Text Books

- **Distributed Systems Principles and Paradigms** by Andrew S. Tanenbaum and Maarten van Steen
- **Distributed Systems Concepts and Design** by George Coulouris, Jean Dolimore, Tim Kindberg

A little bit of History

- Computer Systems are undergoing a revolution
- From 1945, when the modern computer era began, until about 1985, computers were large and expensive. A a result most organisations had only a handful of computers, and for lack of a way to connect them, these operated independently from on an other.

Two important developments

- Starting from 1980s, two advances in technology began to change the situation.
- The first was the development of powerful microprocessors. Initially these were 8-bit, but soon 16 & 32 and 64-bit CPUs became common, Many of them had a computing power of a mainframe computer, but for a fraction of the price.

Two important developments

- The second development was the invention of high speed computer networks.
- Local Area Network (LANs) allow hundreds of machines within a building to be connected in such a way, that small amount of information can be transferred between machines in a few microseconds or so. Larger amount of data can be moved between machines at rates of 10 to 1000 bits/sec.
- Wide Area Networks (WANs) allow millions of machines all over the earth to be connected at speeds varying from 64 Kbps to gigabits per second.
- The result of these technologies is that it is now not only feasible, but easy to put together computing systems composed of large numbers of computers connected by high speed network. They are usually called distributed system.

Definition of a Distributed System

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

- This definition has two aspects: first one deals with hardware, that machines are autonomous, the second one deal with software, the users think they are dealing with a single system.
- Both are essential.

Definition of a Distributed System

A distributed system in one in which components located at networked computers communicate and coordinate their actions only by passing messages.

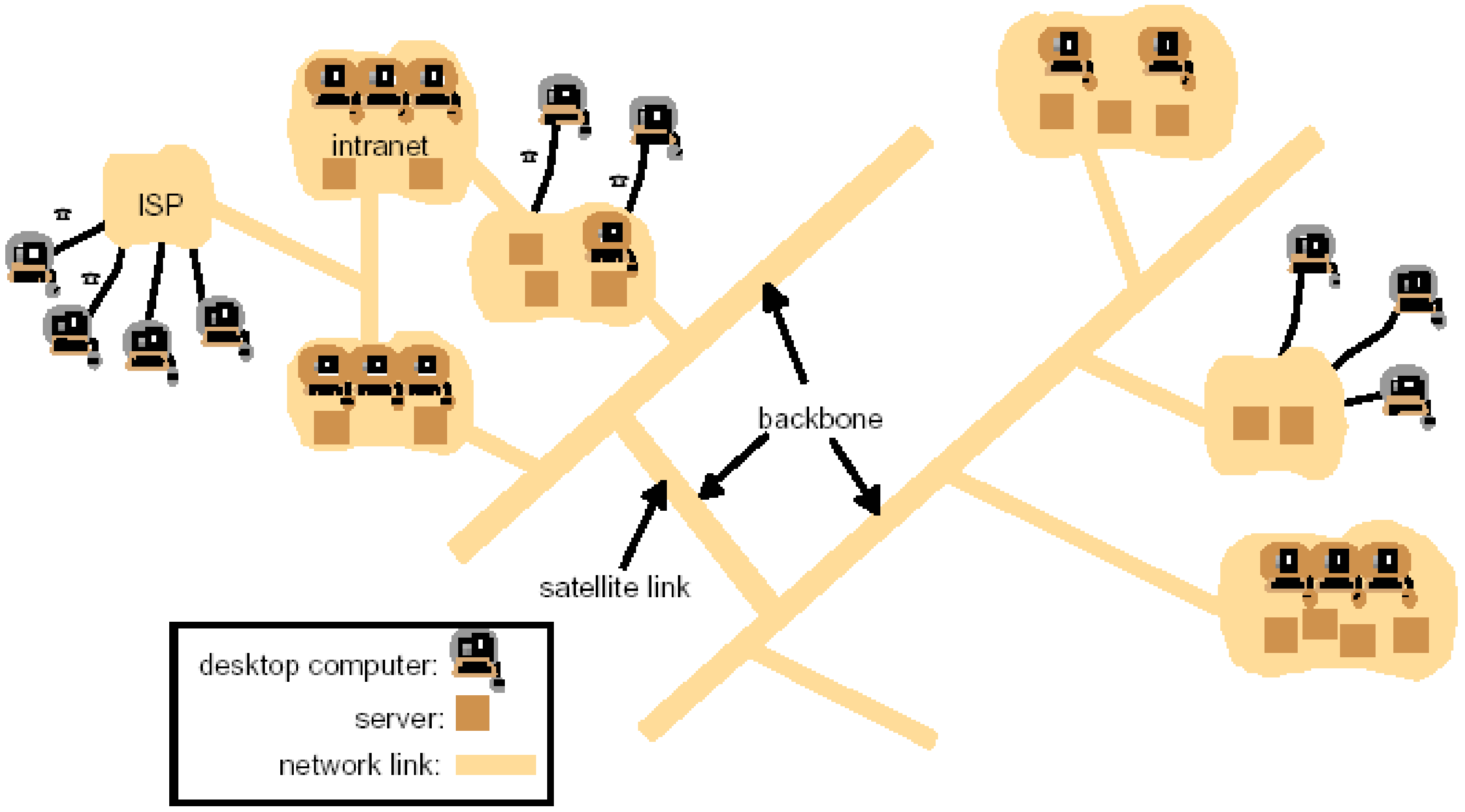
This definition leads to the following characteristics of distributed system:

- Concurrency of the components
- Lack of global clock
- Independent failures of components

Motivation & Challenges

- The sharing of resources is a main motivation for constructing distributed systems. The resources may be managed by servers and accessed by clients or they may be encapsulated as objects and accessed by other client objects.
- The challenges arising from the construction of distributed systems are the heterogeneity of its components, openness -which allows components to be added or replaced, security, scalability – the ability to work well when number of the users increases- failure handling, concurrency of components and transparency.

