name space structure
 - Name resolution process
- **Domain Name System**
 - Partitioning, replication, caching

7. Time and Global States

Introduction

- Recall one fundamental characteristic of distributed systems: loosely coupled
 - No globally shared clock – computers have their own physical clocks that deviate from one another
- Many distributed applications depend on timing
 - UNIX **make** facility compiling source files



- Electronic commerce, e.g., auction and bidding

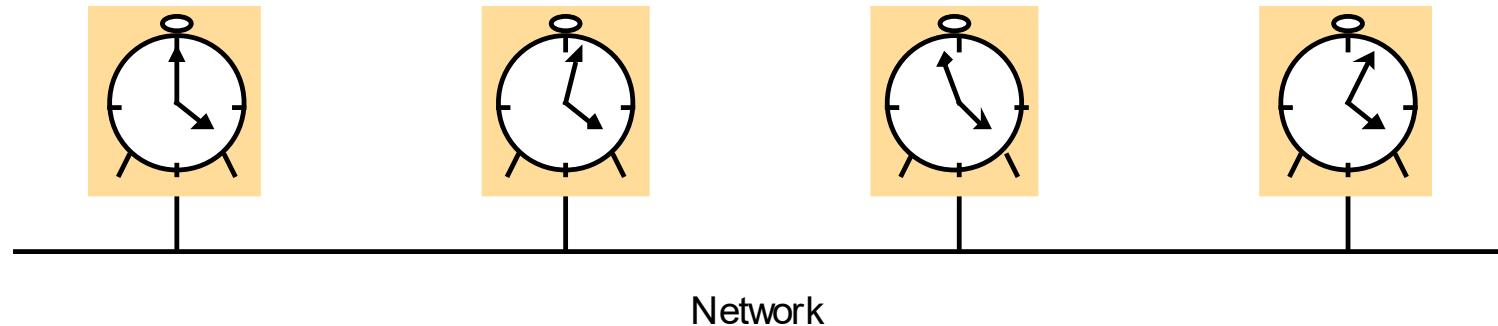
Introduction

- Solutions
 - Synchronize physical clocks of all computers
 - Design algorithms that do not rely on physical times

Outline

- Synchronizing Physical Clocks
- Causal Ordering and Logical Clocks
- Global States
- Distributed Debugging
- Summary

Computer Clocks

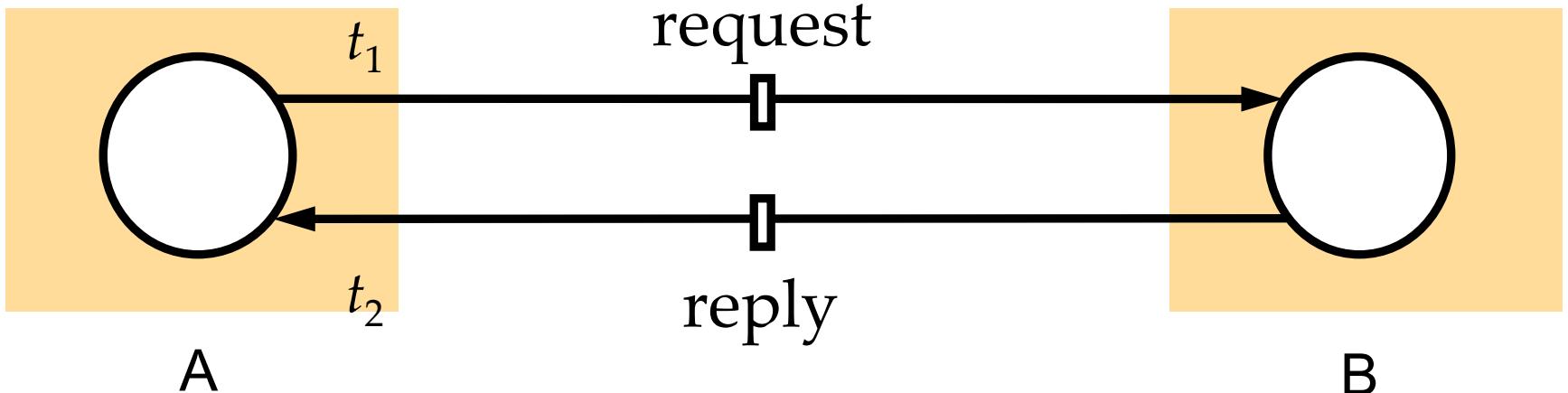


- Computers have their own physical clocks that deviate from one another
- **Offset:** instantaneous difference between the readings of two clocks
- Commonly used reference clock – **Universal Coordinated Time (UTC)**
 - UTC signals are broadcast regularly from land-based radio stations and satellites around the world

Computer Clocks

- **Clock drift:** clocks count time at different rates
- **Drift rate:** the change in offset between a clock and a perfect reference clock per unit of time measured by the reference clock
 - If a clock ticks 1,000,001 seconds while the reference clock ticks 1,000,000 seconds, drift rate = 1 $\mu\text{s}/\text{second}$
 - If a clock ticks 999,999 seconds while the reference clock ticks 1,000,000 seconds, drift rate = 1 $\mu\text{s}/\text{second}$
 - Drift rate of quartz crystal: about 10^{-6}

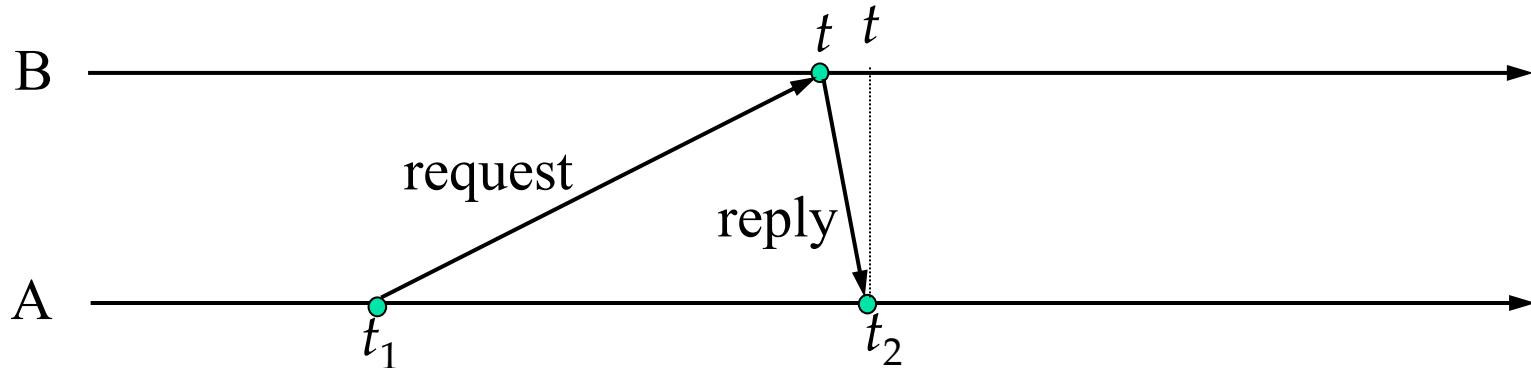
Cristian's Method



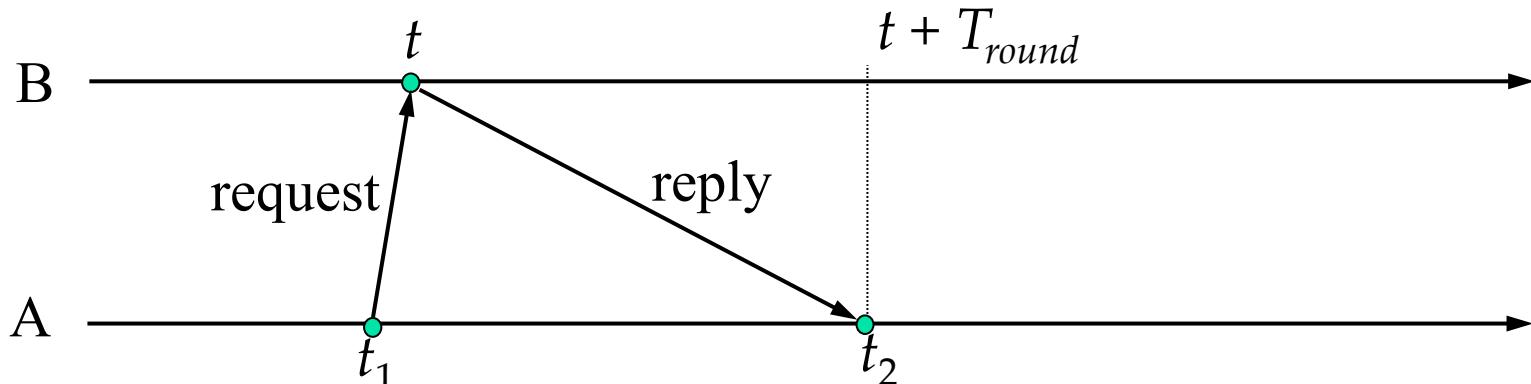
- Cristian's method
 - A requests the time of B
 - B replies with the time value t
 - A records the total round-trip time $T_{round} = t_2 - t_1$
 - A sets its clock to $t + T_{round}/2$
- So, what is the accuracy?

Cristian's Method

Case 1: transmission time of reply message approaches 0 → when A receives the reply, B's clock reading is t



Case 2: transmission time of reply message approaches T_{round} → when A receives the reply, B's clock reading is $t + T_{round}$



Cristian's Method

- When A receives the reply message, the time reading on B should be in $[t, t + T_{round}]$
- A sets its clock to $t + T_{round}/2$
→ it is at most $T_{round}/2$ faster and is at most $T_{round}/2$ slower than B
- So, A is accurate within bound $T_{round}/2$
- To handle variability, make several requests to B and take the minimum value of T_{round} to give the most accurate estimate

Berkeley Algorithm

- Proposed by Gusella & Zetti for synchronizing a group of computers running Berkeley UNIX
- **Master:** a coordinator computer
- **Slaves:** other computers whose clocks are to be synchronized

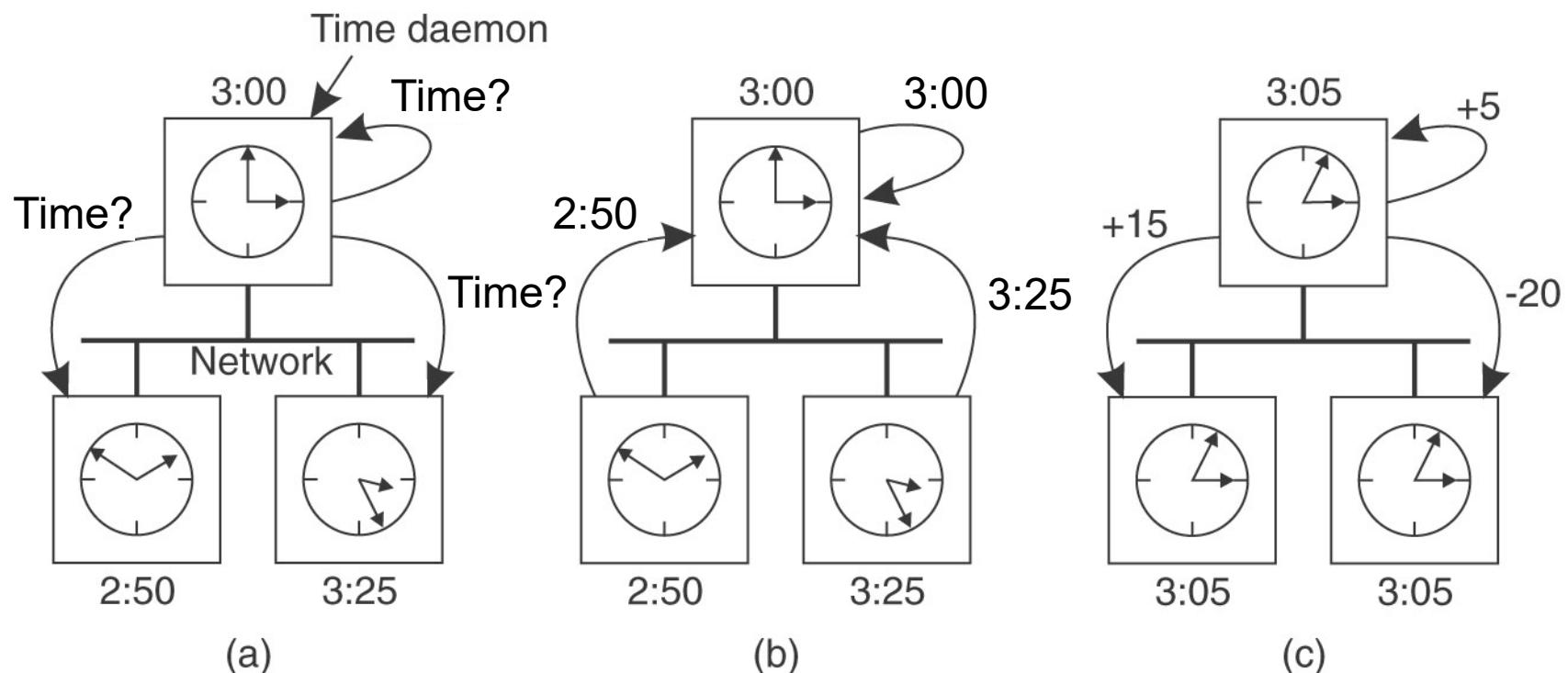
Berkeley Algorithm

- How does it work?

- Master periodically **polls** slaves
- Slaves send their clock readings back
- Master evaluates the local times of slaves by observing roundtrip times (similar to Cristian's method)
- Master **averages** the values obtained (including its own clock reading)
- Master sends **the amount for adjustment** to each slave whose clock requires adjustment

Berkeley Algorithm

- A simplified example



Berkeley Algorithm

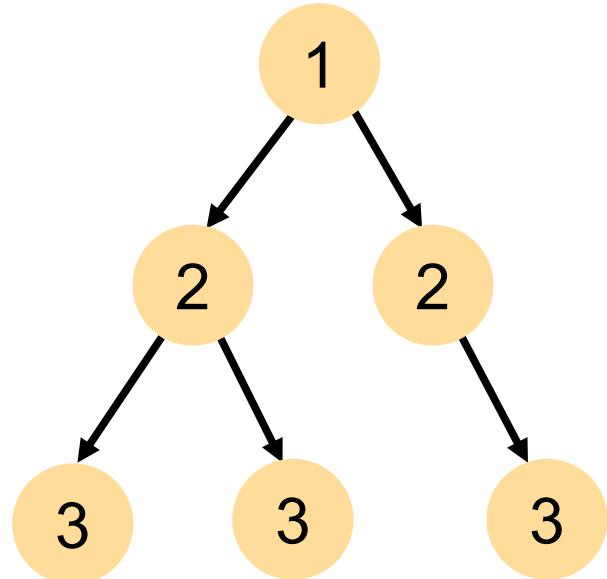
- Why average?
 - Average clock reading **cancels out** individual clocks' tendencies to run fast or slow
- Why sending amounts of adjustment to slaves?
 - Sending average clock reading back to slaves introduce **further uncertainty** due to message transmission time
- If master fails, another computer can be elected to take over its responsibility
- Empirical results: 15 computers on LAN (roundtrip time ~ 10 ms; clock drifts $\sim 20 \mu\text{s/s}$) synchronized to agree within 20-25 ms

Network Time Protocol

- Cristian's method and Berkeley algorithm
 - Primarily for intranets
 - Centralized design (not scalable)
- Network Time Protocol (NTP)
 - For distributing time information over the Internet

Network Time Protocol

- A network of servers structured hierarchically into a synchronization subnet
 - Primary servers (stratum 1) are connected directly to a time source (e.g., radio clock receiving UTC)
 - Stratum 2 servers are synchronized with primary servers
 - Stratum 3 servers are synchronized with stratum 2 servers
 -
 - Lowest-level servers execute in users' workstations



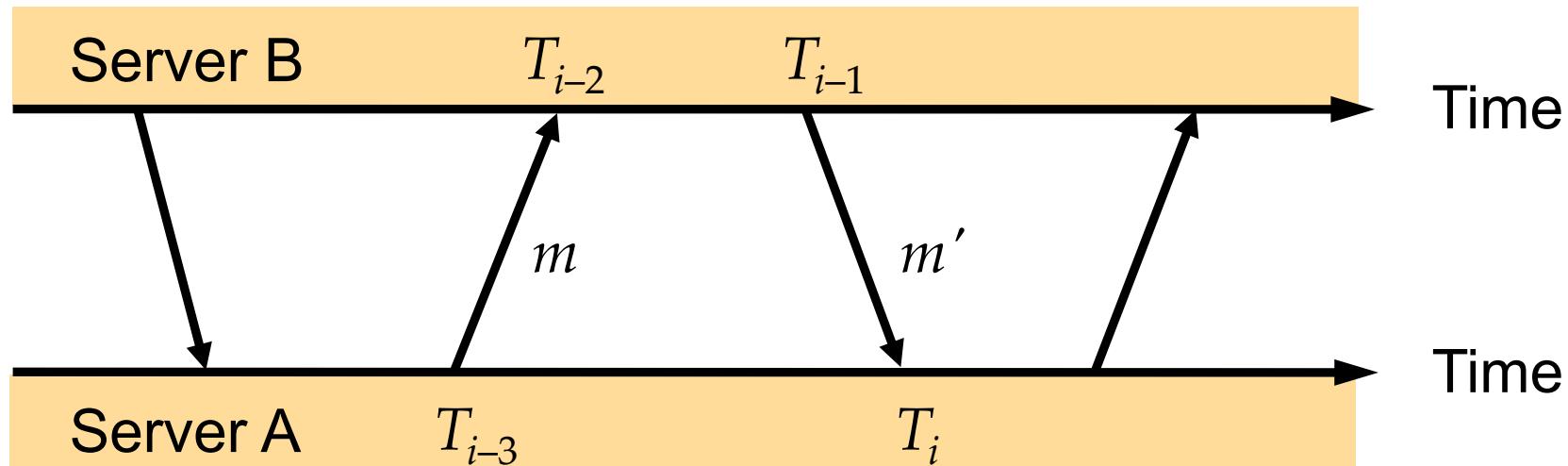
Arrows denote synchronization control, numbers denote strata

Network Time Protocol

- Fault-tolerance
 - Servers can reconfigure themselves if someone becomes unreachable
 - e.g., if a primary server's UTC source fails, it can become a stratum 2 server
 - e.g., if a stratum 2 server's normal source of synchronization (which is a primary server) fails or becomes unreachable, it may synchronize with another primary server
 - ...

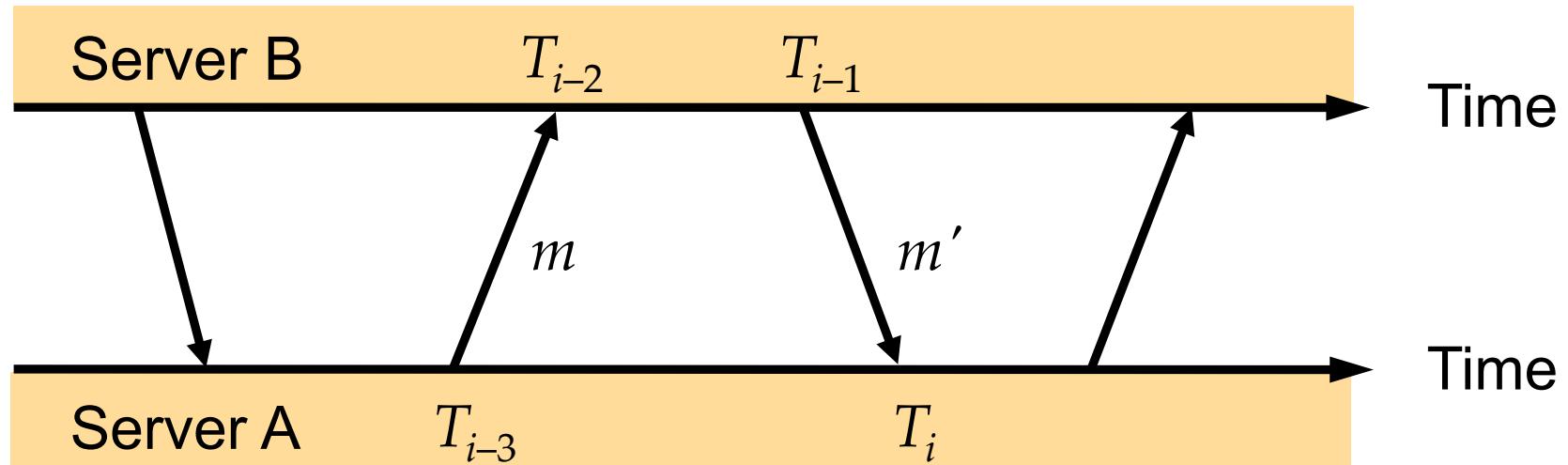
Network Time Protocol

- How to perform synchronization?
 - A pair of servers exchange messages bearing timing information

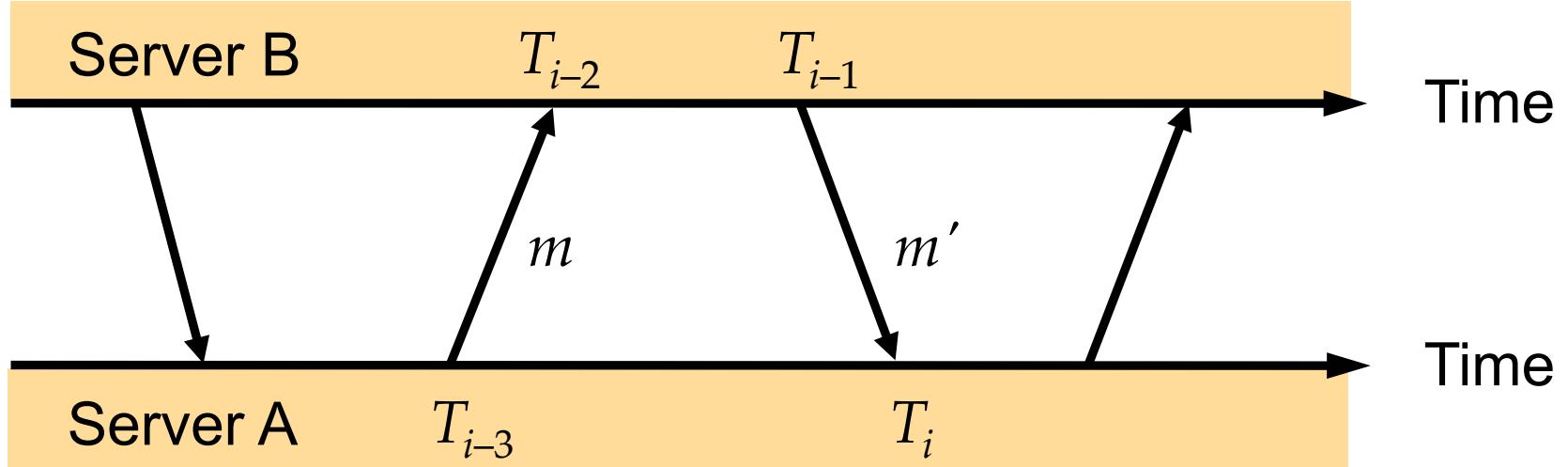


NTP Message Exchange

- Each NTP message contains:
 - Local times when the previous NTP message between the pair was sent and received
 - Local time when the current NTP message was sent
- Recipient of NTP message records local time of receipt

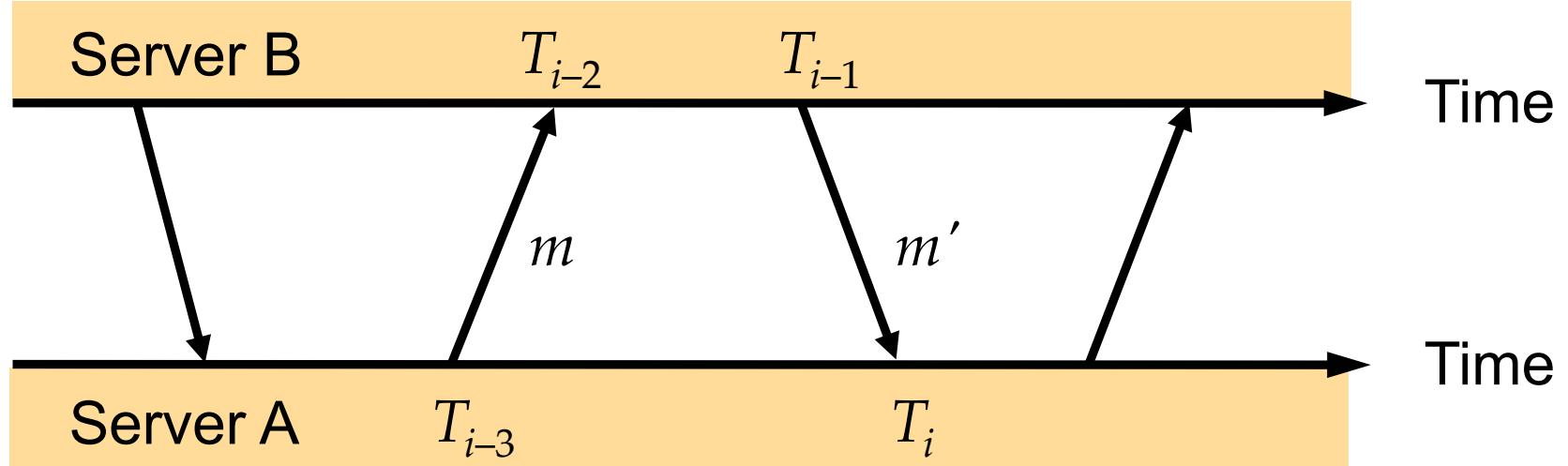


NTP Message Exchange



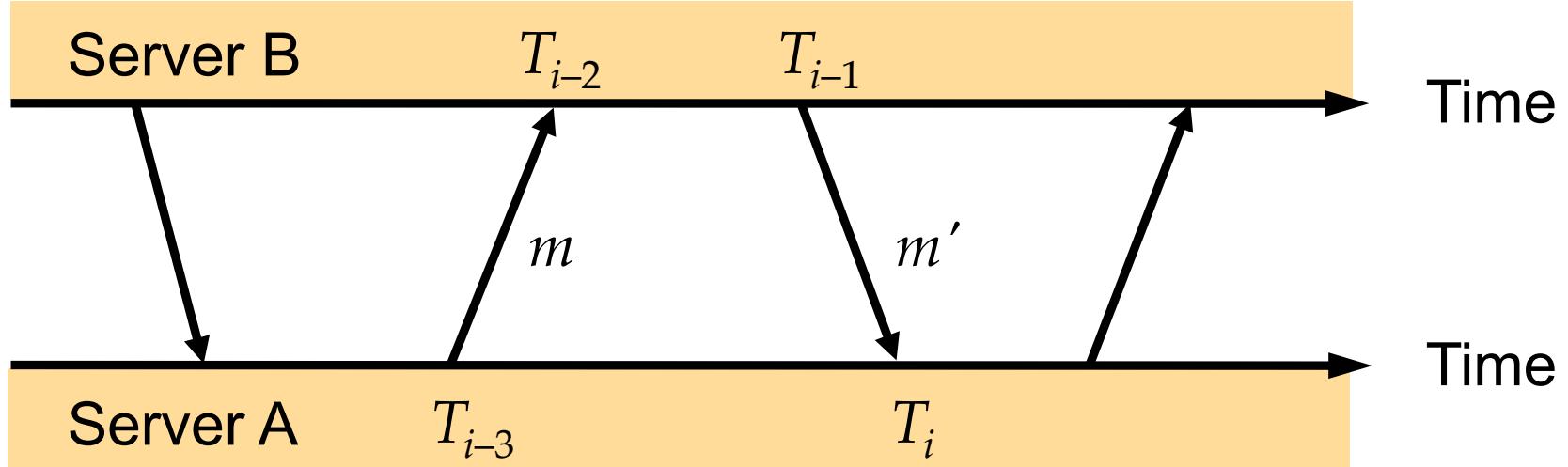
- For each pair of messages m and m' between 2 servers, how does A synchronize with B?
- Unlike Cristian's method, there can be a **non-negligible delay** between the arrival of m and the dispatch of m' at server B → consider this delay

NTP Message Exchange



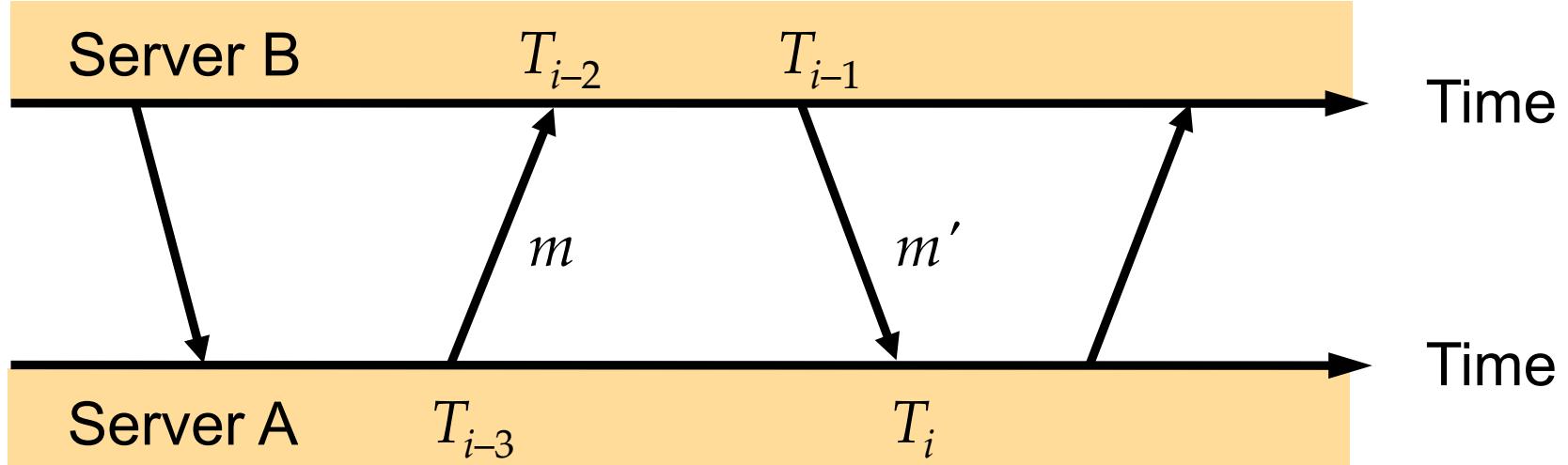
- Time between sending of m and receipt of m' is $\textcolor{violet}{T_i - T_{i-3}}$
- Time between receipt of m and sending of m' is $\textcolor{green}{T_{i-1} - T_{i-2}}$
- So, total transmission time of m and m' is $(\textcolor{violet}{T_i - T_{i-3}}) - (\textcolor{green}{T_{i-1} - T_{i-2}})$
- Thus, transmission time of m' is between 0 and $(\textcolor{violet}{T_i - T_{i-3}}) - (\textcolor{green}{T_{i-1} - T_{i-2}})$

NTP Message Exchange



- If transmission time of m' approaches 0, when A receives m' , B's clock reading is $\textcolor{red}{T_{i-1}}$
- If transmission time of m' approaches $(\textcolor{violet}{T_i} - T_{i-3}) - (\textcolor{green}{T_{i-1}} - \textcolor{green}{T_{i-2}})$, when A receives m' , B's clock reading is $T_{i-1} + (\textcolor{violet}{T_i} - T_{i-3}) - (\textcolor{green}{T_{i-1}} - \textcolor{green}{T_{i-2}}) = \textcolor{blue}{T_i} - T_{i-3} + T_{i-2}$

NTP Message Exchange



- If A wants to synchronize with B as accurately as possible, A should set its clock to $\frac{1}{2} (\textcolor{red}{T}_{i-1} + (T_i - T_{i-3} + T_{i-2})) = \frac{1}{2} (T_i + T_{i-1} + T_{i-2} - T_{i-3})$ when receiving m'
- This setting has an accuracy of $\pm \frac{1}{2} (\textcolor{red}{T}_{i-1} - (T_i - T_{i-3} + T_{i-2}))$

Network Time Protocol

- NTP applies data filtering to improve the accuracy of synchronization over time
 - Retain estimates of 8 most recent message exchanges and their accuracies
 - Choose the one with the highest accuracy for synchronization
- To choose a peer for primary use of synchronization, an NTP server engages in message exchanges with several of its peers and applies a peer-selection algorithm
- Empirical results: synchronization accuracy, 1 ms on LAN, tens of milliseconds over Internet

Revisit

■ Solutions

- Synchronize physical clocks of all computers
- Design algorithms that do not rely on physical times
 - In many cases, knowing the absolute time is not necessary
 - What counts is the order in which related events at different processes occur

Outline

- Synchronizing Physical Clocks
- Causal Ordering and Logical Clocks
- Global States
- Distributed Debugging
- Summary

System Model

- **Distributed system**

- A collection of N processes p_1, p_2, \dots, p_N
- Each process executes on a single processor
- Processors do not share memory
- So, processes cannot communicate with one another except sending messages through the network
- Communication channels are reliable
- Asynchronous system: no bounds on process execution speeds, message transmission delays and clock drift rates

System Model

- State of a process
 - Values of all variables within the process
 - Values of any objects in the local OS environment that the process affects (e.g., files)
 - We refer to p_i 's state as s_i
- Execution of a process
 - Process takes **a series of actions**, each of which is either sending a message, receiving a message, or an internal action that transforms the state of the process
 - An event: **the occurrence of a single action** that a process carries out as it executes

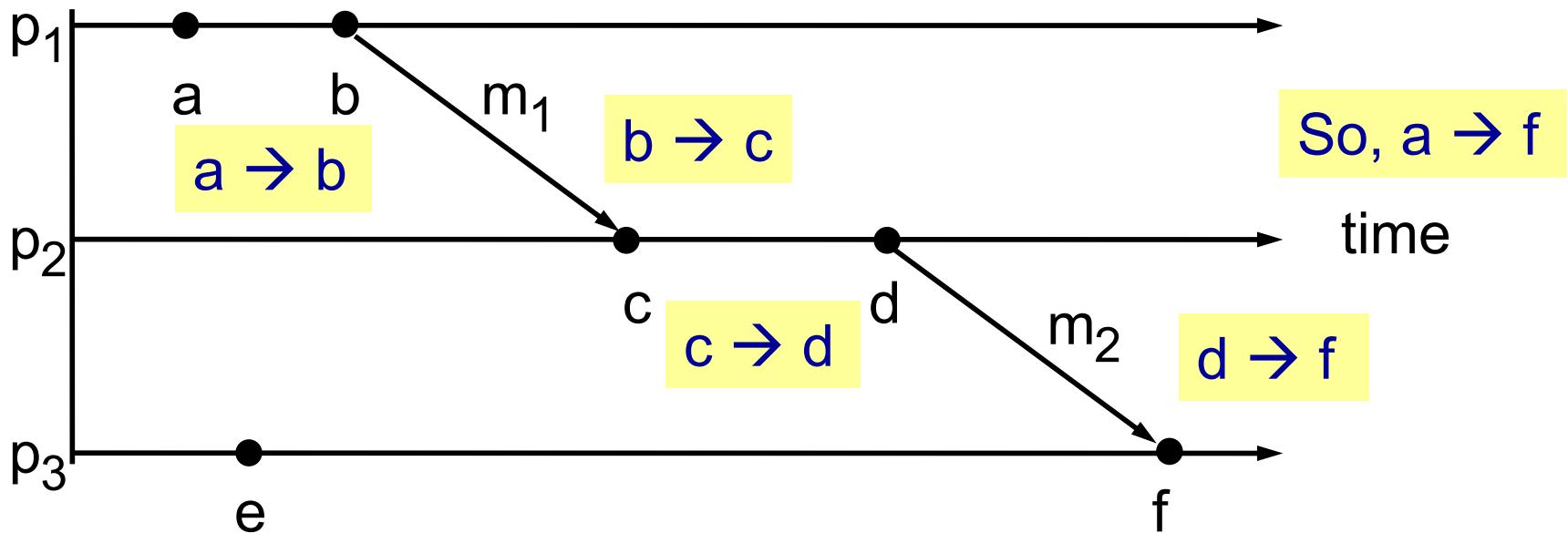
Causal Ordering

- In an asynchronous system where no globally shared clock exists, events can be ordered based on “cause-and-effect” (similar to physical causality)
 - Events within a single process p_i can be placed in a unique total ordering
 - Since each process executes on a single processor whose clock always advances, the sequence of events within a single process p_i is well defined
 - Whenever a message is sent between processes, the event of sending the message occurs before the event of receiving the message

Causal Ordering

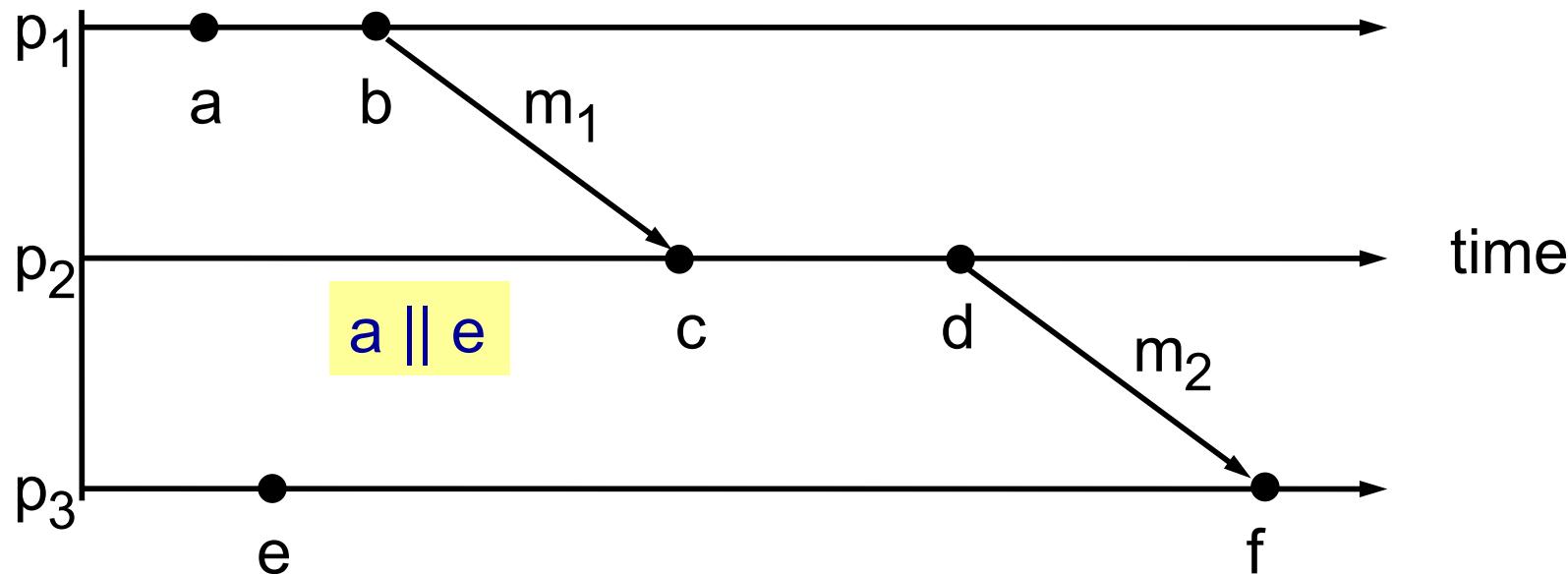
- Causal ordering (happened-before relation)
 - Denoted by “ \rightarrow ”
 - $a \rightarrow b$ is read as “*a happened-before b*”
 - If e and e' are two events in the same process p_i and e occurs before e' at p_i , then $e \rightarrow e'$
 - For any message m , let $send(m)$ be the event of sending m , and $receive(m)$ be the event of receiving m , then $send(m) \rightarrow receive(m)$
 - Note that $send(m)$ and $receive(m)$ occur at different processes
 - If e, e', e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Example of Causal Ordering



- If $e \rightarrow e'$, we can find a sequence of events e_1, e_2, \dots, e_n occurring at one or more processes such that
 - $e = e_1, e' = e_n$
 - For each $1 \leq i \leq n-1$, either e_i and e_{i+1} occur in succession at the same process, or there is a message m such that $e_i = \text{send}(m)$ and $e_{i+1} = \text{receive}(m)$

Example of Causal Ordering



- Causal ordering is **a partial ordering** – it may not be true that all pairs of distinct events are ordered
- If neither $e \rightarrow e'$ nor $e' \rightarrow e$, we say that they are **concurrent** and denote it by $e \parallel e'$

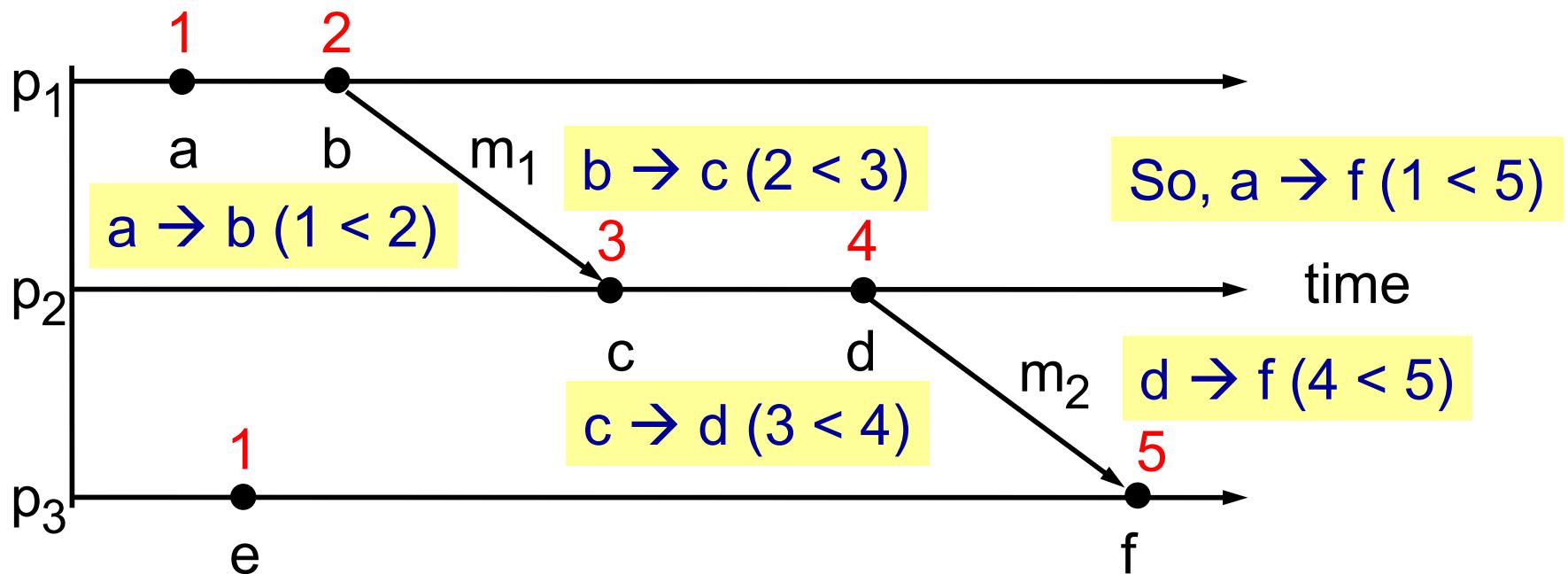
Lamport's Logical Clocks

- Logical clock
 - A monotonically increasing software counter whose value bears no particular relationship with real time
- Each process p_i keeps its own logical clock L_i to timestamp events

Lamport's Logical Clocks

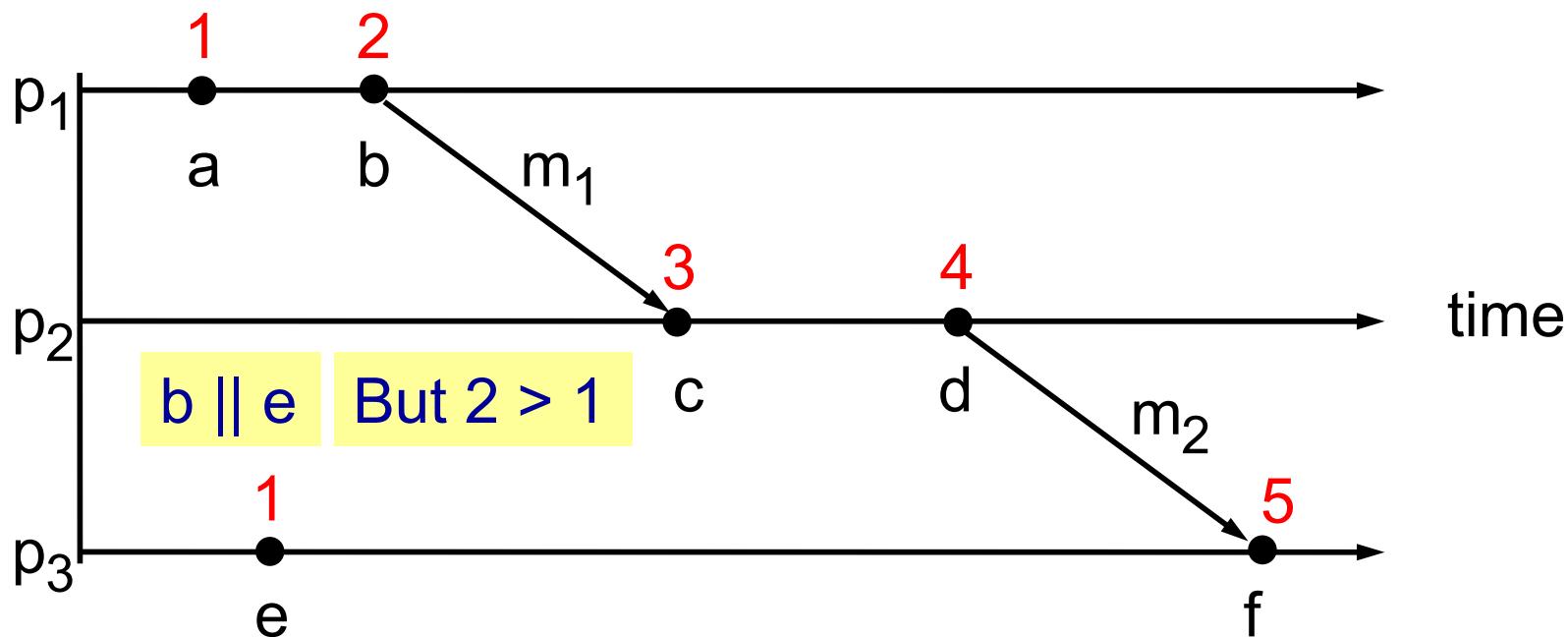
- How does it work? – update rules
 - All processes initialize their logical clocks to 0
 - Process p_i sets $L_i = L_i + 1$ before timestamping each event at p_i
 - When process p_i sends a message m , it piggybacks on m the value $t = L_i$
 - When another process p_j receives message (m, t) , it sets $L_j = \max(L_j, t)$ before incrementing L_j to timestamp event $receive(m)$

Example of Lamport's Logical Clocks



- Denote the timestamp of event e by $L(e)$
- If $e \rightarrow e'$, then $L(e) < L(e')$ (How to prove it?)