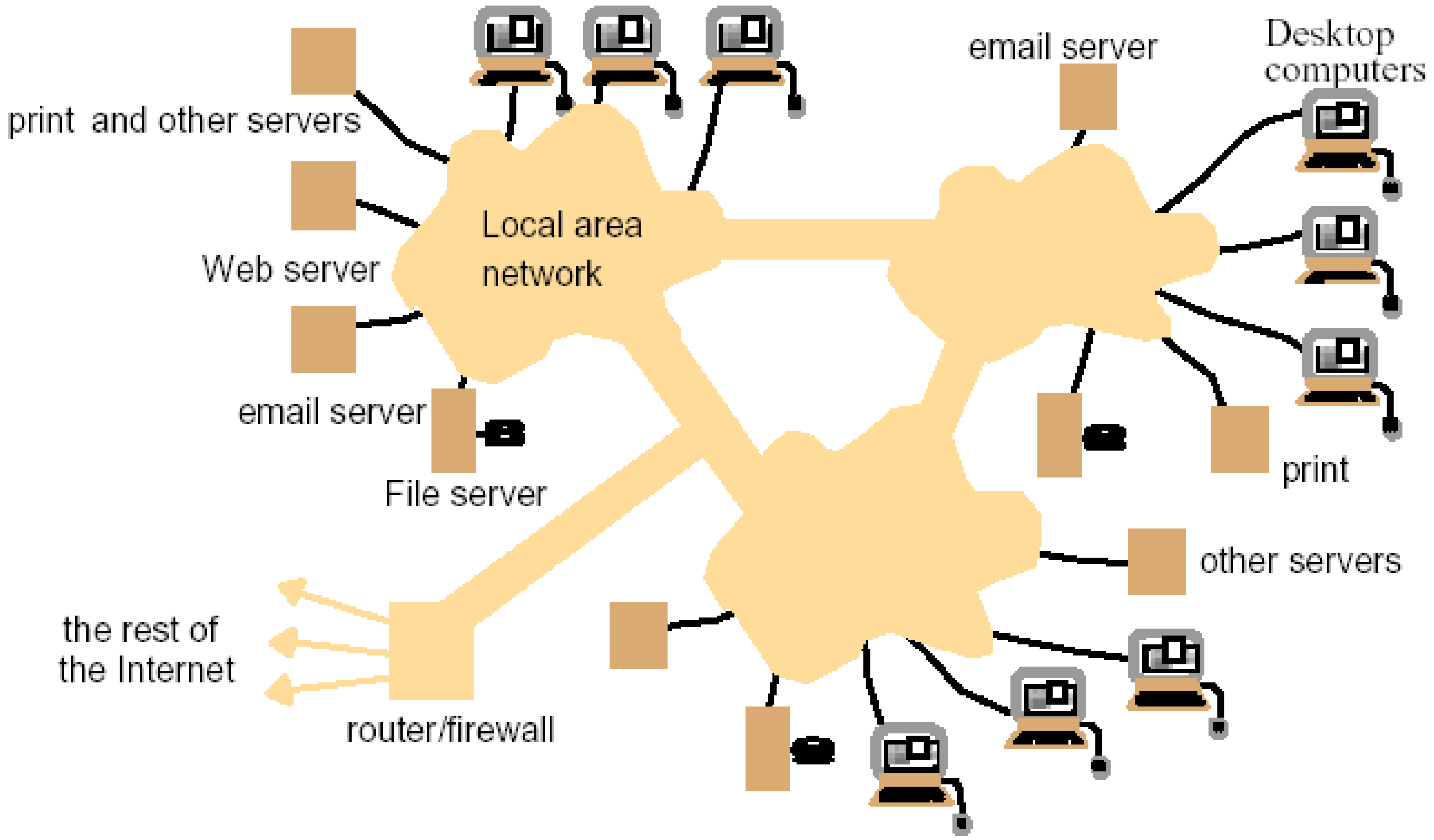
Example: The Internet



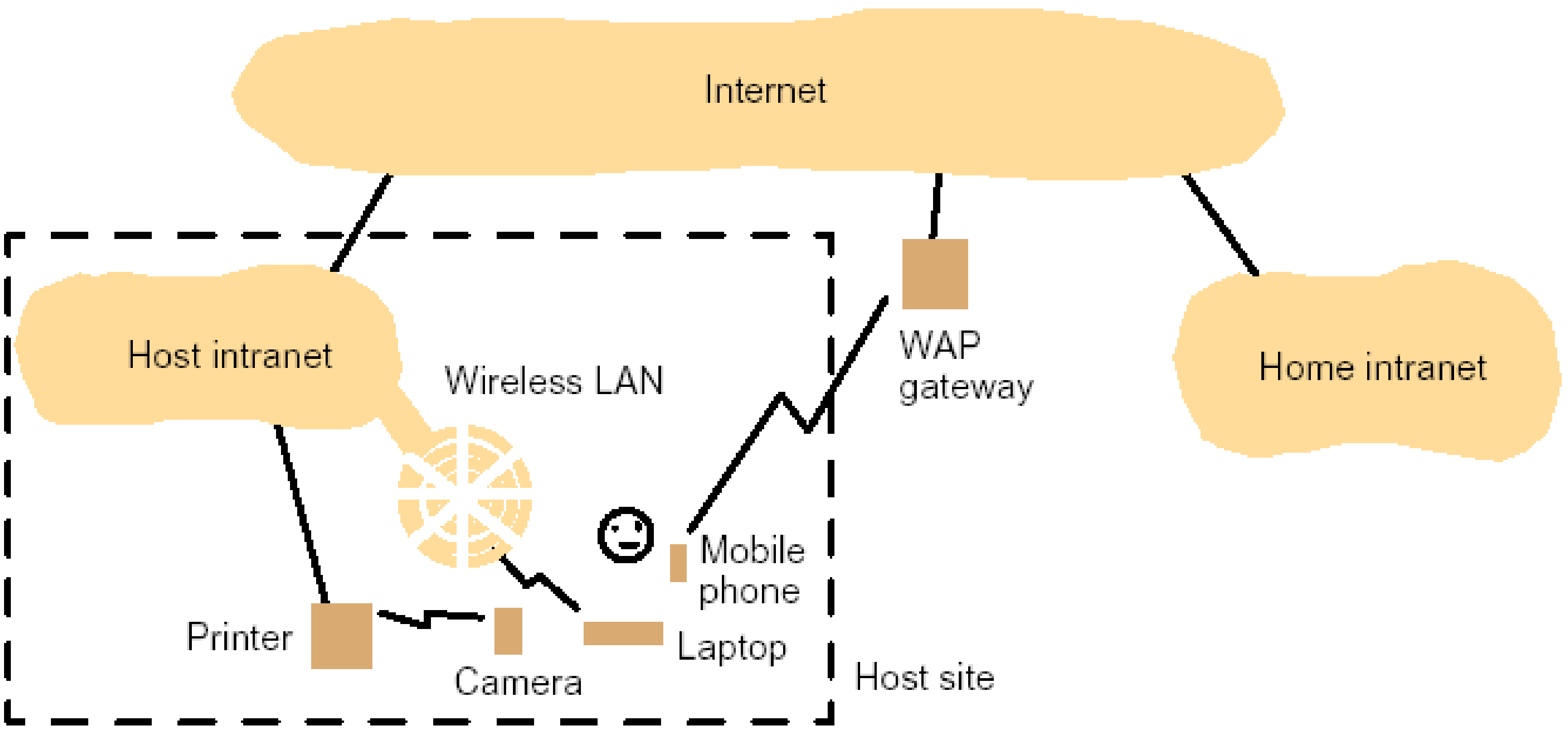
The Internet

- The internet is a vast interconnected collection of computer networks of many different types. Programs running on the computers connected to it interact by passing messages.
- The design and construction of the internet communication mechanisms (the internet protocol) is a major technical achievement, enabling programs running anywhere to address messages to programs anywhere else.
- Large scale
- Enables users, wherever they are, to make use of services such as World Wide Web, e mail, file transfer
- The set of services is open ended – it can be extended by the addition of new server computers and new types of services.

Example: A Typical Intranet



Portable/Handheld Devices in a Distributed System



Key Design Challenges

- connecting users and resources
- heterogeneity
- transparency
- openness
- scalability
- concurrency
- security
- failure handling
- mobility and location dependency

Heterogeneity

- Distributed systems are constructed from a variety of different networks, operating systems, computer hardware and programming languages.
- Data types such as integers may be represented differently on different sorts of hardware. These differences must be dealt with if messages are to be exchanged between programs running on different hardware.
- Operating systems: call for exchanging messages in UNIX is different from calls in Windows NT.
- Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs in different languages are to be able to communicate.

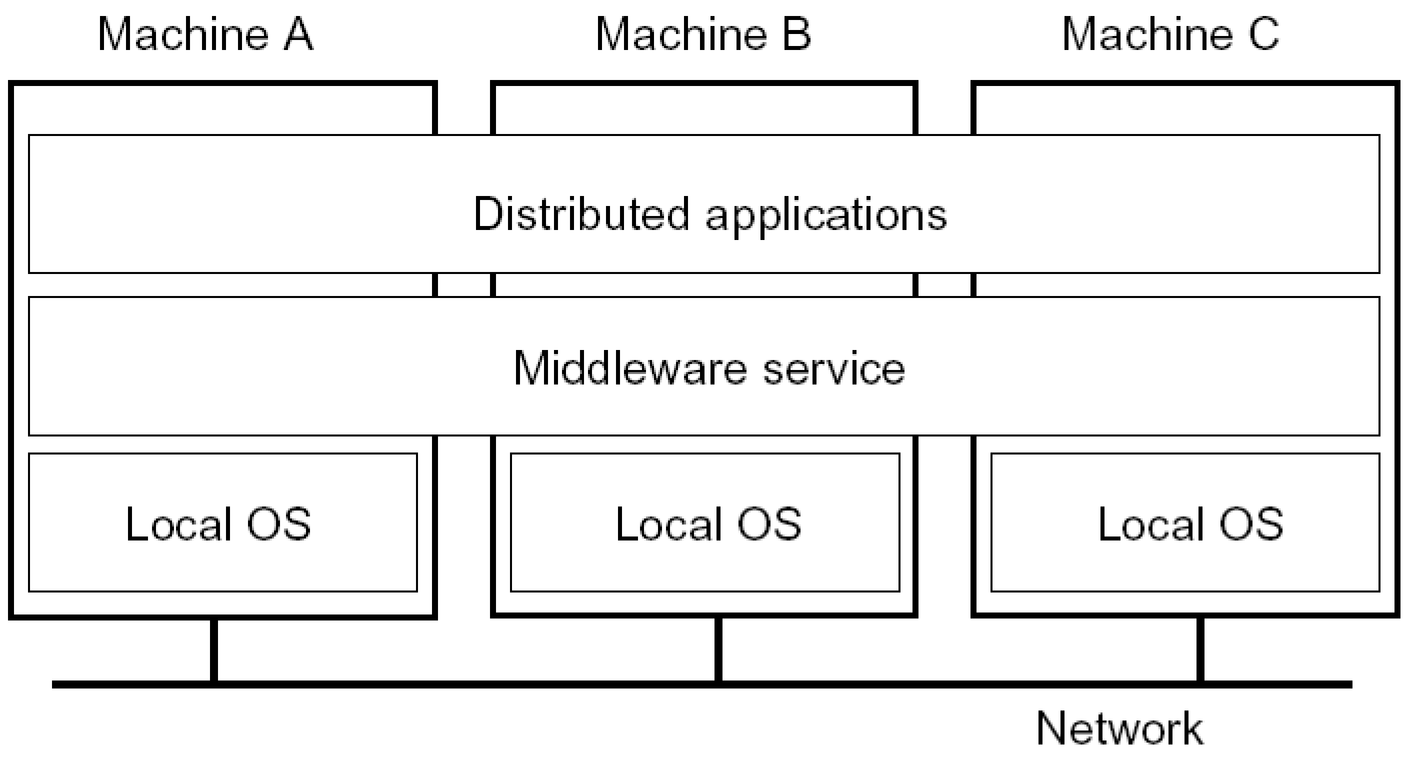
Heterogeneity

- One possible solution is Middleware
- The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. (Java RMI, CORBA)
- Middleware provides a uniform computational model for use by the programmers of servers and distributed applications. Possible models include remote object invocation, remote event notification, remote SQL access and distributed transaction processing.

Middleware

- Software that manages and supports the different components of a distributed system. In essence, it sits in the *middle* of the system.
- Middleware is usually off-the-shelf rather than specially written software.
- Examples
  - Transaction processing monitors;
  - Data converters;
  - Communication controllers.

Distributed Systems as Middleware



To hide the heterogeneity, a distributed system is usually organized as **middleware**. Note that the middleware layer extends over multiple machines.

Middleware: Examples

- CORBA: provides remote object invocation, which allows an object in a program running on one computer to invoke a method on an object in a program running on another computer. Its implementation hides the fact that messages are passed over a network in order to send the invocation request and its reply.
- Java RMI is similar, but it supports only single programming language.

Transparency

- Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than a collection of independent components.

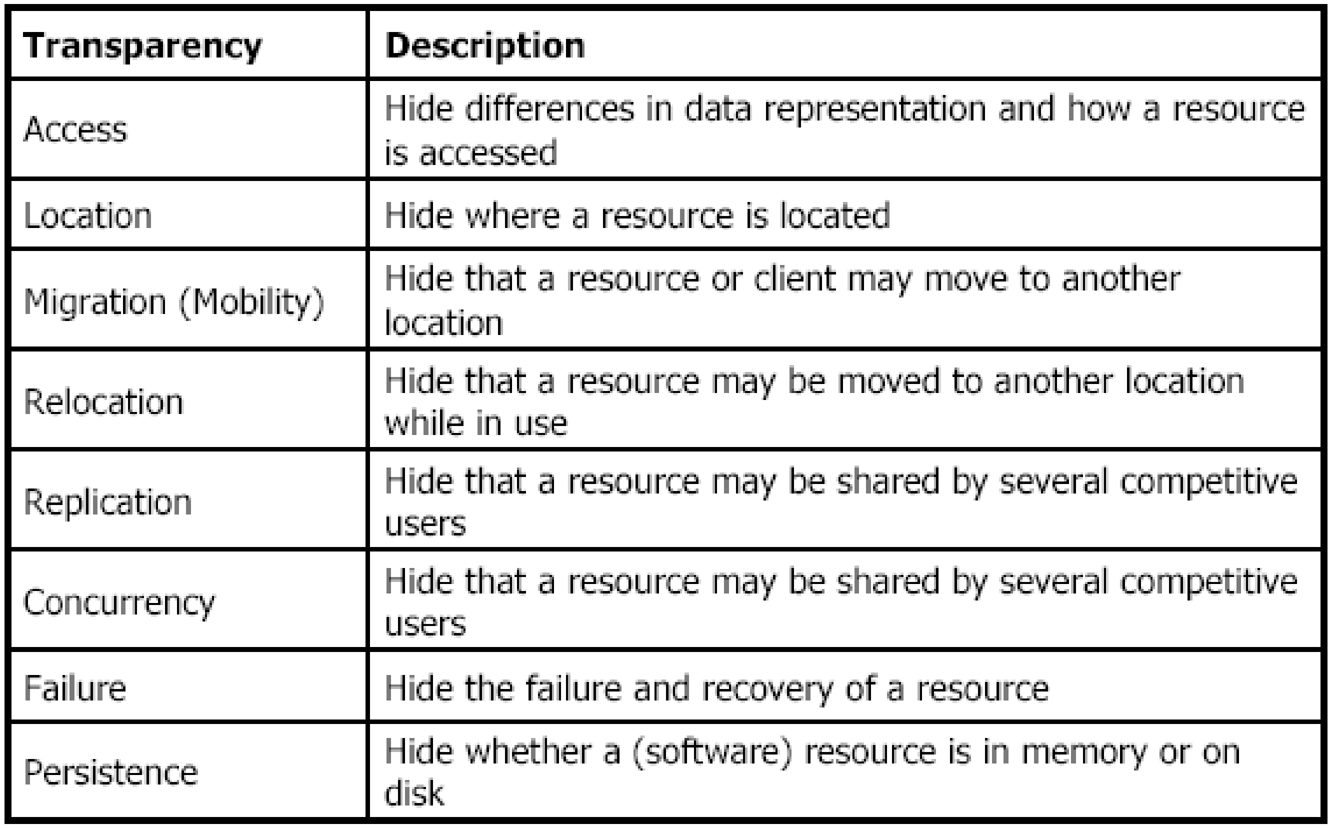
Eight Identified Forms of Transparency

- **Access**: Enables local and remote resources to be accessed using identical operations
- **Location**: Enables resources to be accessed without the knowledge of their location.
- **Concurrency**: Enables several processes to operate concurrently using shared resources without interference between them
- **Replication**: Enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers

Eight Identified Forms of Transparency

- **Failure**: Enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software.
- **Mobility**: Allows the movement of resources and client within a system without affecting the operation of users or programs.
- **Performance**: Allows the system to be reconfigured to improve performance as load vary.
- **Scaling**: Allows the system and applications to expand in scale without change to the system structure or the application.