Example of Lamport's Logical Clocks



- However, if $L(e) < L(e')$, we cannot infer that $e \rightarrow e'$

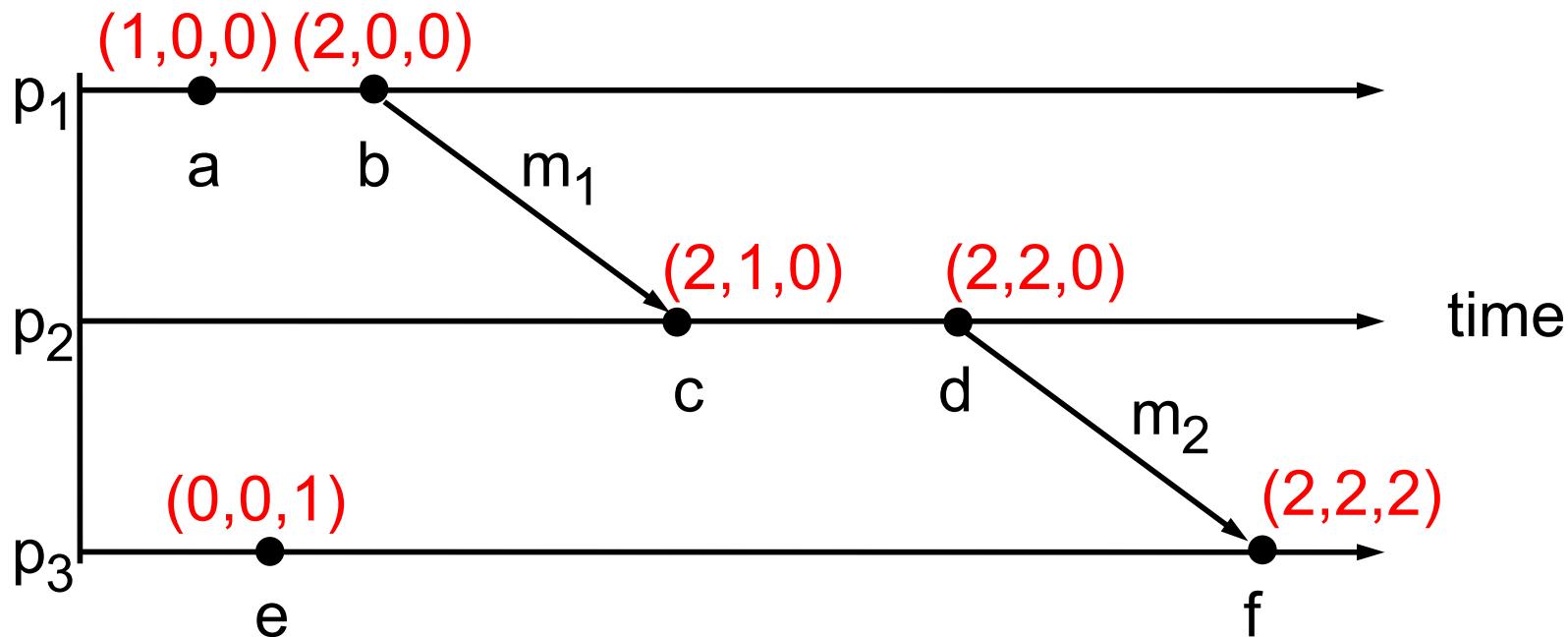
Vector Clocks

- Vector clock
 - A vector clock for a system of N processes is **an array of N integers**
- Each process p_i keeps its own vector clock V_i to timestamp events
 - Denote the k th component of clock V_i by $V_i[k]$

Vector Clocks

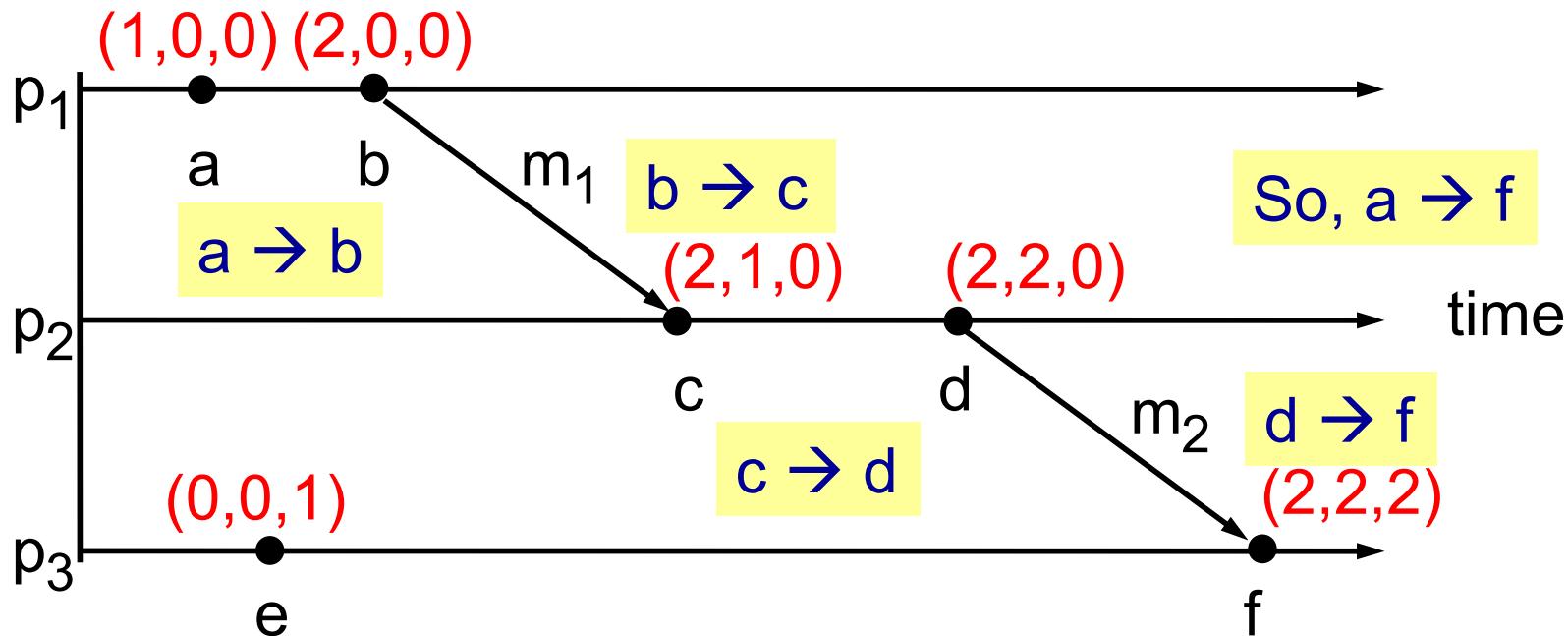
- How does it work? – update rules
 - All processes initialize their vector clocks to $(0, 0, \dots, 0)$, i.e., $V_i[k] = 0$ for $i, k = 1, 2, \dots, N$
 - Process p_i sets $V_i[i] = V_i[i] + 1$ before timestamping each event at p_i
 - When process p_i sends a message m , it piggybacks on m the value $t = V_i$
 - When another process p_j receives message (m, t) , it sets $V_j[k] = \max(V_j[k], t[k])$ for $k = 1, 2, \dots, N$ before incrementing $V_j[j]$ to timestamp event $receive(m)$

Example of Vector Clocks



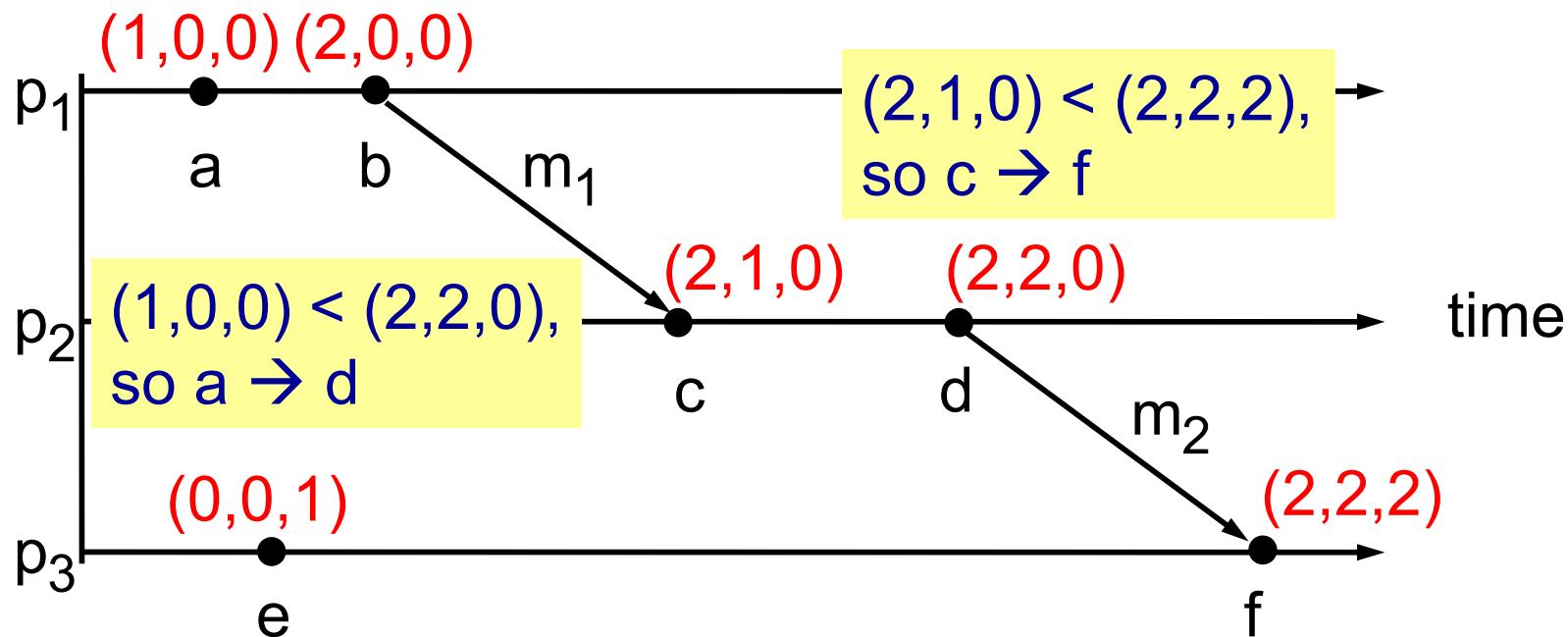
- Rules to compare vector timestamps V and V'
 - $V = V'$ if and only if $V[k] = V'[k]$ for $k = 1, 2, \dots, N$
 - $V \leq V'$ if and only if $V[k] \leq V'[k]$ for $k = 1, 2, \dots, N$
 - $V < V'$ if and only if $V \leq V'$ and $V \neq V'$

Example of Vector Clocks



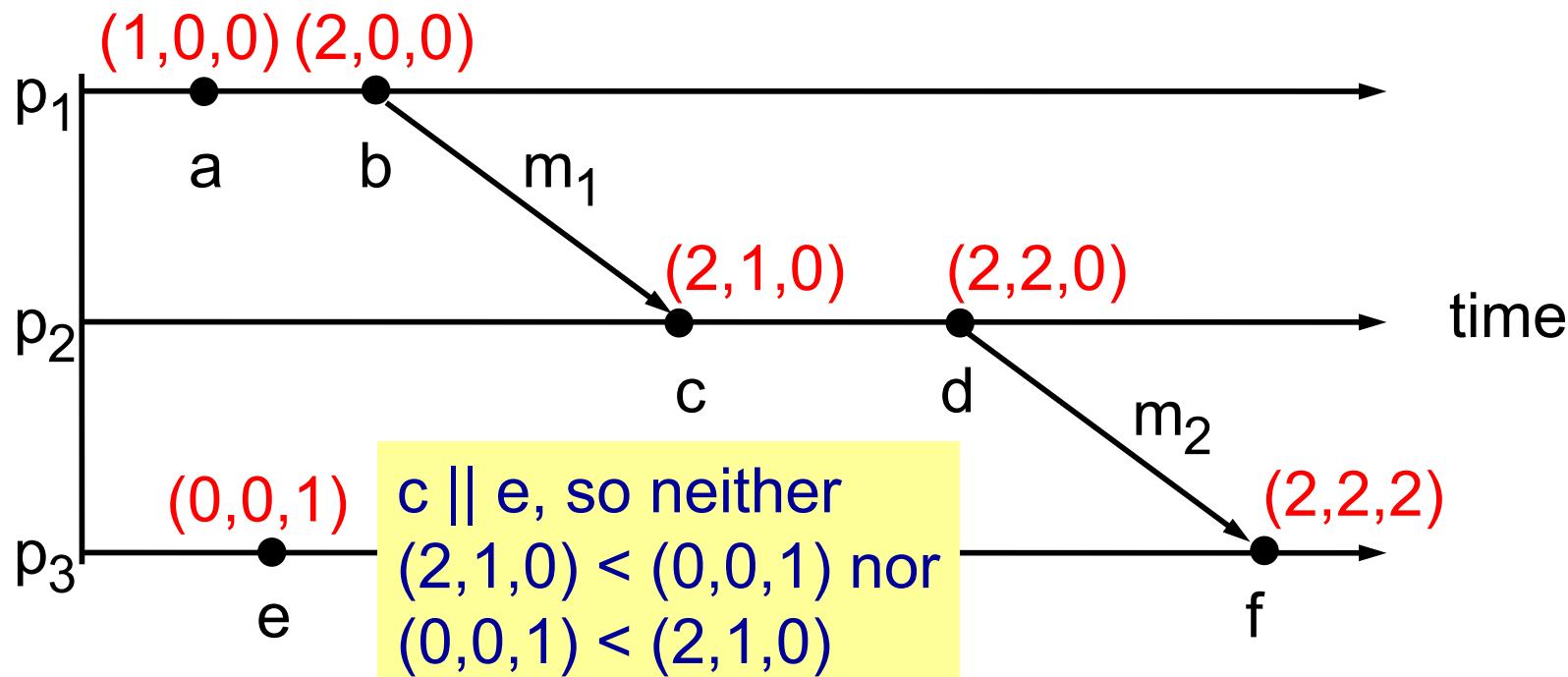
- Denote the timestamp of event e by $V(e)$
- If $e \rightarrow e'$, then $V(e) < V(e')$ (How to prove it?)

Example of Vector Clocks



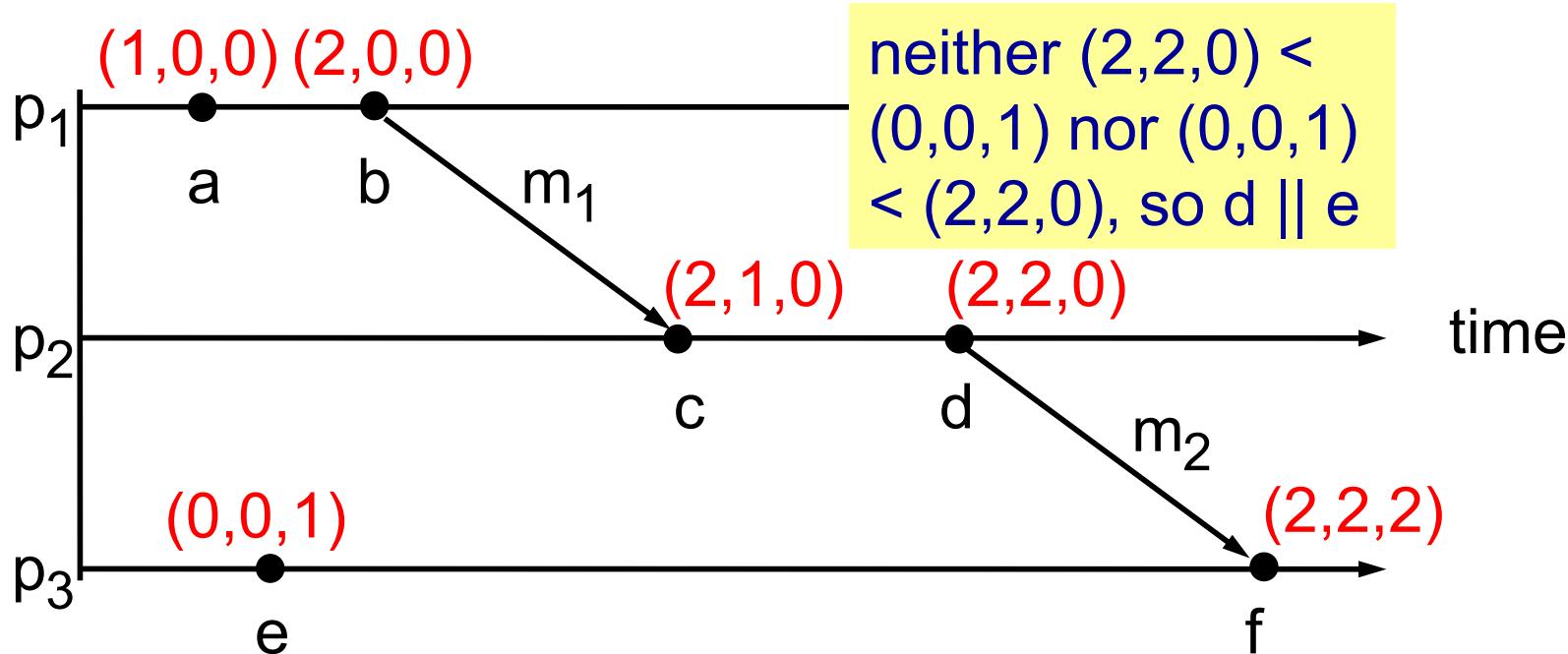
- If $V(e) < V(e')$, then $e \rightarrow e'$ (How to prove it?)

Example of Vector Clocks



- If $e \parallel e'$, then neither $V(e) \leq V(e')$ nor $V(e') \leq V(e)$
 - their timestamps are not comparable

Example of Vector Clocks

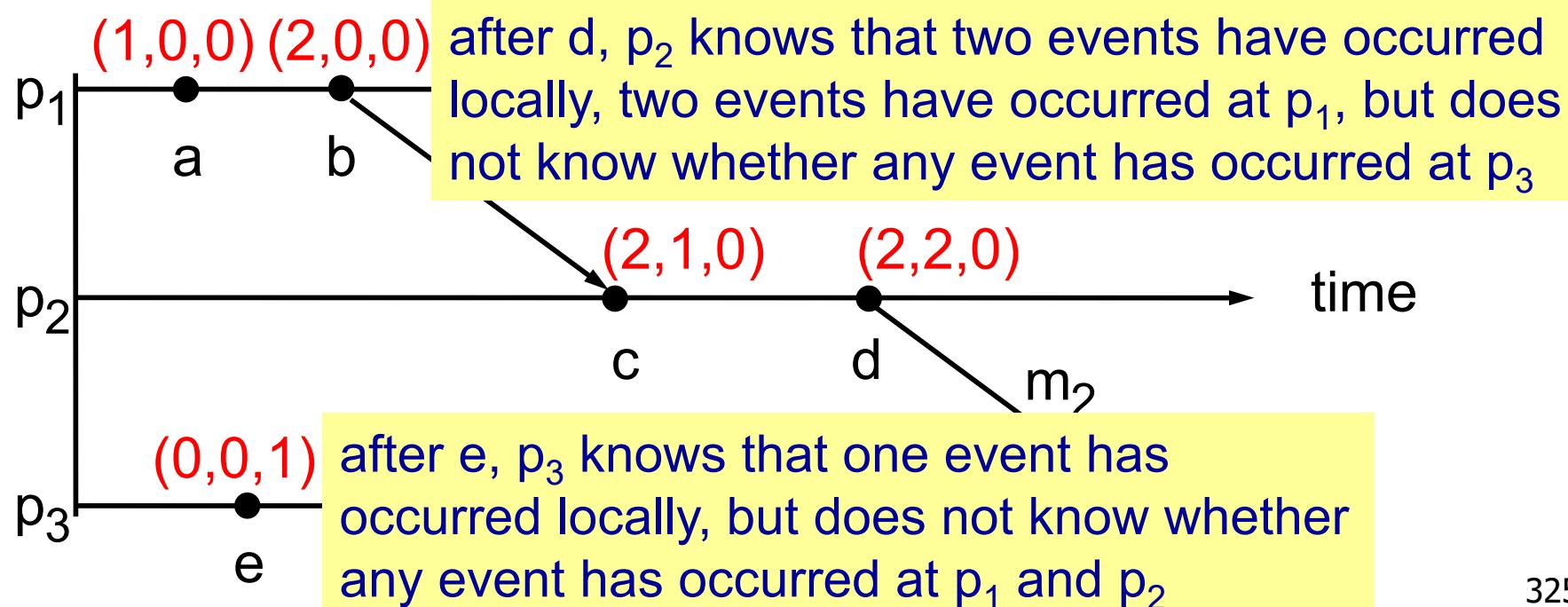


- If neither $V(e) \leq V(e')$ nor $V(e') \leq V(e)$, then $e \parallel e'$

Vector Clocks

Properties

- For a vector clock V_i , $V_i[i]$ is the number of events that have occurred at p_i so far
- For a vector clock V_i , $V_i[k]$ ($k \neq i$) is the number of events that have occurred at p_k that p_i knows



Outline

- Synchronizing Physical Clocks
- Causal Ordering and Logical Clocks
- **Global States**
- Distributed Debugging
- Summary

Motivation

- In many situations, we need to find out whether a particular property is true when a distributed system executes
- Example: distributed deadlock detection



- A distributed deadlock occurs when
 - Each of a collection of processes waits for another process to send it a message
 - There is a cycle in the graph of this ‘wait-for’ relationship

System Model

- Without global time, can we assemble a meaningful global state from local states recorded by different processes at different real times?
- History of process p_i
 - p_i 's history: all events that take place within p_i and ordered by their occurrences

$$h_i = \langle e_i^1, e_i^2, \dots \rangle$$

- Prefix of p_i 's history containing the first k events:

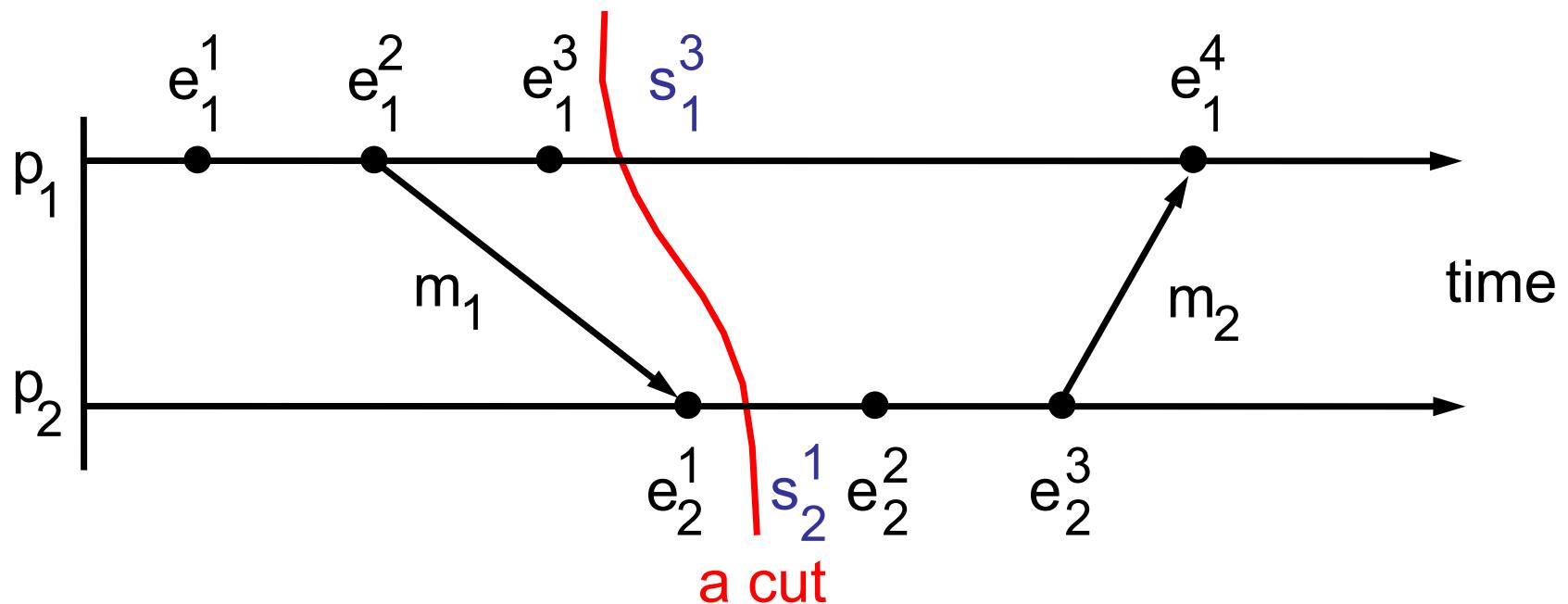
$$h_i^k = \langle e_i^1, e_i^2, \dots, e_i^k \rangle$$

System Model

- **Cut**
 - A union of the prefixes of process histories:
$$C = h_1^{c_1} \cup h_2^{c_2} \cup \dots \cup h_N^{c_N}$$
- **Global state**
 - A set of the states of all processes
 - We refer to the state of process p_i immediately after its k th event occurs as s_i^k (s_i^0 denotes the initial state of p_i)
 - Each global state corresponds to a cut – the set of the process states immediately after the occurrence of the last event of each process in the cut:

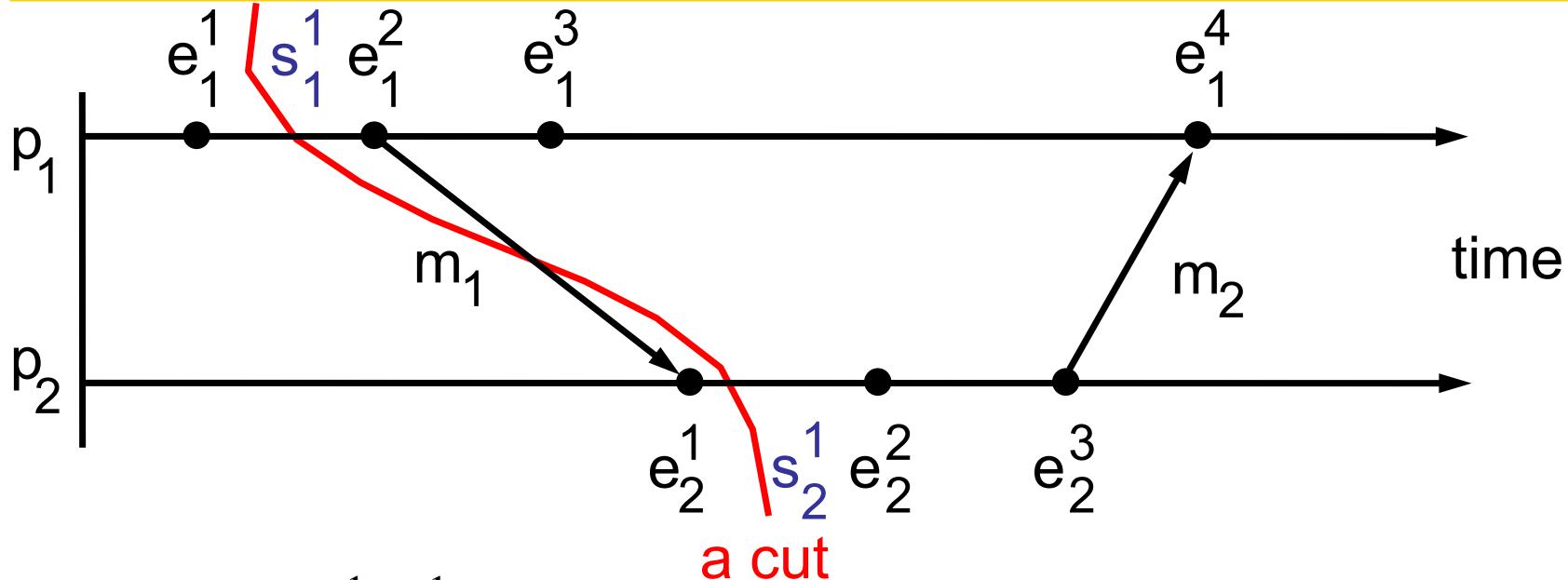
$$\langle s_1^{c_1}, s_2^{c_2}, \dots, s_N^{c_N} \rangle$$

Example of Cut



- Cut: $\langle e_1^1, e_1^2, e_1^3, e_2^1 \rangle$
- Global state corresponding to the cut: $\langle s_1^3, s_2^1 \rangle$

Example of Cut

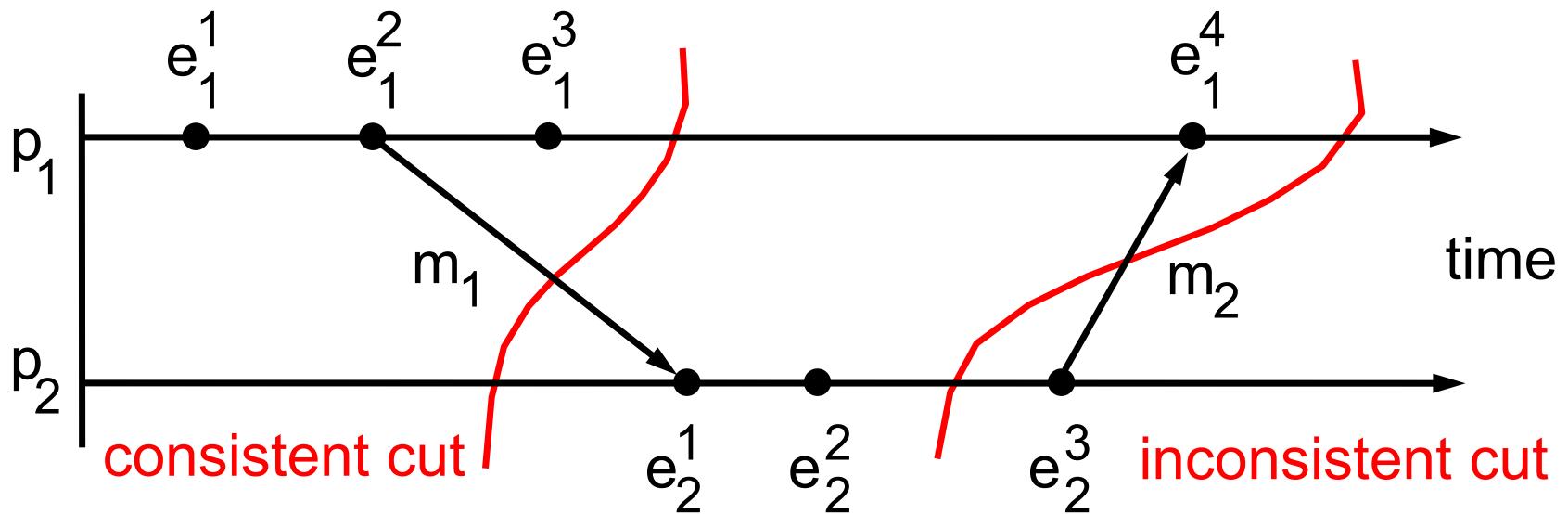


- Cut: $\langle e_1^1, e_2^1 \rangle$
- Global state corresponding to the cut: $\langle s_1^1, s_2^1 \rangle$
- However, are the processes likely to be in these states at the same time in actual execution? – No, because p_2 cannot have received m_1 before p_1 sends it

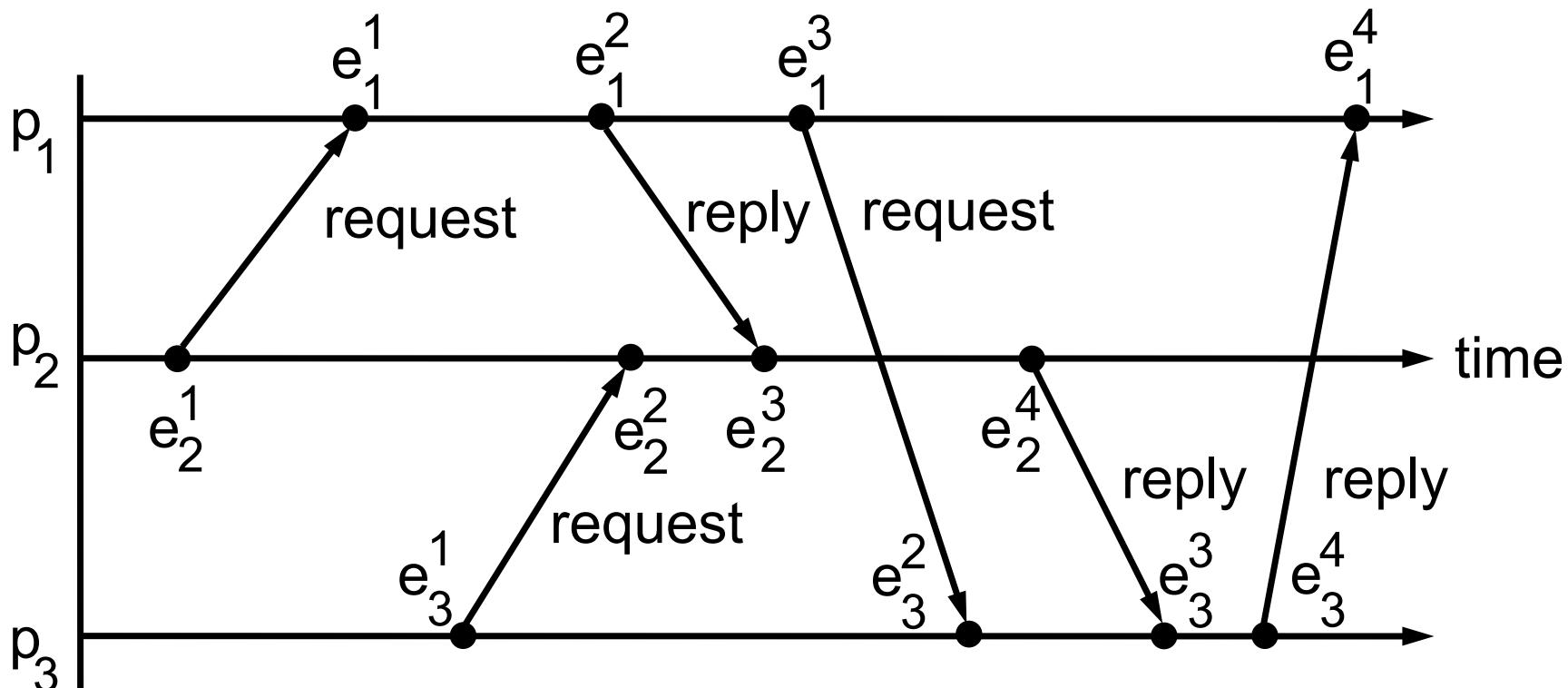
Consistent Cut

- We are interested in the cuts that are consistent with the progress of events
- Consistent cut
 - A cut C is **consistent** if, for all events $e \in C$,
 $f \rightarrow e \Rightarrow f \in C$
 - Otherwise, the cut is known as **inconsistent**
- Consistent global state
 - A consistent global state corresponds to a consistent cut
 - An inconsistent global state corresponds to an inconsistent cut
 - In actual execution, a system passes through consistent global states only

Example of Consistent Cut

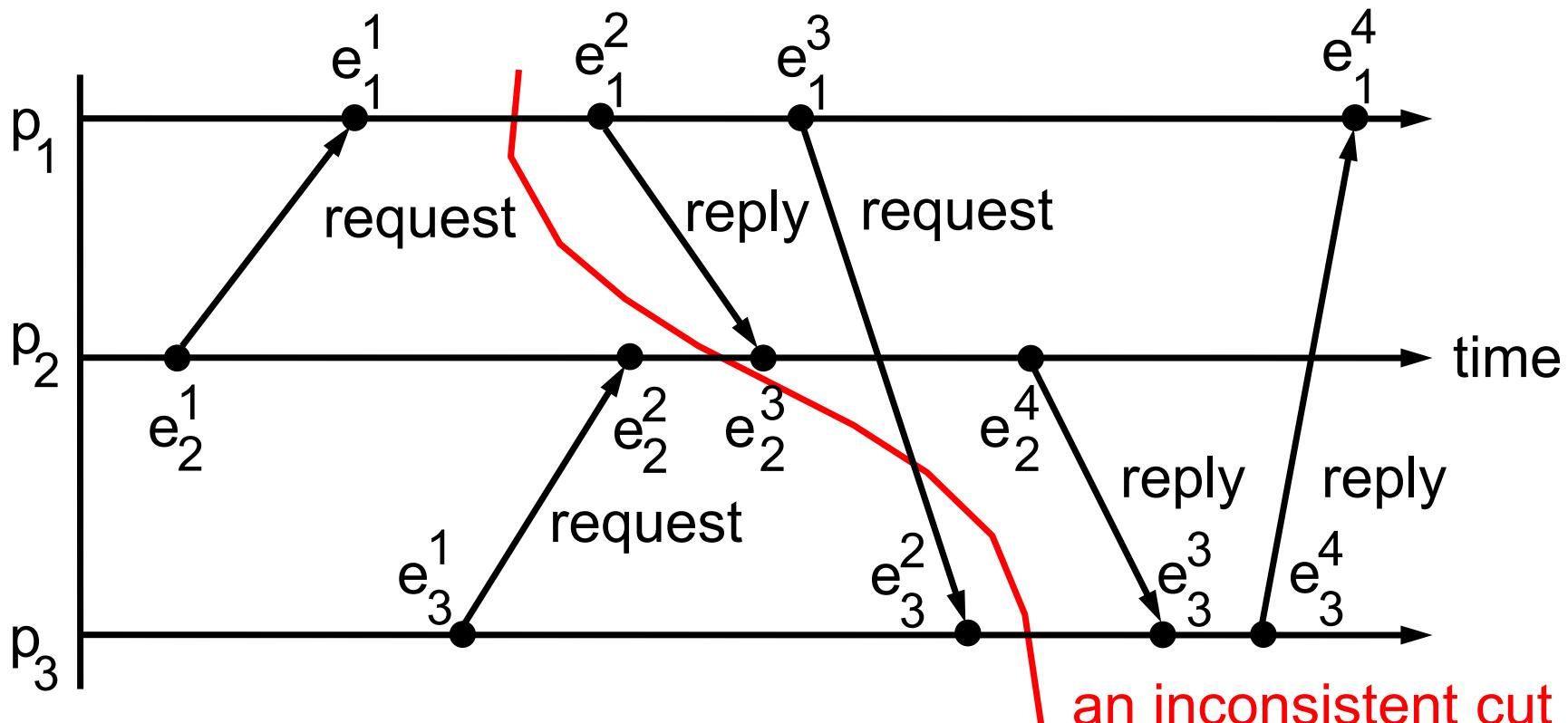


Why Is Consistent Cut Important?



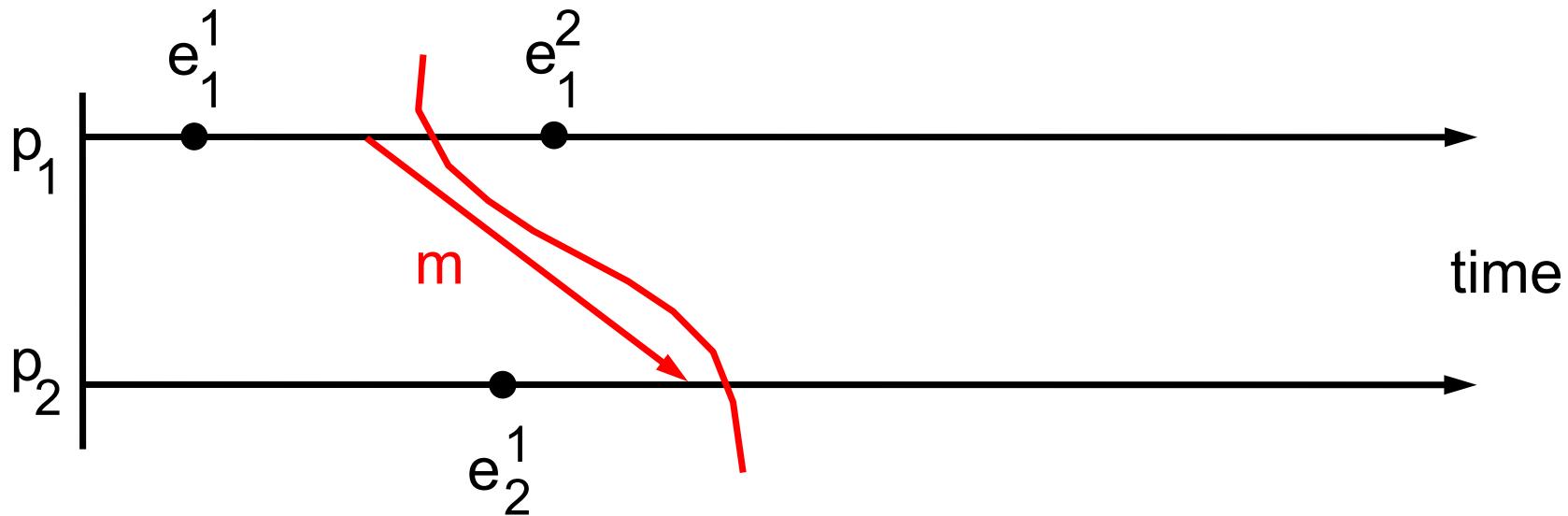
- Each process either waits for requests from other processes and handles them or sends a request to another process and waits for reply
- There is no deadlock in the above execution

Why Is Consistent Cut Important?



- However, if someone derives process states from an inconsistent cut, he may conclude that there is a deadlock because p_1 is waiting for p_3 , p_2 is waiting for p_1 , and p_3 is waiting for p_2

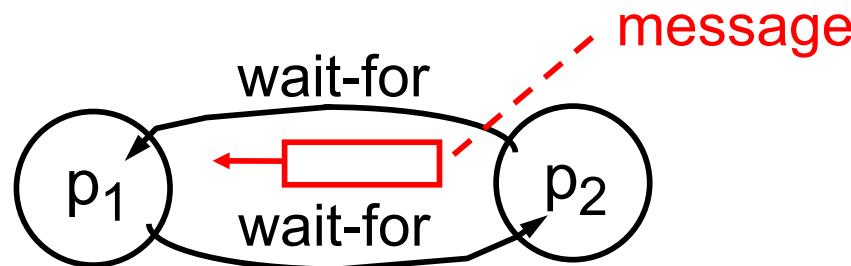
How to Record A Snapshot?



- p_1 records its local state and then sends a marker message m to p_2 who records its local state as soon as it receives the message
- Good enough? – in many applications, the state of communication channels is also relevant

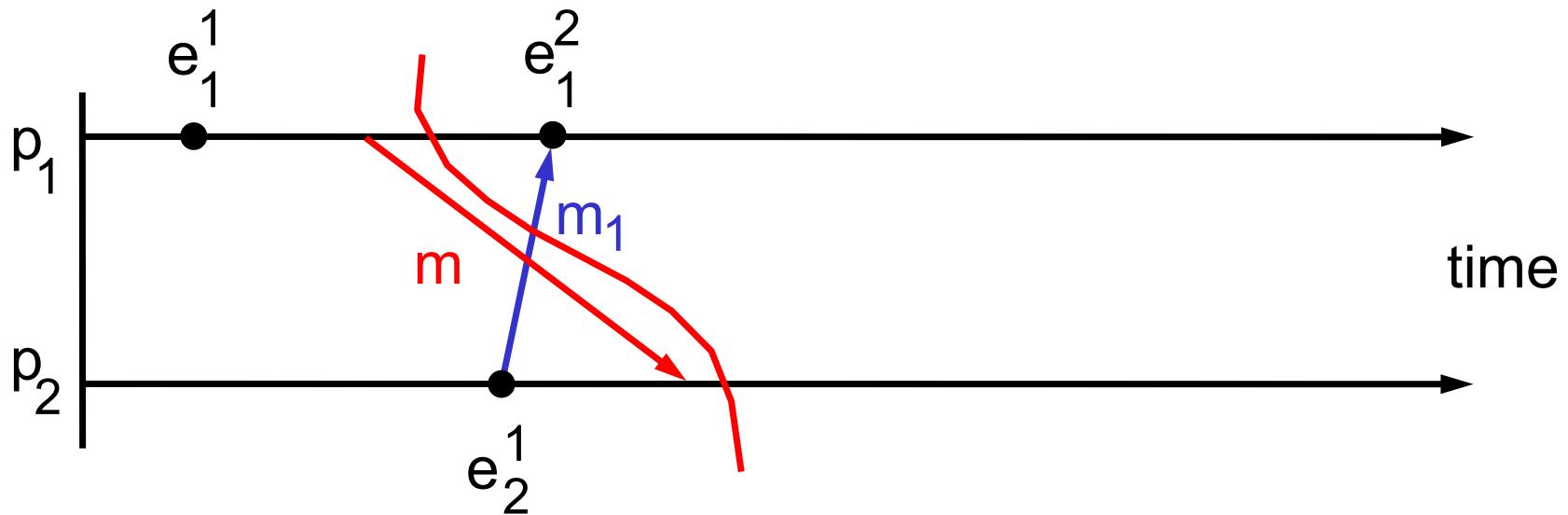
How to Record A Snapshot?

- Revisit example: distributed deadlock detection



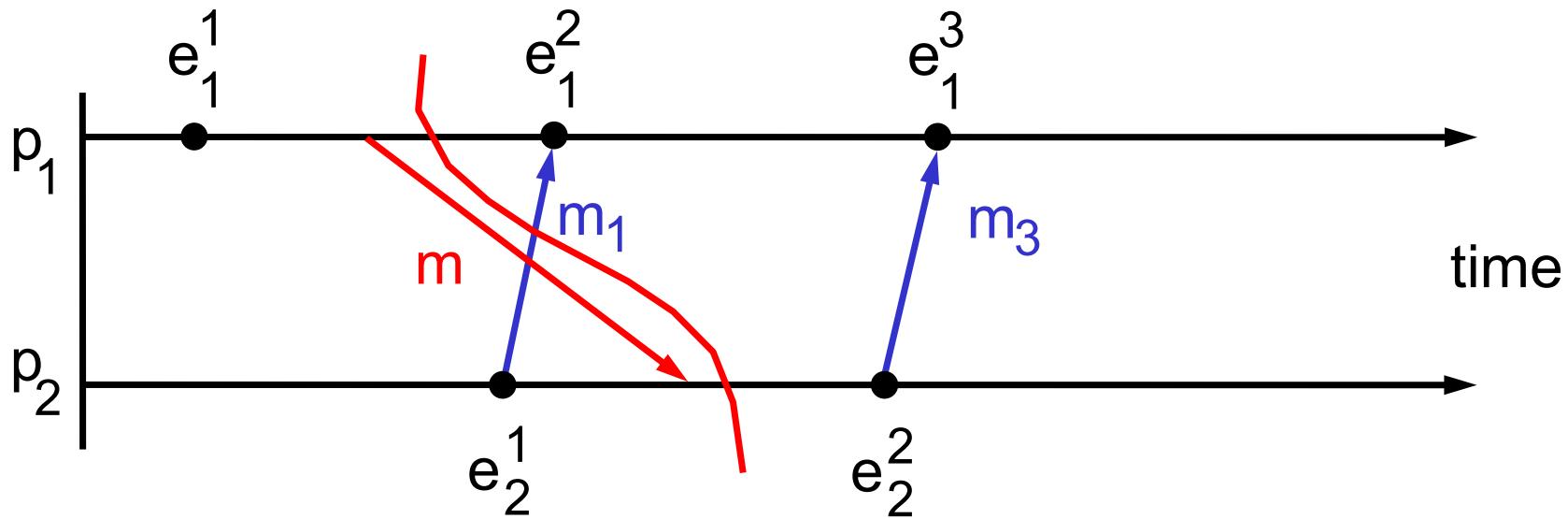
- If there is a message in transition at this time, there may not be a deadlock
 - For example, p_1 is a server and p_2 is a client, p_1 is waiting for the request, p_2 just sends out a request and is waiting for the reply
 - So, we must include the state of communication channels as well as the state of processes in the snapshot

How to Record A Snapshot?



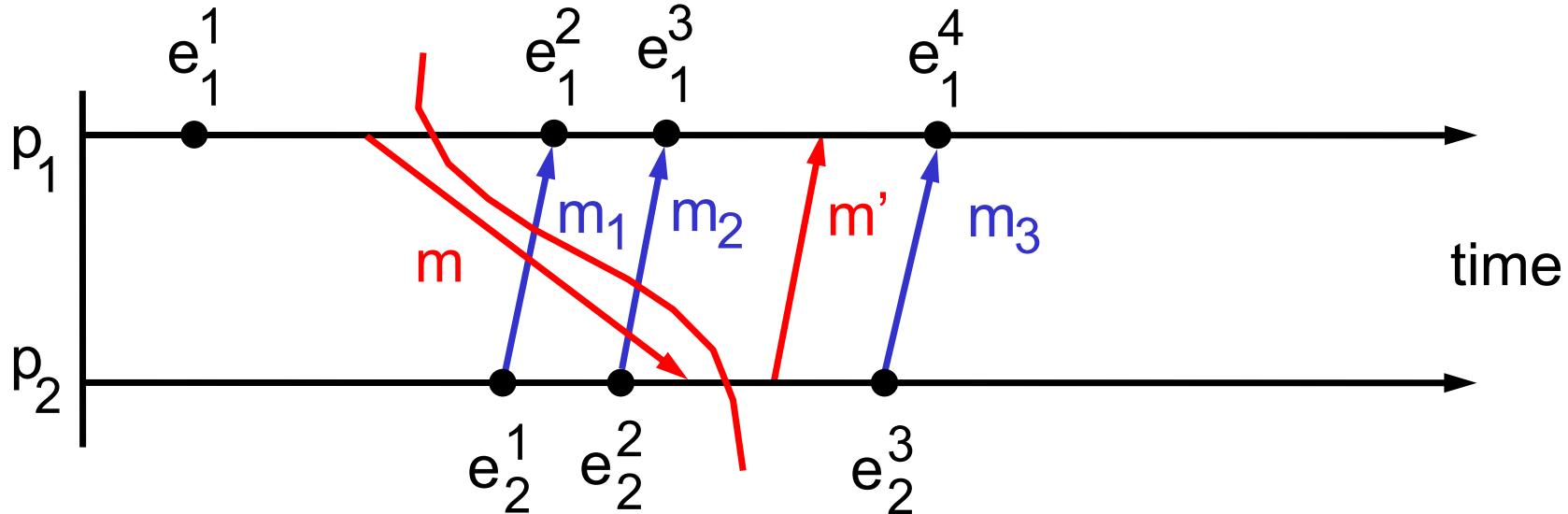
- What if p_2 also sends a message to p_1 ?
- We also need to record the state of the channel
- So, p_1 needs to record the messages (e.g., m_1) from p_2 after it sends m to p_2

How to Record A Snapshot?



- But how does p_1 determine when to stop recording?
- Should p_1 record m_3 as well? – No

How to Record A Snapshot?