Transparency: In a nutshell



Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration (Mobility)	Hide that a resource or client may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be shared by several competitive users
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

Different forms of transparency in a distributed system.

Openness

- The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways.
- The openness of a distributed system is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.
- Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers. That is, key interfaces are published.

Openness

- Open systems are characterized by the fact that their key interfaces are published
- Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.
- Open distributed systems can be constructed from heterogeneous hardware and software possibly from different vendors.
- The confirmation of each component to the published standards must be tested and verified if the system is to work correctly.

Scalability

- A system is described as scalable if will remain effective when there is a significant increase in the number of resources and the number of users.
- The Internet provides an illustration of a distributed system in which the number of computers and services has increased dramatically.
- Ideally the system and application software should not need to change when the scale of the system increases.

Scalability Challenges

The design of scalable distributed systems presents the following challenges:

- Controlling the cost of physical resources: As the demand for resources grows, it should be possible to extend the system, at a reasonable cost, to meet it. For example, if the demand of files increases in an intranet, it must be possible to add server computers.
- Controlling the performance loss: For a system to be scalable, the maximum performance loss should be no worst than a certain acceptable value:

$O(\log n)$

Scalability Challenges

- Preventing Software resources running out: An example of lack of scalability is shown by the numbers used as Internet addresses.1970, it was decided, it should be 32 bits, but will probably run out in early 2000s. For this reason, the new version of the protocol will use 128 bit Internet addresses. (To be fair to early designers, sometimes it is difficult to predict the demand that will be put on the system years ahead.
- Avoiding Performance Bottlenecks: Algorithms should be decentralized to avoid performance bottlenecks. An example is the domain name system, where initially the name table was kept in a single master file, but it soon became a serious performance and administrative bottleneck. This bottleneck was removed by partitioning the name table between servers located throughout the Internet and administered locally.

Security

Three components of security

- **<u>Confidentiality</u>**: Protection against disclosure to unauthorized individuals.
- **<u>Integrity:</u>** Protection against alteration or corruption.
- **<u>Availability</u>**: Protection against interference with the means to access the resources.

<u>Challenges that have not yet been fully met:</u> Denial of Service attacks and Security of mobile code.

Security

Strategies for securing Distributed Systems.

Generally very similar to techniques used in a non-distributed system, only much more difficult to implement ….

Difficult to get right, impossible to get perfect!

Security Topics

1. Providing a *secure communications channel* – authentication, confidentiality and integrity.
2. Handling *authorization* – who is entitled to use what in the system?
3. Providing effective *Security Management.*
4. Example systems: *SESAME* and *e-payment systems*.

Security Mechanisms

- **Encryption** – fundamental technique: used to implement confidentiality and integrity.
- **Authentication** – verifying identities.
- **Authorization** – verifying allowable operations.
- **Auditing** – who did what to what and when/how did they do it?

Concurrency

- In a distributed system, there is a possibility that several clients will attempt to access a shared resource at the same time. The process that manages a shared resource could take one client at a time. But this approach limits throughput. Therefore services and applications generally allow multiple clients requests to be processed concurrently. In this case the clients operations on the resources may conflict with one another and produce consistent results.
- For a resource (object, data source etc,) to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent.

Failure Handling

- Failure in distributed systems are partials – that is, some components fail while others continue to function. Therefore handling of failures is particularly difficult.
- Important Issues: Detecting failures, Masking failures, Tolerating failures, Recovery from failures, Redundancy

Failure Handling

- **Detecting Failures:** Some failures can be detected (checksum for data corruption), some other failures are very difficult (sometimes even impossible) to detect (remote crash server). The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

- **Masking Failures:** Some failures that have been detected can be hidden or made less severe. For example: messages can be retransmitted whey they fail to arrive, file data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

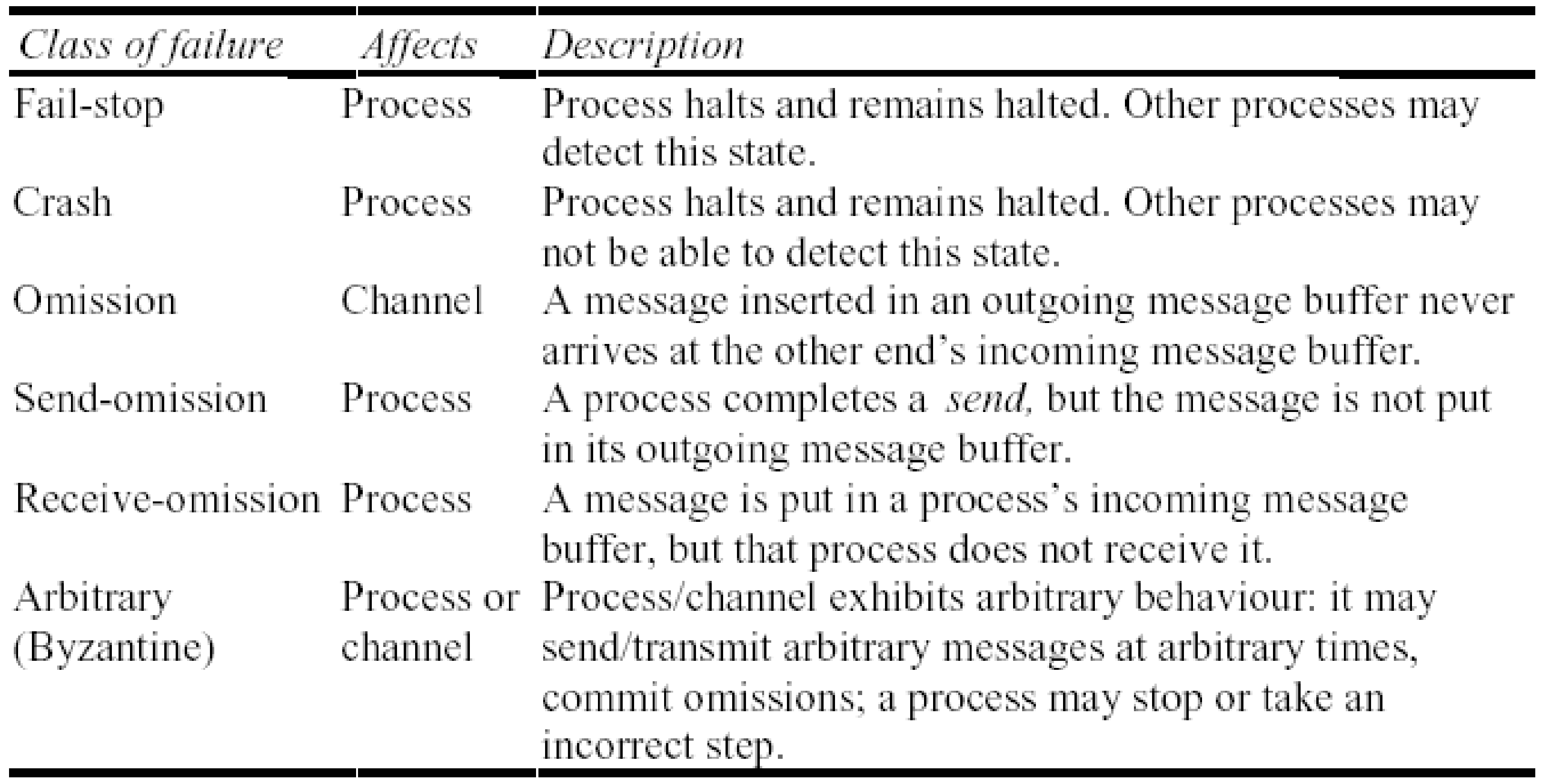
Failure Handling

- **Tolerating Failures / Redundancy :** Services can be made to tolerate failures by the use of redundant components. For example: 1) There should always be at least two different routes between any two routers in the Internet. 2) In domain name system every name table is replicated in at least two different servers. 3) A data base may be replicated in several servers to ensure that the data remain accessible after failure, clients are redirected to the remaining servers.

Failure Handling

- Recovery from Failures: Recovery involves the design of software so that the state of the permanent data can be recovered or “Rolled Back” after a server has crashed. In general the computations performed by some programs will be incomplete when a fault occurs, and the permanent data that they update may not be in a consistent state. Recovery will be discussed in detail latter.