- On receiving m , before sending any other messages, p_2 sends a marker message m' to p_1 who uses it to finish recording channel states
- p_1 should record all messages between sending of m and receipt of m' only (i.e., m_1 and m_2)
- Now, let's extend this idea to N processes

Chandy and Lamport Algorithm

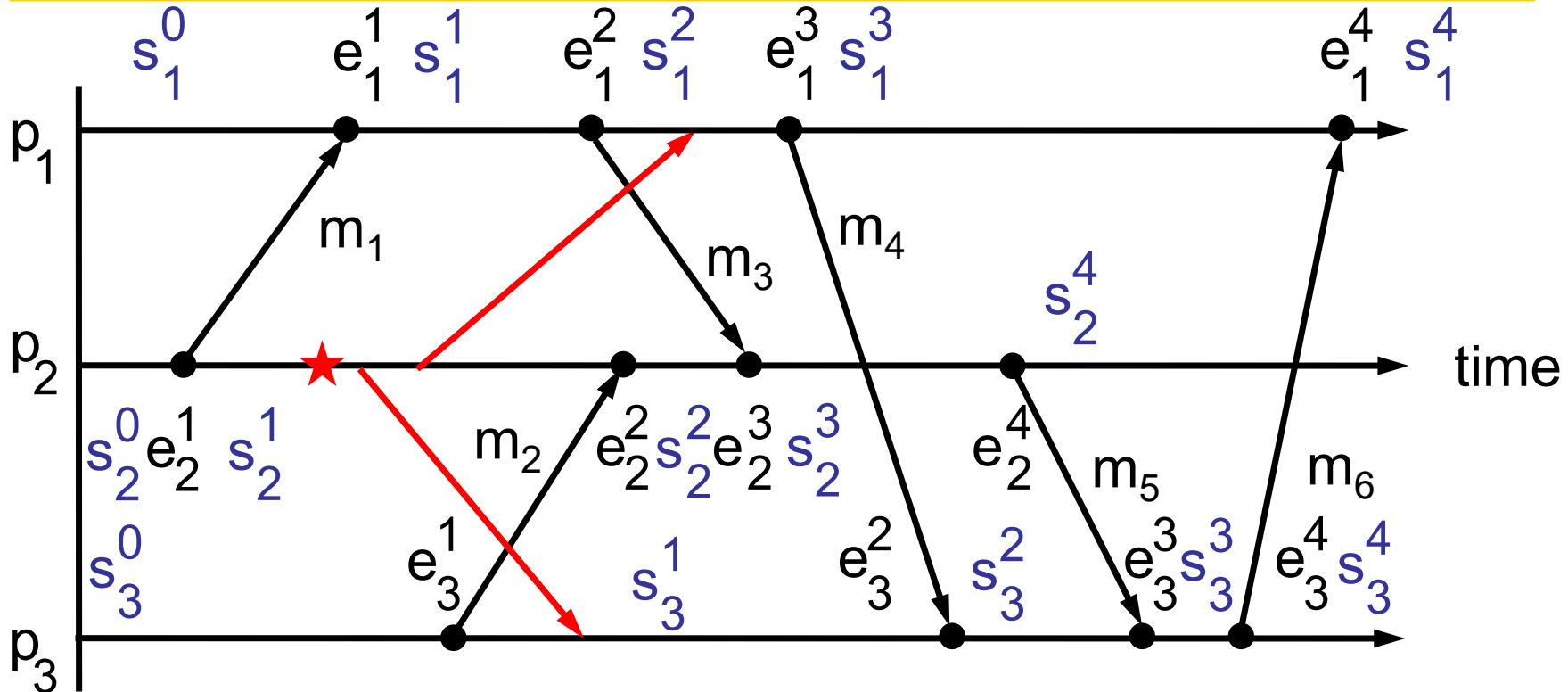
- Objective: to record a snapshot of a distributed system during execution
 - Any process may initiate a snapshot recording at any time
- Assumptions
 - Processes are connected to each other through unidirectional point-to-point channels
 - N processes $\rightarrow N(N - 1)$ channels
 - For each process, we shall distinguish its incoming channels and outgoing channels
 - Neither channels nor processes fail
 - Reliable communication (validity and integrity)
 - FIFO-ordered message delivery on each channel

Chandy and Lamport Algorithm

- Approach: **use marker messages**
 - As a prompt for the receiver to save its own local state if it has not already done so
 - As a means of determining which messages to include in the channel state

Example

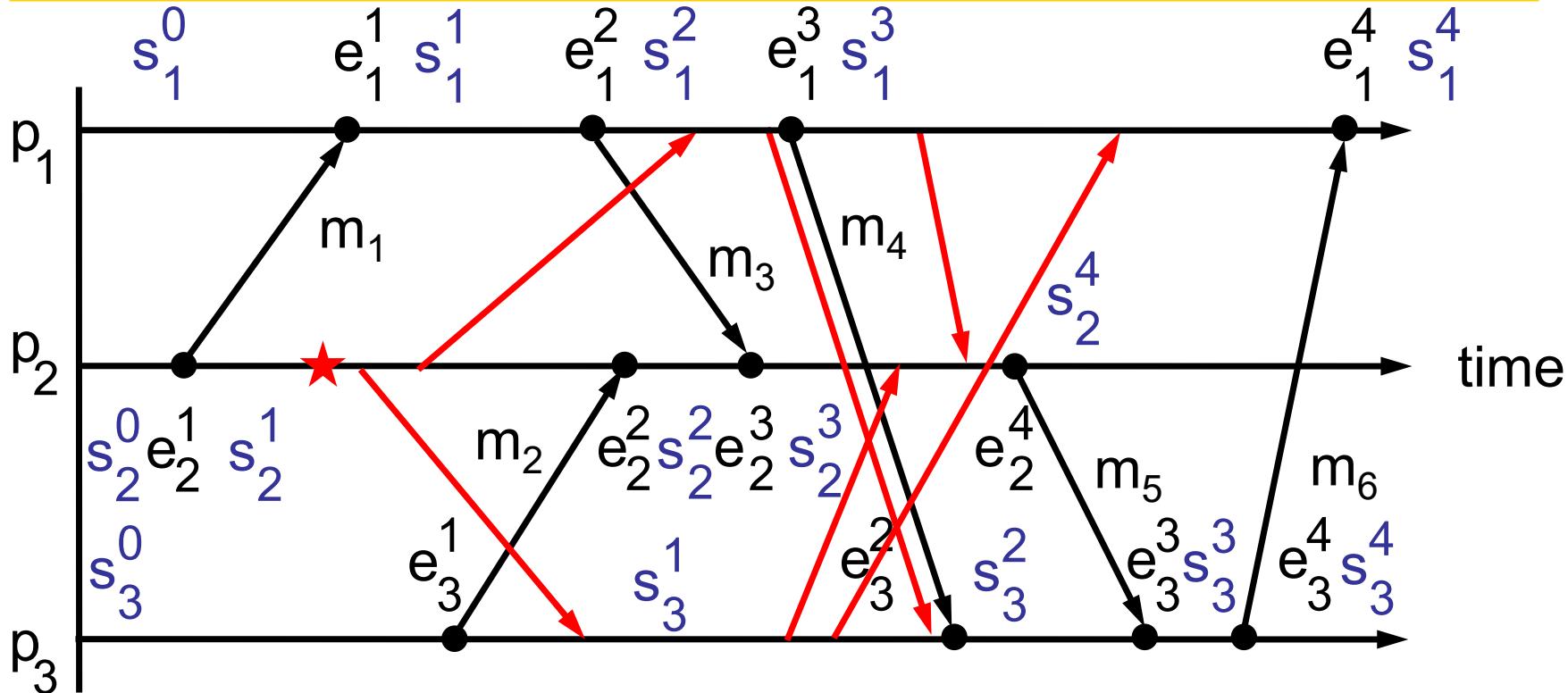
→ marker message



- p_2 initiates a snapshot recording at ★ between e_2^1 and e_2^2
- The initiating process **sends a marker message** along each outgoing channel to all other processes **before it sends any other application message** along that channel

Example

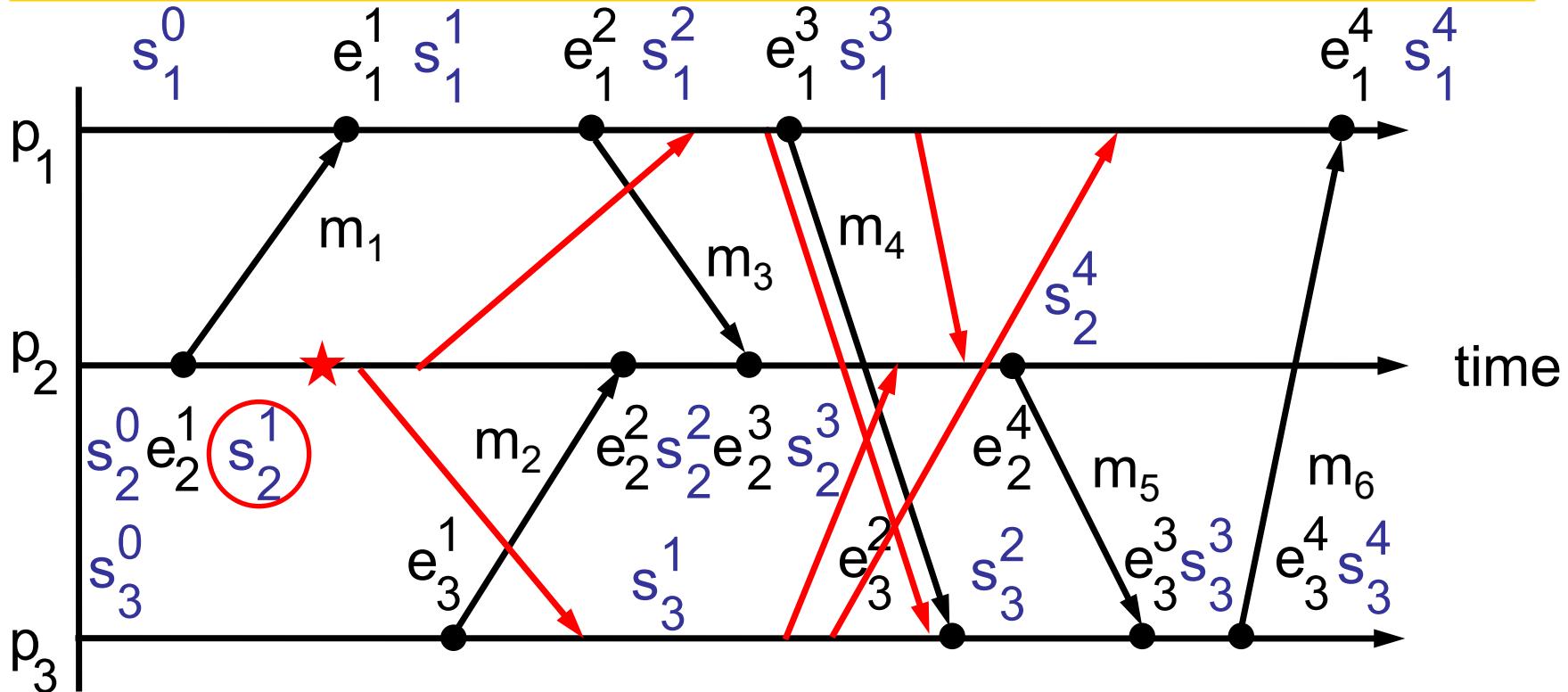
→ marker message



- On receiving the first marker message, each non-initiating process **sends a marker message** along each outgoing channel to all other processes **before it sends any other application message** along that channel

Example

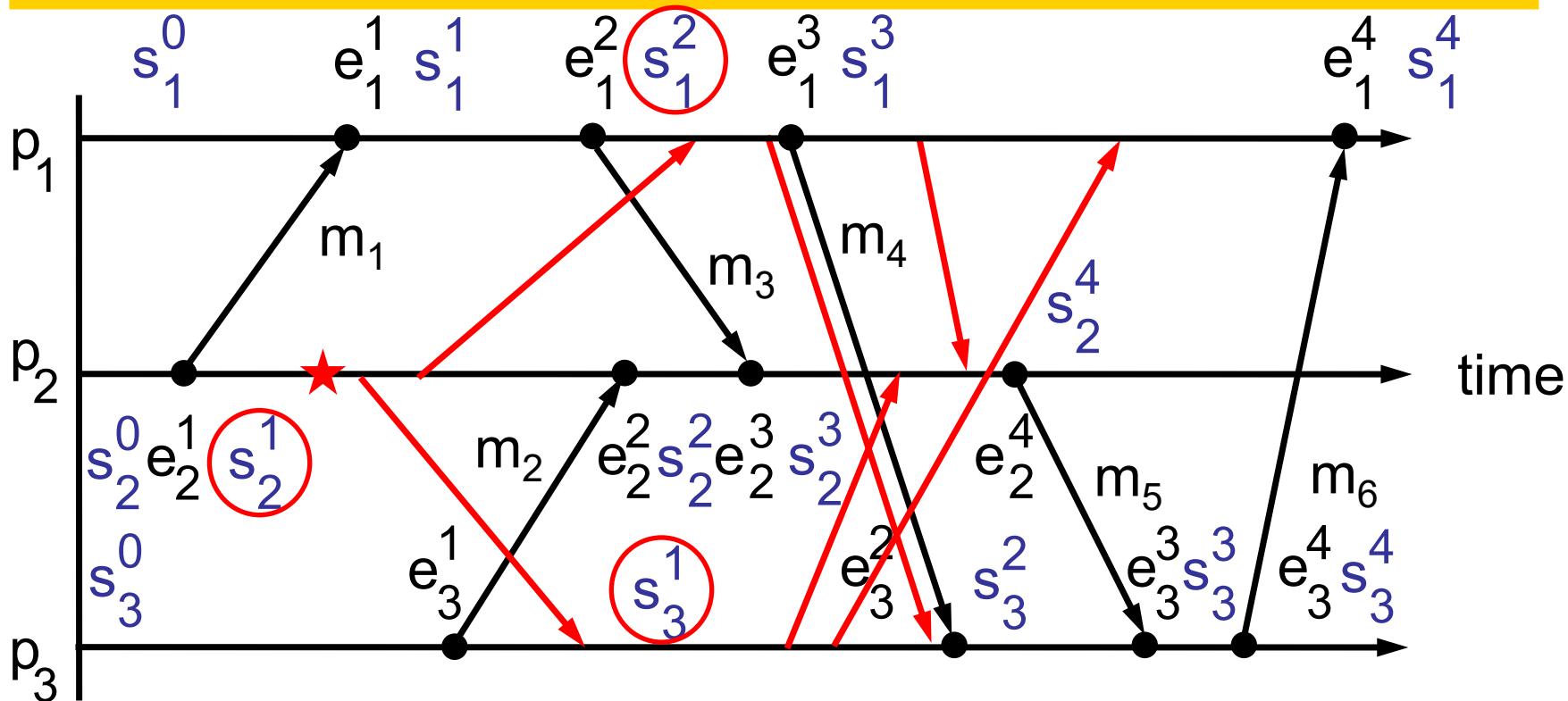
→ marker message



- p_2 initiates a snapshot recording at ★ between e_2^1 and e_2^2
- The initiating process records its process state before sending out the marker messages

Example

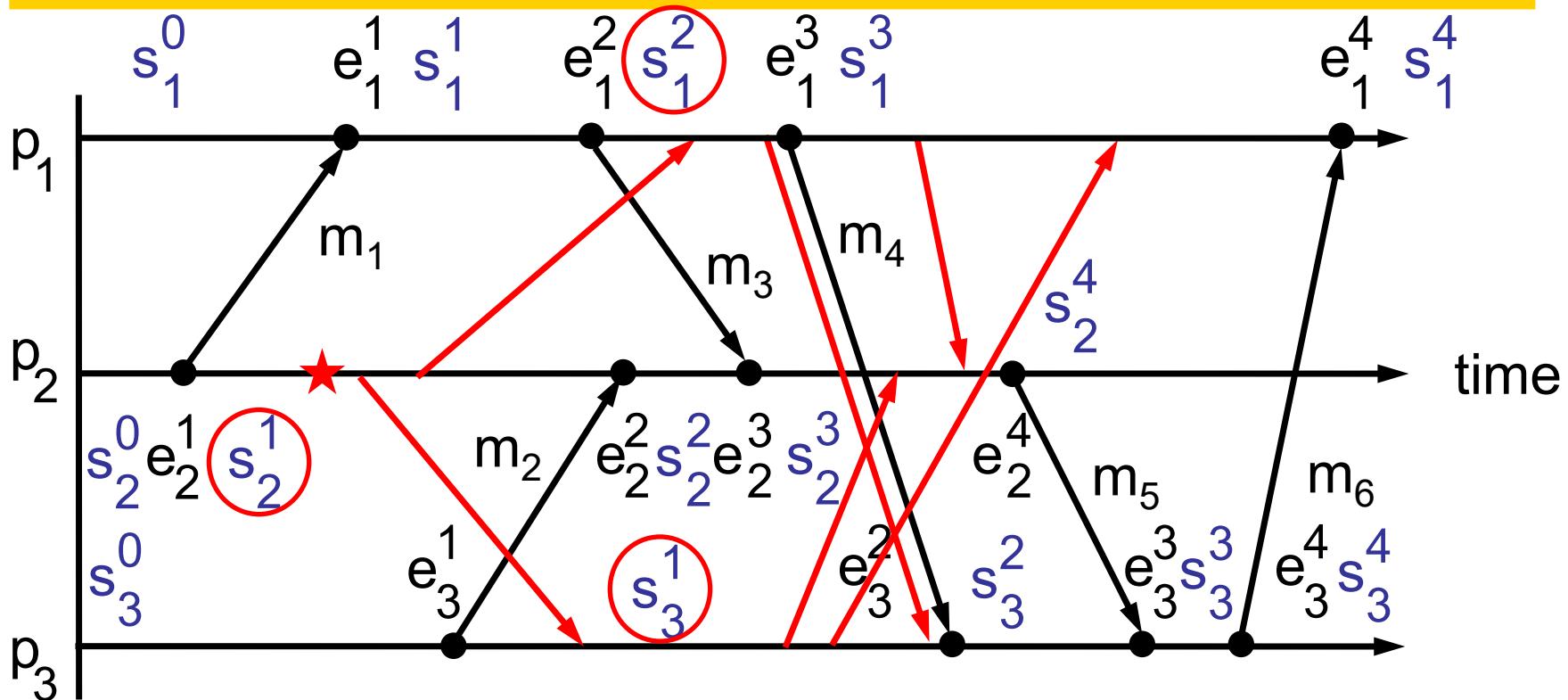
marker message →



- p_2 initiates a snapshot recording at ★ between e_2^1 and e_2^2
- Each non-initiating process records its process state on receiving the first marker message

Example

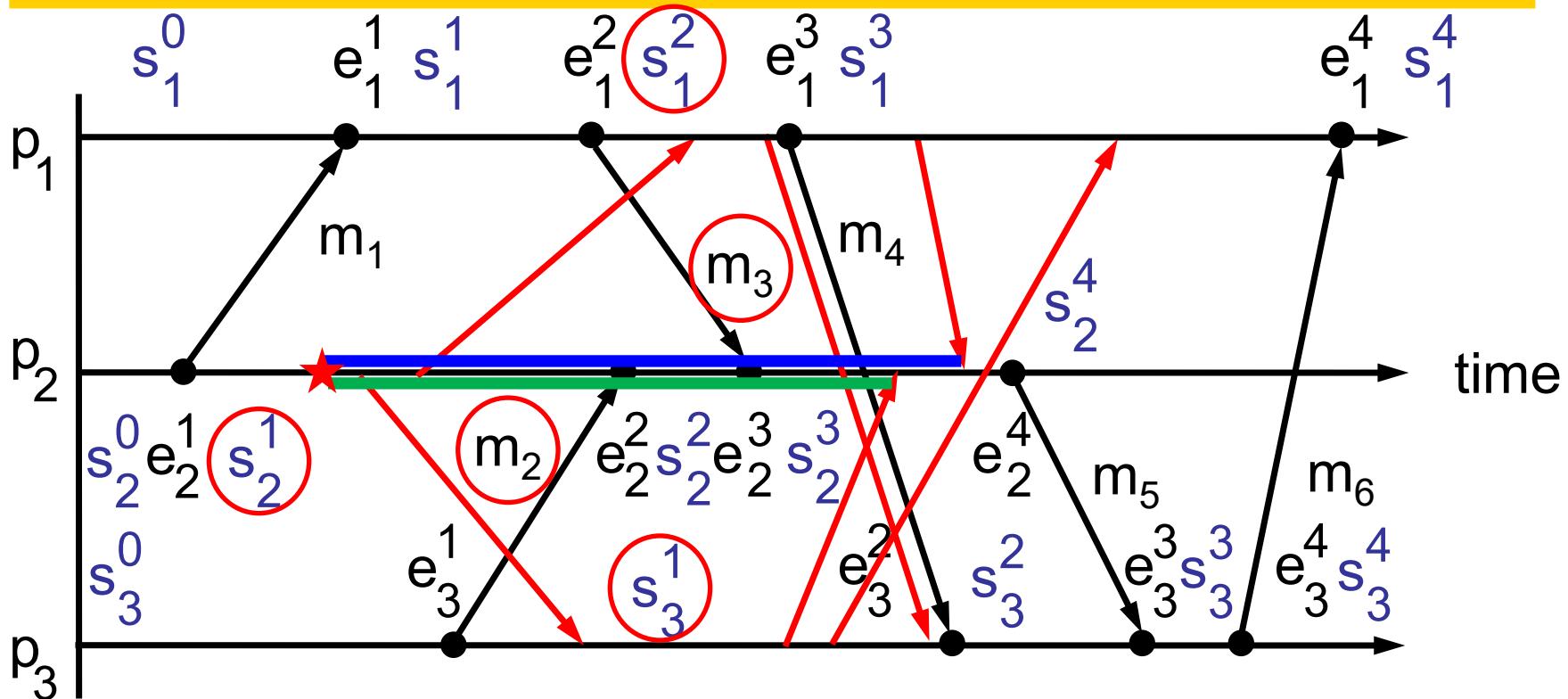
marker message →



- Denote the unidirectional channel from process p_i to process p_j by $c_{i \rightarrow j}$
- Channel $c_{i \rightarrow j}$'s state is recorded by process p_j

Example

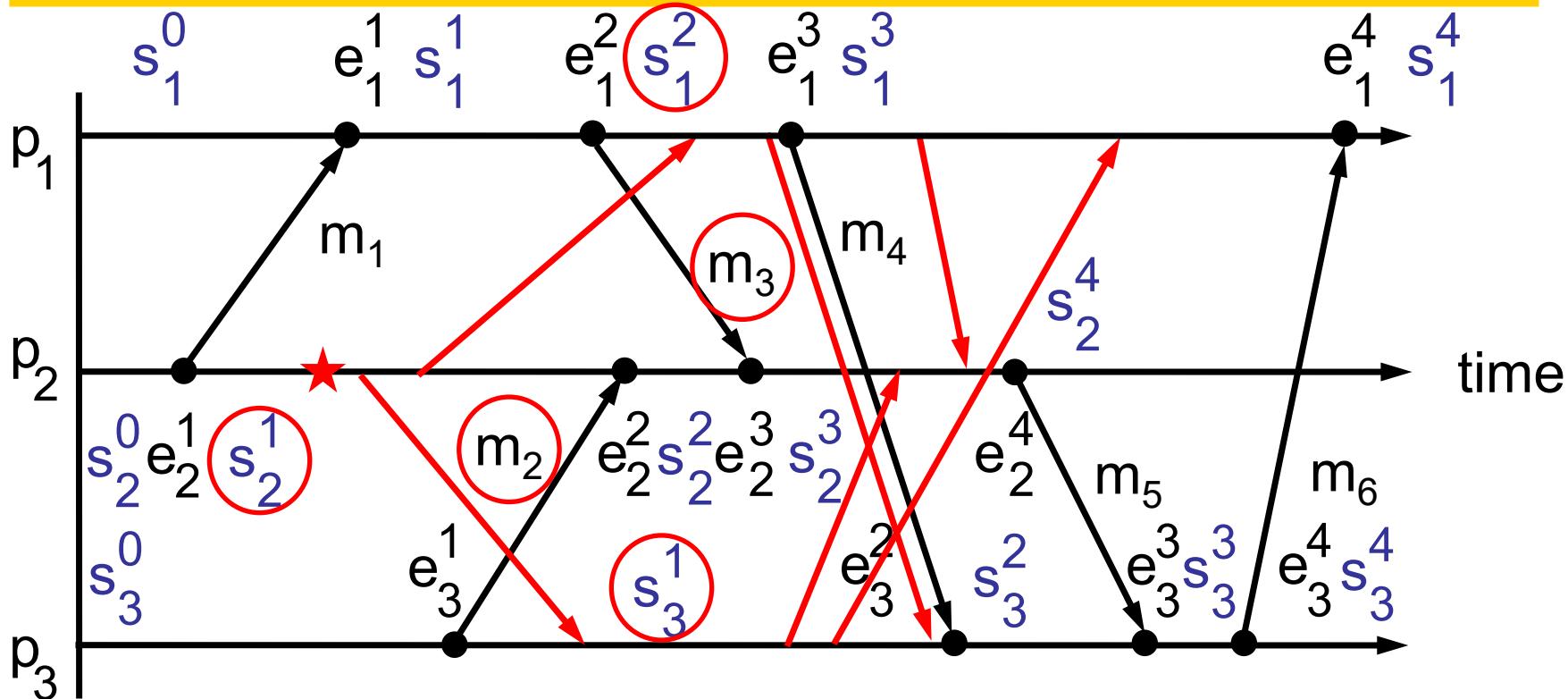
→ marker message



- If p_j is the initiating process, p_j records the messages arriving on channel $c_{i \rightarrow j}$ between p_j initiates the snapshot recording and receives the marker message from p_i
- So, $c_{1 \rightarrow 2}$ state = $\langle m_3 \rangle$, $c_{3 \rightarrow 2}$ state = $\langle m_2 \rangle$

Example

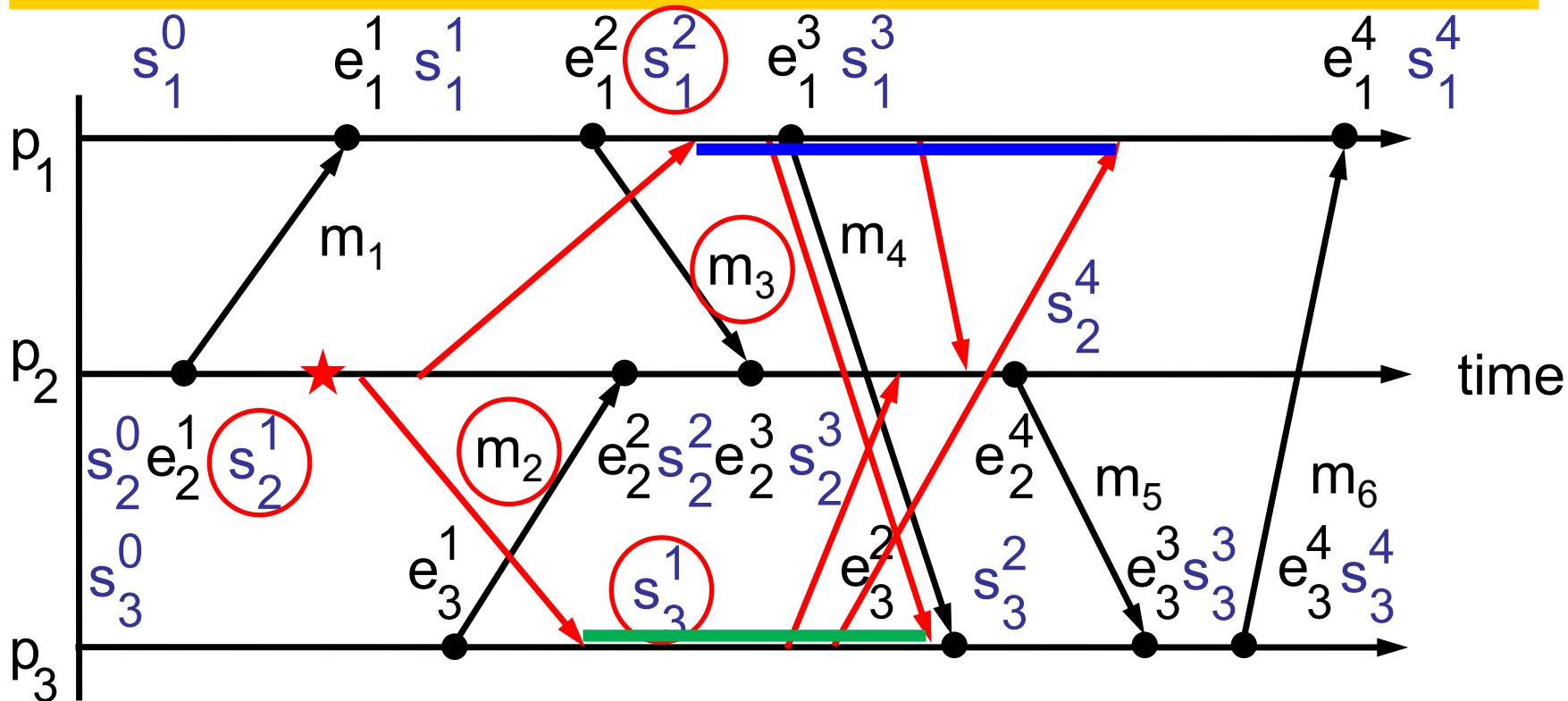
→ marker message



- If p_j is a non-initiating process and $c_{i \rightarrow j}$ is the channel through which p_j receives its first marker message, $c_{i \rightarrow j}$'s state is recorded as an empty set
- So, $c_{2 \rightarrow 1}$ state = $\langle \rangle$, $c_{2 \rightarrow 3}$ state = $\langle \rangle$

Example

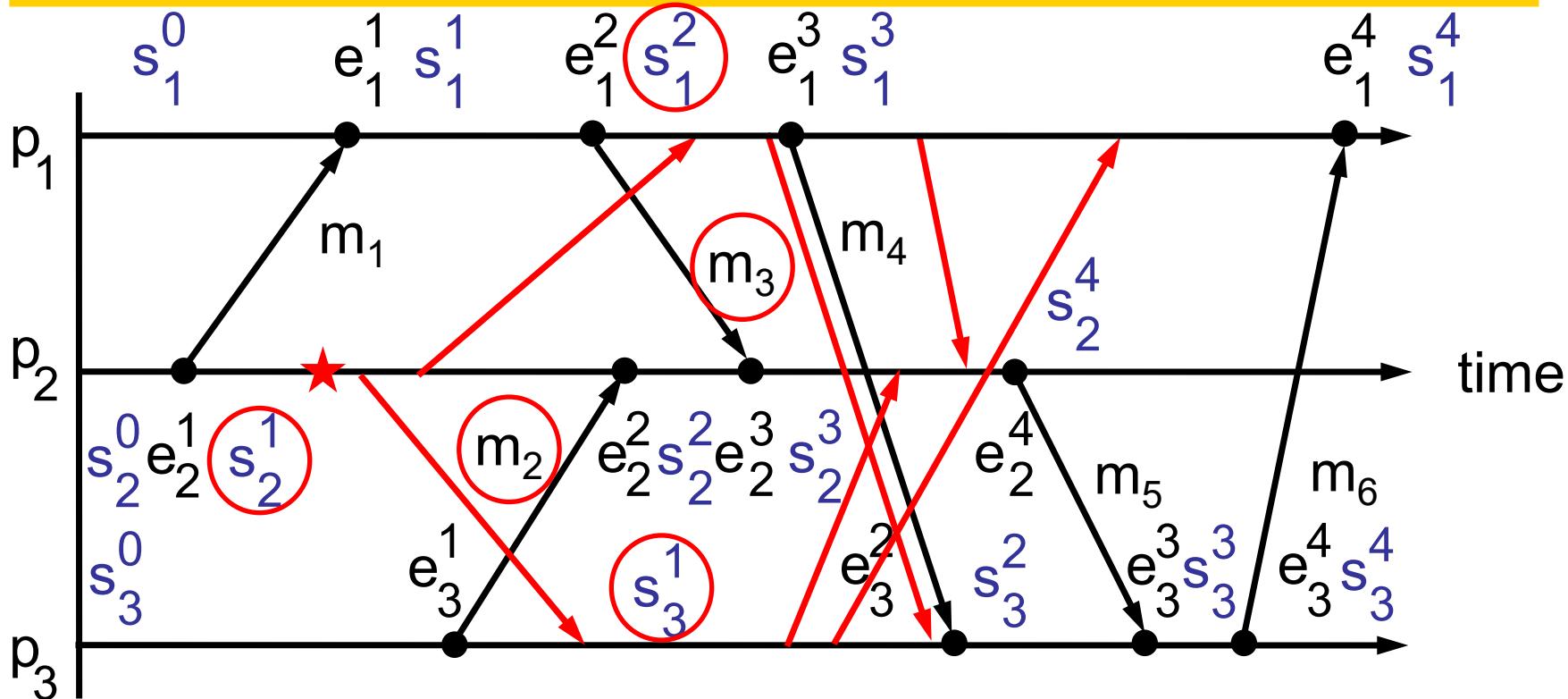
marker message →



- If p_j is a non-initiating process and $c_{i \rightarrow j}$ is not the channel through which p_j receives its first marker message, p_j records the messages arriving on channel $c_{i \rightarrow j}$ between p_j receives the first marker message and the marker message from p_i
- So, $c_{1 \rightarrow 3}$ state = $\langle \rangle$, $c_{3 \rightarrow 1}$ state = $\langle \rangle$

Example

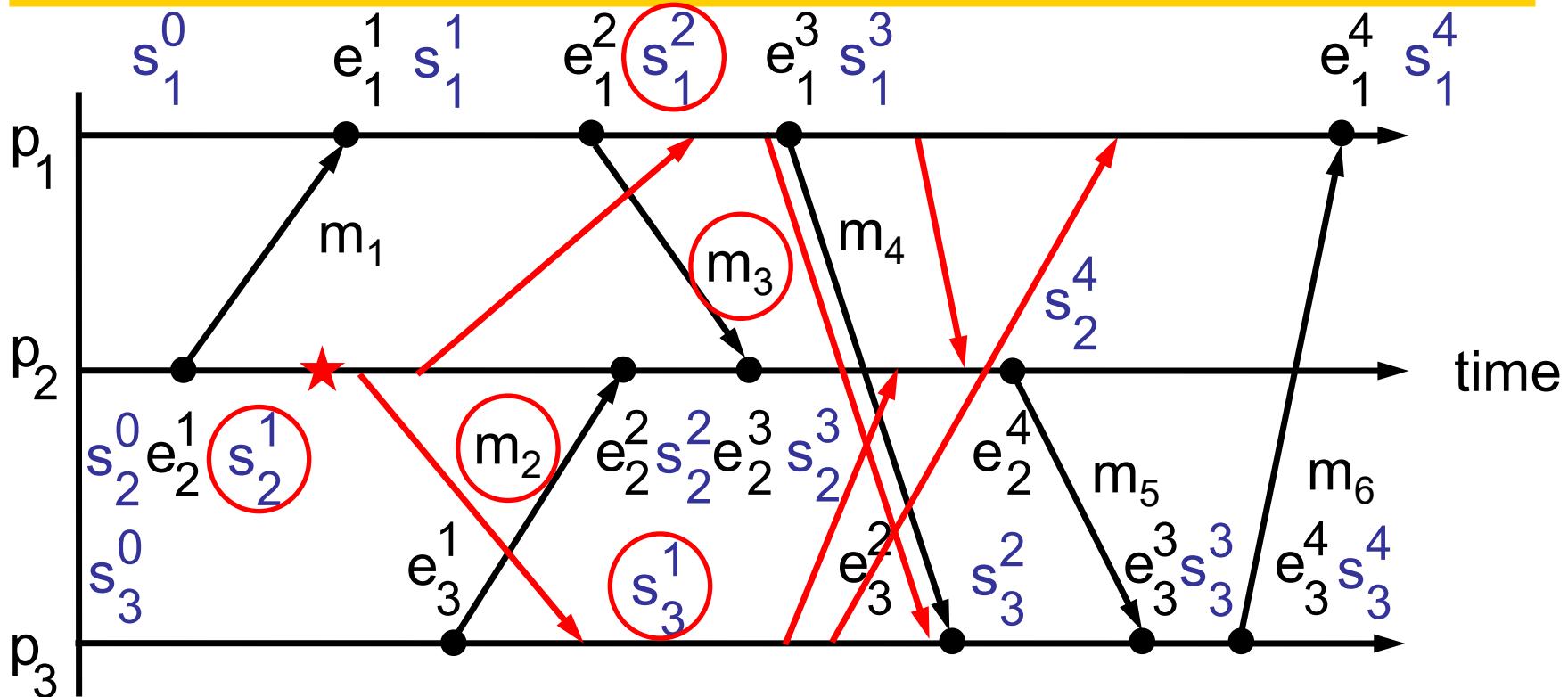
marker message →



- Each process completes all recording activities when it has received a marker message from all other processes
- Then, it may send the recorded states (local process state + channel states) to the initiating process

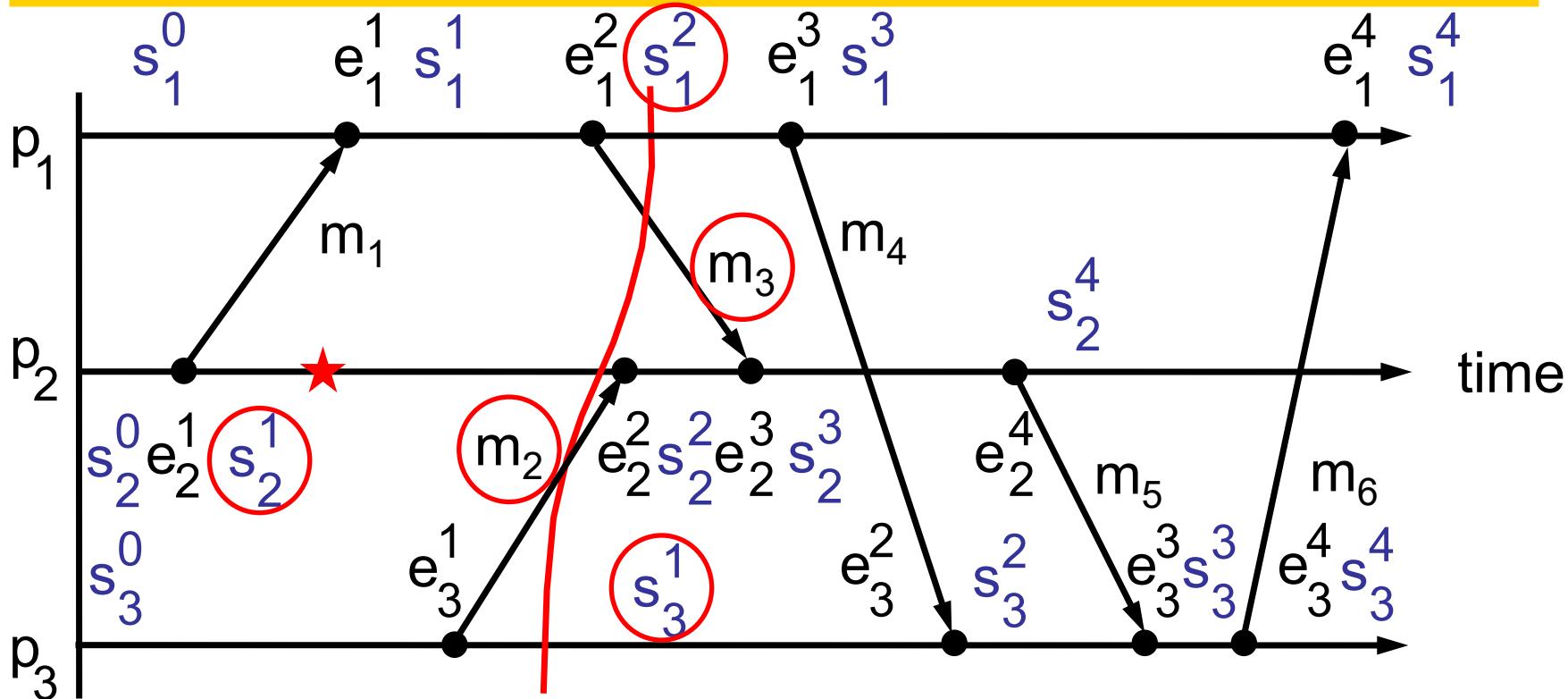
Example

→ marker message



- Final recorded snapshot: p_1 state = s_1^2 , p_2 state = s_2^1 , p_3 state = s_3^1 , $c_{1 \rightarrow 2}$ state = $\langle m_3 \rangle$, $c_{2 \rightarrow 1}$ state = $\langle \rangle$, $c_{1 \rightarrow 3}$ state = $\langle \rangle$, $c_{3 \rightarrow 1}$ state = $\langle \rangle$, $c_{2 \rightarrow 3}$ state = $\langle \rangle$, $c_{3 \rightarrow 2}$ state = $\langle m_2 \rangle$

Example



- Chandy and Lamport algorithm always selects a **consistent cut**
 - Proof omitted

Outline

- Synchronizing Physical Clocks
- Causal Ordering and Logical Clocks
- Global States
- **Distributed Debugging**
- Summary

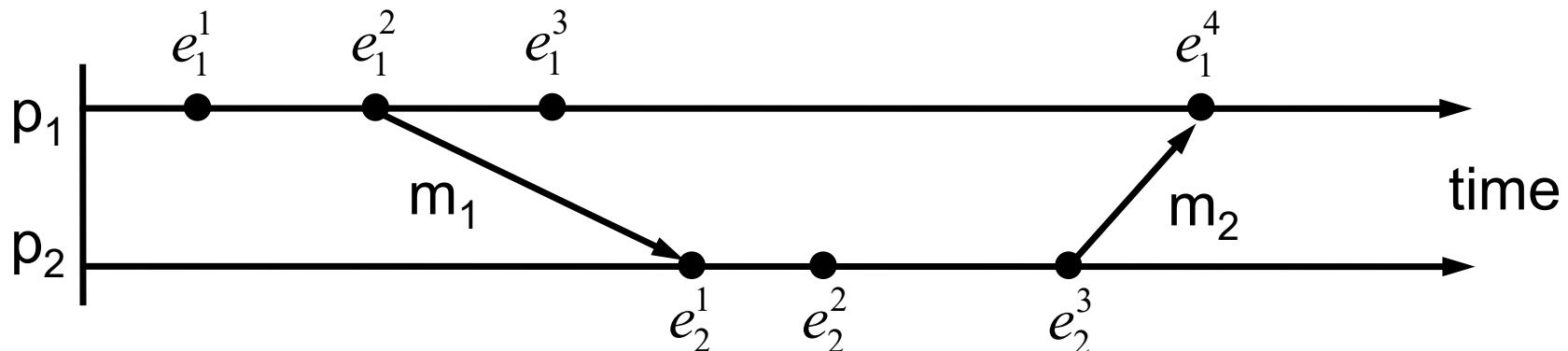
Distributed Debugging

- To establish what occurred during the execution of a distributed system and to check whether they are correct
- Example
 - An application: each process p_i contains a variable x_i
 - The variables change as the application executes
 - They are always required to be within a value δ of one another, i.e., $|x_i - x_j| \leq \delta$ for all i and j
 - How to check whether the constraints were broken during the execution? – must find out the values of variables that occur at the same time and evaluate their relationship

Distributed Debugging

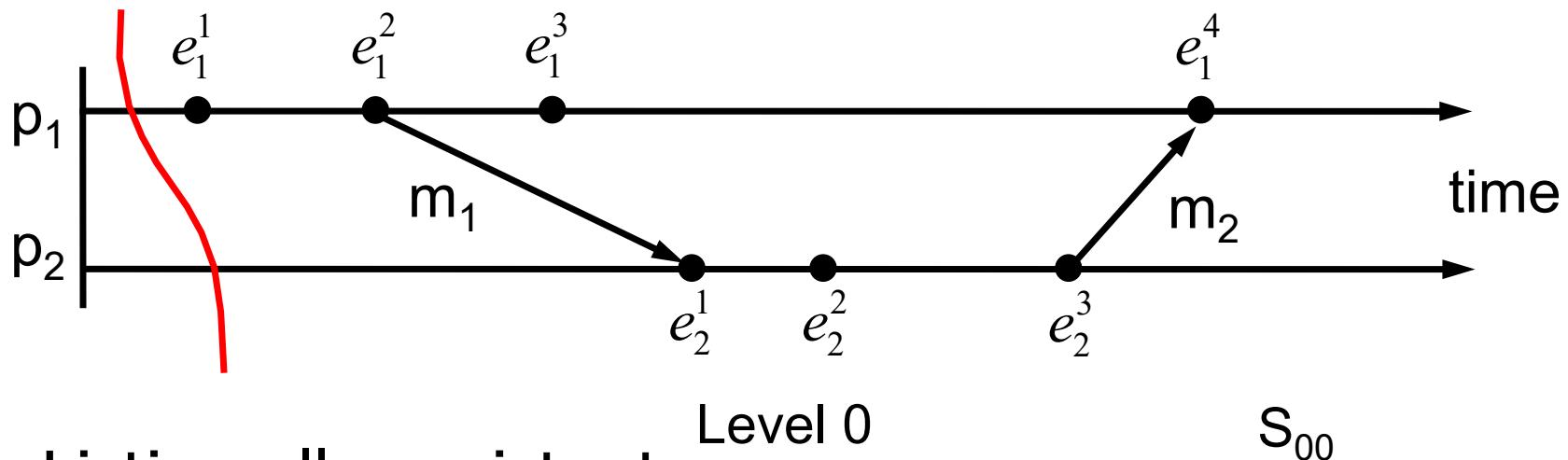
- Without global time, how to check?
- Can we use Chandy and Lamport algorithm to repeatedly record snapshots?
 - High communication overhead
 - Cannot continuously and completely “trace” global states over time
- Our approach: enumerate all consistent global states and check them exhaustively

Lattice of Consistent Global States



- Listing all consistent global states through a lattice
 - Starting from the initial state, advance the global state by one event at a time
 - A global state reachable from state $\langle s_1^{c_1}, s_2^{c_2}, \dots, s_N^{c_N} \rangle$ has the form of $\langle s_1^{c_1}, s_2^{c_2}, \dots, s_i^{c_i+1}, \dots, s_N^{c_N} \rangle$

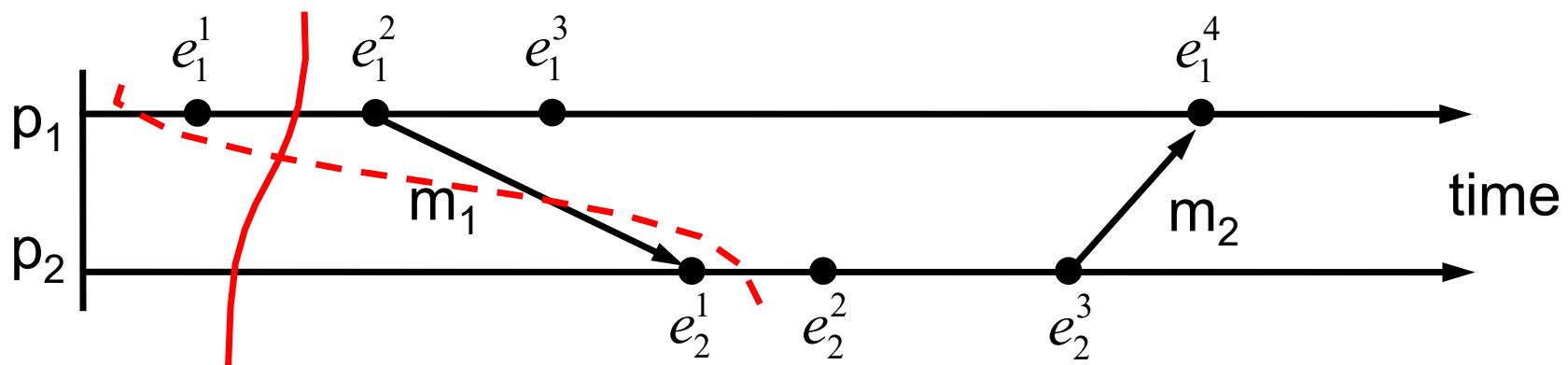
Lattice of Consistent Global States



- Listing all consistent global states through a lattice

- $S_{ij} = \langle s_1^i, s_2^j \rangle$: global state after i events at process p_1 and j events at process p_2
- Initial global state is S_{00}

Lattice of Consistent Global States



- Listing all consistent global states through a lattice

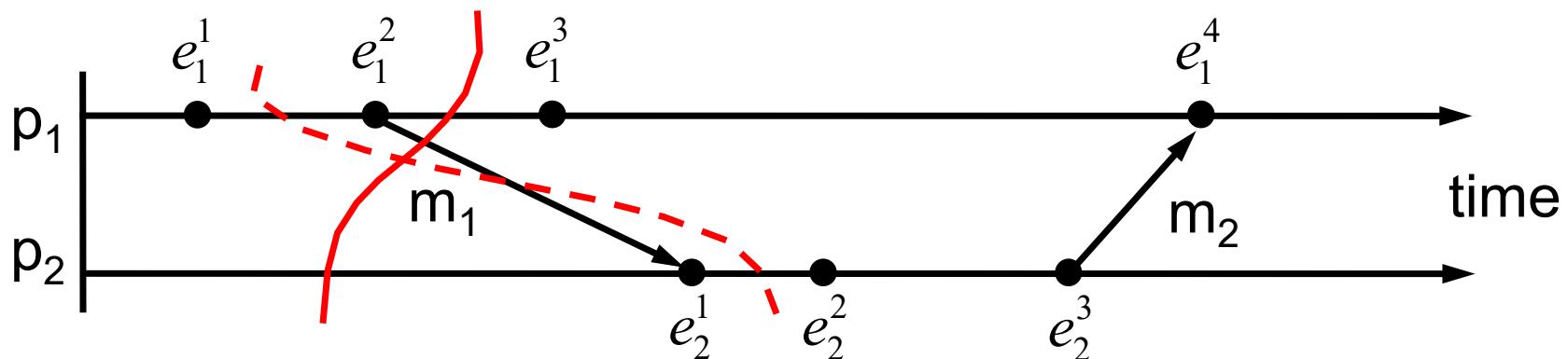
- Global states reachable from S_{00} include S_{10} , S_{01}
- S_{10} is consistent, S_{01} is not consistent

Level 0

1

S_{00}
 S_{10}

Lattice of Consistent Global States

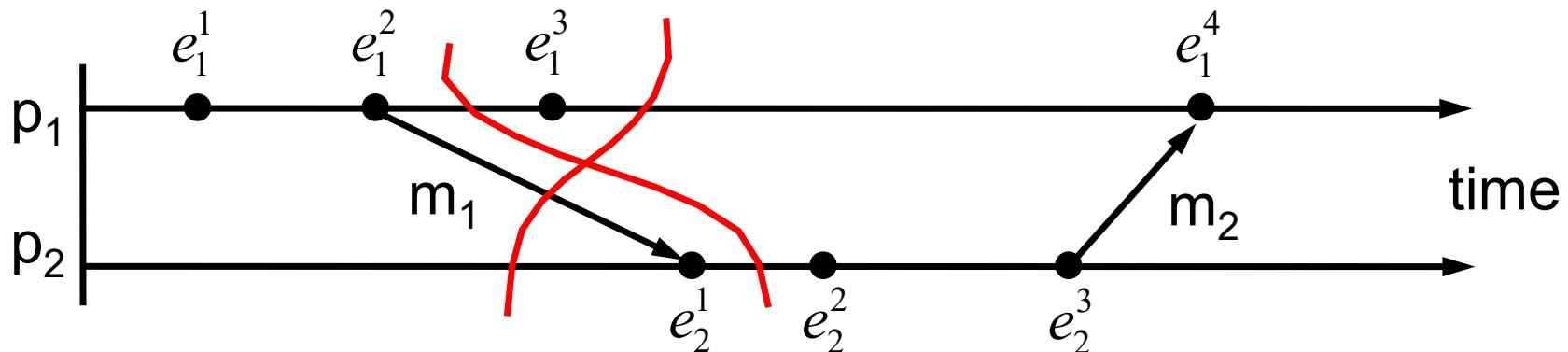


- Listing all consistent global states through a lattice
 - Global states reachable from S_{10} include S_{20}, S_{11}
 - S_{20} is consistent, S_{11} is not consistent

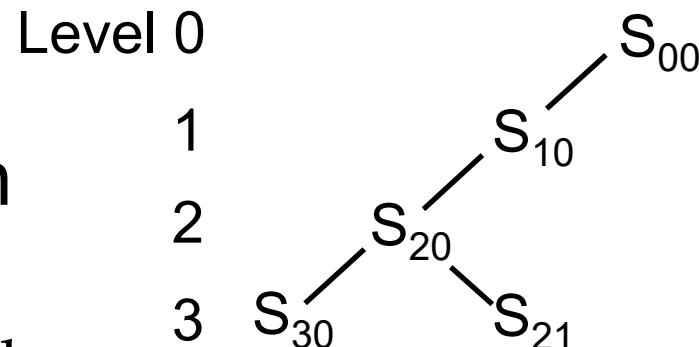
Level 0
1
2

S_{00}
 S_{10}
 S_{20}

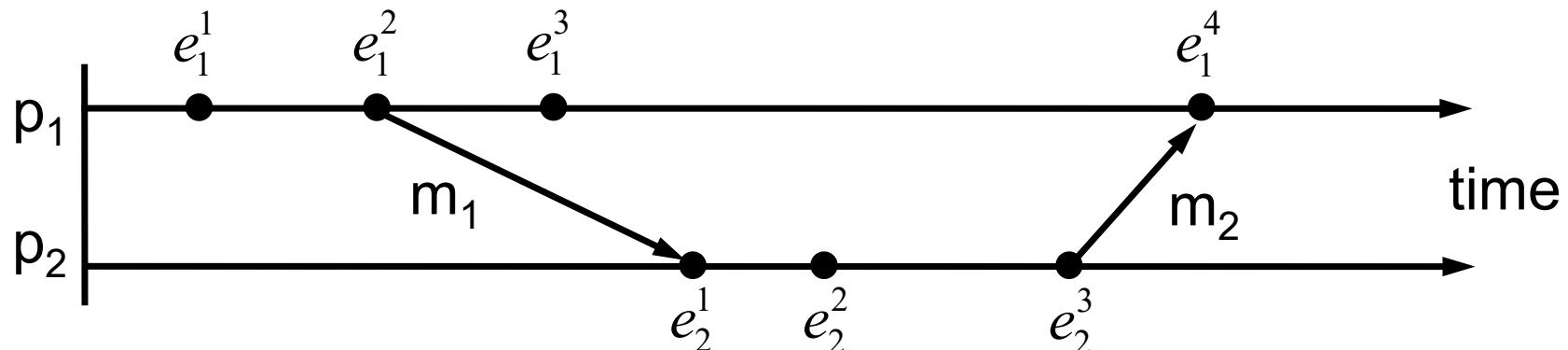
Lattice of Consistent Global States



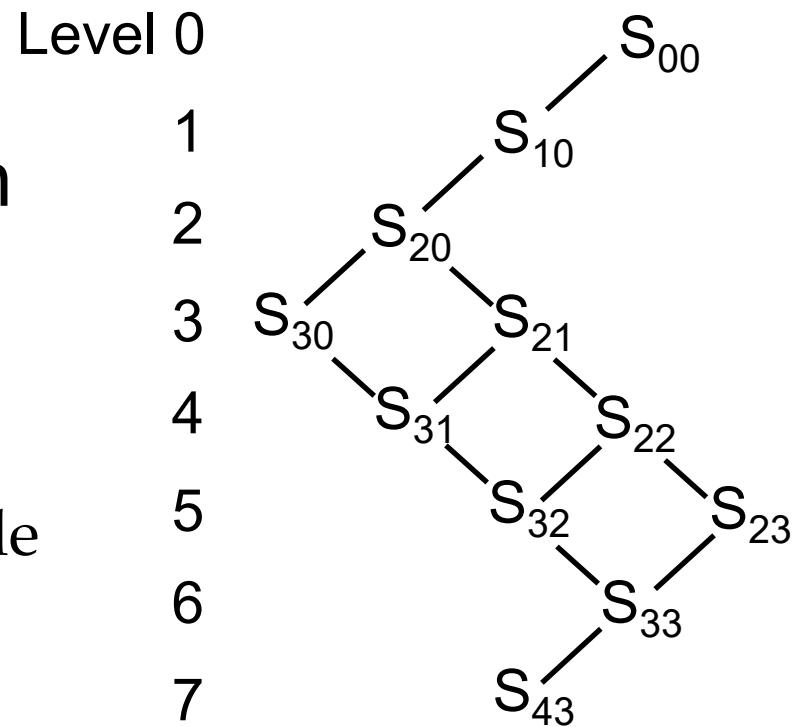
- Listing all consistent global states through a lattice
 - Global states reachable from S_{20} include S_{30}, S_{21}
 - S_{30} and S_{21} are both consistent



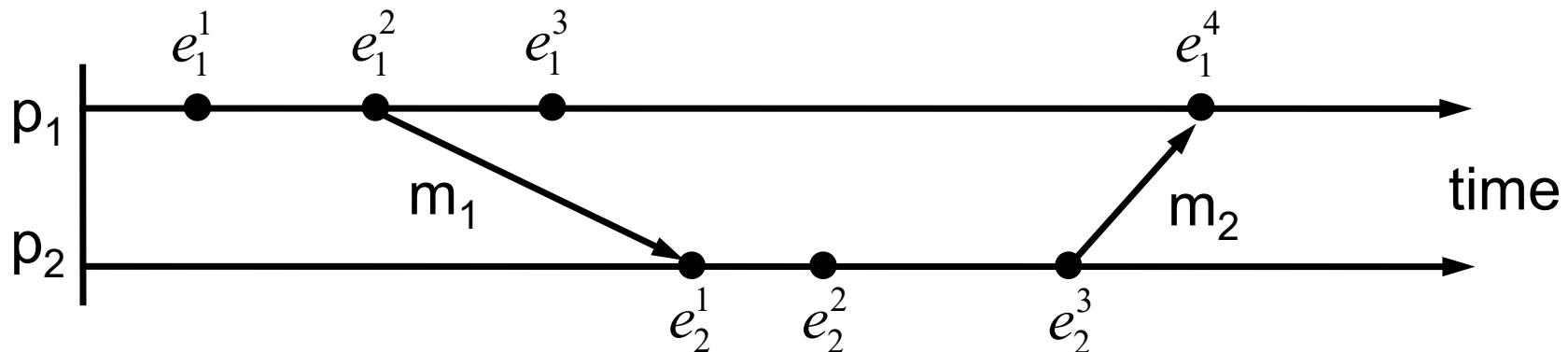
Lattice of Consistent Global States



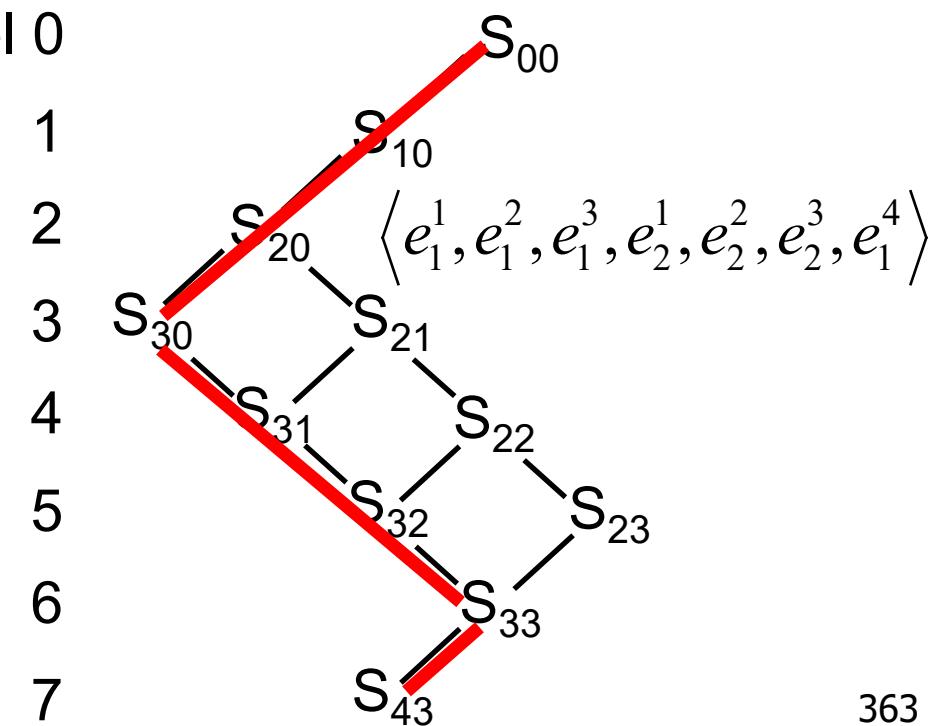
- Listing all consistent global states through a lattice
 - Complete lattice
 - In general, a global state may be reachable from several global states



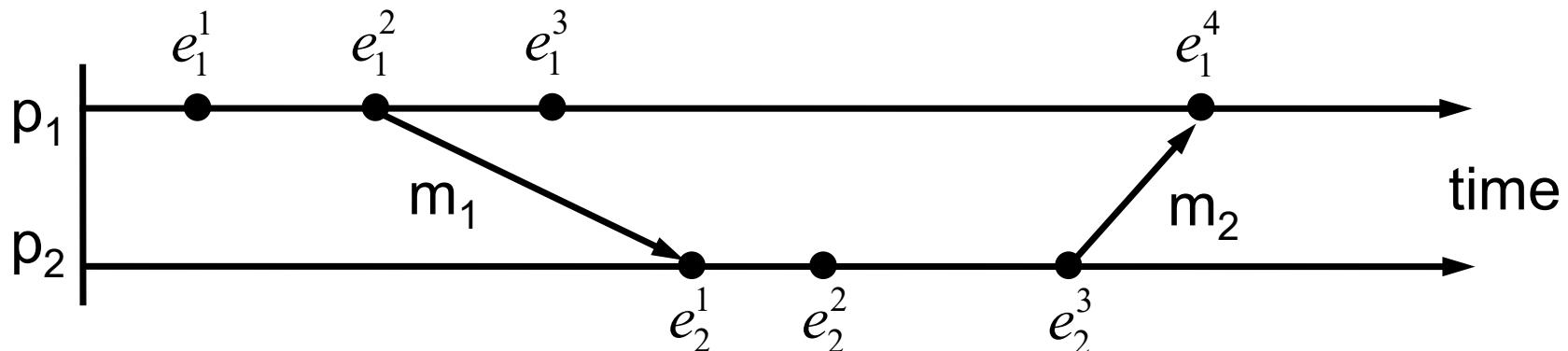
Lattice of Consistent Global States



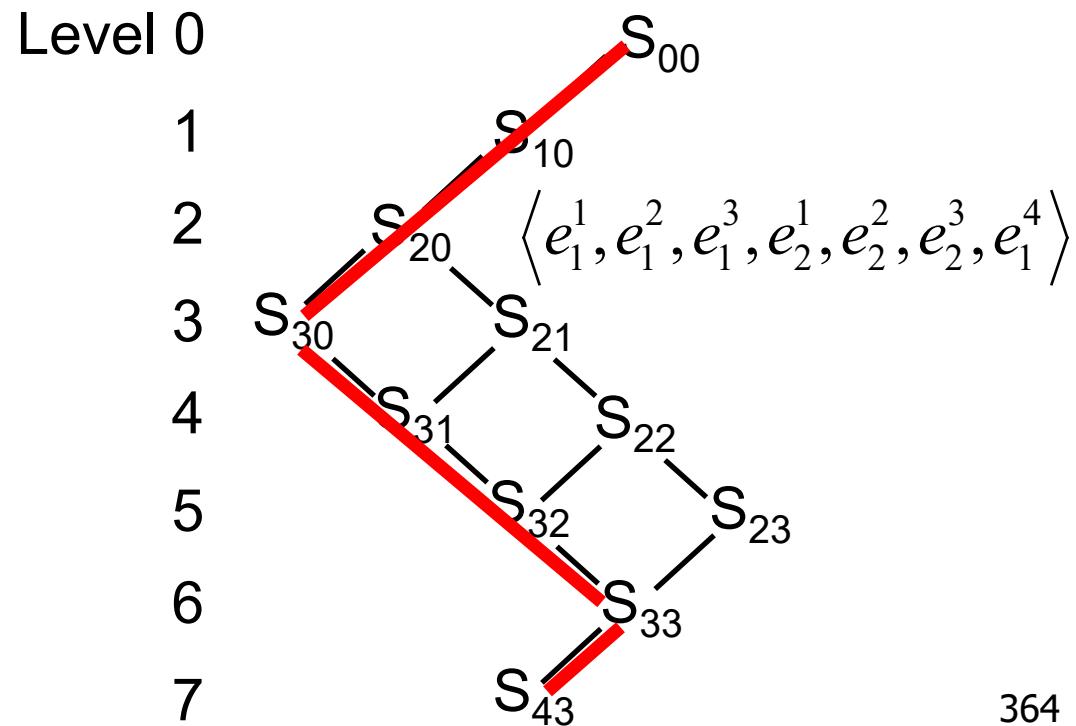
- Each path from the top-level state to the bottom-level state in the lattice represents an ordering of all events that is consistent with causal ordering (i.e., a possible execution of the system)



Lattice of Consistent Global States



- Without global time, we cannot tell which one is the actual execution of the system among all possible executions



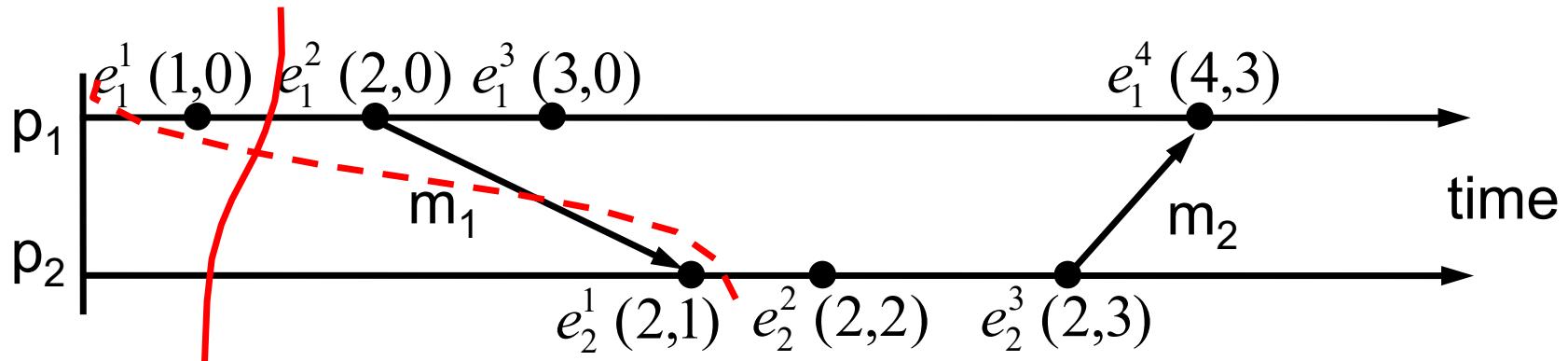
Observing Consistent Global States

- Use an additional monitor process that lies outside the system to observe the execution of the system
- All processes in the system send their states to the monitor process
 - Each process p_i sends its initial state to the monitor process initially
 - Then, when an event $e_i^{c_i}$ occurs at p_i , p_i sends a state message (including its new state $s_i^{c_i}$ and the vector clock timestamp of the event) to the monitor process
- The monitor process assembles consistent global states

Observing Consistent Global States

- In general, how to assemble consistent global states?
 - $\langle s_1^{c_1}, s_2^{c_2}, \dots, s_N^{c_N} \rangle$: a global state drawn from the state messages
 - $V(e_i^{c_i})$: vector clock timestamp of event $e_i^{c_i}$
 - $\langle s_1^{c_1}, s_2^{c_2}, \dots, s_N^{c_N} \rangle$ is a consistent global state if and only if for any i and j
$$V(e_i^{c_i})[i] \geq V(e_j^{c_j})[i]$$
- How to prove it?

Example

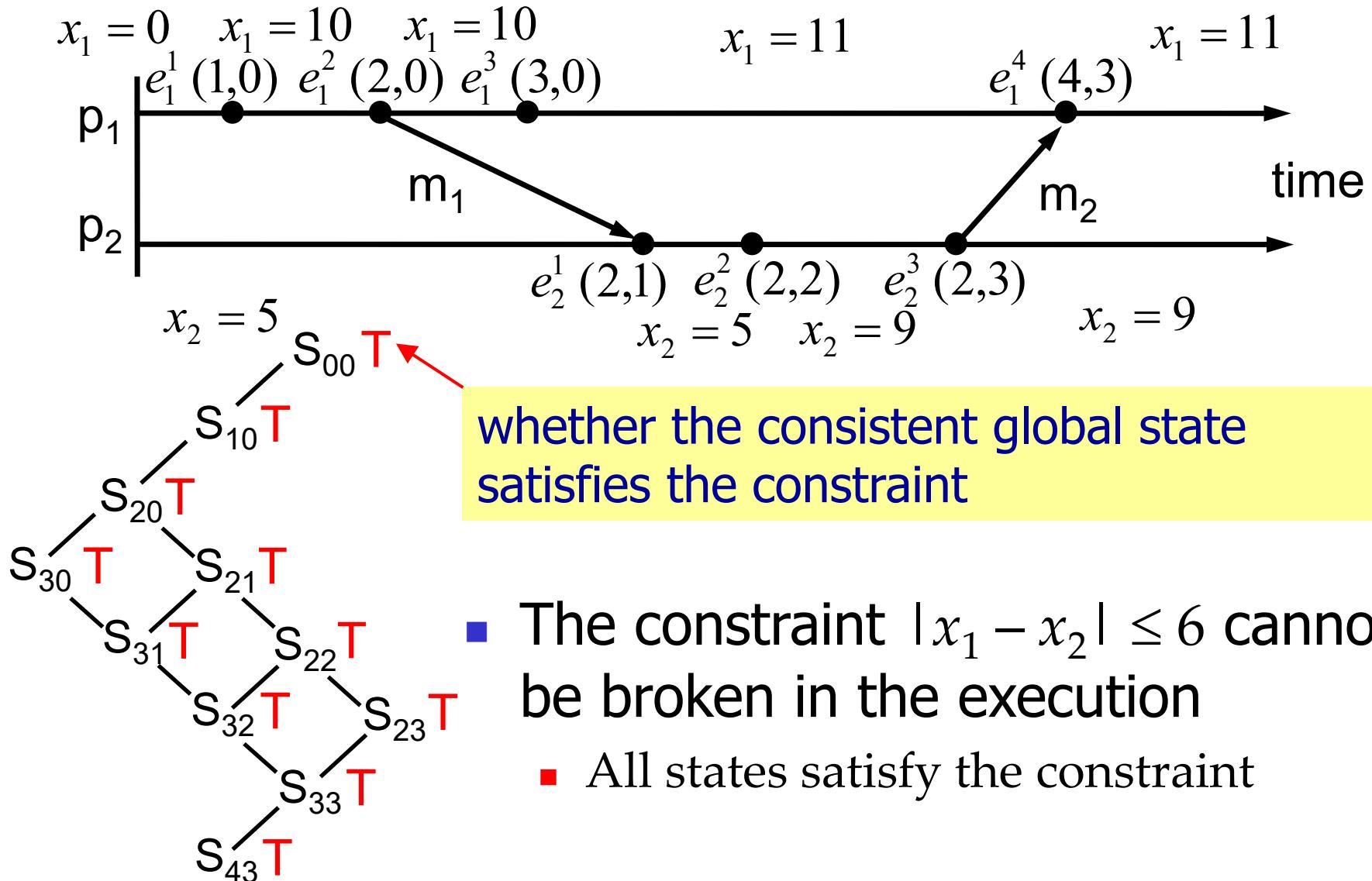


- $S_{ij} = \langle s_1^i, s_2^j \rangle$: global state after i events at process p_1 and j events at process p_2
- S_{10} is a consistent global state
 - $V(e_1^1) = (1,0)$ and $V(e_2^0) = (0,0)$
- S_{01} is not a consistent global state
 - $V(e_1^0) = (0,0)$ and $V(e_2^1) = (2,1)$

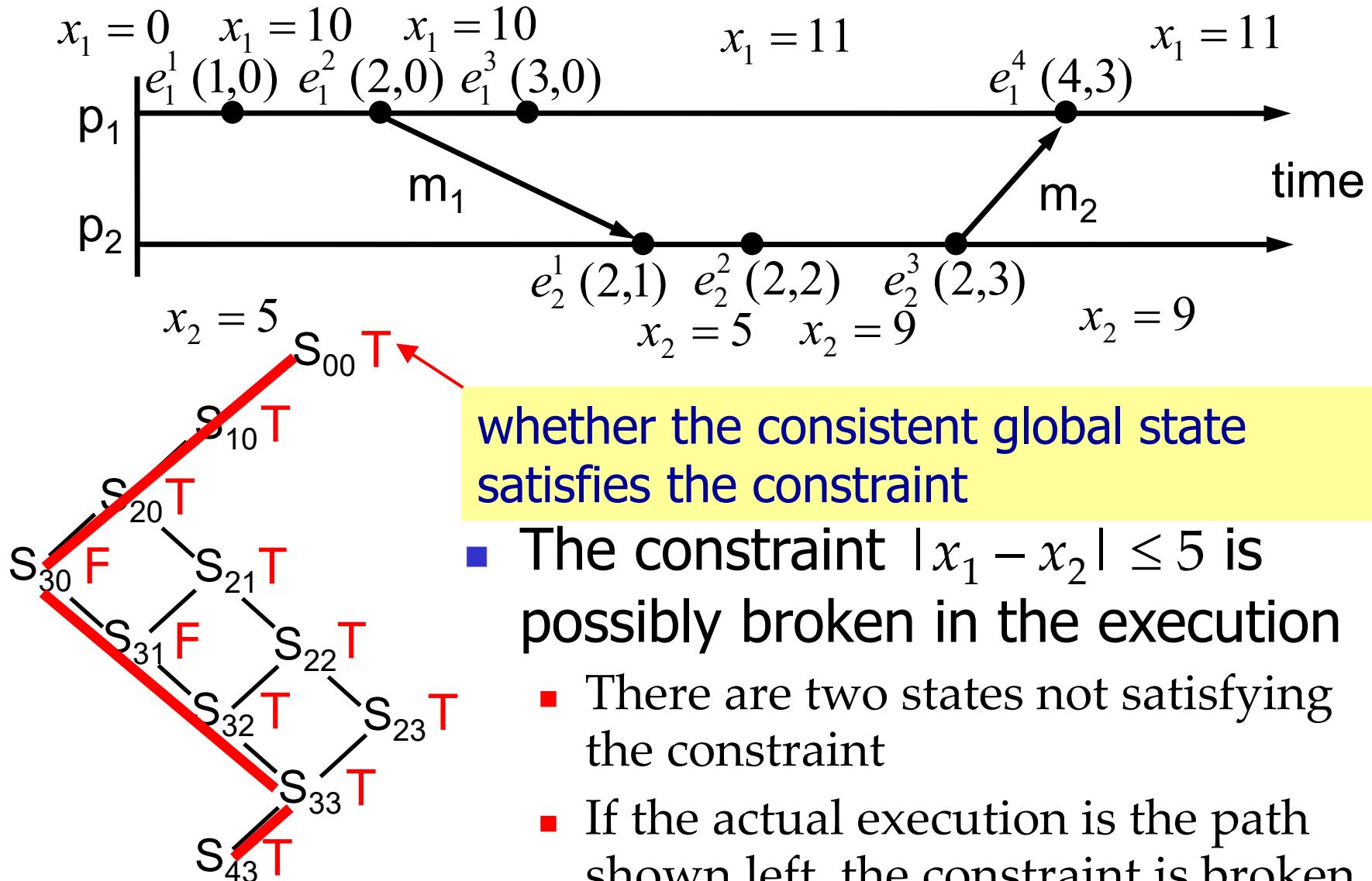
Distributed Debugging

- How to check whether a constraint is broken during execution?
- Construct the lattice of consistent global states
- For each consistent global state, evaluate whether it satisfies the constraint
 - If all states satisfy the constraint, the constraint **cannot be broken** in the execution
 - If there exists a state not satisfying the constraint, the constraint **is possibly broken** in the execution
 - If every path from the top-level state to the bottom-level state passes through at least one state not satisfying the constraint, the constraint **must be broken** in the execution

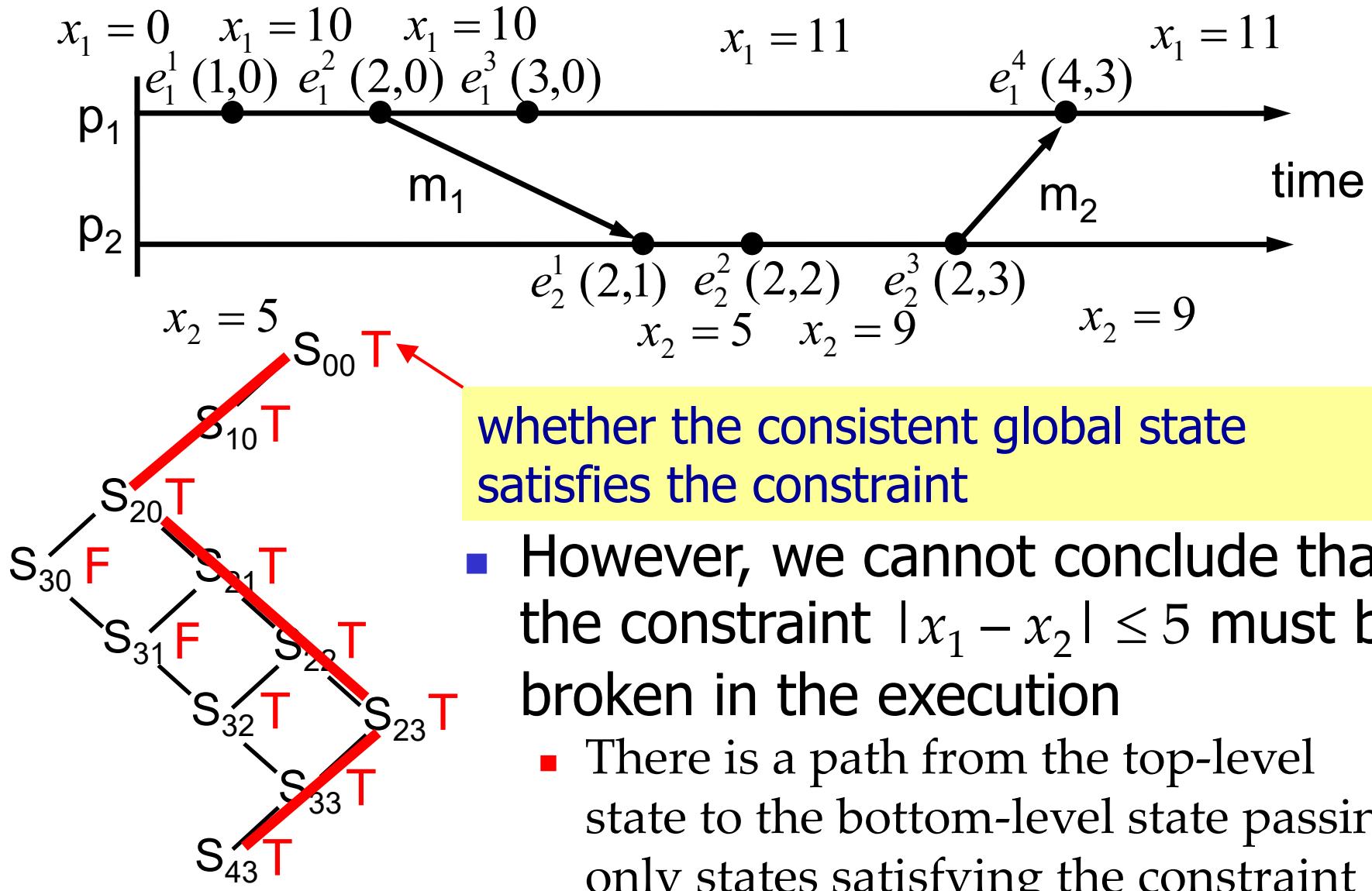
Distributed Debugging



Distributed Debugging



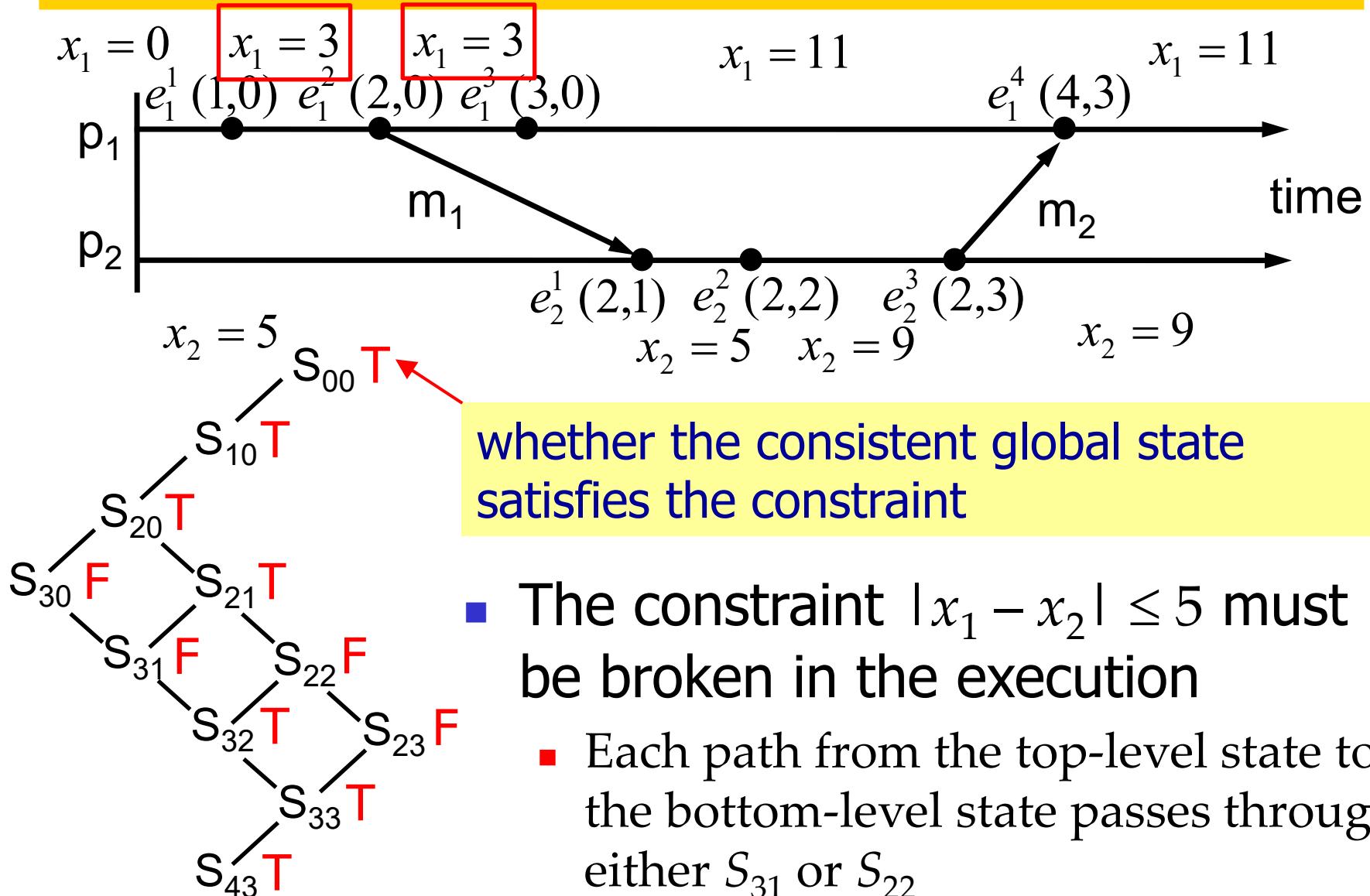
Distributed Debugging



whether the consistent global state satisfies the constraint

- However, we cannot conclude that the constraint $|x_1 - x_2| \leq 5$ must be broken in the execution
 - There is a path from the top-level state to the bottom-level state passing only states satisfying the constraint

Distributed Debugging



Summary

- How to synchronize different computer clocks?
 - Cristian's method
 - Berkeley algorithm
 - Network Time Protocol
 - These methods are based on exchanging clock values and taking into account message transmission times
 - Variations in message transmission times and the way those variations are handled determine the accuracy of synchronization
 - Practically obtainable synchronization accuracy fulfills many requirements but is not sufficient to determine the ordering of arbitrary events at different computers

Summary

- How to order events without global time?
 - Causal ordering
 - Lamport's logical clocks and vector clocks
- How to observe the global state of a distributed system?
 - Cut and consistent cut
 - Chandy and Lamport algorithm to record a snapshot of a distributed system during execution
- Distributed debugging
 - How to enumerate all consistent global states?
 - How to check whether a constraint is broken during execution?

8. Coordination and Agreement

Introduction

- How do processes coordinate their actions and agree on shared values in distributed systems?
 - e.g., among a set of computers, one is selected as a designated time server with which the remaining computers synchronize their clocks – if this server fails, how to choose another computer to take over?
- Problems and issues
 - **Mutual exclusion** – for processes to coordinate their accesses to shared resources
 - **Election** – for processes to agree on a coordinator
 - **Consensus** – for processes to agree on some value

Assumptions

- Each pair of processes is connected by reliable channels – any message sent is eventually delivered intact, exactly once
- No process failure implies a threat to the other processes' ability to communicate – processes do not depend on one another to forward messages
- Communication channels between processes are private
 - A process does not know the messages sent over channels irrelevant to it
 - A process cannot inject messages into the channel between a pair of other processes

Outline

- **Distributed Mutual Exclusion**
- Election
- Consensus Problem
- Summary

Mutual Exclusion

- Objective
 - To prevent interference when a set of processes access shared resources that require exclusive use, e.g., printers, databases, data structures
 - An extension to the critical section problem in operating systems
- Distributed mutual exclusion
 - Unlike stand-alone systems, in a distributed system, neither shared variables nor facilities supplied by a single local kernel can solve the problem
 - So, we require a solution that is based solely on message passing

System Model

- A collection of N processes p_1, p_2, \dots, p_N that do not share variables
 - The processes access shared resources in the critical sections
 - Application-level protocol for a critical section
- enter() //enter critical section – block if necessary
- resourceAccess() //access shared resources in critical section
- exit() //leave critical section – other processes may now enter
- Asynchronous system (delays are unbounded)

Requirements for Mutual Exclusion

- Safety
 - At most one process may execute in the critical section at a time
- Liveness
 - Requests to enter and exit the critical section eventually succeed
 - → no deadlock (deadlock: several processes are waiting for each other to proceed)
 - → no starvation (starvation: a requesting process is infinitely postponed for entry to the critical section)

Performance Metrics

- Bandwidth consumption
 - Number of messages sent in each entry and exit operation
- Client delay
 - Delay incurred at a process at each entry operation
 - When the time accessing the resources in the critical section is short, the dominant factor in the delay is the actual mechanism for mutual exclusion
 - When the resources are used for a long time period in the critical section, the dominant factor is waiting for everyone else to take their turn
 - We focus on the former case

Performance Metrics

- Throughput
 - Rate at which the set of processes as a whole can access the critical section
 - **Synchronization delay** – between one process exiting the critical section and the next process entering it
 - Shorter synchronization delay → higher throughput
 - We focus on the synchronization delay when every process constantly wants to enter a critical section

An Example

- An example to use throughout this section

- A set of 9 processes p_1, p_2, \dots, p_9
- They require entry to the critical section at the following physical times:

p_1	1 sec
p_6	2 sec
p_2	3 sec
p_8	5 sec

- Once entering the critical section, a process accesses the shared resource for 3 seconds before exiting the critical section
- For simplicity, we ignore message transmission delay in the network and message processing delay at the processes in the examples

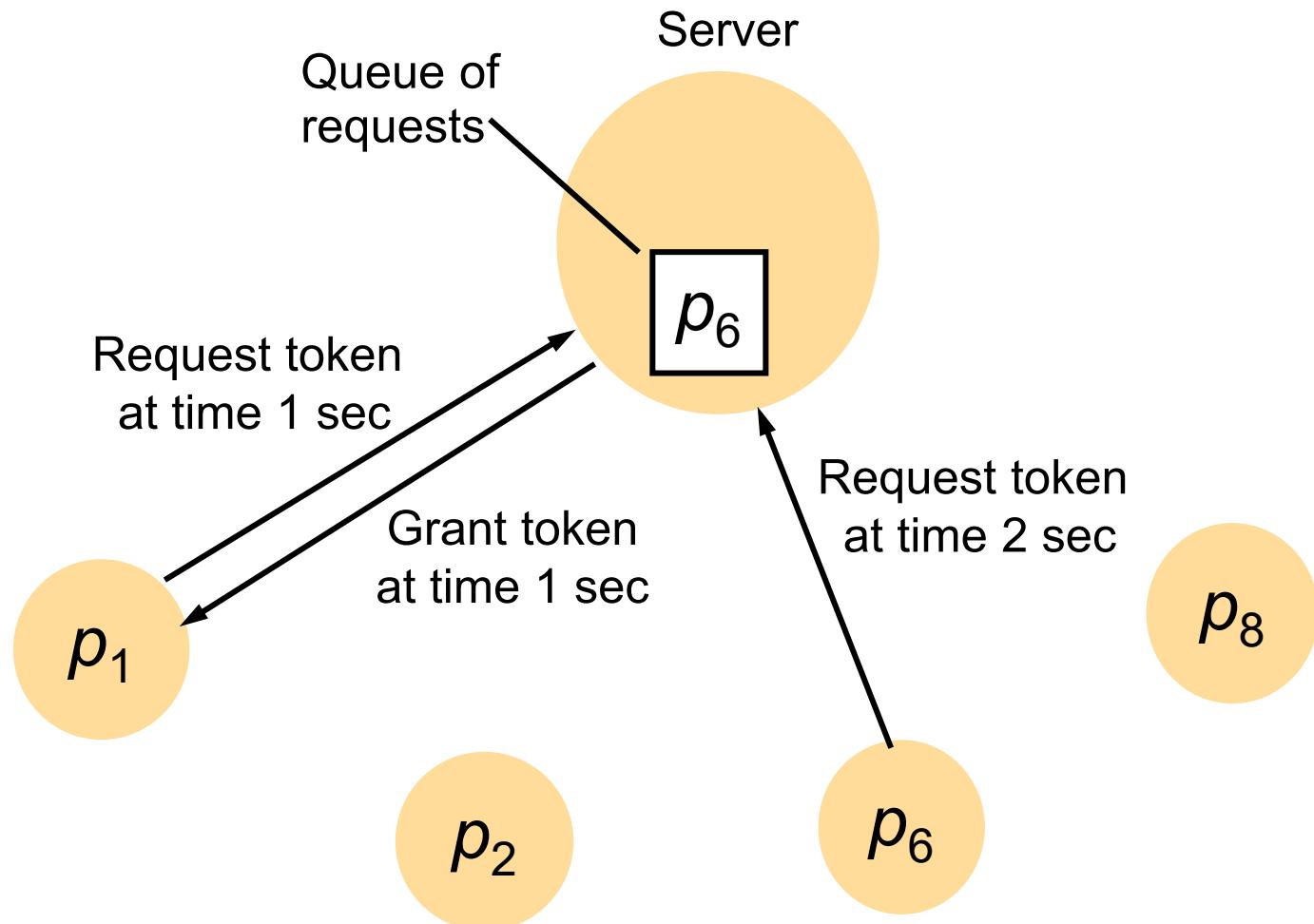
Central Server Algorithm

- Employ a server that grants permission to enter the critical section
 - The server can be an **external process** or **one of the N processes** p_1, p_2, \dots, p_N to be coordinated
- To enter a critical section
 - A process sends a request to the server and awaits a reply from it
 - **Token** – permission to enter the critical section
 - If no other process has the token, the server replies immediately, granting the token, on receiving which the requesting process enters the critical section
 - Otherwise, the server queues the request

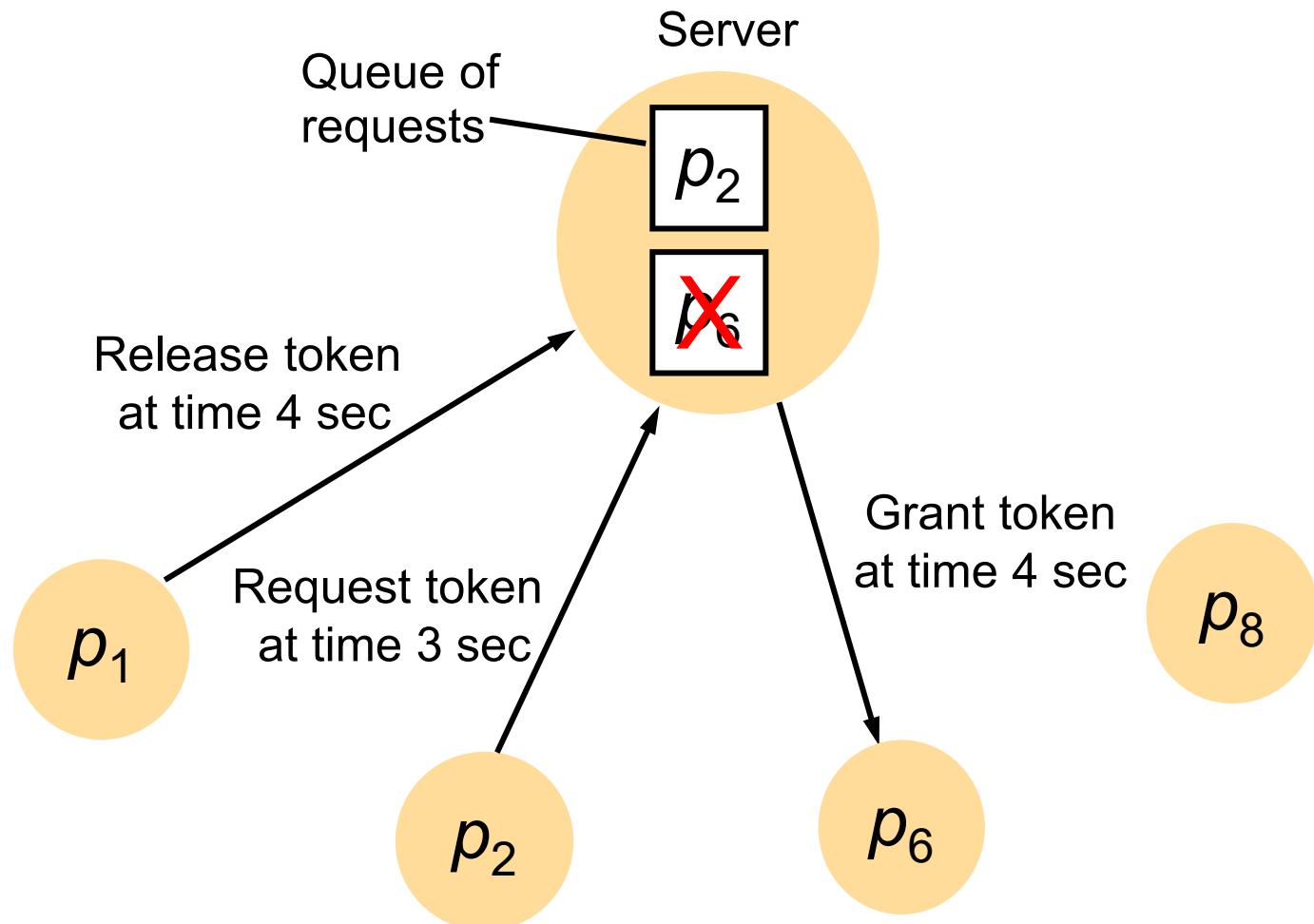
Central Server Algorithm

- To exit a critical section
 - A process sends the token back to the server
 - If the waiting queue is not empty, the server chooses the oldest request in the queue, removes it and replies to the corresponding process with the token

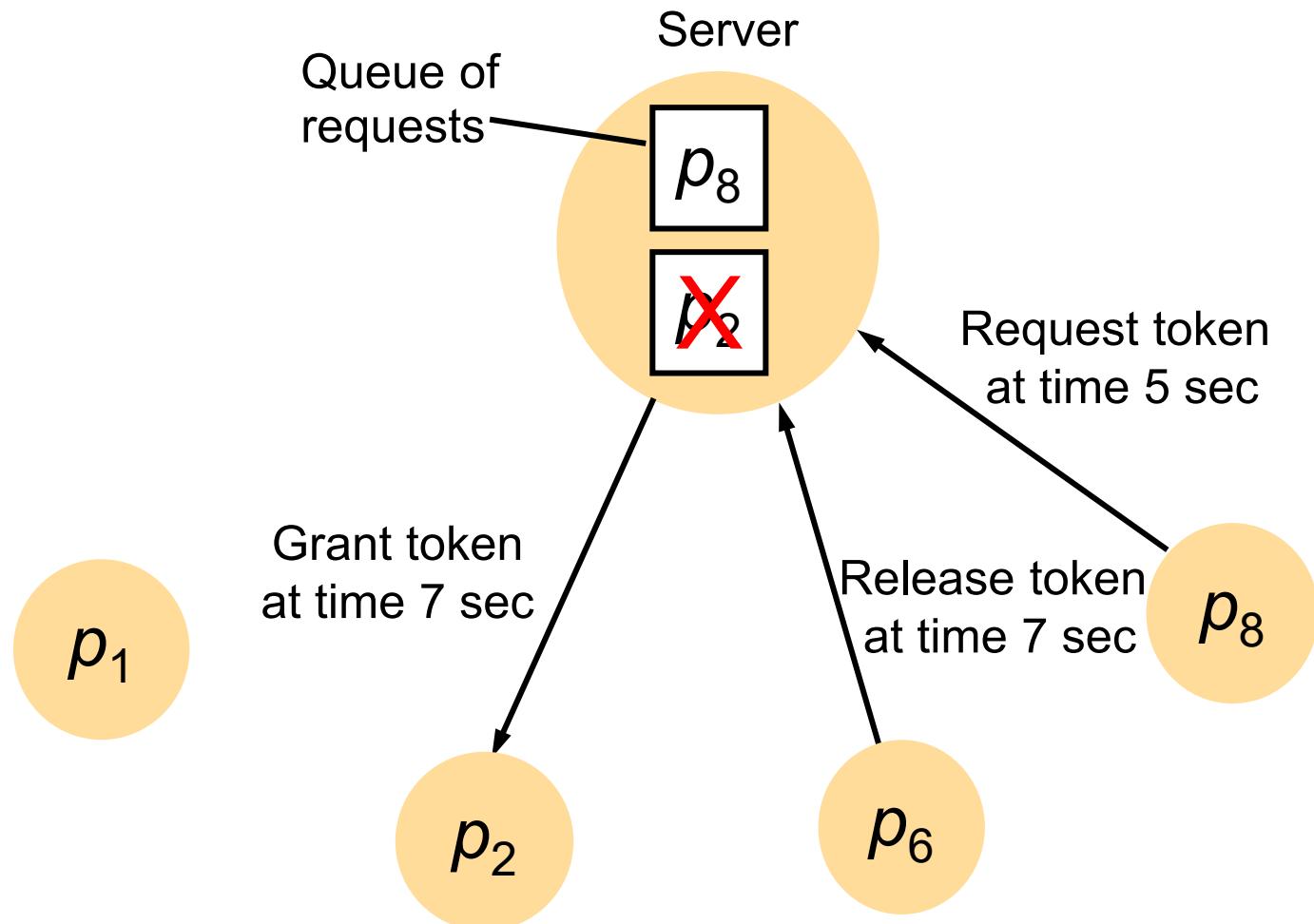
Example of Central Server Algorithm



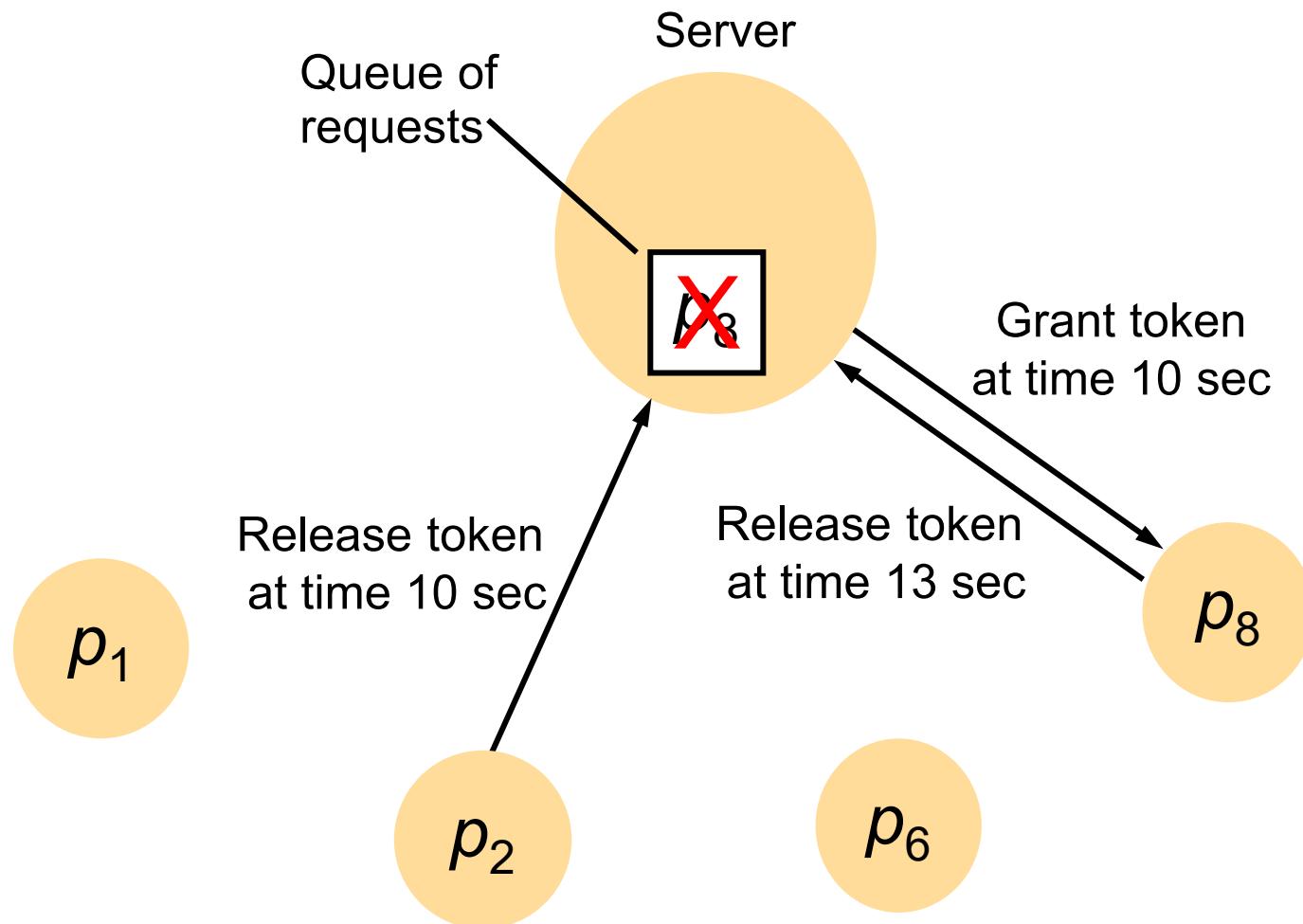
Example of Central Server Algorithm



Example of Central Server Algorithm



Example of Central Server Algorithm



Analysis of Central Server Algorithm

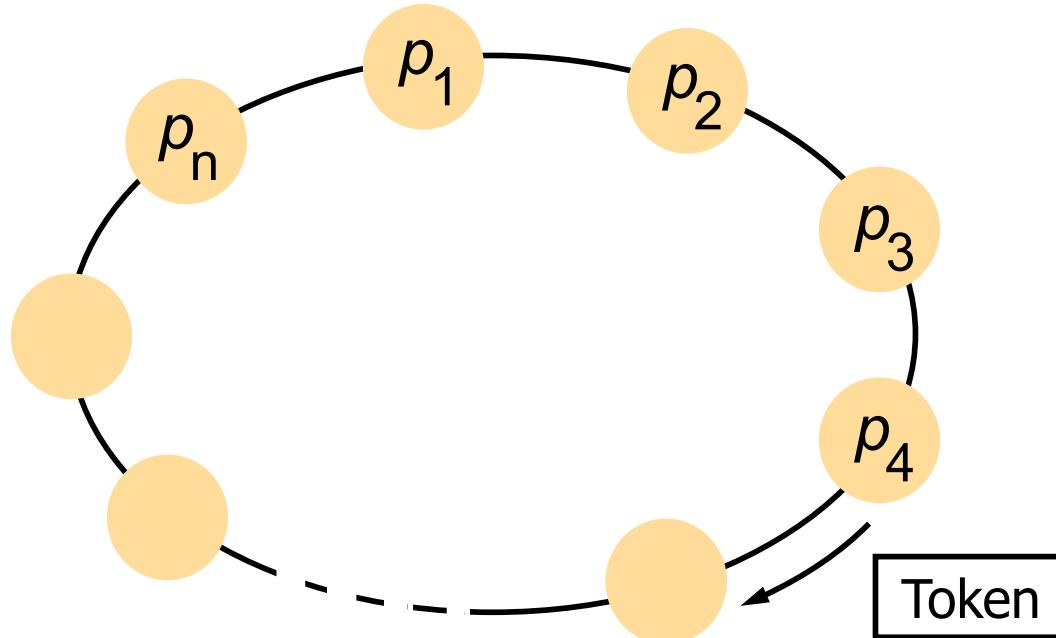
- Safety – satisfied since there is only one token
- Liveness – satisfied due to FIFO request queue
 - Requests are granted in the order in which they are received by the central server
 - No process waits forever → no starvation
- Disadvantage – the server is a single point of failure and can become a performance bottleneck

Analysis of Central Server Algorithm

- Bandwidth consumption
 - Entering a critical section takes **two messages** (a request + a grant)
 - Exiting a critical section takes **one message** (release)
- Client delay
 - Entering a critical section delays the process by **two message transmissions**
- Synchronization delay
 - **Two message transmissions** (a release message to the server followed by a grant message to the next process to enter the critical section)