Omission and Arbitrary Failures



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

- All these challenges / requirements must be considered in designing distributed systems.

Two Concepts to look at ! ! !

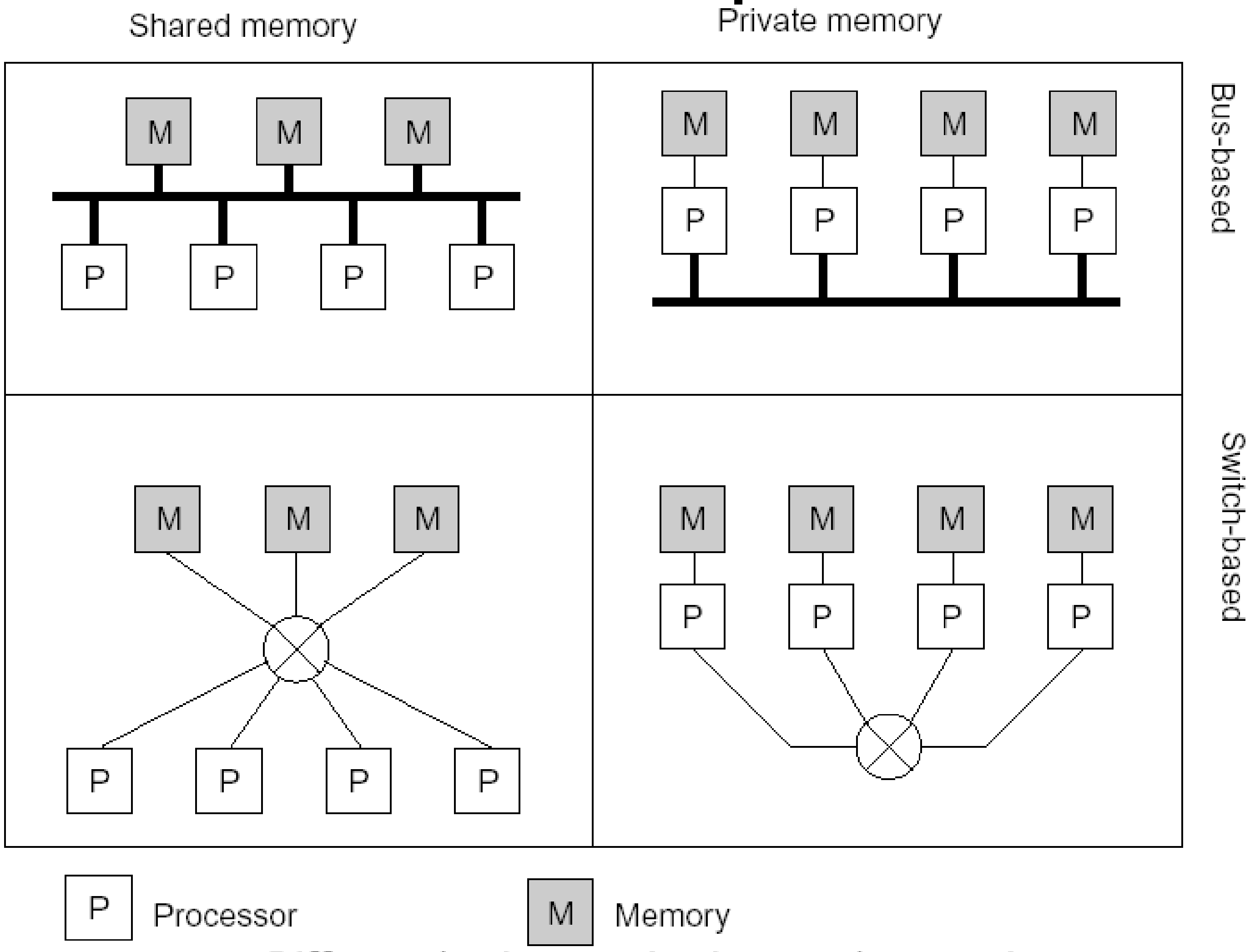
- No Global Clock: Need to understand the impact of no global clock. Such as Distributed Transactions, Program Compilations etc and explain the concept of logical clock based on the “happened before” relationship.
- Distributed Algorithms: Need to understand the concept of distributed algorithms, use the example of mutual exclusion, centralized mutual exclusion (one server controls the access to a shared variable) and distributed mutual exclusion (a node sends a message to all other nodes, waits for all “OK” replies, and then go ahead to use the shared variable, if a node is using or already waiting for the same access right, will not reply “OK”… explain with diagrams)
- Discuss Middleware again in detail…