Ring-Based Algorithm

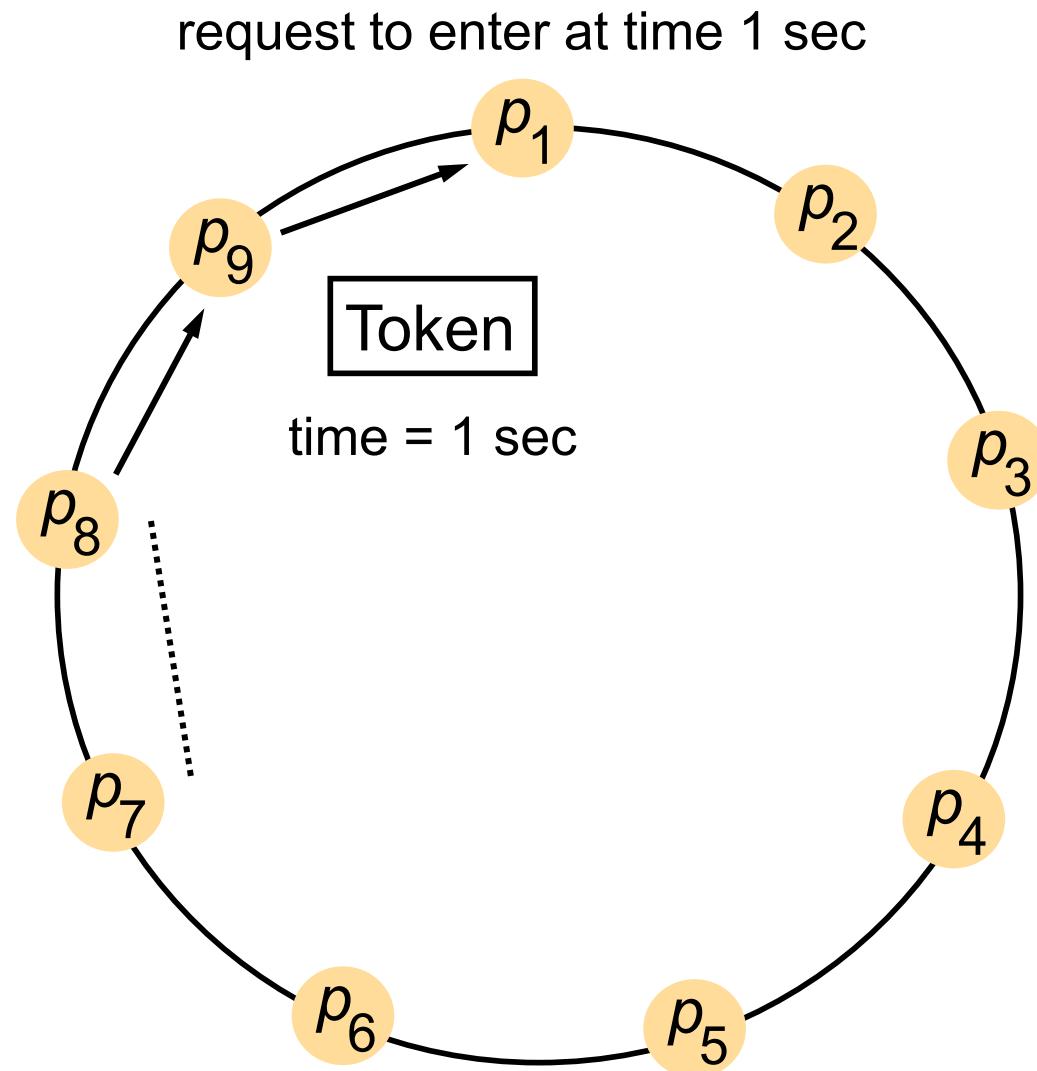
- How to achieve mutual exclusion without a central server process?
- Arrange processes in **a logical ring**
 - Each process knows the next process around the ring



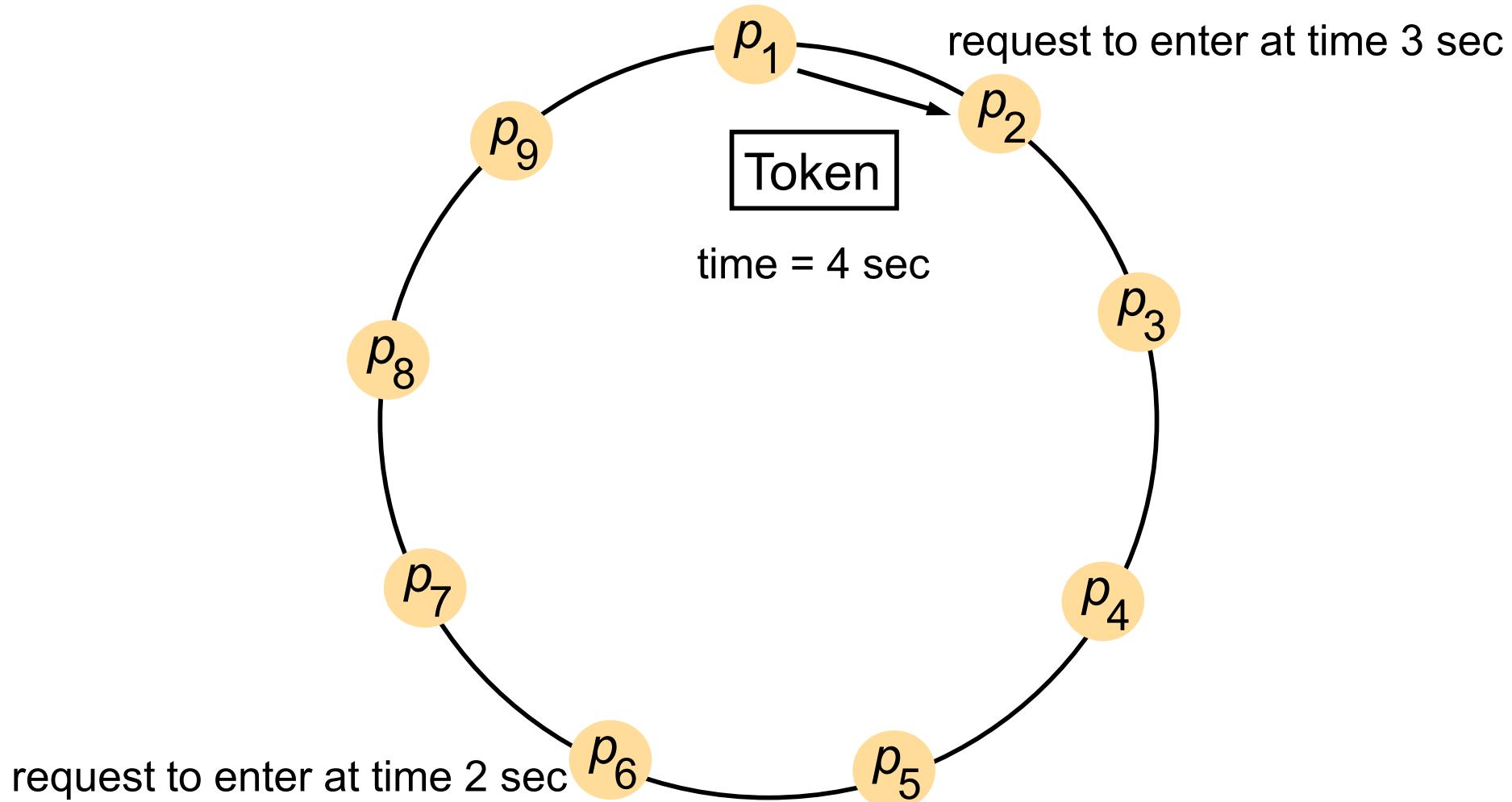
Ring-Based Algorithm

- A token is circulated in clockwise direction around the ring
 - On receiving the token, if the process does not require to enter the critical section, it immediately forwards the token to its neighbor
 - If the receiving process requires to enter the critical section, it retains the token and enters the critical section
 - To exit the critical section, the process sends the token on to its neighbor

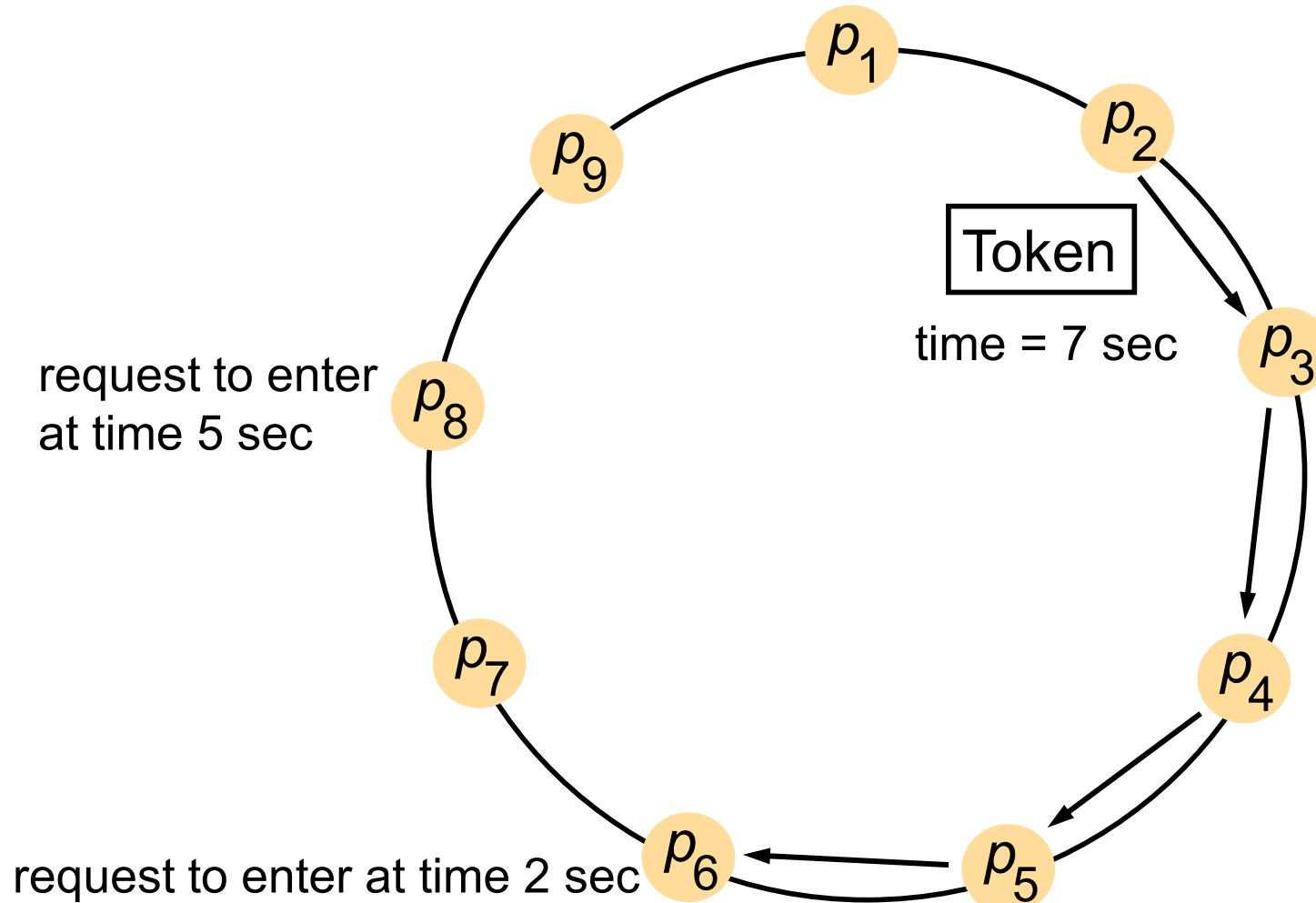
Example of Ring-Based Algorithm



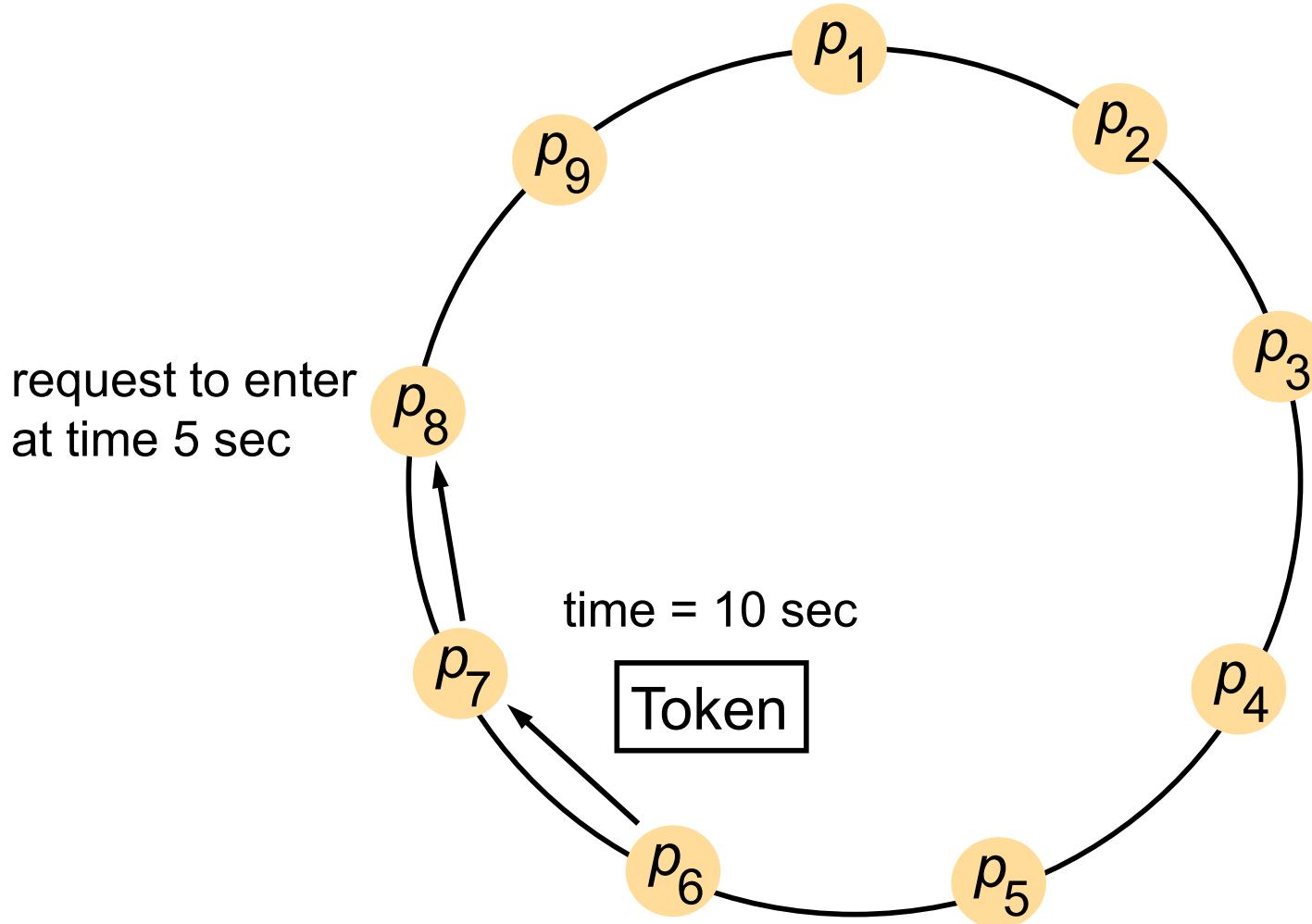
Example of Ring-Based Algorithm



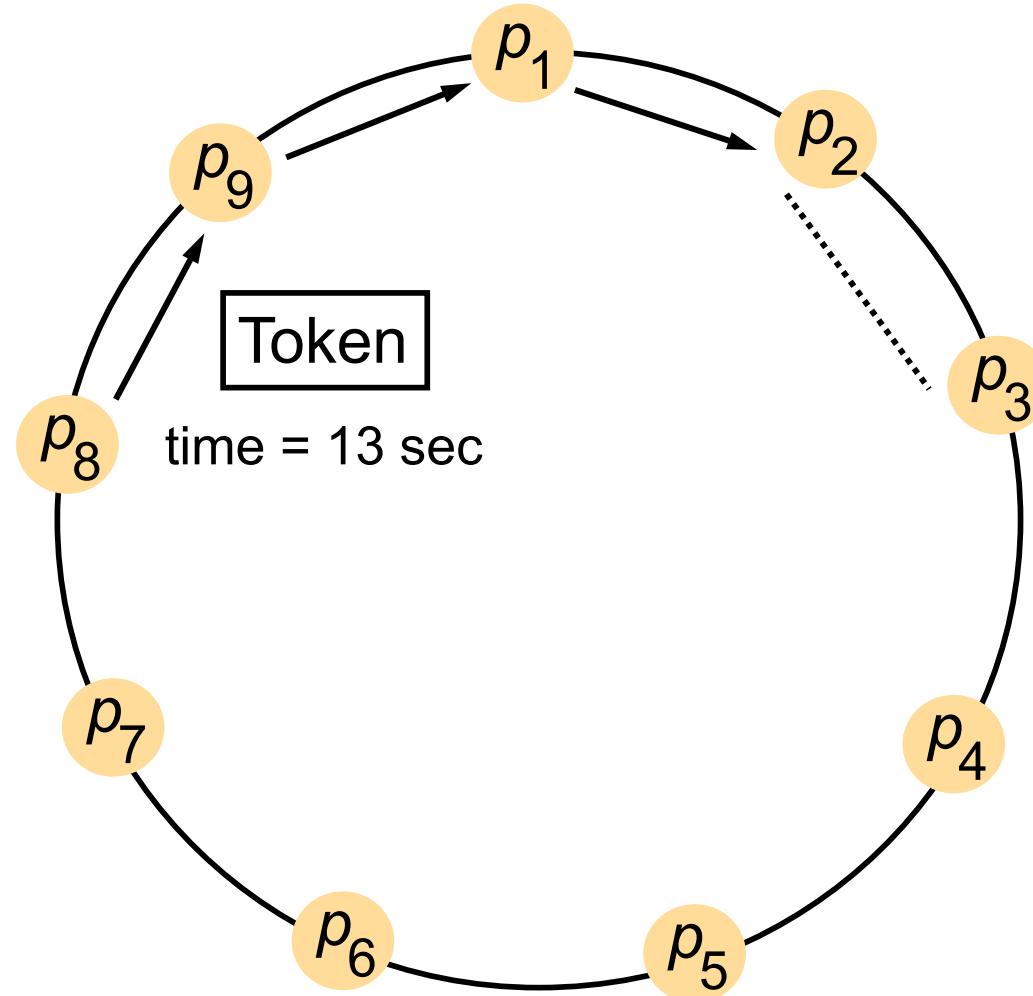
Example of Ring-Based Algorithm



Example of Ring-Based Algorithm



Example of Ring-Based Algorithm



Analysis of Ring-Based Algorithm

- Safety – satisfied
 - Only one token in the system → only one process can execute in the critical section at a time
- Liveness – satisfied
 - Token is circulated among all processes → it will find outstanding requests, so progress will always be made
 - No process can enter the critical section again without passing the token to the others first (hence granting others a chance) → no starvation
- Entry is not necessarily granted in the time order of requests
 - It depends on process locations in the ring

Analysis of Ring-Based Algorithm

- Bandwidth consumption
 - If every process constantly wants to enter a critical section, each token pass results in one entry and exit
 - On the other hand, processes send the token around the ring even when no process requires entry to the critical section → in this case, the number of messages per entry into a critical section is unbounded
- Client delay
 - Entering a critical section delays the process **between 0** (when it has just received the token) **and N message transmissions** (when it has just passed on the token)
- Synchronization delay
 - **Between 1 and N message transmissions**

Ricart and Agrawala Algorithm

- Require a total ordering of all events
 - For example, use Lamport's logical clocks and break ties with process identifiers
- Each process is in one of the following three states:
 - **RELEASED** – outside the critical section
 - **WANTED** – requiring entry to the critical section
 - **HELD** – executing in the critical section

Ricart and Agrawala Algorithm

- When a process requires entry to a critical section
 - Change to WANTED state
 - Send a request message to all other processes and wait for their replies
 - Can enter the critical section only when all other processes have replied to the request message
 - Change to HELD state when entering

Ricart and Agrawala Algorithm

- When a process receives a request message from another process
 - If the receiver is in RELEASED state, it replies immediately
 - If the receiver is in HELD state, it queues the request
 - If the receiver is in WANTED state, it compares the timestamp of the incoming request with its own request
 - If the incoming request has **a smaller timestamp**, reply immediately
 - Otherwise, queue the request

Ricart and Agrawala Algorithm

- When a process exits the critical section
 - Change to RELEASED state
 - Reply to all queued requests

Ricart and Agrawala Algorithm

On initialization

state := RELEASED;

To enter the critical section

state := WANTED;

send *request* to all the other processes;

T := timestamp of the request;

wait until (number of replies received = $(N - 1)$);

state := HELD;

On receipt of a request $\langle T_j, p_j \rangle$ from another process p_j ($j \neq i$)

if (*state* = HELD or (*state* = WANTED and $(T, p_i) < (T_j, p_j)$))

then

queue *request* from p_j without replying;

else

reply immediately to p_j ;

To exit the critical section

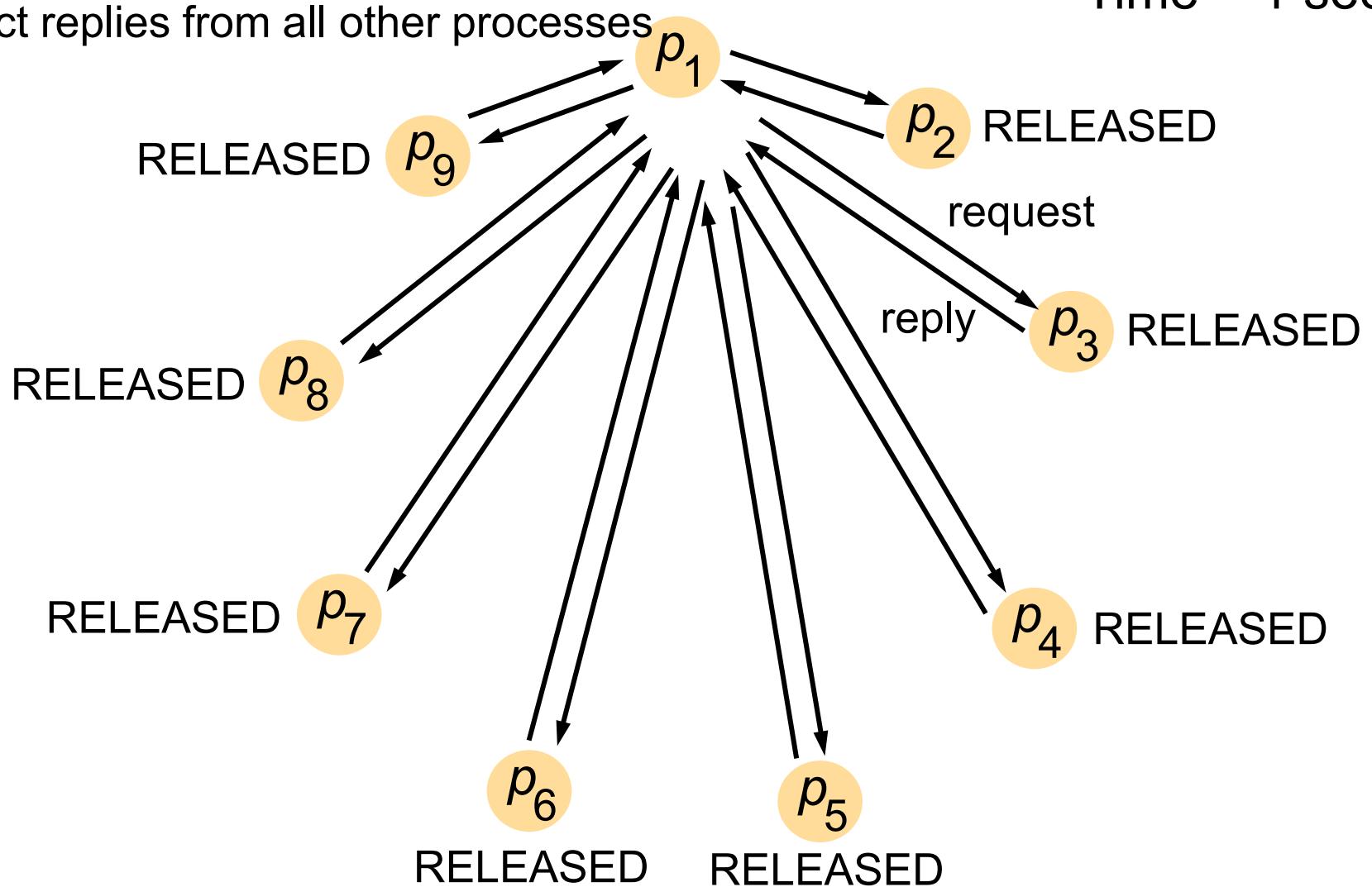
state := RELEASED;

reply to all queued requests;

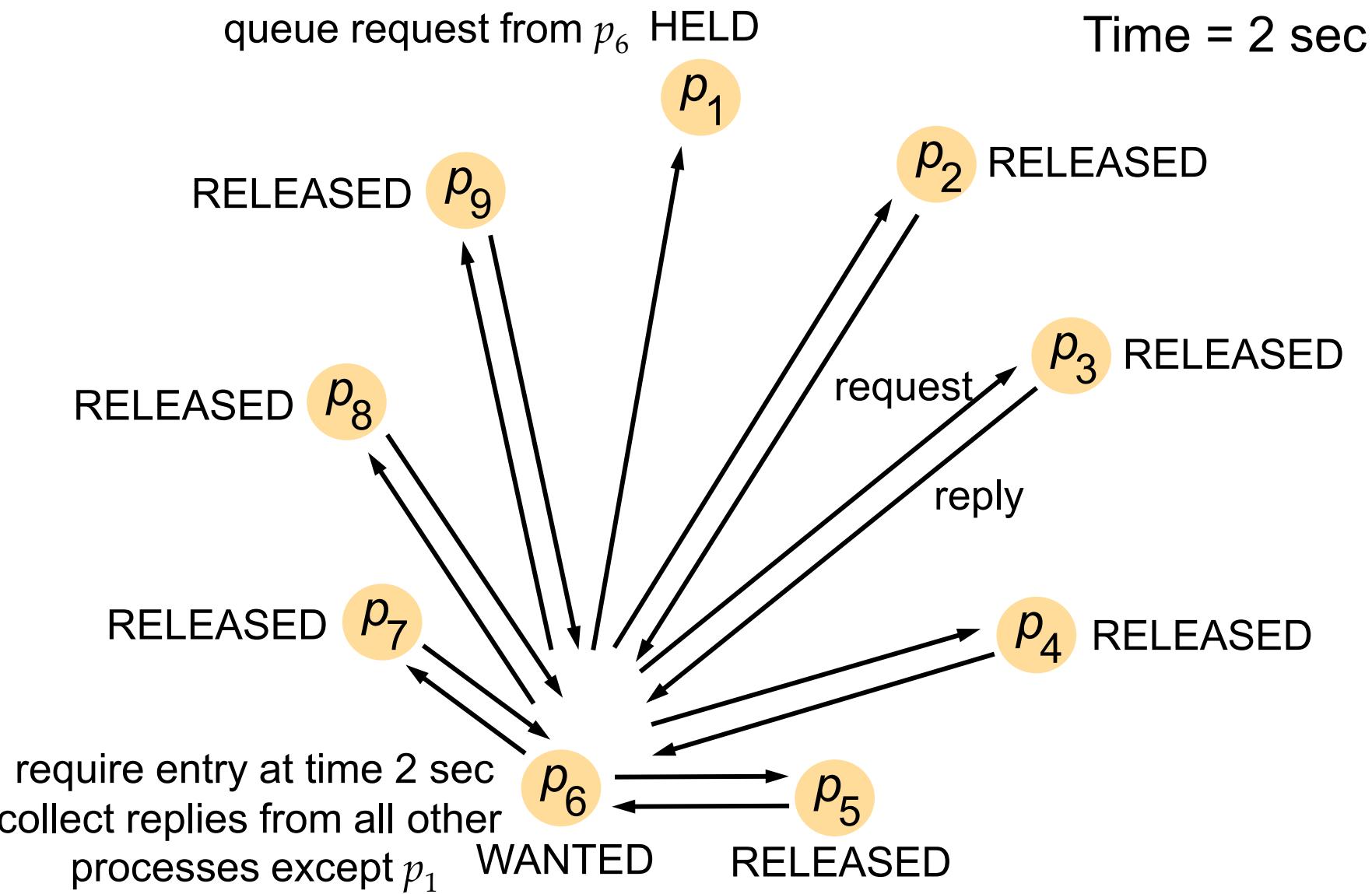
algorithm executed by a process p_i

Example of Ricart and Agrawala Algo

require entry at time 1 sec
collect replies from all other processes



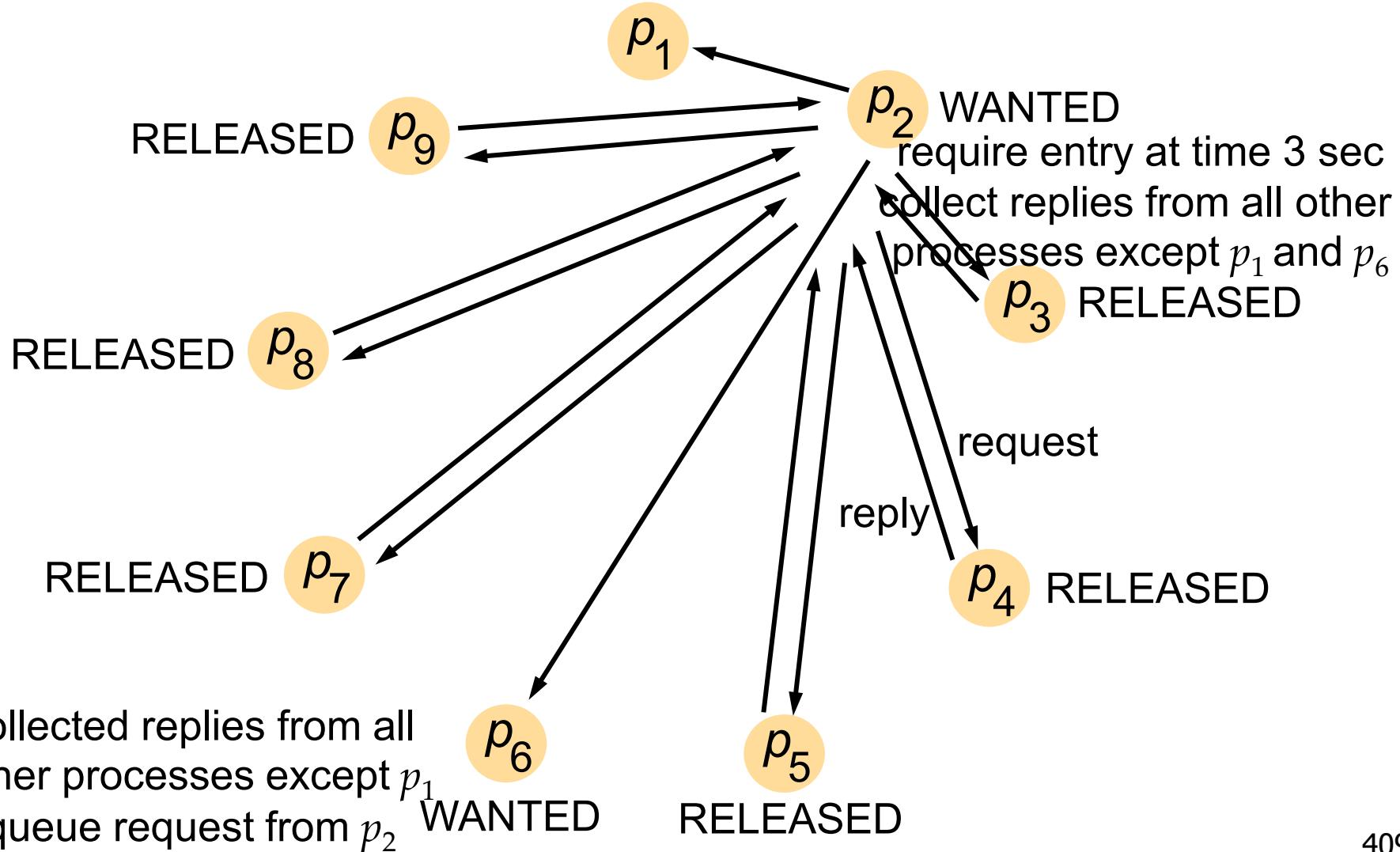
Example of Ricart and Agrawala Algo



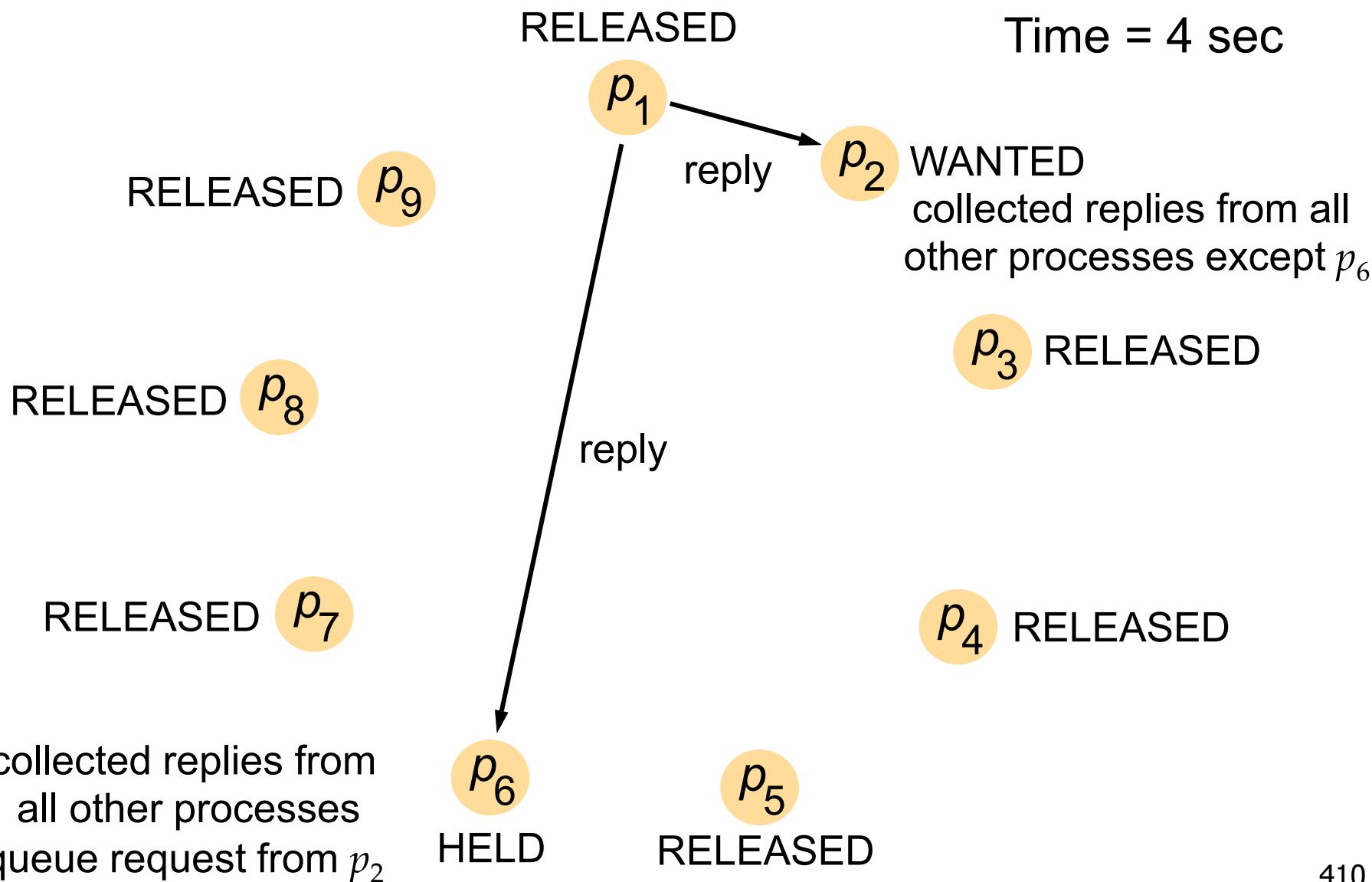
Example of Ricart and Agrawala Algo

queue requests from p_6 and p_2 HELD

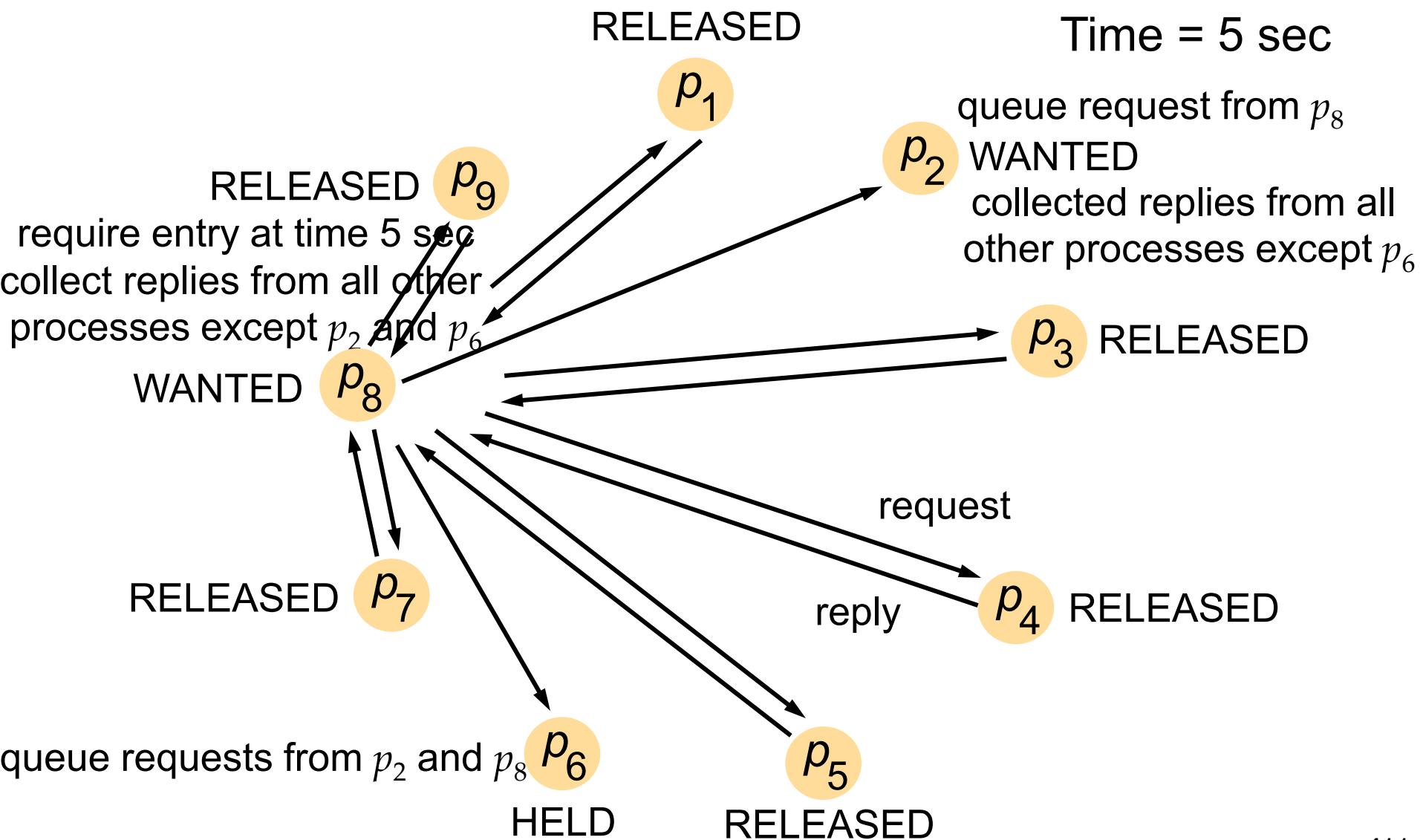
Time = 3 sec



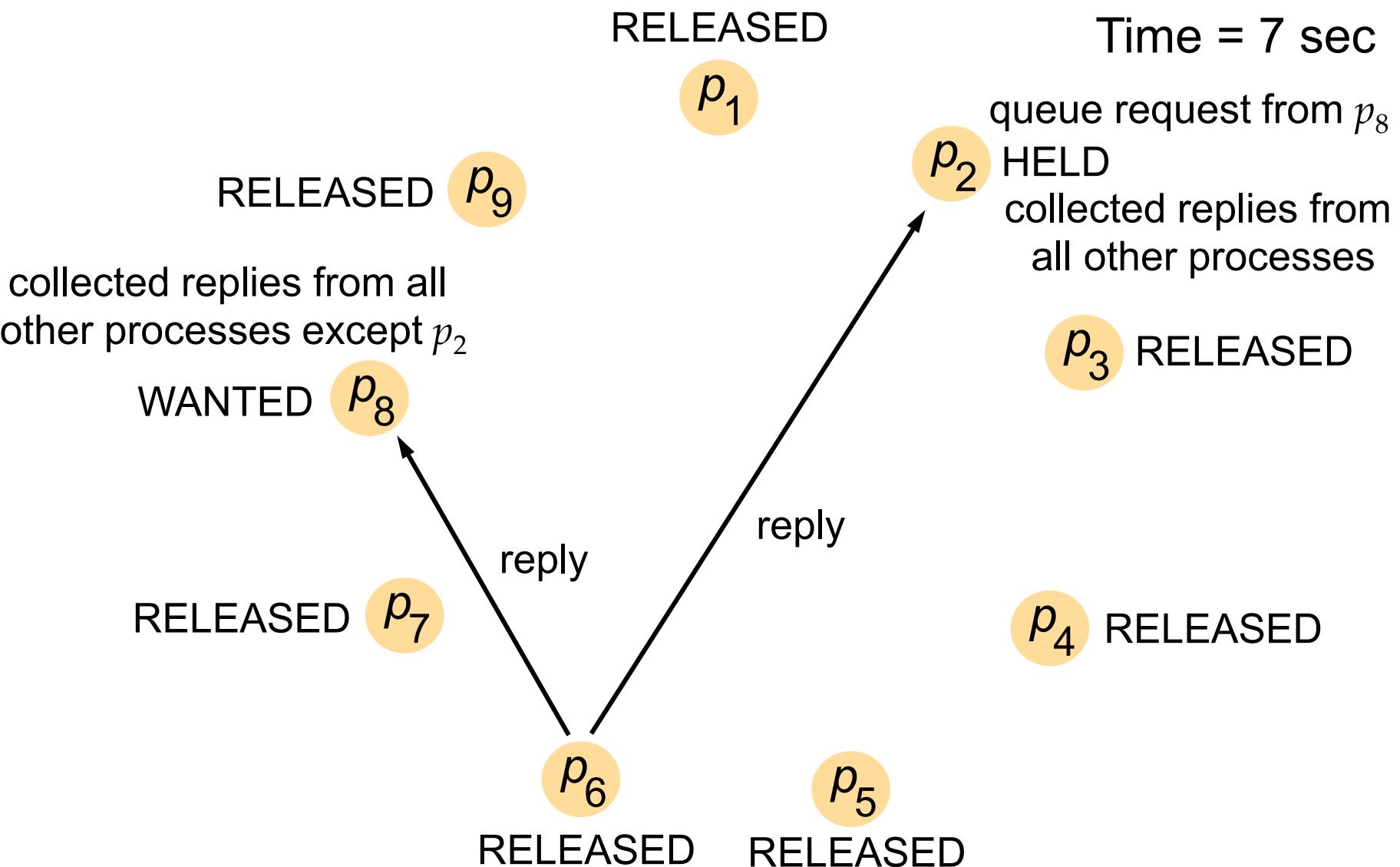
Example of Ricart and Agrawala Algo



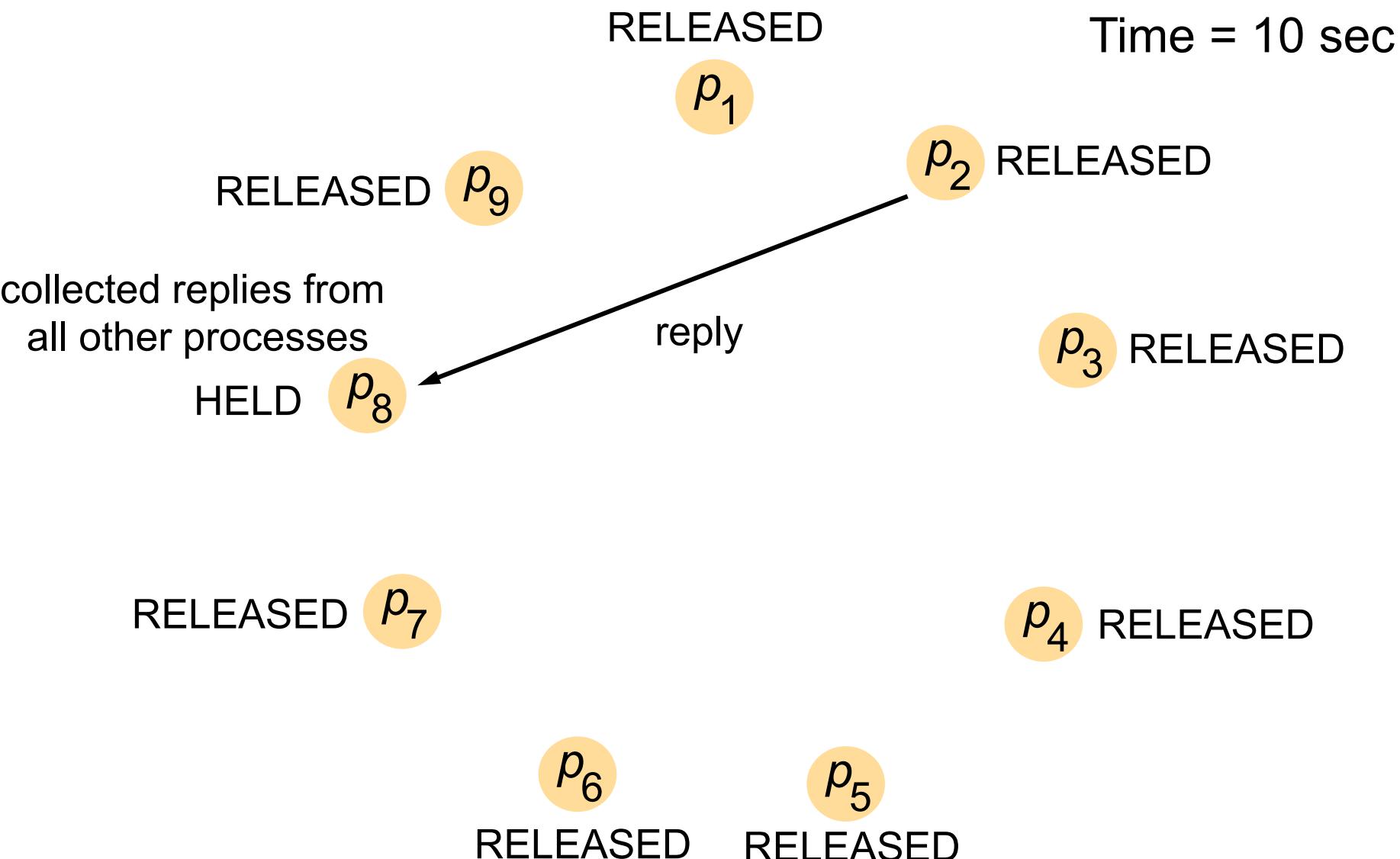
Example of Ricart and Agrawala Algo



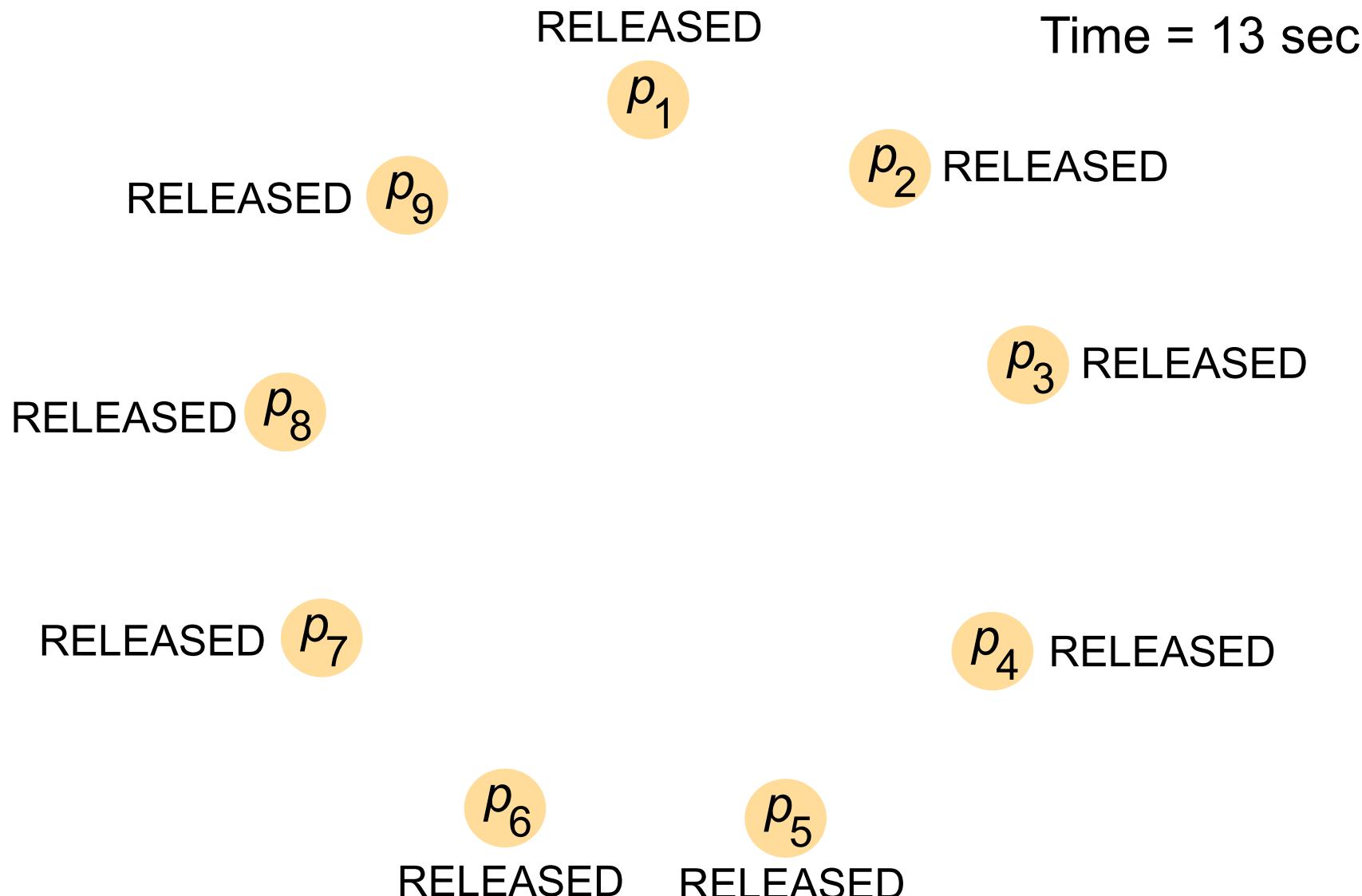
Example of Ricart and Agrawala Algo



Example of Ricart and Agrawala Algo



Example of Ricart and Agrawala Algo



Analysis of Ricart and Agrawala Algo

- Safety – satisfied
 - If it were possible for two processes to enter the critical section at the same time, both of them would have replied to the other, but since requests are totally ordered, this is impossible
- Liveness – satisfied
 - The outstanding request bearing the lowest timestamp would collect all replies and gain entry
 - If Lamport's logical clocks are used, processes are granted entry to the critical section in causal ordering of their requests

Analysis of Ricart and Agrawala Algo

- Bandwidth consumption
 - Entering a critical section takes $2(N-1)$ messages ($N-1$ requests + $N-1$ replies)
- Client delay
 - Entering a critical section delays the process by two message transmissions (if all requests are sent simultaneously)
- Synchronization delay
 - 1 message transmission

Outline

- Distributed Mutual Exclusion
- Election
- Consensus Problem
- Summary

Election

- Objective
 - To choose a **unique** process (called **coordinator**) to play a particular role
 - e.g., a time server among a set of computers
 - e.g., a server in the central server algorithm for mutual exclusion if it is one of the N processes p_1, p_2, \dots, p_N to be coordinated
 - In general, a coordinator may offer any service such as synchronization, breaking deadlocks etc.
- If the current coordinator crashes or wishes to retire, an election is required to choose a new coordinator on which all processes agree

Election

- A process initiates the election if it starts a particular run of the election algorithm
 - An individual process does not initiate more than one election at a time, but **different processes may call elections concurrently** → we need to ensure that a unique coordinator is elected in such cases
 - **More processes may fail during election** → we need to ensure that a non-crashed process is eventually elected in such cases

Election

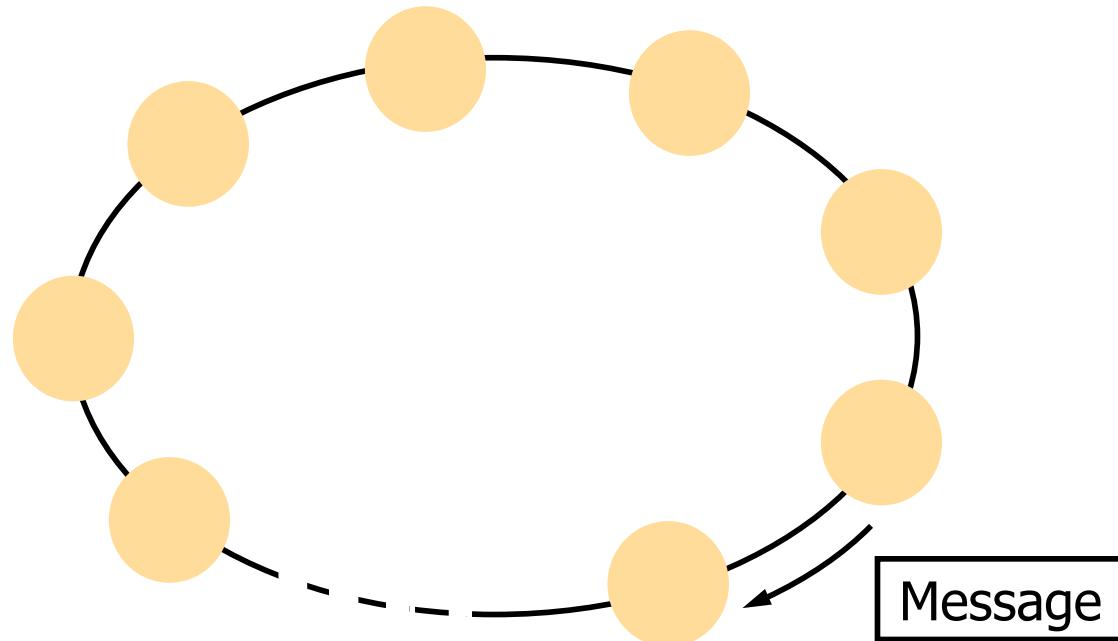
- Assume each process has a unique identifier
- Without loss of generality, we require the coordinator be chosen as the process with largest identifier
 - Assume a total order of process identifiers
 - The identifier may be any useful value, e.g., to elect a process with the lowest computational load, we can take $1/\text{load}$ as the process identifier (or use $\langle 1/\text{load}, i \rangle$ to break ties where i is the process index)

Performance Metrics

- Bandwidth consumption
 - Total number of messages sent
- Turnaround time
 - The number of serialized message transmission times between the initiation and termination of an election

Ring-Based Algorithm

- Arrange processes in a logical ring
 - Each process knows the next process around the ring
 - All messages are sent clockwise around the ring



Ring-Based Algorithm

- Assume asynchronous system and no failure occurs during election
- A process is called a **participant** if it is currently engaged in some election(s), otherwise the process is called a **non-participant**
- Initially, every process is marked as a non-participant in an election
- Any process can initiate an election, it proceeds by marking itself as a participant, placing its identifier in **an election message** and sending it to its clockwise neighbor

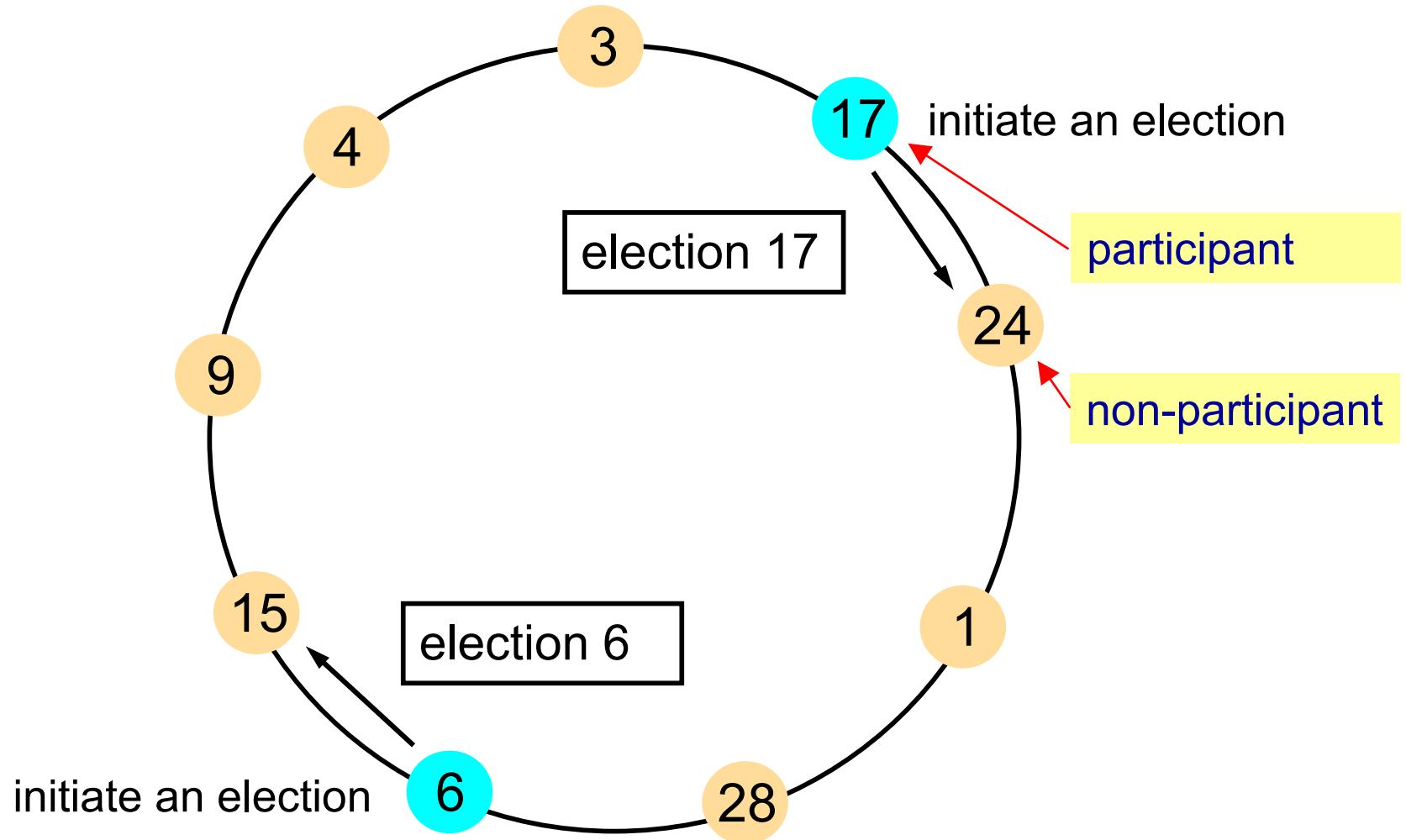
Ring-Based Algorithm

- When a process receives **an election message**, it compares the identifier in the message with its own identifier
 - If the arrived identifier is greater, it forwards the message to its neighbor
 - If the arrived identifier is smaller and the receiver is not a participant yet, it substitutes its own identifier in the message and forwards the message
 - If the arrived identifier is smaller and the receiver is already a participant, it does not forward the message
 - On forwarding an election message in any case, the process marks itself as a participant

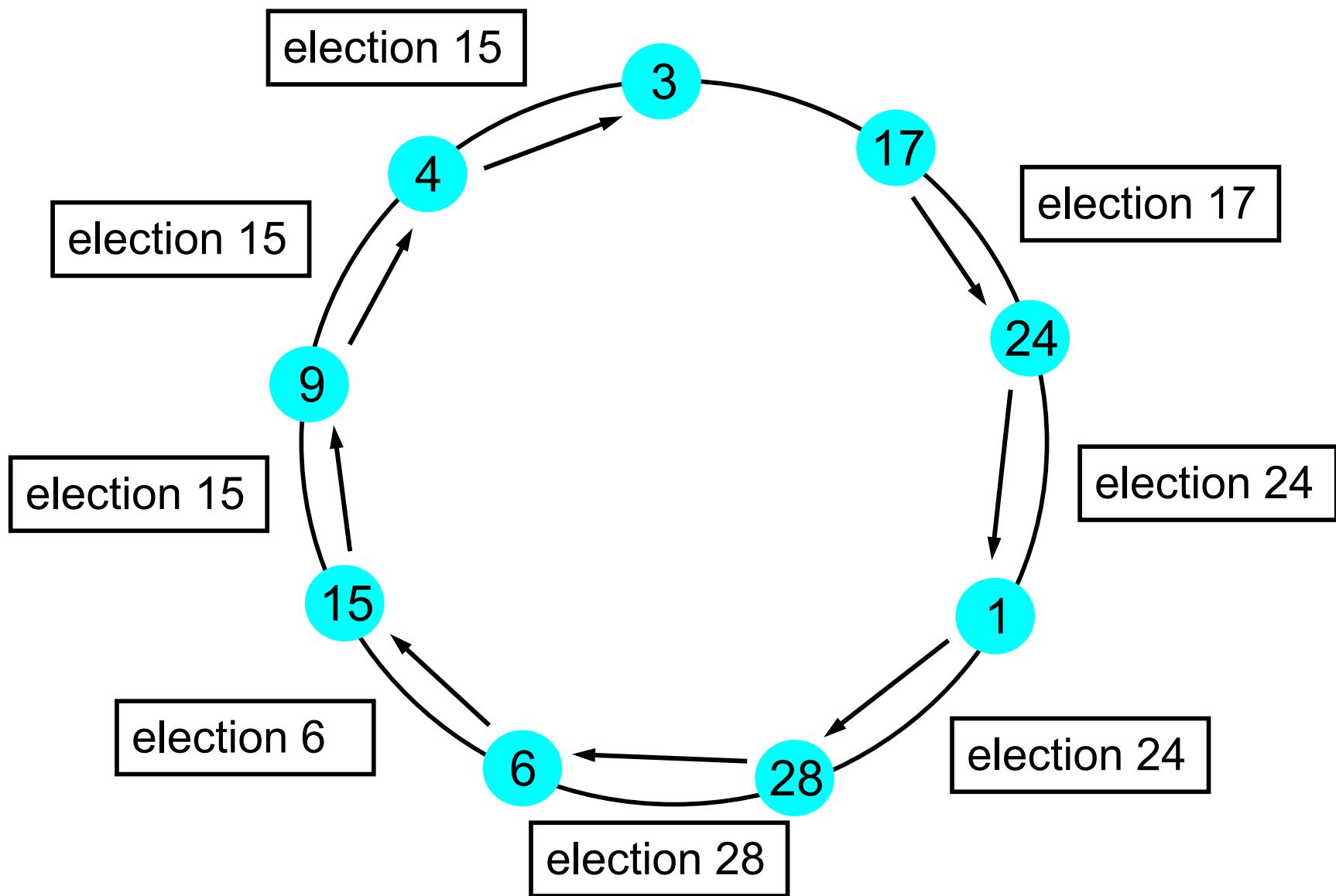
Ring-Based Algorithm

- (cont'd)
 - If the arrived identifier is that of the receiver itself, this process's identifier must be the greatest and it becomes the **coordinator**
 - The coordinator marks itself as a non-participant and sends **an elected message** to its neighbor, announcing its election and enclosing its identity
 - When a process p_i receives an elected message, it marks itself as a non-participant and records the coordinator identity found in the message
 - p_i forwards the elected message to its neighbor unless it is the coordinator

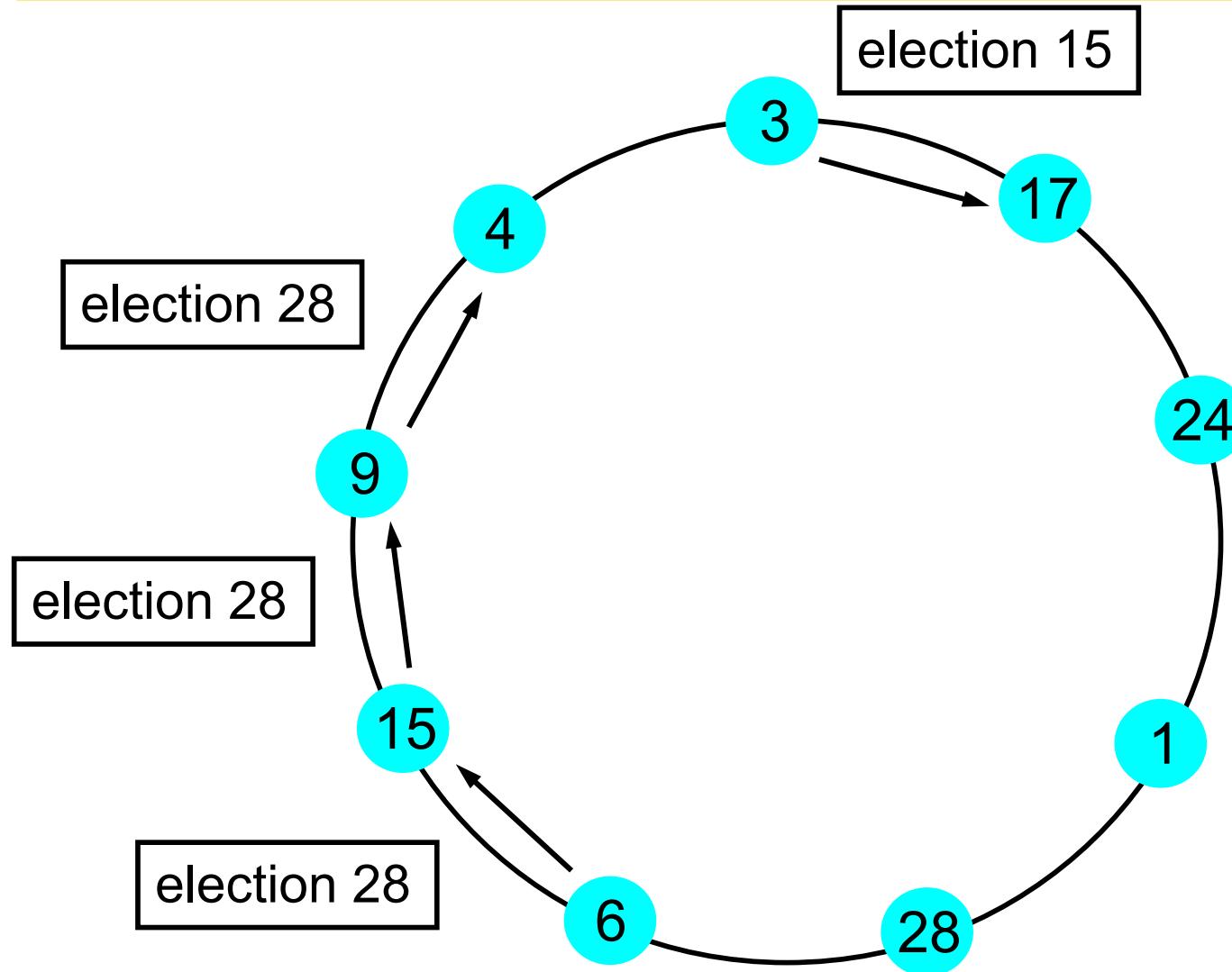
Example of Ring-Based Algorithm



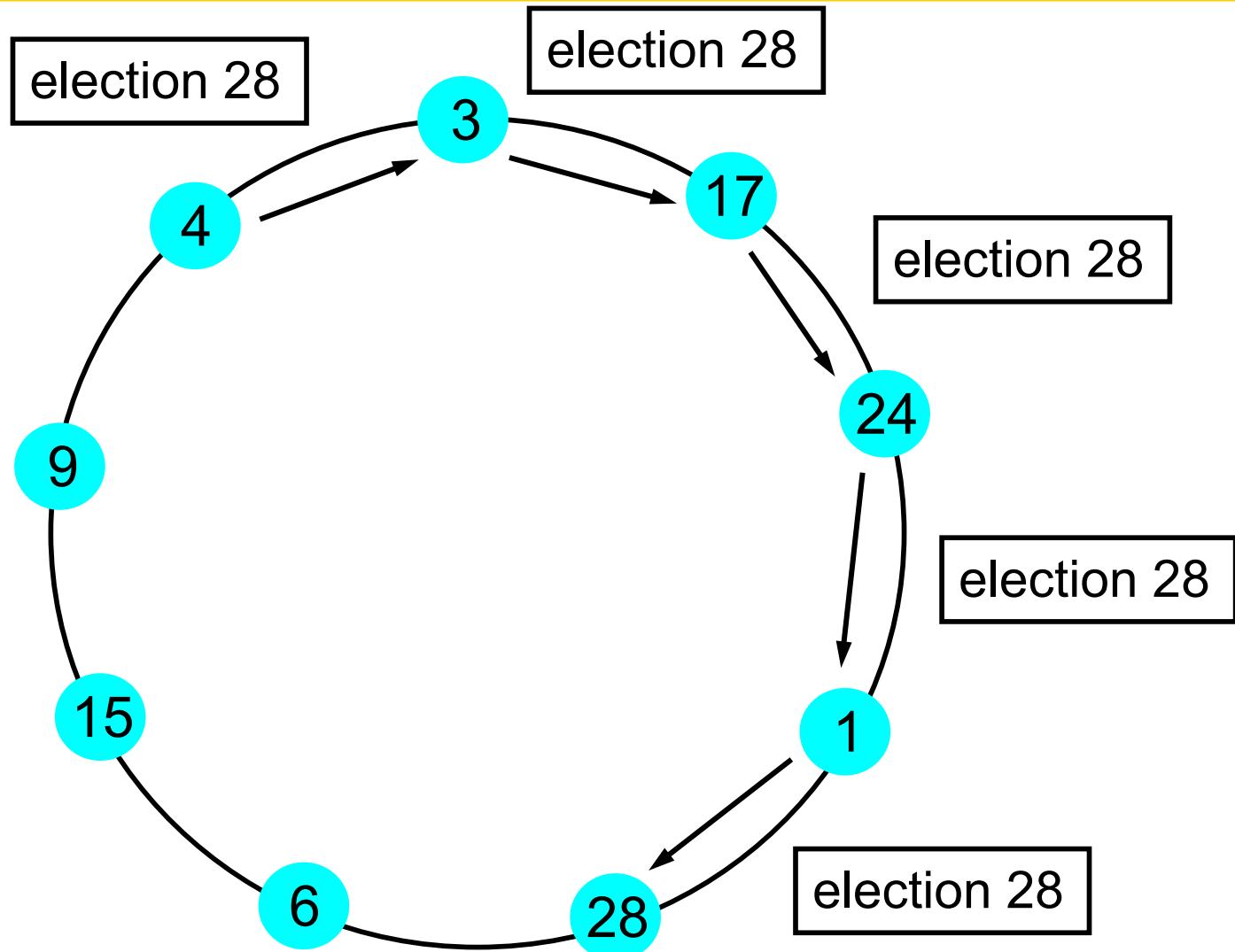
Example of Ring-Based Algorithm



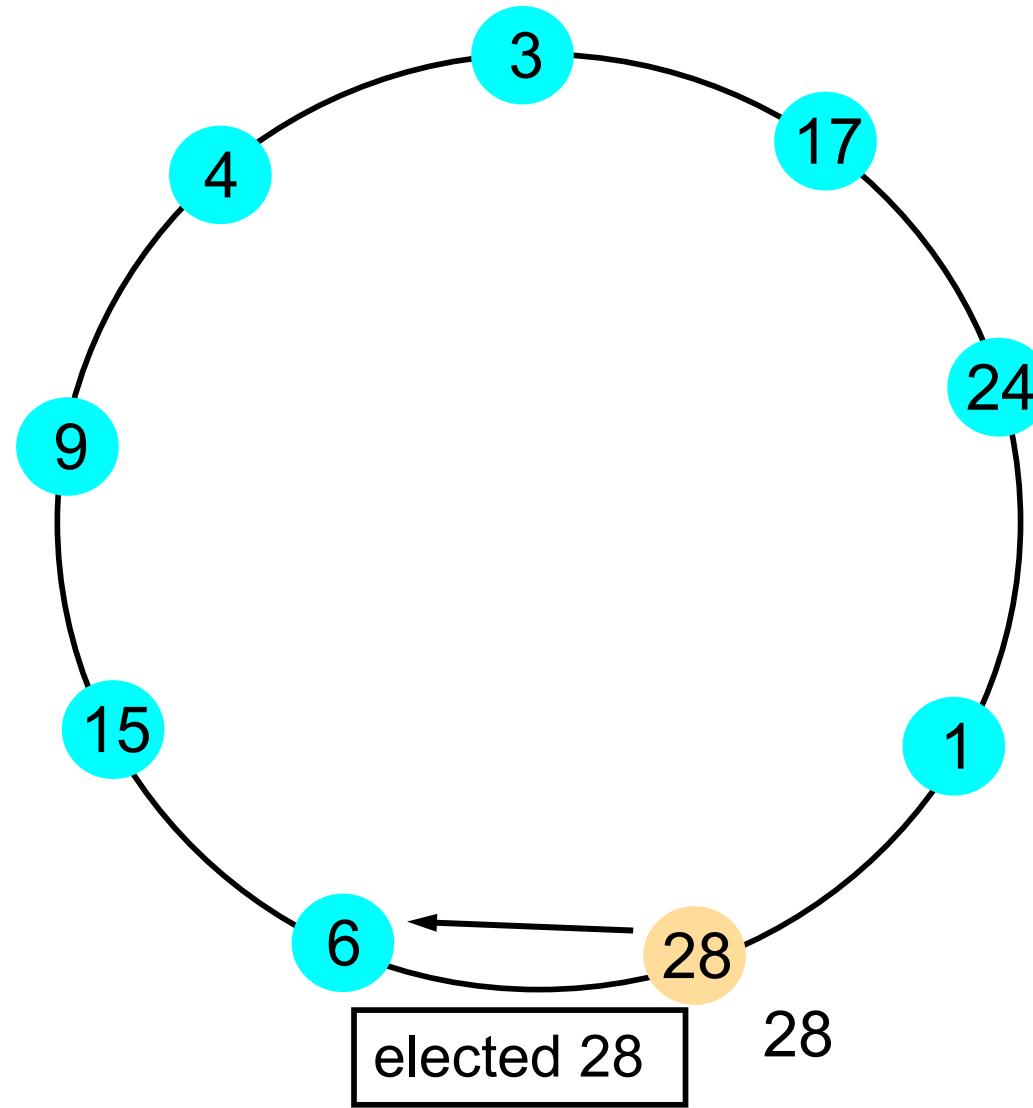
Example of Ring-Based Algorithm



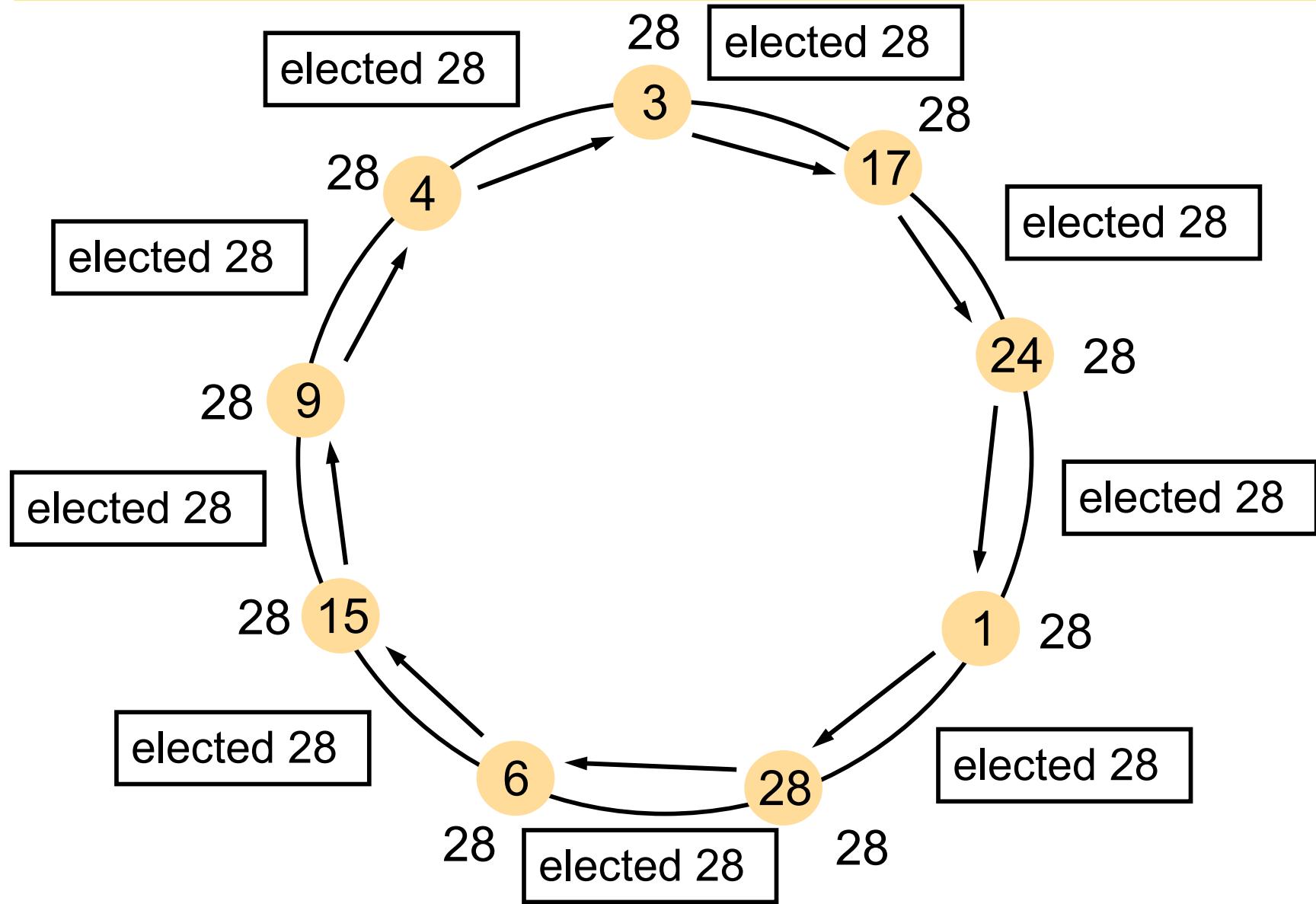
Example of Ring-Based Algorithm



Example of Ring-Based Algorithm



Example of Ring-Based Algorithm



Analysis of Ring-Based Algorithm

- Correctness
 - All identifiers are compared since a process must receive its own identifier back before sending an elected message
 - For any two processes, the one with the larger identifier will not pass on the other's identifier
→ it is impossible that both processes receive their own identifiers back
 - So, only one process would receive its own identifier back and this process's identifier must be the largest

Analysis of Ring-Based Algorithm

- Bandwidth consumption
 - Assume only a single process initiates an election
 - Worst case: when its anti-clockwise neighbor has the largest identifier → it takes **$N-1$ election messages** to reach this neighbor
 - Then, **N more election messages** to complete a circuit before announcing the coordinator
 - Finally, announcing the coordinator takes further **N elected messages**
 - So, a total of $3N-1$ messages
- Turnaround time
 - Worst case: $3N-1$ message transmissions since these messages are sent sequentially

Bully Algorithm

- Allow processes to crash during an election
- Assume synchronous system and use timeouts to detect process crash
 - Let T_{trans} be a maximum message transmission delay, and T_{process} be a maximum time for processing a message, then an upper bound on the total elapsed time from sending a message to another process to receiving a response is $T = 2 T_{\text{trans}} + T_{\text{process}}$
 - If no response arrives within time T , the intended recipient must have crashed

Bully Algorithm

- Assume each process knows which processes have larger identifiers and it can communicate with all such processes
- Three types of messages in Bully algorithm
 - **Election message:** to announce an election
 - **Answer message:** to respond to an election message
 - **Coordinator message:** to announce the identity of the coordinator
- When any process detects that the coordinator crashes, it begins an election
 - Several processes may discover this concurrently

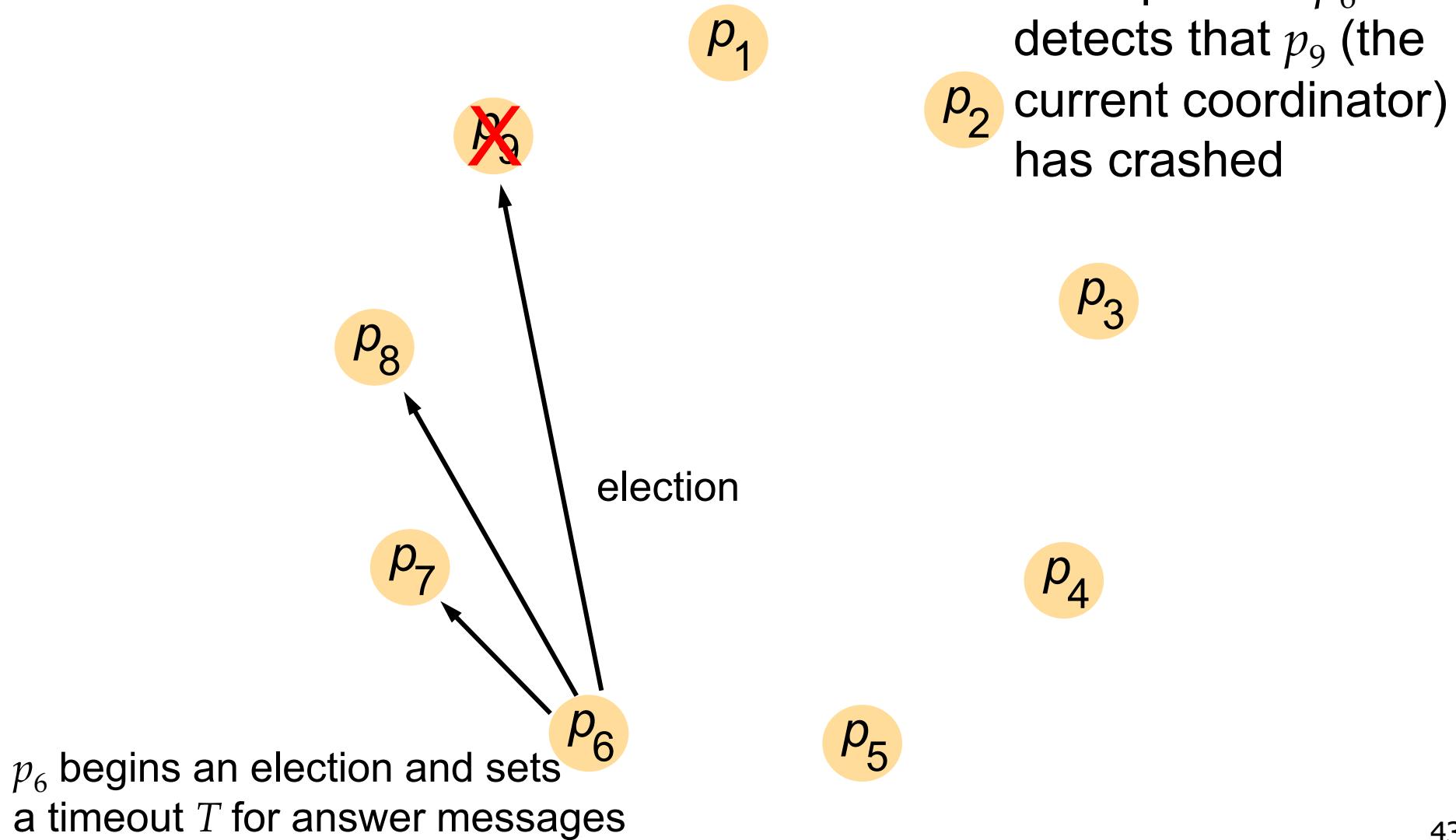
Bully Algorithm

- When a process P begins an election
 - P sends **an election message** to all processes with larger identifiers and awaits **answer messages** in response
 - If no process responds within time T , P wins the election and becomes the coordinator (to announce this, P sends **a coordinator message** to all processes with smaller identifiers)
 - If some process responds with answer message, P waits a further period for a coordinator message to arrive from the new coordinator (if none arrives, P begins another election)
 - When P is waiting for answer messages, P is said to hold an election

Bully Algorithm

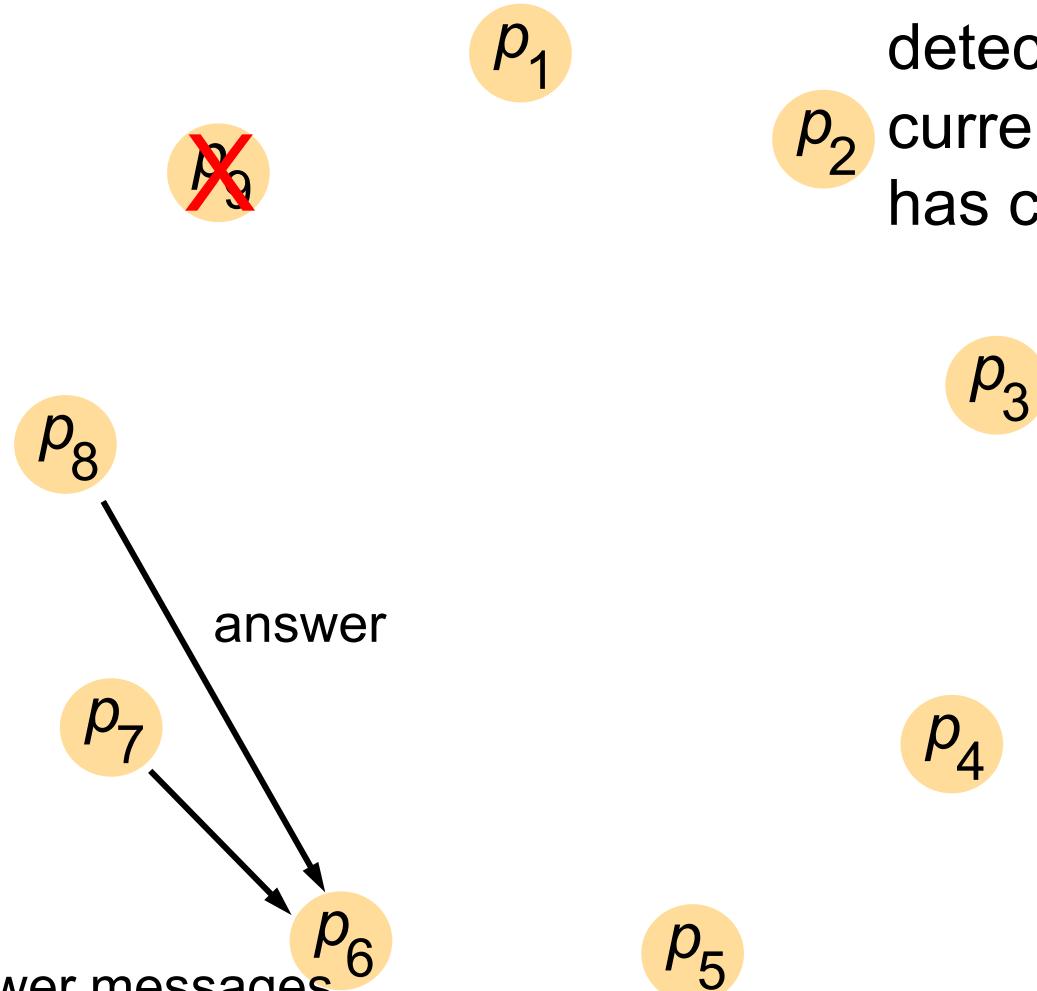
- When a process receives **an election message** (from another process with smaller identifier)
 - The receiving process sends back **an answer message** to indicate that it is alive
 - The receiving process begins an election (as described in the previous slide) if it is not holding an election
- If a crashed process comes back up
 - If it is the process with the largest identifier, it decides that it is the coordinator and announces this to all other processes by sending **a coordinator message**
 - Otherwise, it begins an election

Example of Bully Algorithm



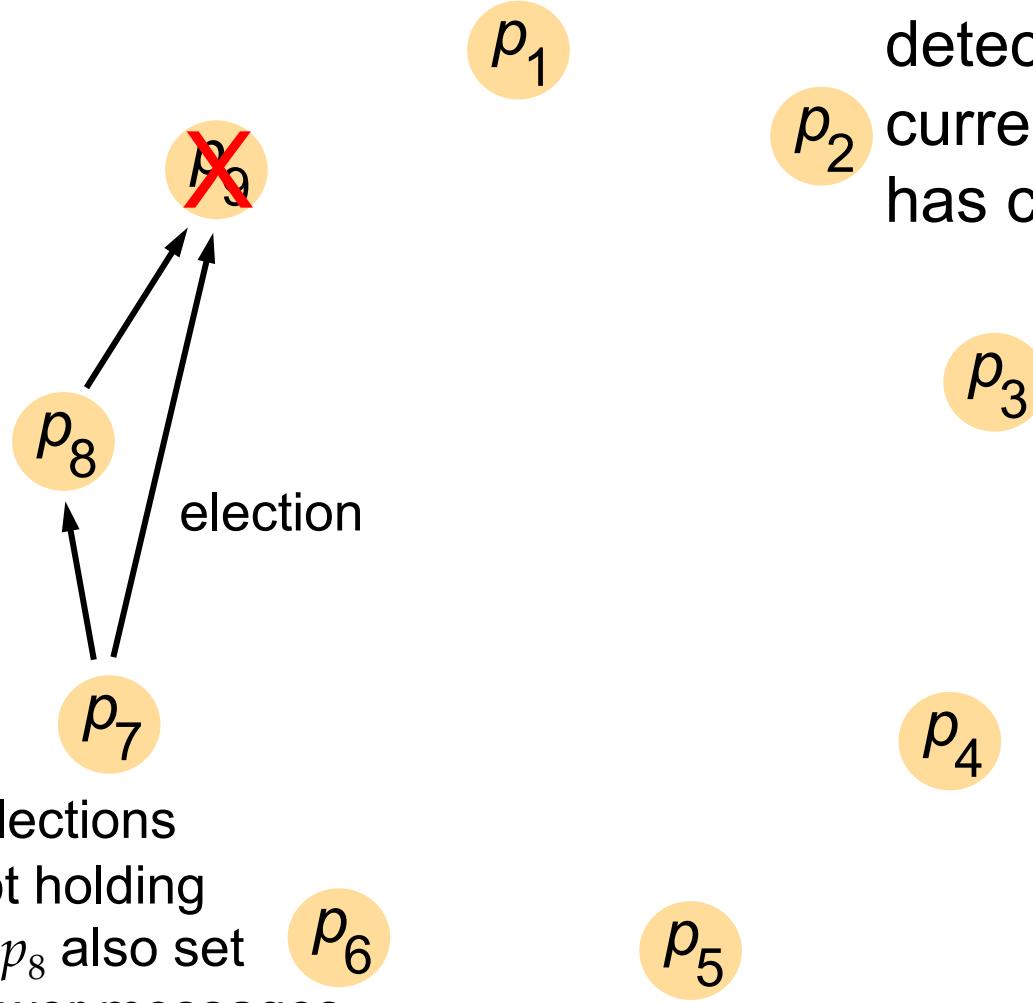
Example of Bully Algorithm

Example 1: if p_6 detects that p_9 (the current coordinator) has crashed



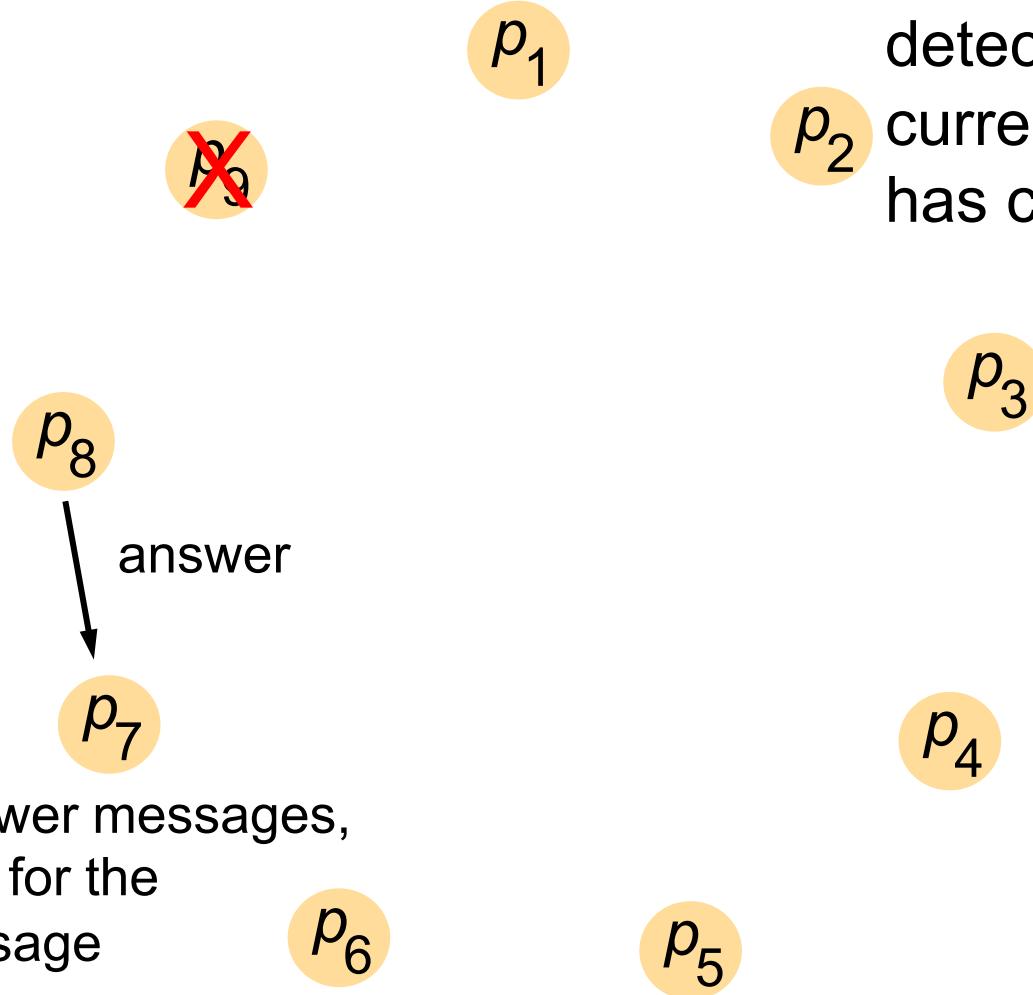
on receiving answer messages,
 p_6 sets a timeout for the coordinator message

Example of Bully Algorithm



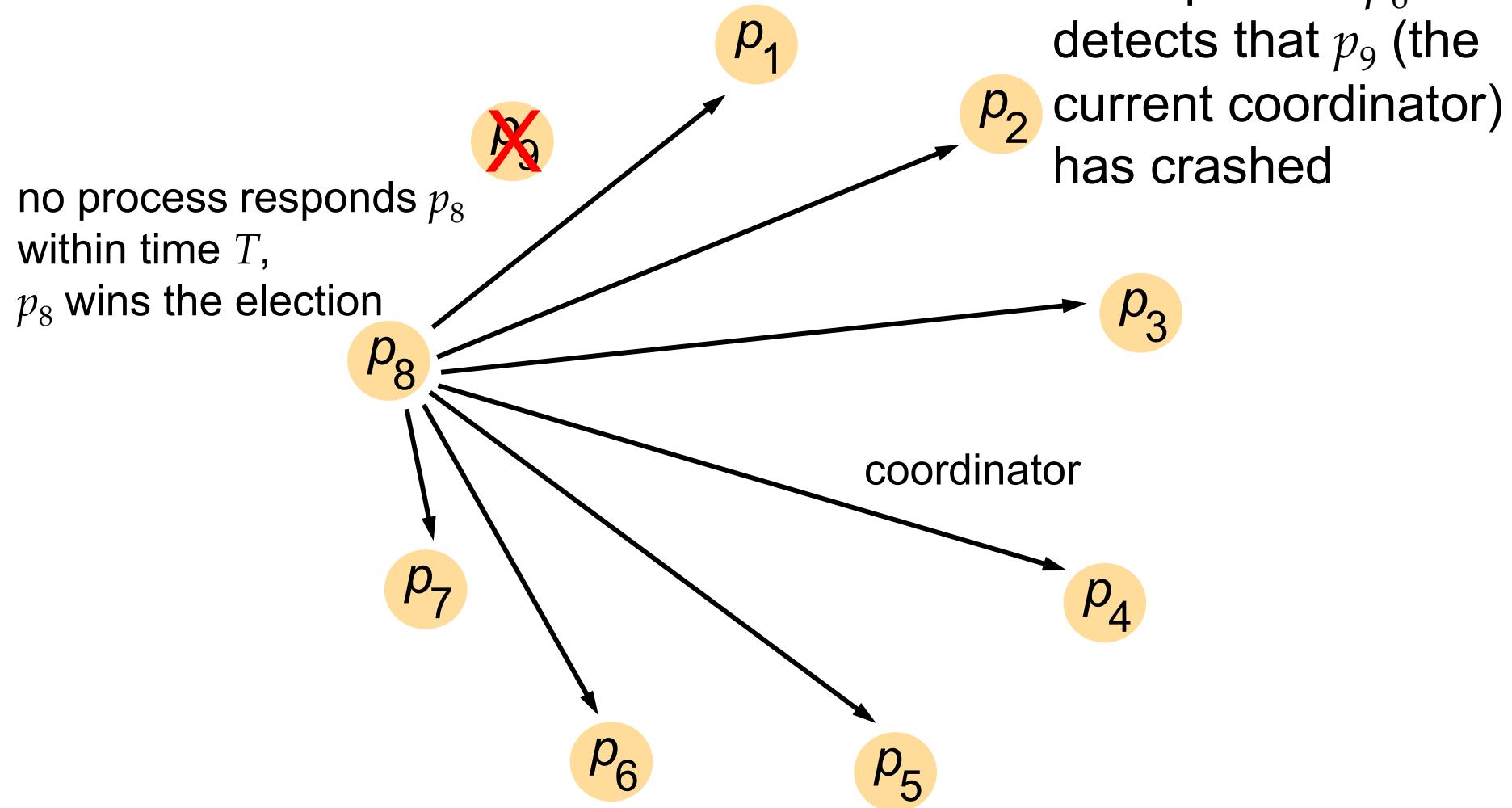
Example 1: if p_6 detects that p_9 (the current coordinator) has crashed

Example of Bully Algorithm

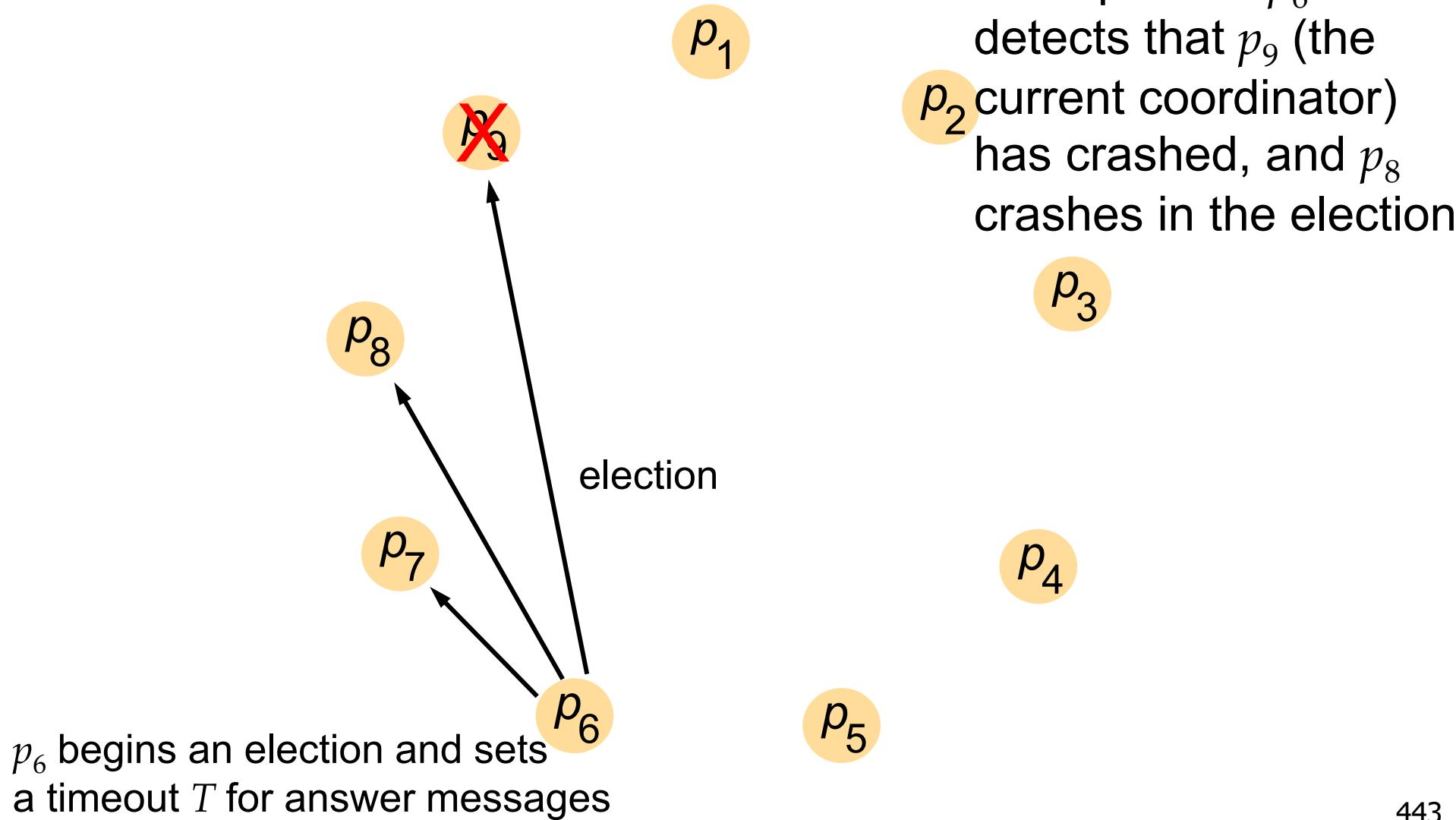


Example 1: if p_6 detects that p_9 (the current coordinator) has crashed

Example of Bully Algorithm

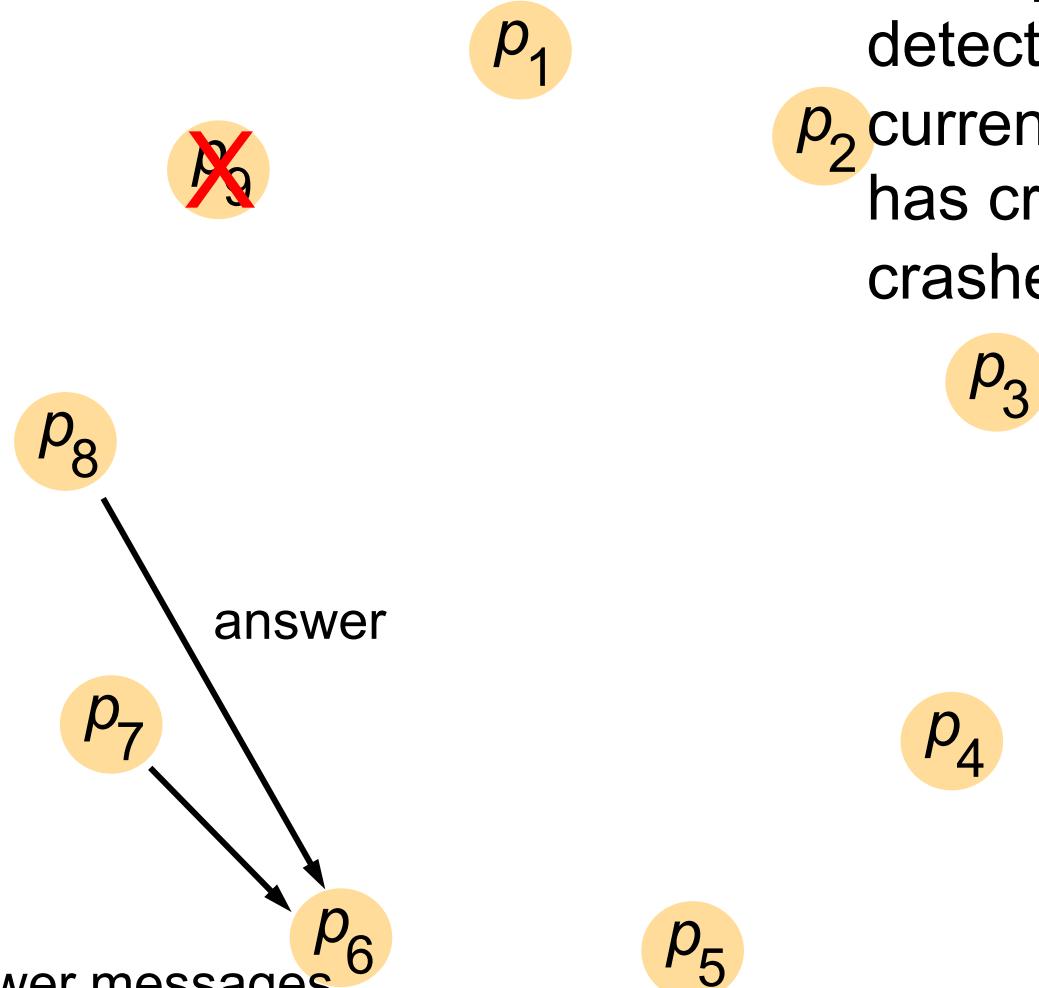


Example of Bully Algorithm



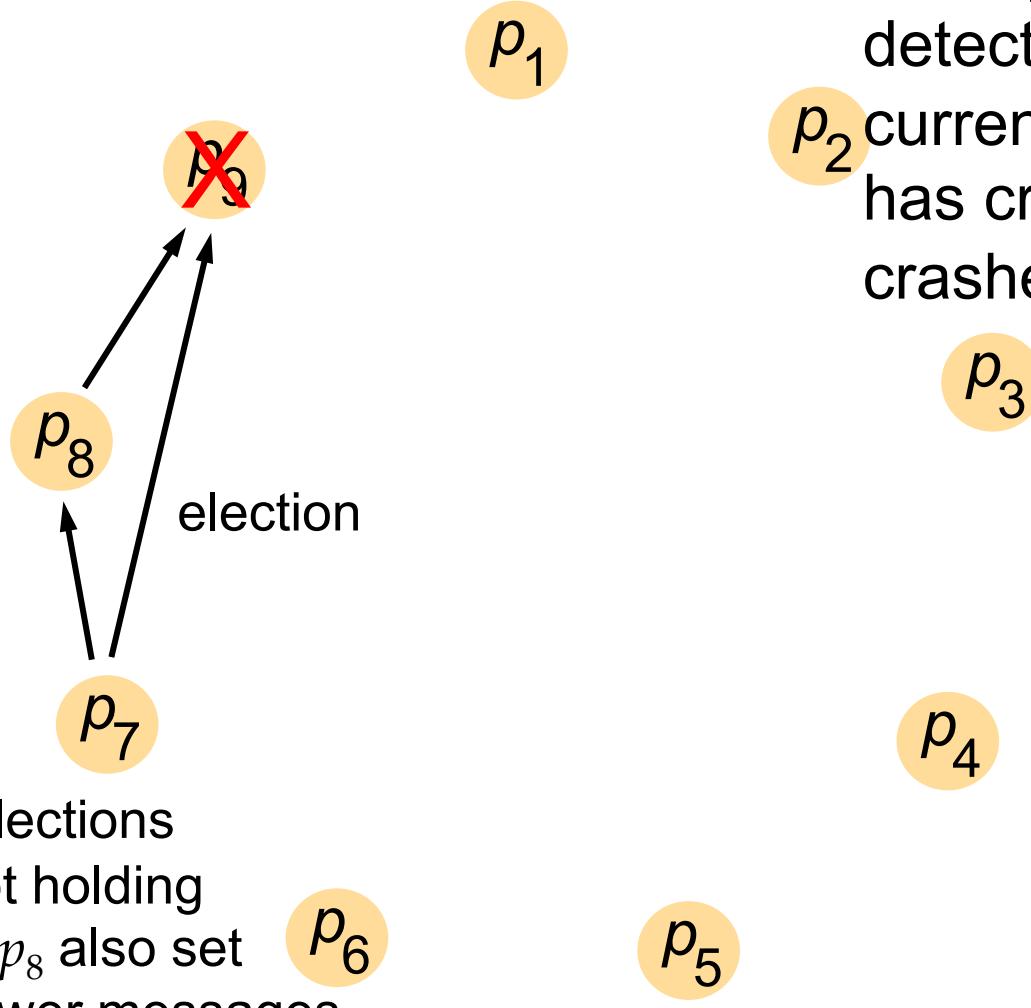
Example of Bully Algorithm

Example 2: if p_6 detects that p_9 (the current coordinator) has crashed, and p_8 crashes in the election