System Models

Multiprocessors Vs Multicomputers

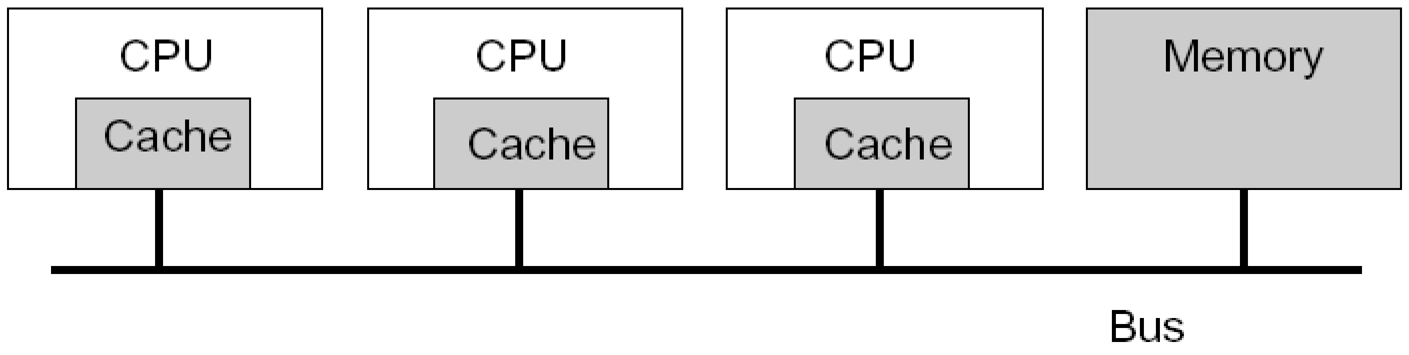
- In multiprocessors, there is a single address space shared by all CPUs, in a multi-computer every machine has its own private memory.
- Bus based & Switch based
- Homogeneous & Heterogeneous
- Tightly Coupled Vs Loosely Coupled Systems.

Hardware Concepts



Different basic organizations and memories in distributed computer systems

Multiprocessors



- A bus-based multiprocessor.

Heterogeneous Multicomputer Systems

- The machines that form the system may vary widely.
- The interconnection network may be heterogeneous.
- Large scale multicomputers using existing networks and backbones.
- Need sophisticated softwares for such systems.