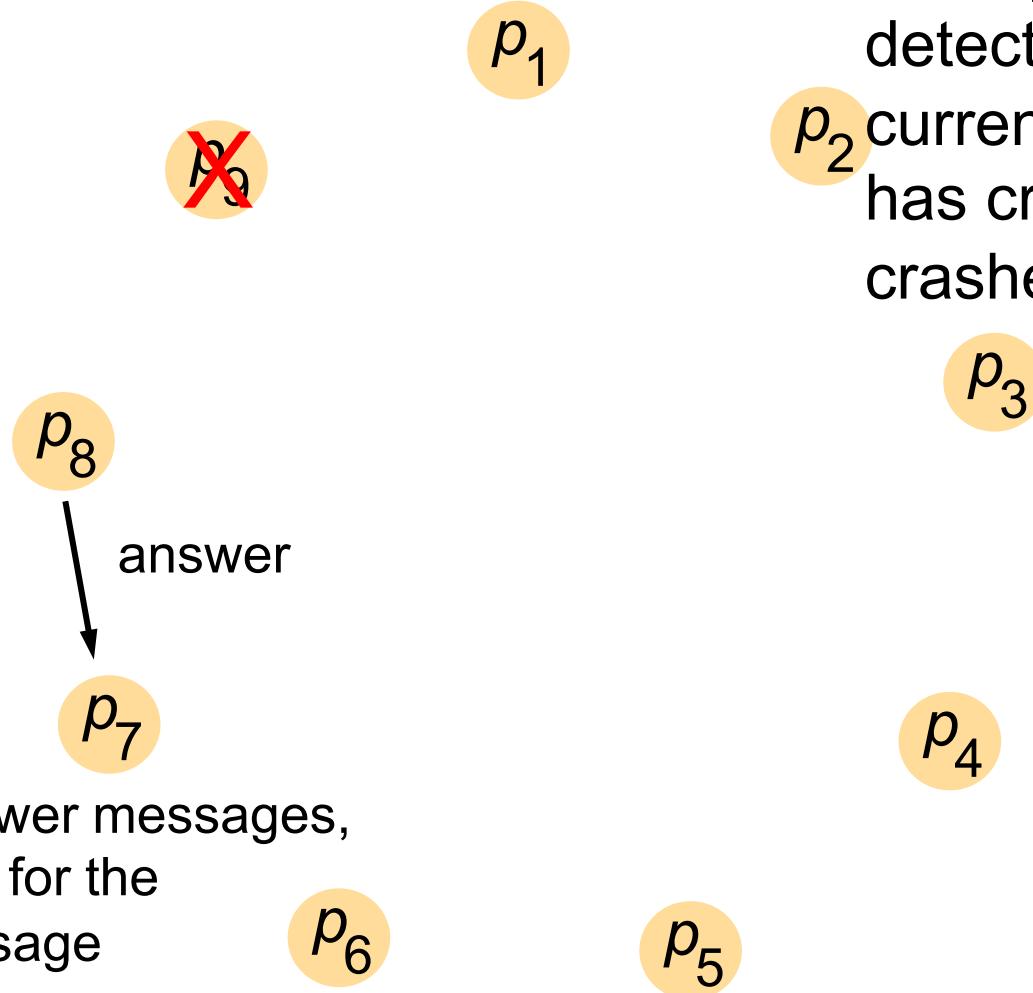
on receiving answer messages,
 p_6 sets a timeout for the coordinator message

Example of Bully Algorithm



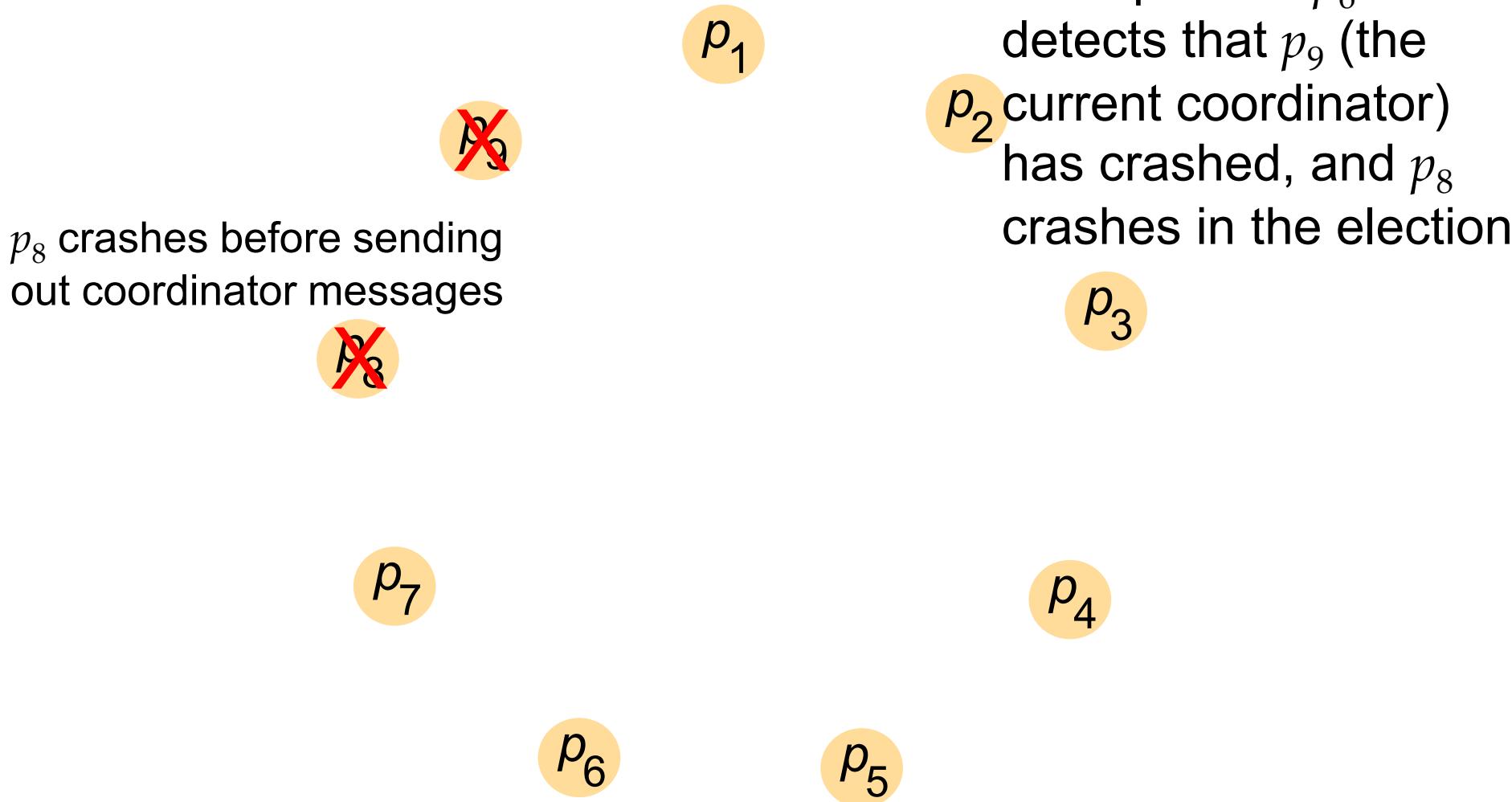
Example 2: if p_6 detects that p_9 (the current coordinator) has crashed, and p_8 crashes in the election

Example of Bully Algorithm

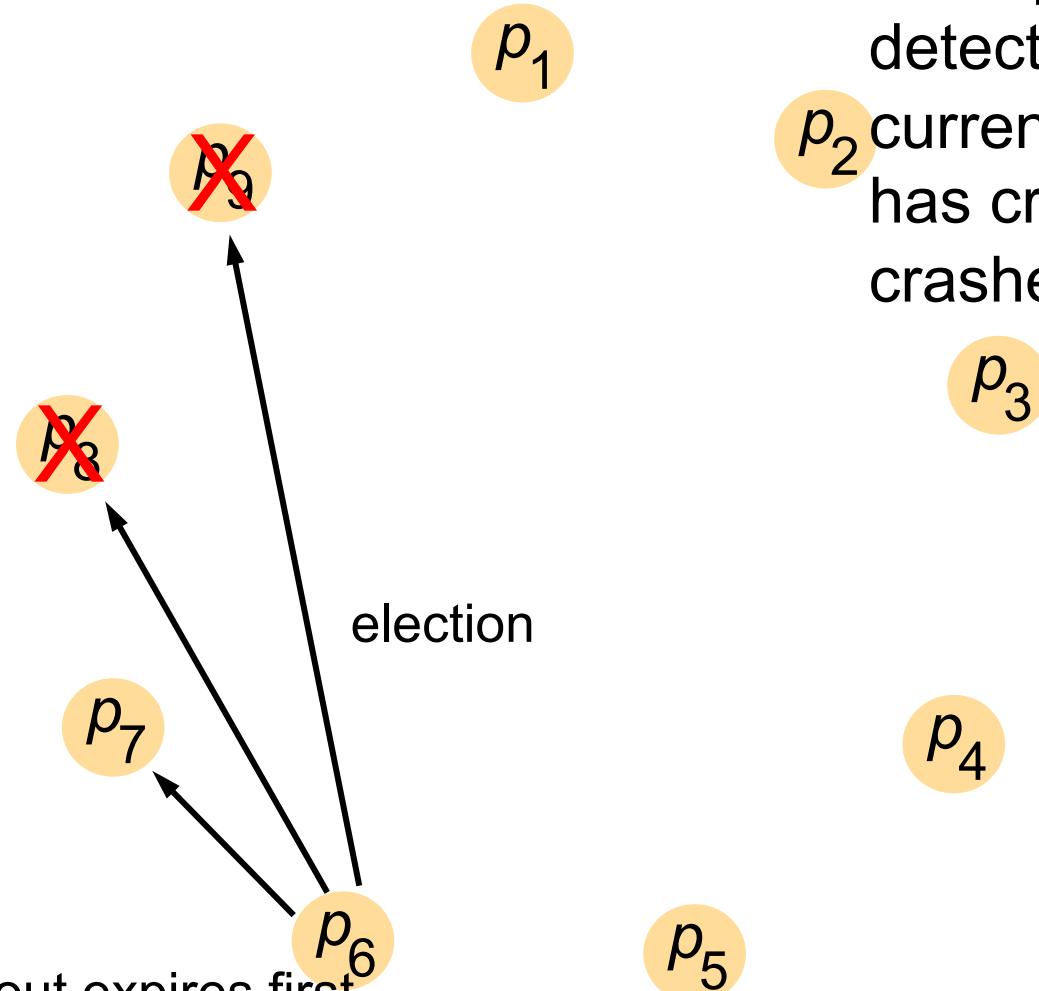


Example 2: if p_6 detects that p_9 (the current coordinator) has crashed, and p_8 crashes in the election

Example of Bully Algorithm



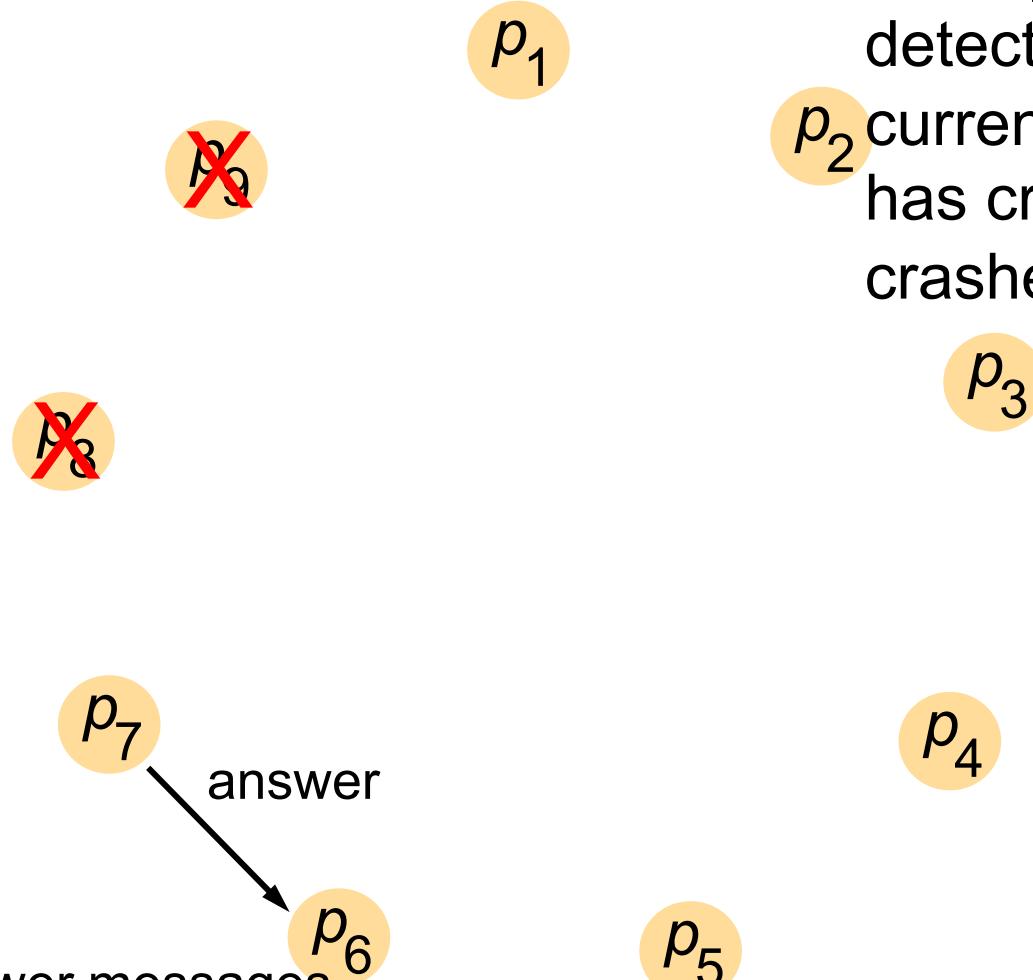
Example of Bully Algorithm



Example 2: if p_6 detects that p_9 (the current coordinator) has crashed, and p_8 crashes in the election

assume p_6 's timeout expires first,
 p_6 begins another election and sets a timeout T for answer messages

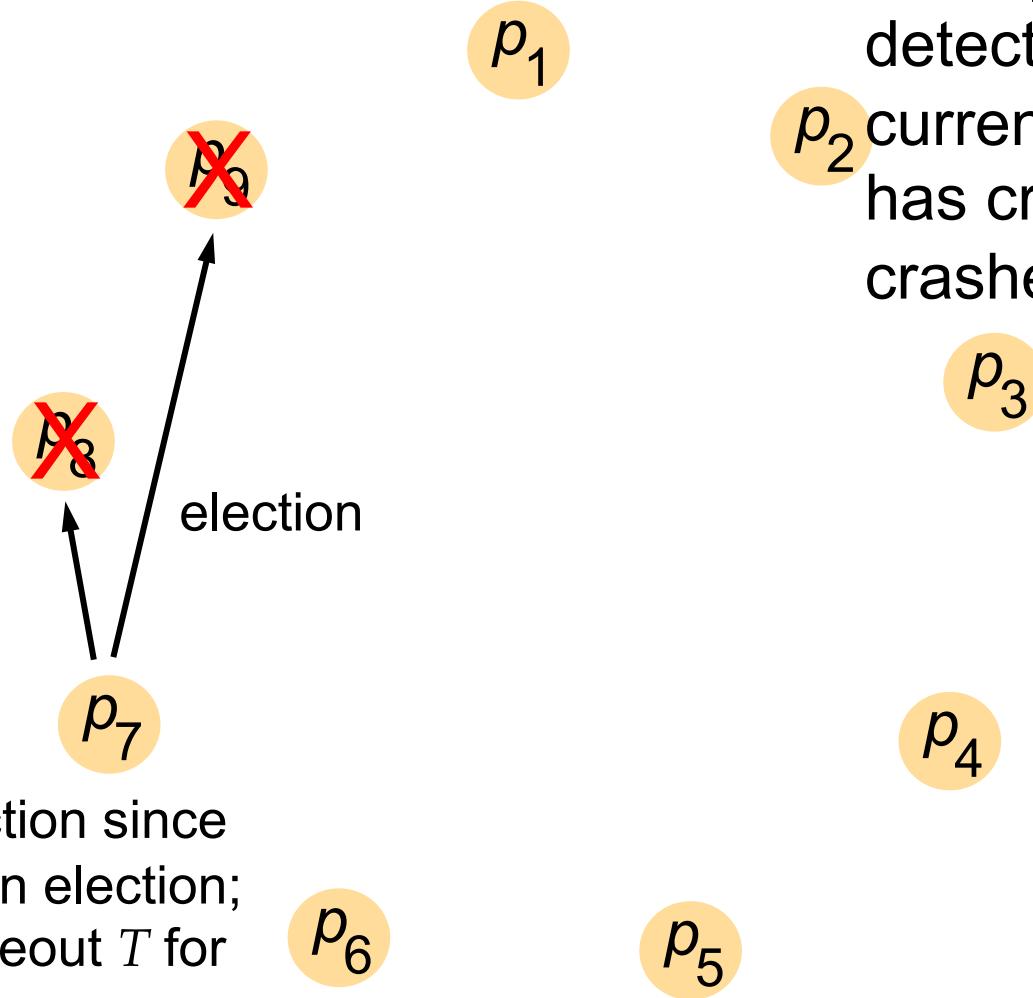
Example of Bully Algorithm



Example 2: if p_6 detects that p_9 (the current coordinator) has crashed, and p_8 crashes in the election

on receiving answer messages,
 p_6 sets a timeout for the coordinator message

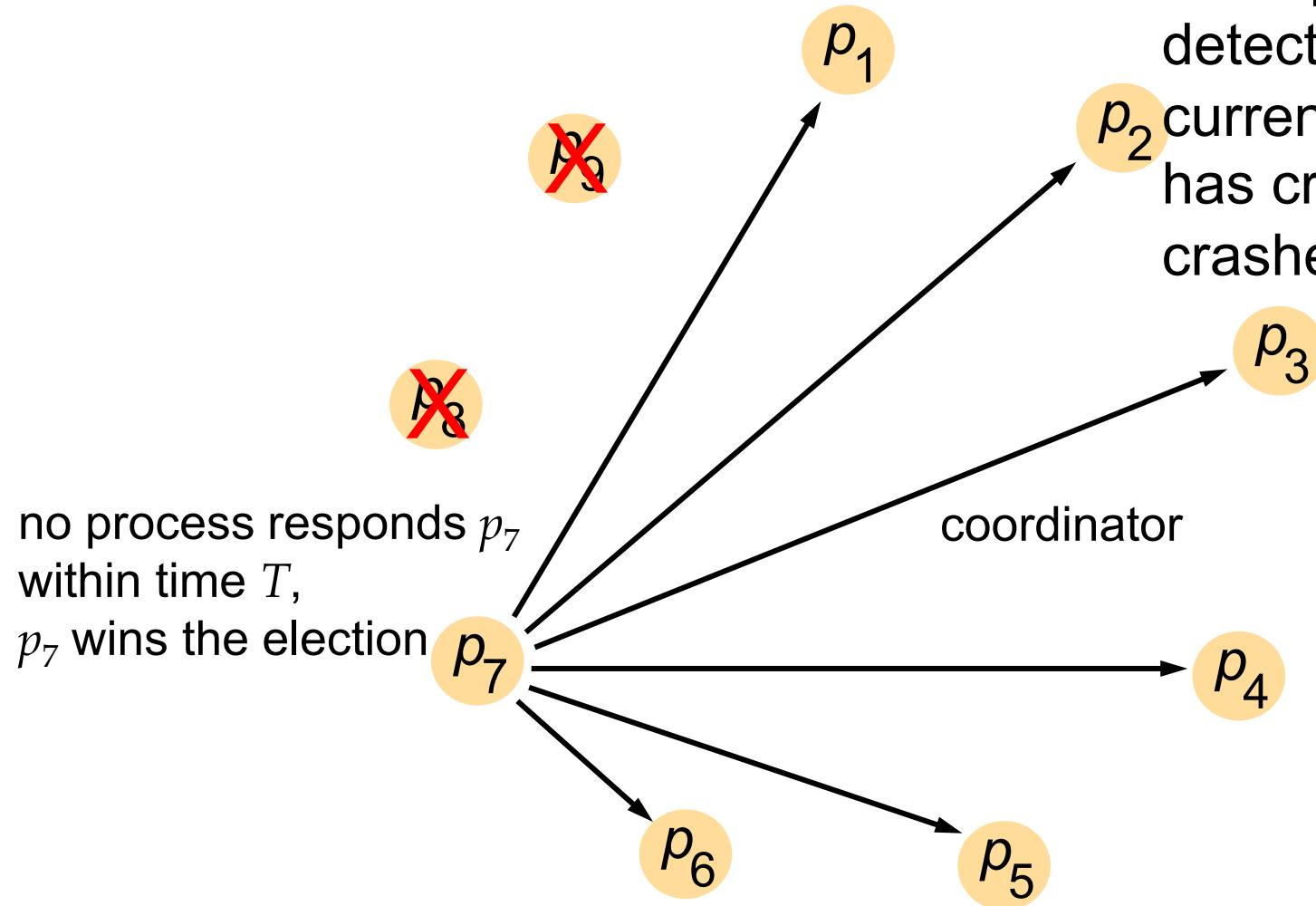
Example of Bully Algorithm



Example 2: if p_6 detects that p_9 (the current coordinator) has crashed, and p_8 crashes in the election

Example of Bully Algorithm

Example 2: if p_6 detects that p_9 (the current coordinator) has crashed, and p_8 crashes in the election



Analysis of Bully Algorithm

- Correctness
 - For any two processes, the one with smaller identifier will discover that the other exists and defer to it
 - → it is impossible for two processes to decide that they are both coordinators

Analysis of Bully Algorithm

- Bandwidth consumption
 - Assume the current coordinator (the process with the largest identifier) crashes and no more process crashes during the election
 - Best case: the process with the second largest identifier detects the crash → 1 election message + $(N-2)$ coordinator messages
 - Worst case: the process with the smallest identifier detects the crash, it sends $O(N)$ election messages, and then each of the other processes that are alive begins an election → $O(N^2)$ messages
- Turnaround time
 - Best case: T + one message transmission

Outline

- Distributed Mutual Exclusion
- Election
- Consensus Problem
- Summary

Consensus Problem

- Objective: for processes to **agree on a value** even in the presence of failures after one or more processes have proposed the value
 - In a transaction to transfer funds from one account to another, the processes involved must consistently agree to perform the respective debit and credit
 - In mutual exclusion, the processes agree on who can enter the critical section
 - In election, the processes agree on who is the coordinator
 - So, consensus problem is a generalization of mutual exclusion, election and many other problems

System Model

- A collection of N processes p_1, p_2, \dots, p_N that communicate by message passing
- Synchronous or asynchronous system
- Up to f of the N processes may fail
 - They may have crash failures or arbitrary failures

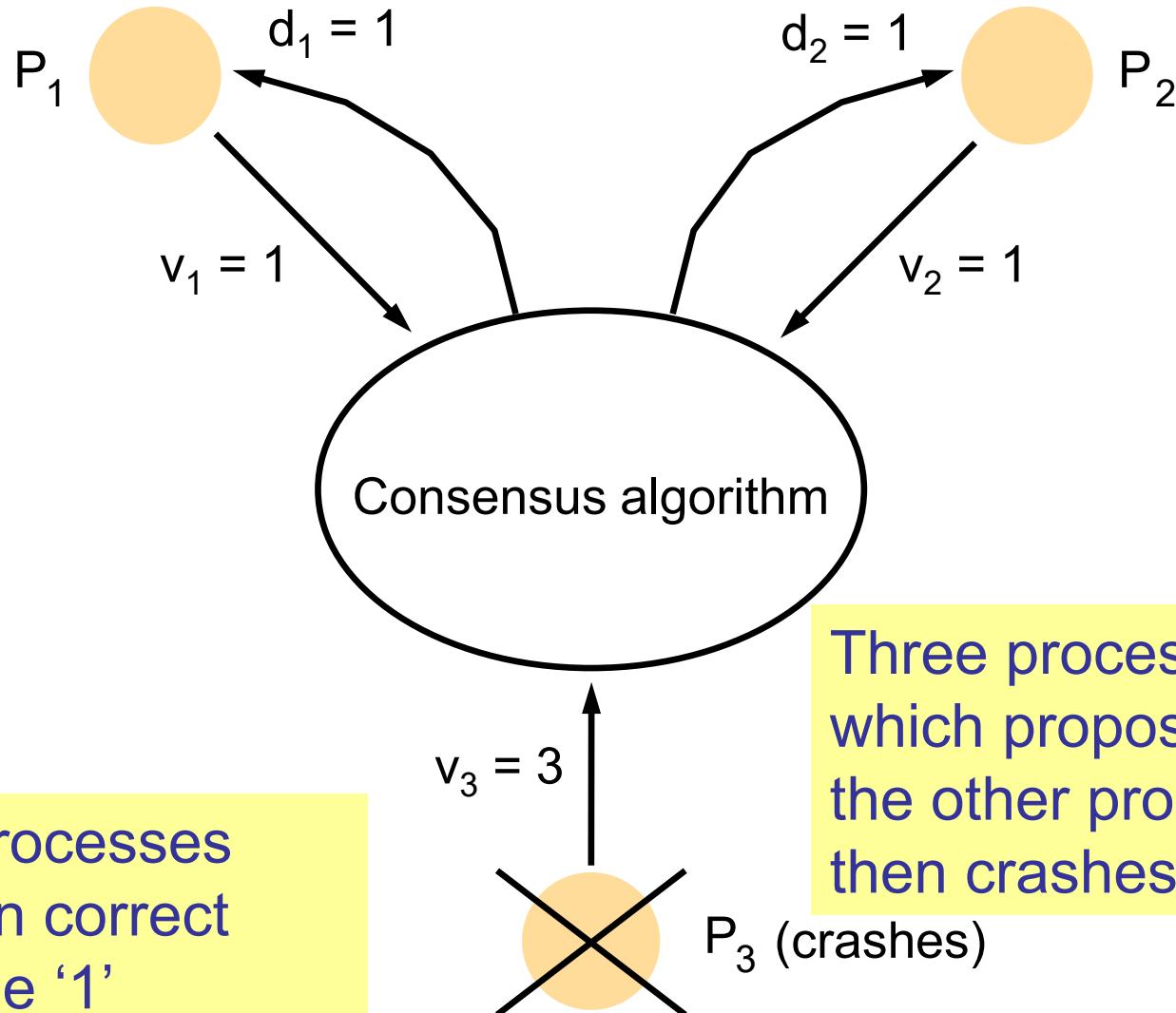
Consensus Problem

- Every process p_i begins in **the undecided state** and proposes a single value v_i
- Processes communicate with each other to exchange values
- Each process p_i then sets the value of a decision variable d_i and enters **the decided state** in which p_i no longer changes d_i

Requirements of Consensus Problem

- Termination
 - Eventually, each correct process sets its decision variable
- Agreement
 - The decision values of all correct processes are the same: for all i and j , if p_i and p_j are correct and have entered the decided state, then $d_i = d_j$
- Integrity
 - If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value

Example of Consensus Problem



Consensus in Synchronous System

- Assumptions
 - Synchronous system
 - Up to f of the N processes may crash (no arbitrary failure)
- In the absence of process crash
 - Each process p_i multicasts its proposed value v_i to **all other processes**
 - Each process waits until it has collected all N values (including its own) and then **evaluates a function** to set its decision variable
 - e.g., $\text{majority}(v_1, v_2, \dots, v_N)$, or $\text{minimum}(v_1, v_2, \dots, v_N)$, or $\text{maximum}(v_1, v_2, \dots, v_N)$

Consensus in Synchronous System

- What is the problem when processes crash?
 - A faulty process may crash in the middle of multicast
→ it may not actually send values to all other processes
 - So, how to guarantee that processes receive the same set of values in the presence of crashes?
- Solution: an algorithm proceeds in $f+1$ rounds
 - In each round, the correct processes multicast the values between themselves (each process multicasts the new values received for the first time in the previous round)
 - At the end of the last round, all correct processes that survive must have received the same set of values
 - Then, each correct process evaluates a function to set its decision variable

Consensus in Synchronous System

- Theoretical results: any algorithm to reach consensus despite up to f crash failures requires at least $f + 1$ rounds of message exchanges
- This lower bound also applies to the case of byzantine failures

Byzantine Generals (BG) Problem

- Every process p_i begins in **the undecided state**
- One process is called **commander**
- The commander proposes a value v that the others are to agree upon
- Processes communicate with each other to exchange values
- Each process p_i then sets the value of a decision variable d_i and enters **the decided state** in which p_i no longer changes d_i

Requirements of BG Problem

- Termination
 - Eventually, each correct process sets its decision variable
- Agreement
 - The decision values of all correct processes are the same: for all i and j , if p_i and p_j are correct and have entered the decided state, then $d_i = d_j$
- Integrity
 - If the commander is correct, then all correct processes decide on the value that the commander proposed

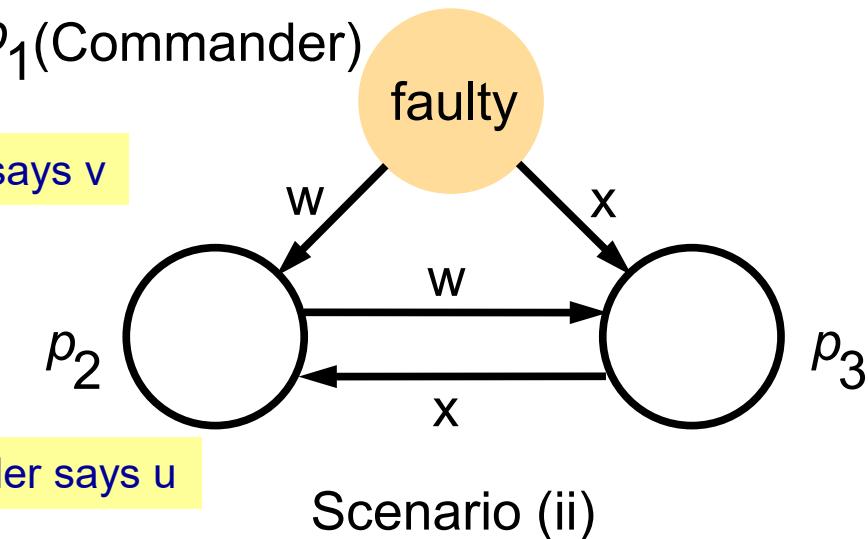
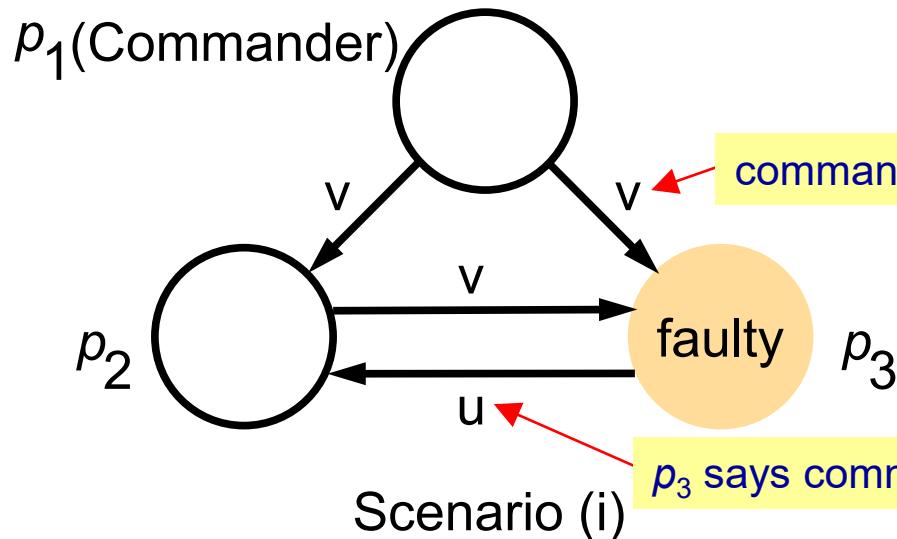
BG Problem in Synchronous System

- Assumptions
 - Synchronous system
 - Up to f of the N processes may exhibit arbitrary failures – they may send any message with any value at any time and may omit to send any message
- In the absence of arbitrary failure
 - The commander sends the proposed value v to all other processes
 - Each process waits until it receives the value from the commander and then sets its decision variable

BG Problem in Synchronous System

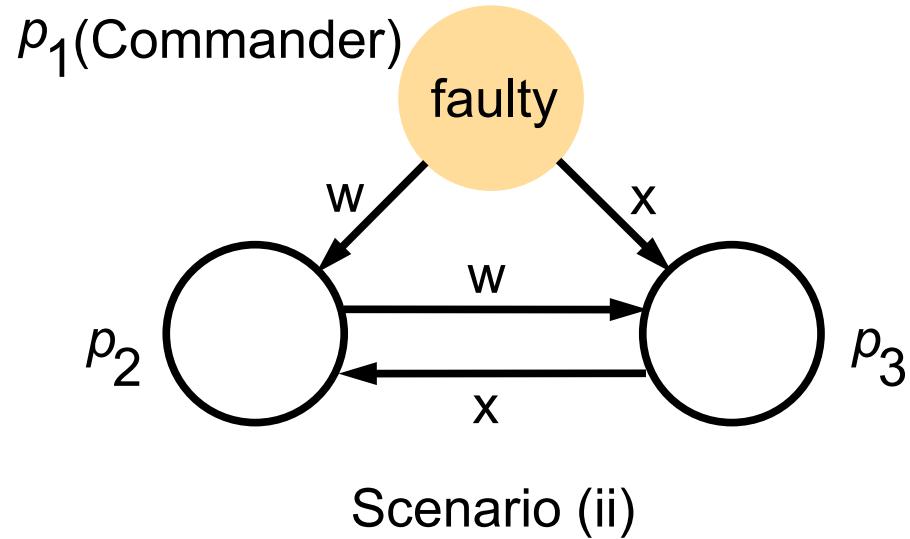
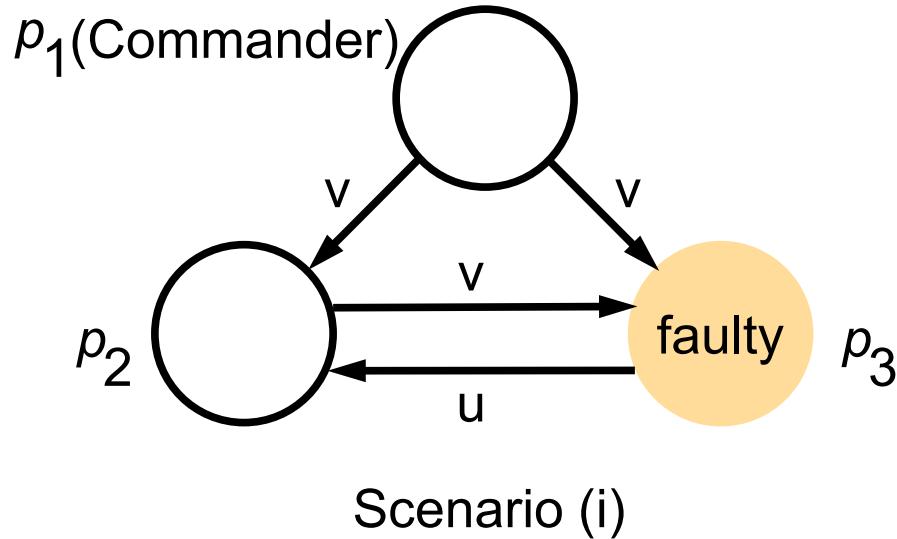
- What is the problem when there are arbitrary failures?
 - A faulty process (including the commander) may send a value to some processes and another value to the other processes → processes do not actually receive the same value
- Given 3 processes sending messages to one another, Lamport showed that there is no solution that guarantees to meet the requirements of the byzantine generals problem if one process is allowed to fail

Impossibility with Three Processes



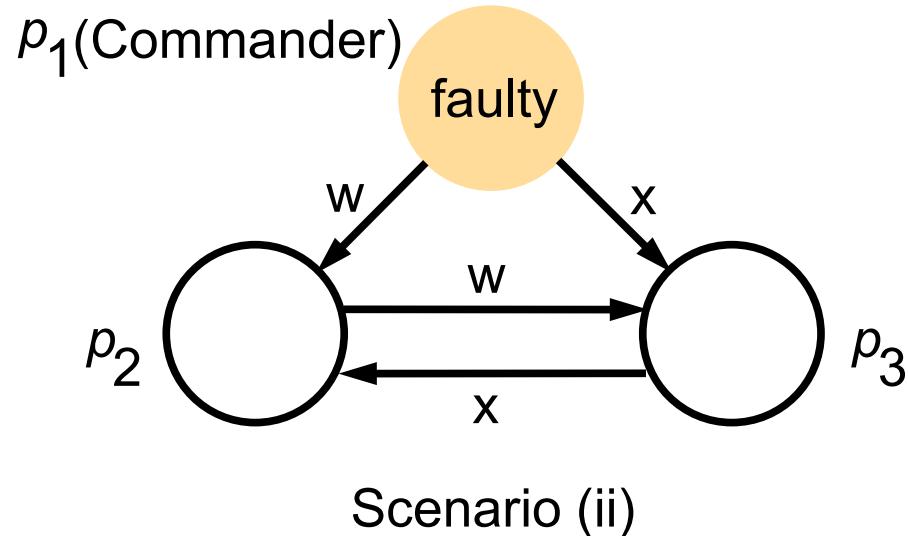
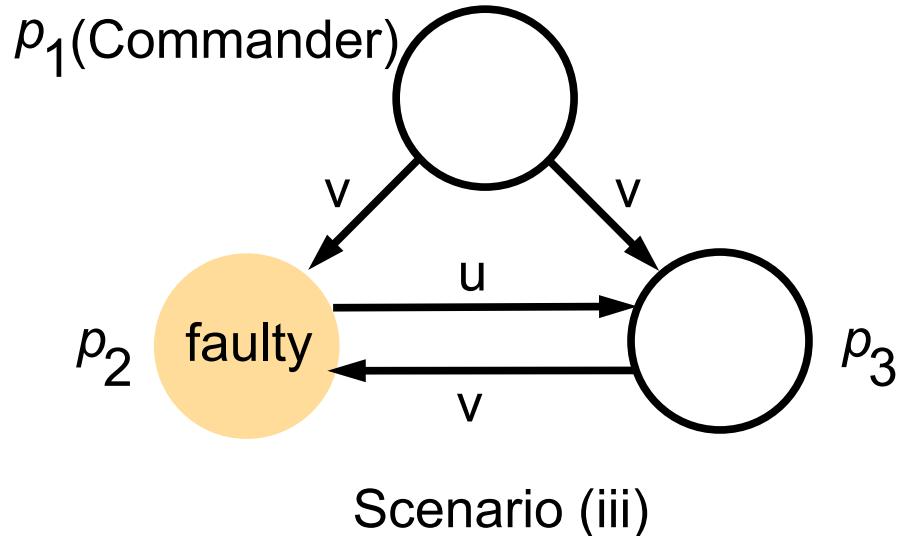
- In both scenarios (i) and (ii), p_2 receives two different values
- p_2 cannot distinguish between these two scenarios

Impossibility with Three Processes



- If there exists a correct solution A, due to the integrity requirement, p_2 must decide on value v in scenario (i) under A
- → p_2 must decide on value w in scenario (ii) under A

Impossibility with Three Processes



- Similarly, due to the integrity requirement, p_3 must decide on value v in scenario (iii) under A
- $\rightarrow p_3$ must decide on value x in scenario (ii) under A
- \rightarrow contradiction with agreement requirement
- \rightarrow no solution exists

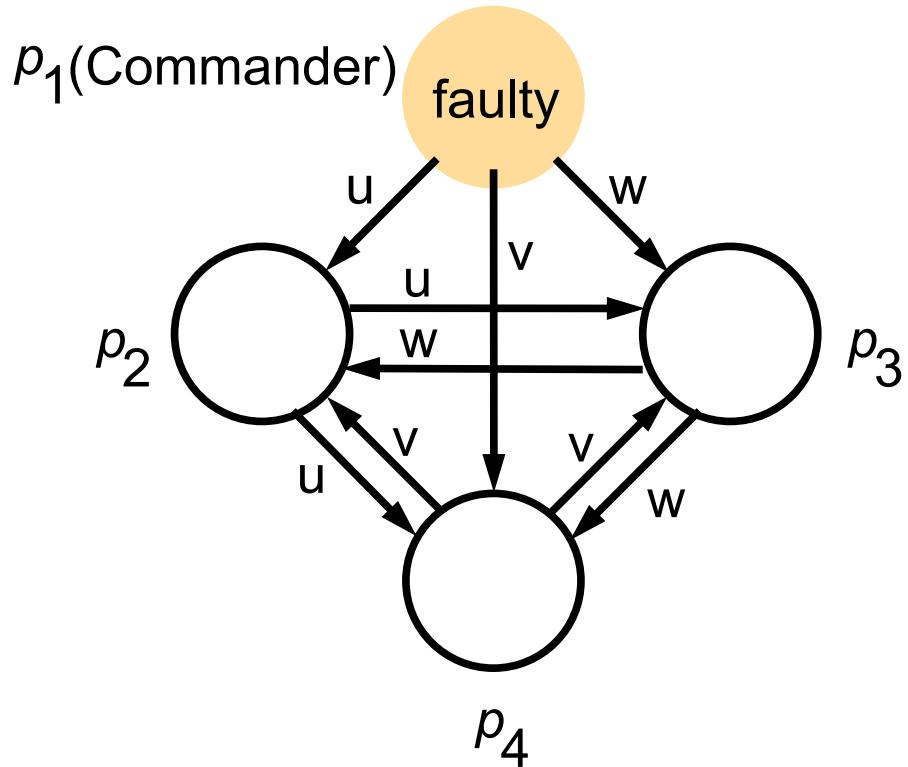
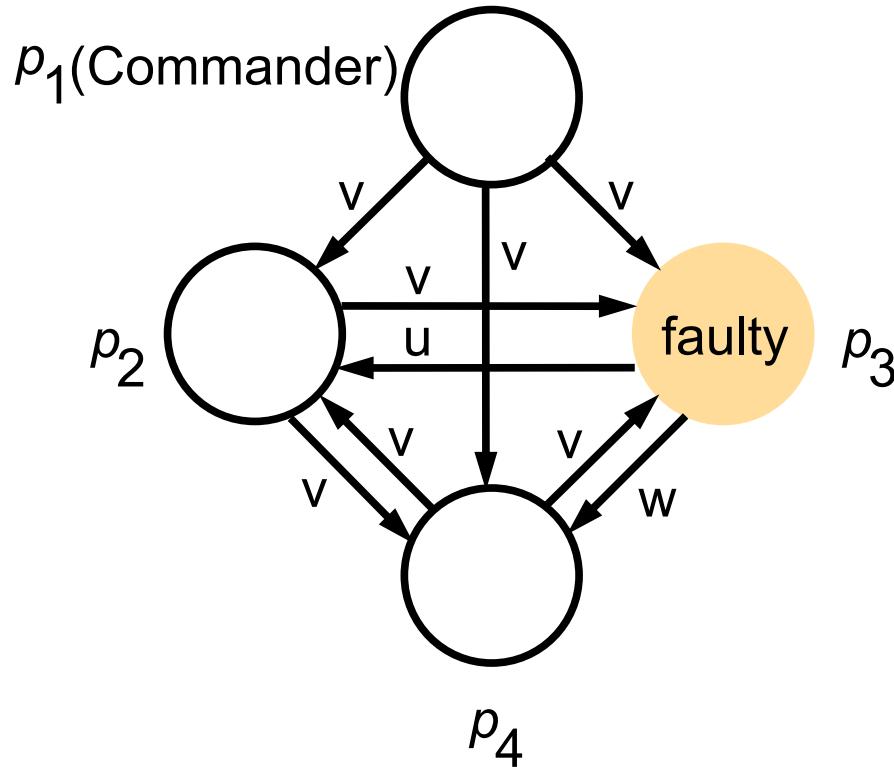
BG Problem in Synchronous System

- This is just a simplified illustration of impossibility with three processes, the complete proof is harder because processes can forward messages for indefinite number of rounds
- The conclusion can be extended – in general, no solution exists to the byzantine generals problem if $N \leq 3f$, and there exists a solution to the byzantine generals problem if $N > 3f$

BG Problem in Synchronous System

- Let's look at a solution when $N = 4$ and $f = 1$, i.e., there are 4 processes and at most 1 process can exhibit arbitrary failure
 - The algorithm proceeds in two rounds
 - In the first round, the commander sends the proposed value v to all other processes
 - In the second round, each non-commander process sends the value it received to all other processes
 - Each process then sets its decision variable by choosing the majority of the received values
 - If a faulty process omits to send a message in any round, a correct process proceeds as though the faulty process had sent it a special value \perp

Example of Four Processes



p_2 receives $\{v, u, v\}$ & decides on v
 p_4 receives $\{v, v, w\}$ & decides on v
So, agreement and integrity requirements are both satisfied

p_2 , p_3 , p_4 all receive $\{u, v, w\}$ and decide on \perp (no majority exists), agreement requirement is satisfied

Impossibility in Asynchronous System

- Studied solutions to consensus and byzantine generals problems in synchronous systems
 - Message exchanges take place in rounds (the length of a round is set to be longer than the maximum delay)
 - A process can be concluded to have crashed if no message is received from it within a round
- In asynchronous systems, processes can respond to messages at arbitrary times, so a crashed process is indistinguishable from a slow one
- Fischer proved that **no algorithm can guarantee to reach consensus in asynchronous systems even if there is only one faulty process**

Summary

- Distributed mutual exclusion
 - Central server algorithm
 - Ring-based algorithm
 - Ricart and Agrawala algorithm
- Election
 - Ring-based algorithm
 - Bully algorithm
- Consensus problem
 - Solution to consensus problem in synchronous system
 - Byzantine generals problem in synchronous system
 - Impossibility if $N \leq 3f$
 - Solution when $N > 3f$

9. Replication and Consistency

Outline

- **Introduction**
- Data-Centric Consistency Models and Protocols
- Client-Centric Consistency Models and Protocols
- Summary

Introduction

- Data replication: maintaining copies of data at multiple computers
- Motivations for replication
 - Improve performance and scalability
 - Increase reliability and fault-tolerance
- Price to pay: keeping all replicas consistent

Benefits of Replication

- Improve performance and scalability
 - Replication for performance is important when a distributed system needs to scale in numbers and geographical area
 - Performance can be improved by replicating the server and subsequently dividing the work
 - Example: several web servers can have the same domain name and the servers are selected in turn for the purpose of sharing load
 - By placing a copy of data in the proximity of the process using them, the time to access the data decreases

Benefits of Replication

- Increase reliability and fault tolerance
 - A service's availability is hindered by server failures
 - Replicate data at failure-independent servers
 - When one server crashes, clients may use another server
 - Example: primary and secondary servers in DNS
 - How to measure availability? – **the portion of time the service/system is available for use**
 - If each of n servers has an independent probability p of crashing, then the availability of an object stored in all of these servers is
$$1 - \text{Prob}(\text{all servers crash}) = 1 - p^n$$
e.g., $p = 10\%$, $n = 1$, availability = 90% $n = 2$, availability = 99% $n = 3$, availability = 99.9%

Benefits of Replication

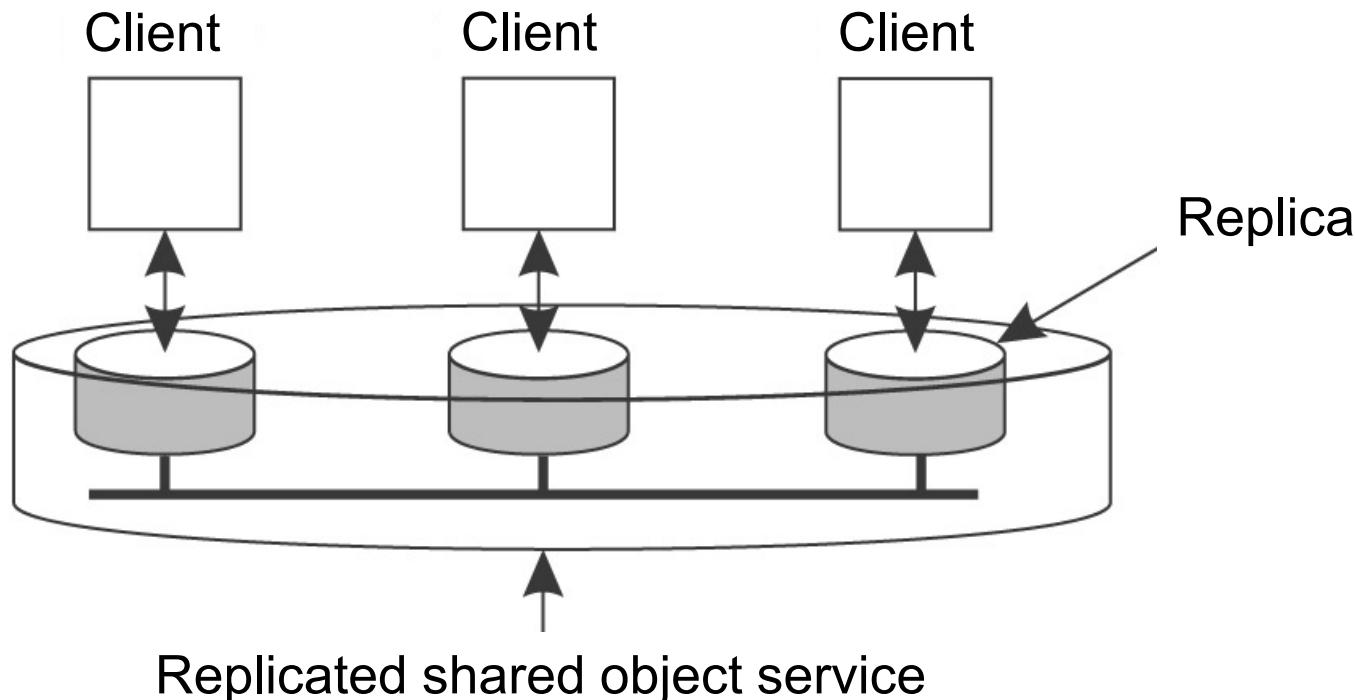
- Increase reliability and fault tolerance (cont'd)
 - Availability is also hindered by network partitions and disconnected operations
 - Users often copy heavily used data, such as the contents of a shared diary, from their usual environment to the laptops
 - They can then access the data on the laptops even if they are disconnected from the network
 - Problem – What if several users update the local copies of the shared diary on their laptops respectively? How to prevent a user from missing the updates by other people?

Problem with Replication

- Having multiple copies of data leads to consistency problems
 - Whenever a copy is modified, that copy becomes different from the rest
 - → Modifications have to be carried out on all copies to ensure consistency
 - When and how those modifications need to be carried out determines the price of replication
 - Example: web browsers cache web pages
 - To let the web server invalidate cached copies when a web page is updated, it requires the server to keep track of all browser caches and send them messages
→ this may degrade the performance of the server

System Model

- Each logical data object is implemented by a collection of physical copies called replicas



System Model

- Static vs. Dynamic
 - A **static system** contains a fixed set of replicas
 - In a **dynamic system**, replicas may be created or destroyed on the fly (not considered in our lecture)
- Transparency
 - Clients are not aware of multiple physical copies
 - Clients see logical objects only (i.e., a service that gives them access to logical objects) – access one logical object and receive a single result for each operation

System Model

- Data operations
 - **Read operation** – an operation that does not change the data
 - **Write operation** (or update operation) – an operation that changes the data
 - Operations are applied to the replicas **atomically**
- Possible failures
 - Crash (process omission failure)
 - Byzantine failure of processes
 - Network partition

System Model

- Consistency model
 - Normally, a read operation is expected to return a value that shows the result of the last write operation
 - But due to inevitable network delay, replicas are not necessarily consistent all the time
 - Some replicas may have received updates that are not yet conveyed to the others → different replicas are likely to have different values
- Consistency models
 - Define what is “correct” behaviour
 - Concern whether operations performed on a collection of replicated objects produce results that meet the correctness criteria for these logical objects

Outline

- Introduction
- Data-Centric Consistency Models and Protocols
- Client-Centric Consistency Models and Protocols
- Summary

Data-Centric Consistency Models

- Target scenario
 - Concurrent processes may simultaneously update data objects
- Objective
 - To provide a system wide consistent view of data objects in the face of such concurrency
- Focus
 - To agree on a consistent ordering of write operations

Strict Consistency

- Strict consistency
 - Any read operation on a data object returns a value corresponding to the result of the **most recent** write operation on the object
- All writes are instantaneously visible to all clients
- Examples in this section
 - A replicated shared object service hosting two objects x and y whose initial values are 0
 - Data operations are performed by multiple clients
 - $\text{read}(x) \rightarrow a$: a read operation on x returning a value a
 - $\text{write}(x, b)$: a write operation setting the value of x to b

Strict Consistency

- Example of a strictly consistent service

Client A	Client B	
	read(x) $\rightarrow 0$	
write($x, 1$)		=
write($y, 2$)		
	read(y) $\rightarrow 2$	

Interleaving
read(x) $\rightarrow 0$
write($x, 1$)
write($y, 2$)
read(y) $\rightarrow 2$

- Example of a service that is not strictly consistent

Client A	Client B	
write($x, 1$)		\neq
	read(y) $\rightarrow 0$	
	read(x) $\rightarrow 0$	
write($y, 2$)		

Interleaving
write($x, 1$)
read(y) $\rightarrow 0$
read(x) $\rightarrow 1$
write($y, 2$)

Strict Consistency

- Practicality
 - Strict consistency assumes the existence of absolute global time
 - In the absence of a global clock in distributed systems
 - It is difficult to assign a unique timestamp to each operation that corresponds to actual global time
 - It is difficult to define precisely which write operation is the “most recent”
 - A client updates an object at a replica at time T_1 ; slightly later, another client reads the object from a different replica at time T_2
 - If $T_2 - T_1$ is extremely small, is it possible for the second client to read the new value of the object?

Sequential Consistency

- Strict consistency may not be practical
- A weaker model is **sequential consistency**
- Sequential consistency
 - The result of any execution is the same as if the (read and write) operations by all clients were executed in some sequential order and the operations of each individual client appear in this sequence in the order specified by its program
 - Key points
 - All clients see **the same interleaving of operations**
 - The interleaving **follows the program order**

Interleaving

- Simply speaking, an interleaving is a serialized sequence of the operations by all clients
- Consider a replicated service with two clients
 - Client A performs operations O_{A0}, O_{A1}, O_{A2}
 - Client B performs operations O_{B0}, O_{B1}, O_{B2}
 - Example interleavings:
 - $O_{B0}, O_{B1}, O_{A0}, O_{B2}, O_{A1}, O_{A2}$
 - $O_{B2}, O_{A1}, O_{A2}, O_{B0}, O_{B1}, O_{A0}$

Interleaving

- Interleaving in program order
 - The order of operations issued by each client is preserved in the interleaving
 - Eligible interleaving may not be unique
 - Client A performs operations O_{A0}, O_{A1}, O_{A2}
 - Client B performs operations O_{B0}, O_{B1}, O_{B2}
 - Interleavings in program order:
 - $O_{B0}, O_{B1}, O_{A0}, O_{B2}, O_{A1}, O_{A2}$
 - $O_{A0}, O_{B0}, O_{B1}, O_{A1}, O_{A2}, O_{B2}$
 - Interleavings not in program order:
 - $O_{B2}, O_{A1}, O_{A2}, O_{B0}, O_{B1}, O_{A0}$
 - $O_{A1}, O_{A2}, O_{B0}, O_{B1}, O_{A0}, O_{B2}$

Sequential Consistency

- Example of a sequentially consistent service

Client A	Client B		Interleaving
write($x, 1$)		=	read(y) $\rightarrow 0$
	read(y) $\rightarrow 0$	=	read(x) $\rightarrow 0$
	read(x) $\rightarrow 0$		write($x, 1$)
write($y, 2$)			write($y, 2$)

Sequential Consistency

- Example of a not sequentially consistent service

Client A	Client B
write($x,1$)	
write($y,2$)	
	read($y \rightarrow 2$)
	read($x \rightarrow 0$)

For any interleaving in program order, write($x,1$) precedes write($y,2$), and read(y) precedes read(x).

If there exists an interleaving showing sequential consistency, since read(y) returns 2, write($y,2$) must precede read(y) in the interleaving. Thus, write($x,1$) precedes read(x). However, this is impossible since read(x) returns 0 in the replicated service.

Sequential Consistency

- Any interleaving of read and write operations in program order is acceptable, but all processes see the same interleaving of operations
- **Any strictly consistent service is sequentially consistent**
 - How to prove it?

Causal Consistency

- Causal consistency is an even weaker model that makes a distinction between events that are potentially causally related and those that are not
- Causal relation
 - Operations by a single client are causally related based on the order in which the client executed them
 - A read is causally related to the write that provided the data the read got
 - Not causally related → concurrent

Causal Consistency

- Example of causal relation

Client A	Client B	Client C	Client D
write($x, 1$)			
	write($x, 2$)	read($x \rightarrow 2$)	
		read($x \rightarrow 1$)	read($x \rightarrow 1$)
			read($x \rightarrow 2$)

The diagram illustrates causal consistency through a grid of client actions. Red curved arrows show dependencies between operations:

- A curved arrow from Client A's $\text{write}(x, 1)$ to Client B's $\text{write}(x, 2)$.
- Two curved arrows from Client B's $\text{write}(x, 2)$ to Client C's $\text{read}(x \rightarrow 2)$.
- One curved arrow from Client C's $\text{read}(x \rightarrow 2)$ to Client D's $\text{read}(x \rightarrow 1)$.

Causal Consistency

- Causal consistency
 - Writes that are potentially causally related must be seen by all clients in the same order
 - Concurrent writes may be seen in different orders by different clients
- For each client, find a valid interleaving of the read operations of this client and the write operations from all clients

Causal Consistency

- Example of a causally consistent service

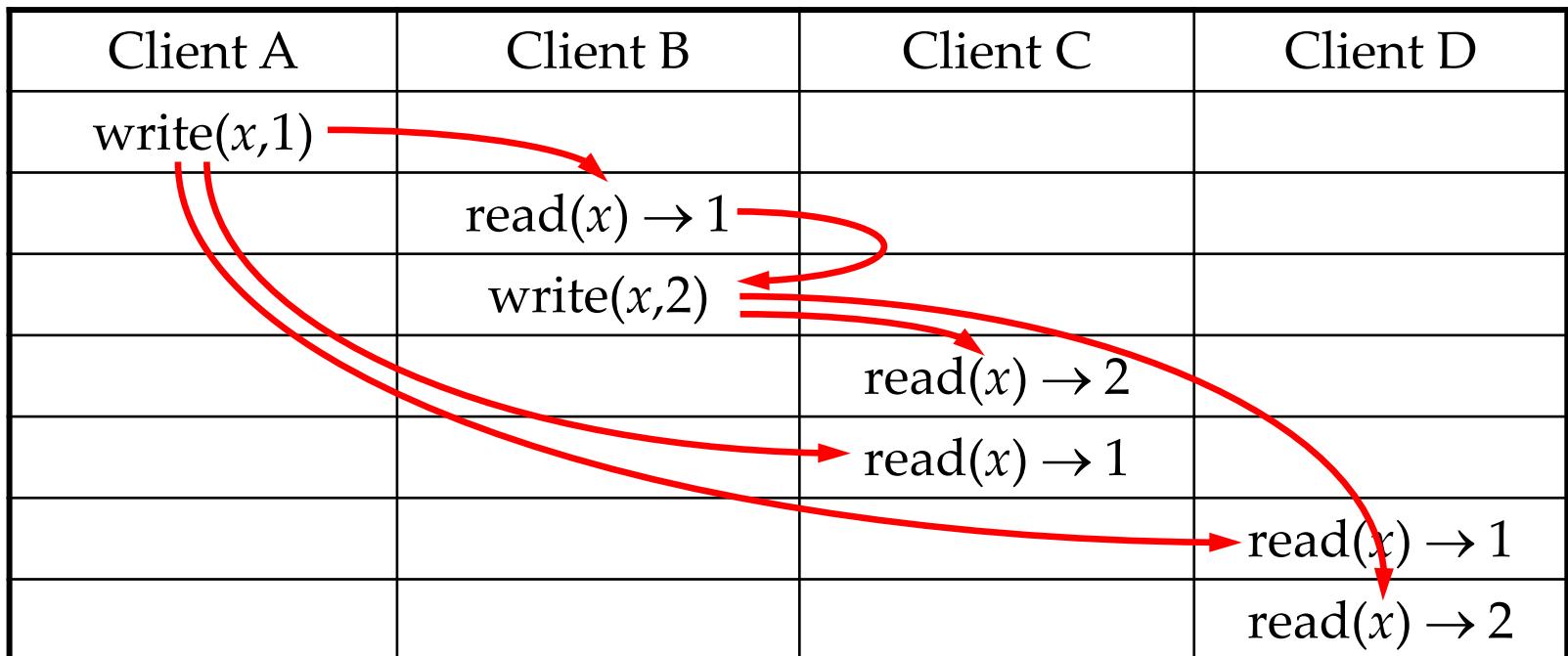
Client A	Client B	Client C	Client D
write($x, 1$)			Client D sees: write($x, 1$) read(x) $\rightarrow 1$
	write($x, 2$)		write($x, 2$) read(x) $\rightarrow 2$
		Client C sees: write($x, 2$) read(x) $\rightarrow 2$ write($x, 1$) read(x) $\rightarrow 1$	read(x) $\rightarrow 2$
			read(x) $\rightarrow 1$
			read(x) $\rightarrow 2$

write($x, 1$) and write($x, 2$) are concurrent \rightarrow it is not required that all clients see them in the same order.

However, the service is not sequentially consistent. Why?

Causal Consistency

- Example of a not causally consistent service



write($x, 1$) and write($x, 2$) are potentially causally related
(write($x, 2$) may depend on read(x) which in turn depends on
write($x, 1$)) \rightarrow a required interleaving for client C cannot be found

FIFO Consistency

- FIFO consistency
 - Writes done by a single client are seen by all other clients in the order in which they were issued
 - Writes from different clients may be seen in different orders by different clients
- For each client, find a valid interleaving of the read operations of this client and the write operations from all clients

FIFO Consistency

- Example of an FIFO consistent service

Client A	Client B	Client C	Client D
write($x, 1$)			
	read(x) $\rightarrow 1$		
	write($x, 2$)		
		Client C sees: write($x, 2$) read(x) $\rightarrow 2$ write($x, 1$) read(x) $\rightarrow 1$	
		read(x) $\rightarrow 2$	
		read(x) $\rightarrow 1$	
			Client D sees: write($x, 1$) read(x) $\rightarrow 1$ write($x, 2$) read(x) $\rightarrow 2$
			read(x) $\rightarrow 1$
			read(x) $\rightarrow 2$

write($x, 1$) and write($x, 2$) are from different clients \rightarrow they may be seen in different orders by different clients.

FIFO Consistency

- Example of a not FIFO consistent service

Client A	Client B	Client C
write($x, 1$)		
write($x, 2$)		
	read($x \rightarrow 2$)	
	read($x \rightarrow 1$)	
		read($x \rightarrow 1$)
		read($x \rightarrow 2$)

write($x, 1$) and write($x, 2$) are from the same client → they are required to be seen by all other clients in their issuing order → a required interleaving for client B cannot be found.

Comparison

- Strict consistency
 - Most restrictive and stringent – absolute time ordering of all operations
- Sequential consistency
 - All clients see all operations in the same order, but the operations may not be ordered in absolute time
- Causal consistency and FIFO consistency
 - Different clients may see operations in different orders
 - Causal consistency – all clients see causally-related operations in the same order
 - FIFO consistency – all clients see writes from each other in the order they were issued

Consistency Protocols

- A consistency protocol describes an actual implementation of a specific consistency model
- We will also look at fault tolerant aspects
 - Does it handle faults?
 - What type of faults does it handle?
 - How many faults can it tolerate?

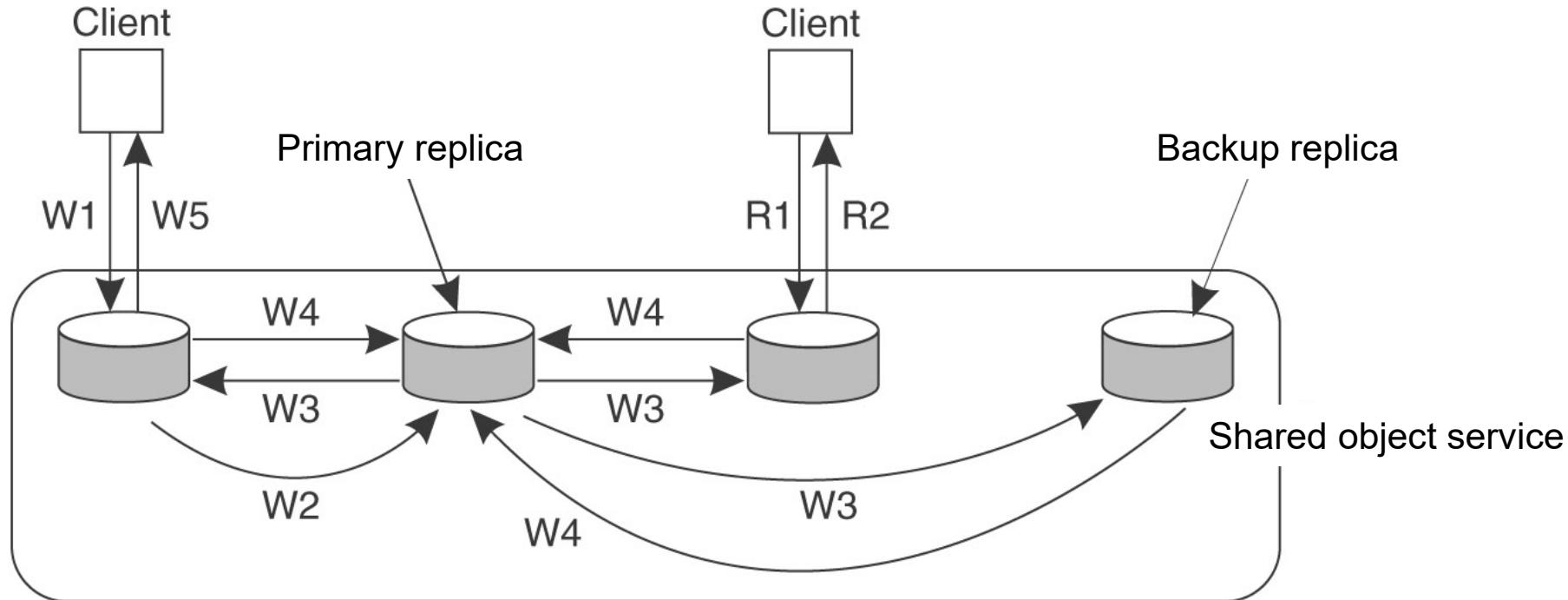
Primary-Based Protocols

- For each data object, one replica is designated as the **primary replica** and the remaining replicas are called **backups** (slaves or secondary replicas)
 - Known as passive replication
 - The primary replica is responsible for coordinating write operations on the object
 - If the primary replica crashes, one of the backups is elected to act as the primary replica
- Primary-based protocols
 - Remote-write protocol: the primary replica is fixed
 - Local-write protocol: the primary replica can be moved for write operations to be carried out locally

Remote-Write Protocol

- Write operations are forwarded to the primary replica
 - The primary replica performs the update and then forwards the update to the backup replicas
 - Each backup replica performs the update as well and sends an acknowledgement back to the primary replica
 - Finally, the primary replica sends an acknowledgment back to the initiating client
 - Write operations are blocking: a client initiating a write operation is not allowed to continue until it receives an acknowledgement from the primary replica
- Read operations can be carried out at any replica

Remote-Write Protocol



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

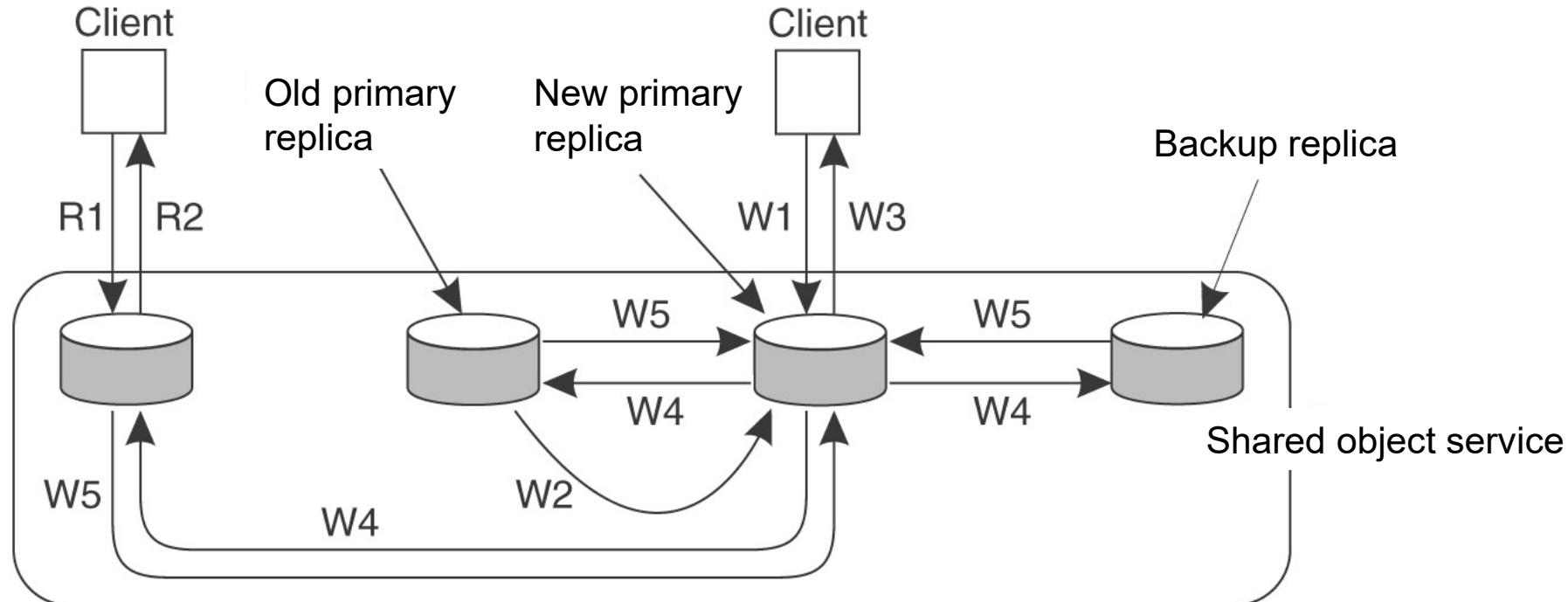
Remote-Write Protocol

- A straightforward implementation of sequential consistency
 - The primary replica orders all write operations in a globally unique order
 - So, all clients see all write operations in the same order, no matter which backup replica they use to perform read operations
 - Also, due to blocking, clients will always see the effects of their most recent write operations

Local-Write Protocol

- The primary migrates between replicas that wish to perform a write operation
- Write operations can be non-blocking
 - As soon as the primary replica has updated its local copy, it returns an acknowledgement to the client
 - After that, the primary replica tells the backup replicas to perform the update as well
- Advantages:
 - Multiple successive write operations can be carried out locally
 - Enable mobile clients to operate in disconnected mode

Local-Write Protocol



W1. Write request

W2. Move primary replica

W3. Acknowledge write completed

W4. Tell backups to update

W5. Acknowledge update

R1. Read request

R2. Response to read

Passive Replication

- Fault tolerance
 - It can handle crashes
 - To tolerate up to f process crashes, $f + 1$ replicas are required
 - It cannot deal with byzantine failures because only one replica executes the operations

Replicated-Write Protocols

- Write operations can be carried out at multiple replicas instead of only one
- Active replication
- Gifford's quorum-based protocol

Active Replication

- All replicas play **the same role**
- Write operations are carried out at all replicas
 - Operations are sent to all replicas via **totally ordered multicast** (which delivers operations to all replicas in the same total order)
 - All replicas start in the same state and **perform the same operations in the same order** → their states remain identical
 - Write operations are blocking: a client initiating a write operation is not allowed to continue until it receives acknowledgements from all replicas
 - If a replica crashes, no action is needed by other replicas
- Read operations can be carried out at any replica

Why is Ordering Important?

Balance = \$1000

Receive ‘deposit \$100’

Balance = \$1100

Receive ‘add 1% interest’

Balance = \$1111

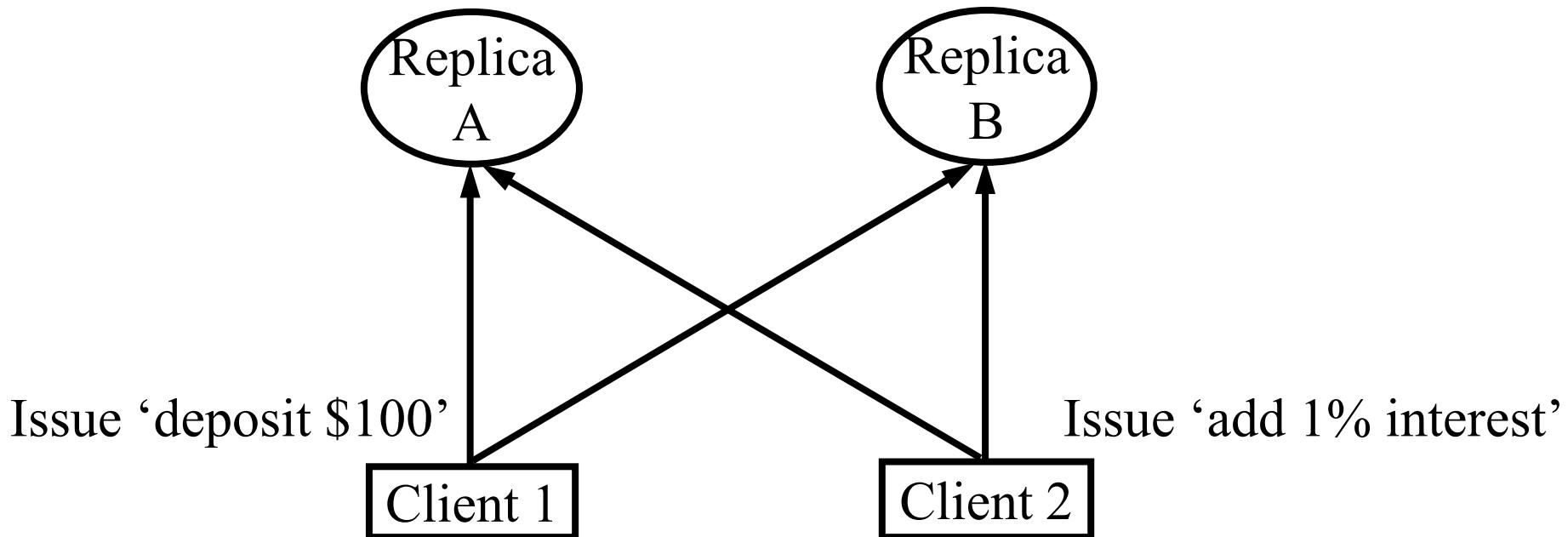
Balance = \$1000

Receive ‘add 1% interest’

Balance = \$1010

Receive ‘deposit \$100’

Balance = \$1110



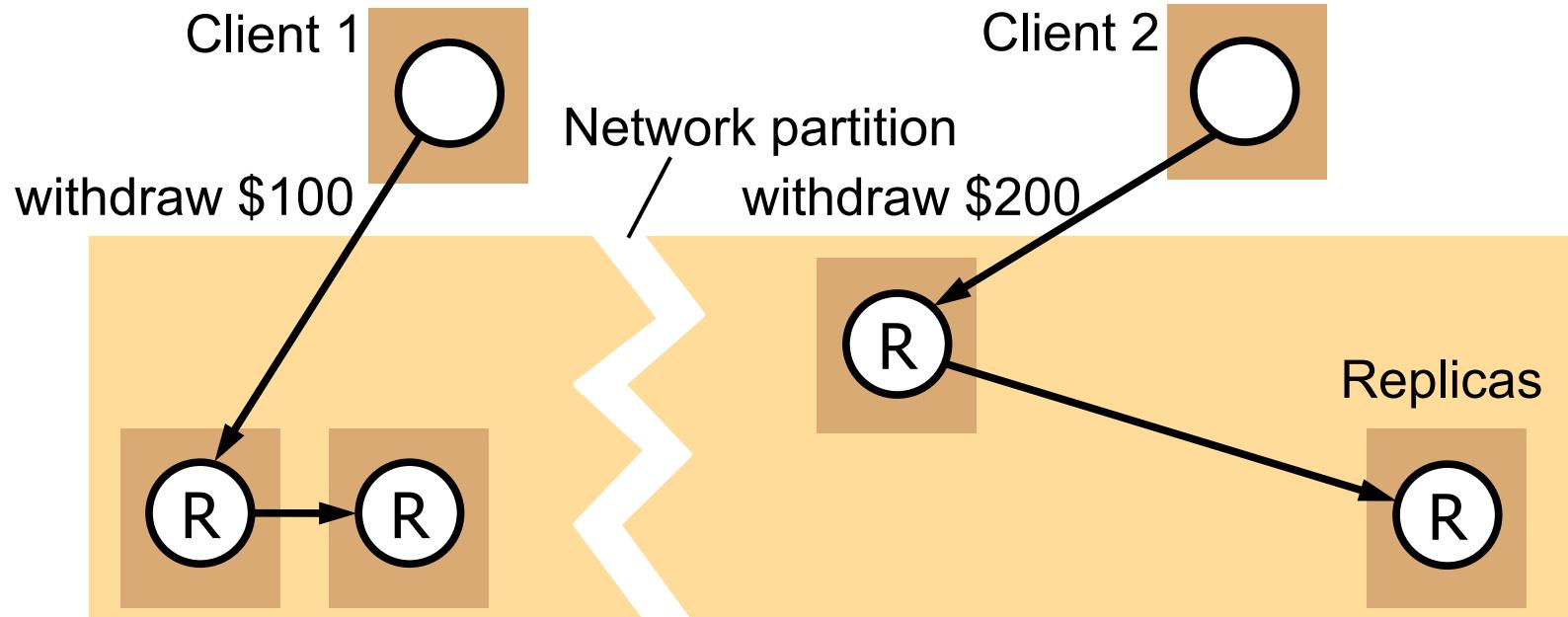
Active Replication

- Active replication achieves sequential consistency
 - Totally ordered multicast orders all write operations
- Fault tolerance
 - To tolerate crashes, the client only needs to wait for the first response
 - It can also handle byzantine failures
 - To tolerate up to f byzantine failures, $2f + 1$ replicas are required
 - The client waits for $f + 1$ identical responses

Network Partitions

- So far, we have assumed no network partition, what if there are network partitions?
- A **network partition** separates a group of replicas into two or more subgroups in such a way that
 - Replicas of one subgroup can communicate with one another
 - But replicas of different subgroups cannot communicate with one another
- Assumption: partitions will eventually be repaired
 - We must ensure that any operation executed during partition **will not make the replicas inconsistent** when the partition is repaired

Example of Network Partitions



- The replicas doing the left withdraw cannot communicate with those doing the right withdraw
- In general, reading during a partition would not cause inconsistency, but writing might

Handling Network Partitions

- Optimistic approach
 - Allow updates in **all partitions** → this can lead to inconsistencies between partitions
 - **Resolve inconsistencies** when the partition is repaired
 - Unsuitable for some applications like banking
- Pessimistic approach
 - Allow updates in **one partition only** → to prevent inconsistencies from occurring
 - **Update the remaining replicas** when the partition is repaired
 - Now let's look at Gifford's quorum-based protocol

Gifford's Quorum-Based Protocol

- Replicas in different partitions cannot communicate
 - Each subgroup must be able to decide **independently** whether they are allowed to carry out operations
- A **quorum** is a subgroup of replicas whose size gives it the right to perform operations
 - For example, having the majority of replicas could be the criterion (in a system of 9 replicas, a subgroup with at least 5 replicas has the right to perform operations)

Gifford's Quorum-Based Protocol

- Update operations may be performed by **one subset of the replicas only**
 - → The other replicas may be out-of-date
 - Version numbers are used to determine which replicas are up-to-date
 - When a replica is updated, its version number is changed accordingly
 - Operations are applied only to replicas with the latest version number

Gifford's Quorum-Based Protocol

- Each read operation must obtain a read quorum $\geq R$ replicas before it can read from any up-to-date replica
- Each write operation must obtain a write quorum $\geq W$ replicas before it can proceed with an update operation
- R and W are set such that
 - $W >$ half the total number of all replicas
 - $R + W >$ the total number of all replicas
 - These constraints ensure that any pair of a read quorum and a write quorum or two write quorums contain common replicas

Gifford's Quorum-Based Protocol

- To perform a read operation, a client must assemble a read quorum
 - Contact a set of available replicas whose total number is not less than R
 - The set must contain at least one up-to-date replica
 - The read operation may be applied to any up-to-date replica in the read quorum (which can be figured out by comparing the version numbers of replicas)

Gifford's Quorum-Based Protocol

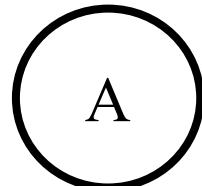
- To perform a write operation, a client must assemble a write quorum
 - Contact a set of available replicas whose total number is not less than W
 - The set must contain at least one up-to-date replica
 - If there are out-of-date replicas in the set, replace them with the up-to-date copy
 - The write operation is then applied to each replica in the write quorum, and the version number is incremented

Example of Gifford's Protocol

$x = 100$ (v0)



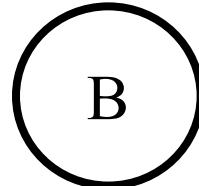
$x = 200$ (v1)



$x = 100$ (v0)



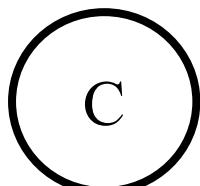
$x = 200$ (v1)



$\text{read}(x) \rightarrow 100$

$\text{write}(x=x+100)$

$x = 100$ (v0)



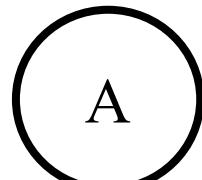
~~$\text{read}(x)$~~

~~$\text{write}(x=x+100)$~~

$R = W = 2$

Example of Gifford's Protocol

$x = 200$ (v1)



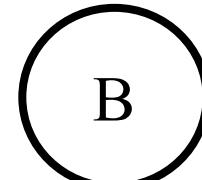
~~read(x)~~
~~write(x=x+20)~~

$R = W = 2$

$x = 200$ (v1)



$x = 400$ (v2)



read($x \rightarrow 200$)

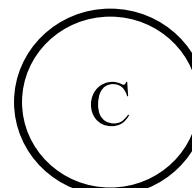
$x = 100$ (v0)



$x = 200$ (v1)



$x = 400$ (v2)



write($x = x + 200$)

Gifford's Quorum-Based Protocol

- There might be multiple eligible pairs of R and W
 - The system performance can be optimized by setting R and W appropriately under different situations
- Consider an object with three replicas
- If the object has a very high read to write ratio, we can set $R = 1$ and $W = 3$ to reduce traffic
 - Reads can be done at any replica, but writes must access all replicas (known as Read-One-Write-All)
- If the object has a very high write to read ratio, we can set $R = 2$ and $W = 2$ to reduce traffic
 - Writes need to access two replicas only (why not 1 replica?), and reads should also access two replicas

Outline

- Introduction
- Data-Centric Consistency Models and Protocols
- Client-Centric Consistency Models and Protocols
- Summary

Client-Centric Consistency Models

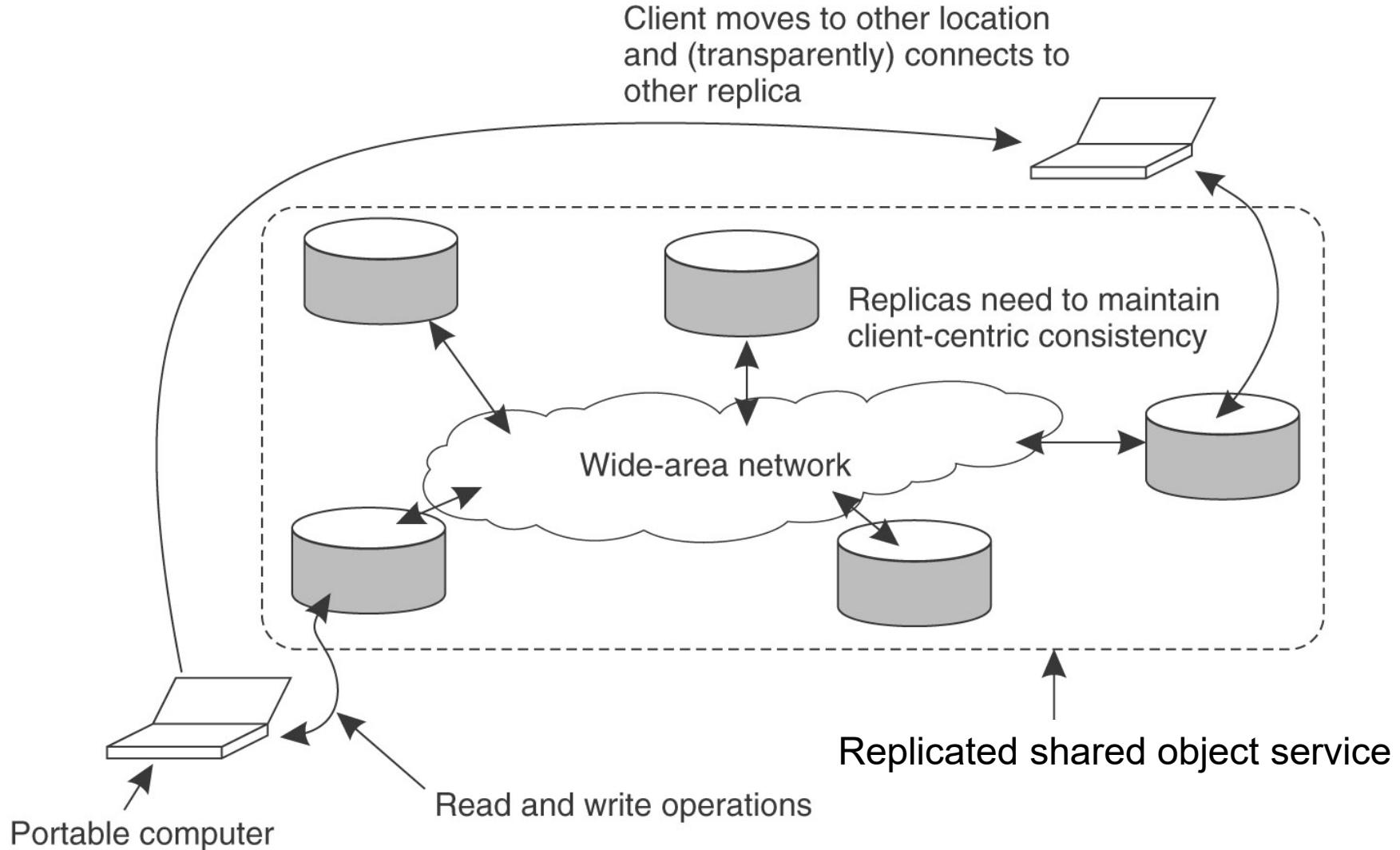
- Target scenario
 - Lack of simultaneous updates (or when they happen, they can easily be resolved)
 - Most operations involve reading data
- Examples
 - In many database systems, only one or very few processes perform update operations
 - In DNS, only the naming authority is allowed to update the name space
 - In World Wide Web, web pages are normally updated by a web master only
 - These applications can tolerate a relatively high degree of inconsistency

Client-Centric Consistency Models

- Eventual consistency

- If no update takes place for a long time, all replicas will gradually become consistent
- Essentially only require that updates are guaranteed to propagate to all replicas (possibly in a lazy fashion)
- Eventual consistency works fine as long as clients always access the same replica
- But problems arise when different replicas are accessed

Client-Centric Consistency Models



Client-Centric Consistency Models

- Client-centric consistency provides guarantees **for a single client** concerning the consistency of its accesses to data objects
- No guarantees are given concerning concurrent accesses by different clients

Client-Centric Consistency Models

- Monotonic reads
- Monotonic writes
- Read your writes
- Writes follow reads
- Examples in this section
 - In most cases, only one client updates an object
 - Updates are eventually propagated to all replicas
 - Initial value of an object x is o
 - Read operations r_1, r_2, \dots ; write operations w_1, w_2, \dots
 - $r_2 \rightarrow o + w_1 + w_2$: a read operation on x returning a value integrating write operations w_1 and w_2

Monotonic Reads

- If a client reads the value of a data object x , any subsequent read operation on x by that client will always return that same value or a more recent value
- That is, if a client has seen a value of x at time t , it will never see an older version of x later
- Sample application: Distributed email database
 - Suppose a user reads his email in Singapore and then later flies to Hong Kong and opens his mailbox again
 - Messages that were in the mailbox in Singapore will also be in the mailbox when it is opened in Hong Kong

Monotonic Reads

- Example of a monotonic-read consistent service

Replica A	Replica B
	w_1 (become $o + w_1$)
propagate w_1 (become $o + w_1$)	
$r_1 \rightarrow o + w_1$	
	w_2 (become $o + w_1 + w_2$)
	$r_2 \rightarrow o + w_1 + w_2$
propagate w_2 (become $o + w_1 + w_2$)	

w_1 and w_2 may be performed by a different client from r_1 and r_2

Monotonic Reads

- Example of a not monotonic-read consistent service

Replica A	Replica B
	w_1 (become $o + w_1$)
propagate w_1 (become $o + w_1$)	
	w_2 (become $o + w_1 + w_2$)
	$r_1 \rightarrow o + w_1 + w_2$
$r_2 \rightarrow o + w_1$	
propagate w_2 (become $o + w_1 + w_2$)	

w_1 and w_2 may be performed by a different client from r_1 and r_2

Monotonic Writes

- A write operation by a client on a data object x is completed before any subsequent write operation on x by the same client
- In general, a write operation on a copy of 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
 - Always necessary to bring the copy up to date first?
- Sample application: Software library
 - Updating the library is done by replacing functions
 - If an update is performed on a copy of the library, all preceding updates will be performed first

Monotonic Writes

- Example of a monotonic-write consistent service

Replica A	Replica B
w_1 (become $o + w_1$)	
	propagate w_1 (become $o + w_1$)
	w_2 (become $o + w_1 + w_2$)
	$r_1 \rightarrow o + w_1 + w_2$

r_1 may be performed by a different client from w_1 and w_2

Monotonic Writes

- Example of a not monotonic-write consistent service

Replica A	Replica B
w_1 (become $o + w_1$)	
	w_2 (become $o + w_2$)
	$r_1 \rightarrow o + w_2$

r_1 may be performed by a different client from w_1 and w_2

Read Your Writes

- The effect of a write operation by a client on a data object x will always be seen by a subsequent read operation on x by the same client
- That is, a write operation is always completed before a subsequent read operation by the same client, no matter where that read takes place
- Sample application: World Wide Web
 - A user's web browser caches a local copy of a web page after accessing it from a local web server
 - When the web page is updated at the server, the user will not see the new version if the browser returns the cached copy

Read Your Writes

- Example of a read-your-writes consistent service

Replica A	Replica B
	w_1 (become $o + w_1$)
propagate w_1 (become $o + w_1$)	
	w_2 (become $o + w_1 + w_2$)
propagate w_2 (become $o + w_1 + w_2$)	
$r_1 \rightarrow o + w_1 + w_2$	

w_1 , w_2 and r_1 are performed by the same client

Read Your Writes

- Example of a not read-your-writes consistent service

Replica A	Replica B
	w_1 (become $o + w_1$)
propagate w_1 (become $o + w_1$)	
	w_2 (become $o + w_1 + w_2$)
$r_1 \rightarrow o + w_1$	
propagate w_2 (become $o + w_1 + w_2$)	

w_1 , w_2 and r_1 are performed by the same client

Writes Follow Reads

- A write operation by a client on a data object x following a previous read operation on x by the same client is guaranteed to take place on the same or a more recent value of x that was read
- That is, any subsequent write operation by a client on a data object x will be performed on a copy of x that is up to date with the value most recently read by that client
- Sample application: Network newsgroup
 - Users of a network newsgroup see a posting of a reaction to an article only after they have seen the original article

Writes Follow Reads

- Example of a writes-follow-reads consistent service

Replica A	Replica B
w_1 (become $o + w_1$)	
	propagate w_1 (become $o + w_1$)
$r_1 \rightarrow o + w_1$	
	w_2 (become $o + w_1 + w_2$)

w_1 may be performed by a different client from r_1 and w_2

Writes Follow Reads

- Example of a not writes-follow-reads consistent service

Replica A	Replica B
w_1 (become $o + w_1$)	
$r_1 \rightarrow o + w_1$	
	w_2 (become $o + w_2$)

w_1 may be performed by a different client from r_1 and w_2

Client-Centric Consistency Protocols

- Each write operation is assigned a globally unique identifier
 - Can be assigned by the replica to which the write had been submitted (known as the origin of the write operation)
- Each client keeps track of two sets of write operations
 - Read set: (identifiers of) writes relevant to the read operations performed by the client
 - Write set: (identifiers of) writes performed by the client

Client-Centric Consistency Protocols

- To implement monotonic-read consistency
 - When a client performs a read operation at a replica, that replica is handed the client's read set to check whether all the identified writes have taken place locally
 - If not, the replica contacts other replicas to ensure that it is brought up to date for carrying out the read operation
 - Alternatively, the read operation is forwarded to a replica where the identified writes have already taken place
 - After the read operation is performed, the write operations that have taken place at the selected replica and which are relevant for the read operation are added to the client's read set

Client-Centric Consistency Protocols

- The write identifier could include the identifier of the replica to which the operation was submitted
- That replica could be required to log the write operation so that it can be replayed at another replica
- Other client-centric consistency models can be implemented in similar ways

Summary

- Replication helps services to provide good performance, reliability and fault tolerance
- Meanwhile, replication introduces the consistency problem
- Data-centric consistency models
 - Strict, sequential, causal and FIFO consistency
- Client-centric consistency models
 - Monotonic-read, monotonic-write, read-your-writes, and writes-follow-reads consistency

Revisit

This is meant to help you recall some of the topics covered, but do not take it to guide your entire effort in review. There is no replacement for a thorough readup and think through.

Introduction and System Models

- What is a distributed system?
- Fundamental characteristics of distributed systems
- Main motivation for building distributed systems
- Architectural models
 - Client-server & peer-to-peer
- Interaction models
 - Synchronous & asynchronous
- Failure models
 - Omission and byzantine failures

Interprocess Communication

- UDP/TCP communication
- External data representation
 - Marshalling and unmarshalling
 - CORBA's common data representation (CDR)
 - Java object serialization
- Request-reply protocols
 - How to build reliable request-reply protocols over UDP?
 - How to reduce overhead of request-reply protocols over TCP?

Distributed Objects & Remote Invocation

- Remote Method Invocation (RMI)
 - Remote object
 - Remote method invocation
 - Remote object reference
 - Remote interface
 - Invocation semantics
 - Java RMI architecture
 - How to write Java remote interfaces?
 - How to write Java RMI programs?

Distributed File Systems

- Network File System
 - Architecture, stateless servers
 - Block-based file serving and caching
 - Timeout-based cache consistency maintenance
 - Approximate one-copy update semantics
- Andrew File System
 - Architecture, stateful servers
 - Whole file serving and caching
 - Callback-based cache consistency maintenance
 - Session update semantics
- Coda File System
 - Hoarding

Peer-to-Peer File Sharing Systems

- What are peer-to-peer systems?
- Unstructured P2P file sharing
 - Napster – centralized directory service
 - Gnutella – search by flooding
 - KaZaA – hierarchical architecture
- Structured P2P file sharing
 - Consistent hashing
 - Chord routing

Name Services

- Names and addresses
- Flat and hierarchical name spaces
- Name resolution process
 - Iterative client-controlled navigation
 - Non-recursive server-controlled navigation
 - Recursive server-controlled navigation
- Domain Name System
 - Naming data are partitioned and distributed to a large number of name servers
 - Replication for reliability
 - Caching for faster resolution

Time and Global States

- How to synchronize different computer clocks?
 - Cristian's method
 - Berkeley algorithm
 - Network Time Protocol
- How to order events without global time?
 - Causal ordering
 - Lamport's logical clocks
 - Vector clocks

Time and Global States

- How to observe the global state of a distributed system?
 - Cut and global state
 - Consistent and inconsistent cuts
 - Chandy and Lamport algorithm to record a snapshot of a distributed system during execution
- Distributed debugging
 - Lattice of consistent global states
 - How to determine whether a constraint is broken in the execution?

Coordination and Agreement

- Distributed mutual exclusion
 - Central server algorithm
 - Ring-based algorithm
 - Ricart and Agrawala algorithm
- Election
 - Ring-based algorithm
 - Bully algorithm
- Consensus and related problems
 - Solution to consensus problem in synchronous system
 - Byzantine generals problem in synchronous system
 - Impossibility if $N \leq 3f$
 - Solution when $N > 3f$

Replication

- Motivations for replication
- Data-centric consistency models and protocols
 - Strict consistency, sequential consistency, causal consistency, FIFO consistency
 - Primary-based protocols: remote-write protocol, local-write protocol
 - Replicated-write protocols: active replication, Gifford's quorum-based protocol
- Client-centric consistency models and protocols
 - Monotonic-read, monotonic-write, read-your-writes, writes-follow-reads consistency
 - Client-centric consistency protocols

Notes on Exam

- Give **clear and concise** answers to the questions
- Do not interleave your solutions
 - Do not: 1(a),2(c),3(b),2(b),4(a),1(b)...
- Leave a blank page (with the question number and part) if you want to skip a part temporarily
- Consultation hours:
 - To be announced