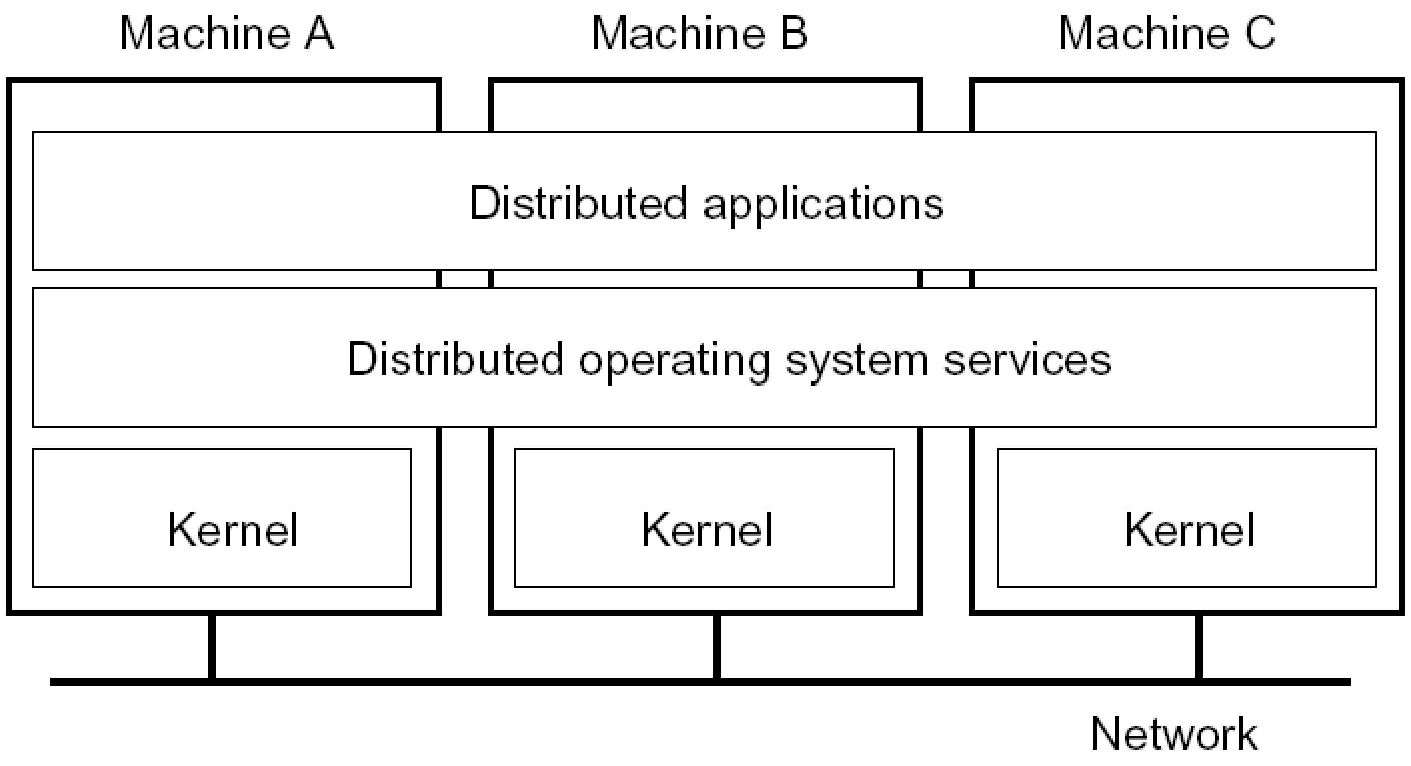
Software Concepts

System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

- An overview of
- DOS (Distributed Operating Systems)
- NOS (Network Operating Systems)
- Middleware

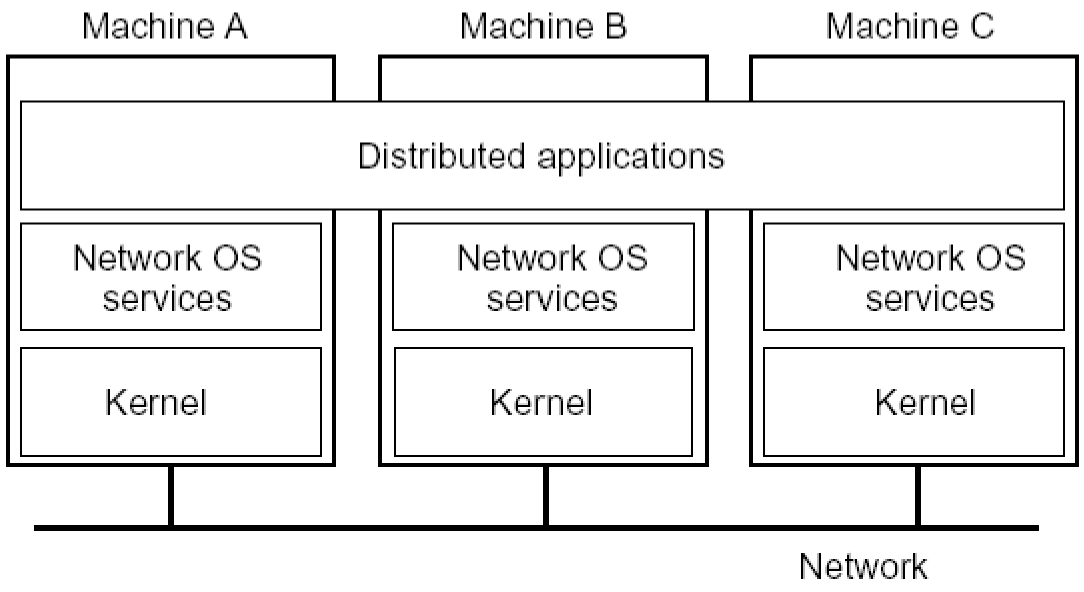
Multicomputer Operating Systems

- General structure of a multicomputer operating system

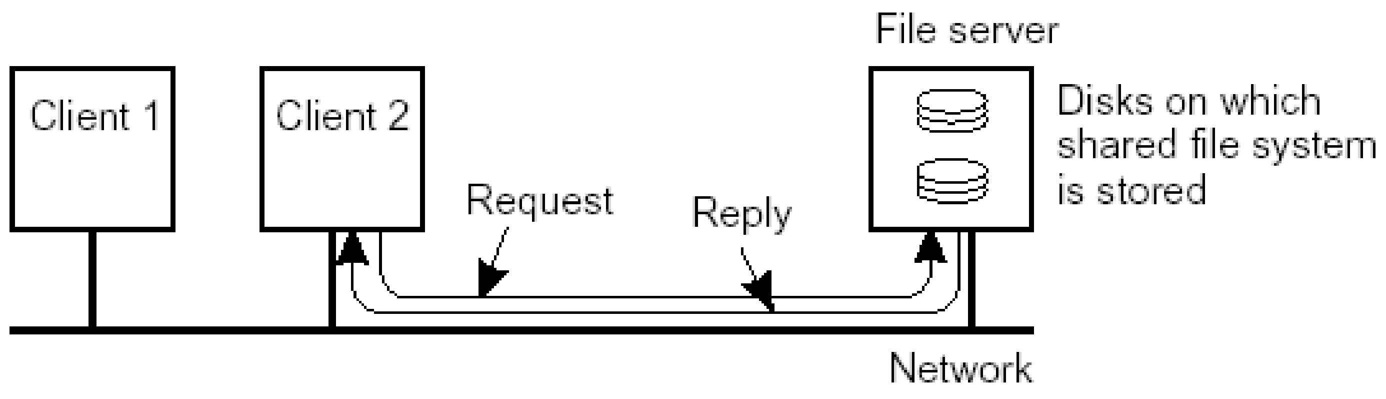


Network Operating System

- General structure of a network operating system.
- Only primitive services such as `rlogin`, `rcp`, ….



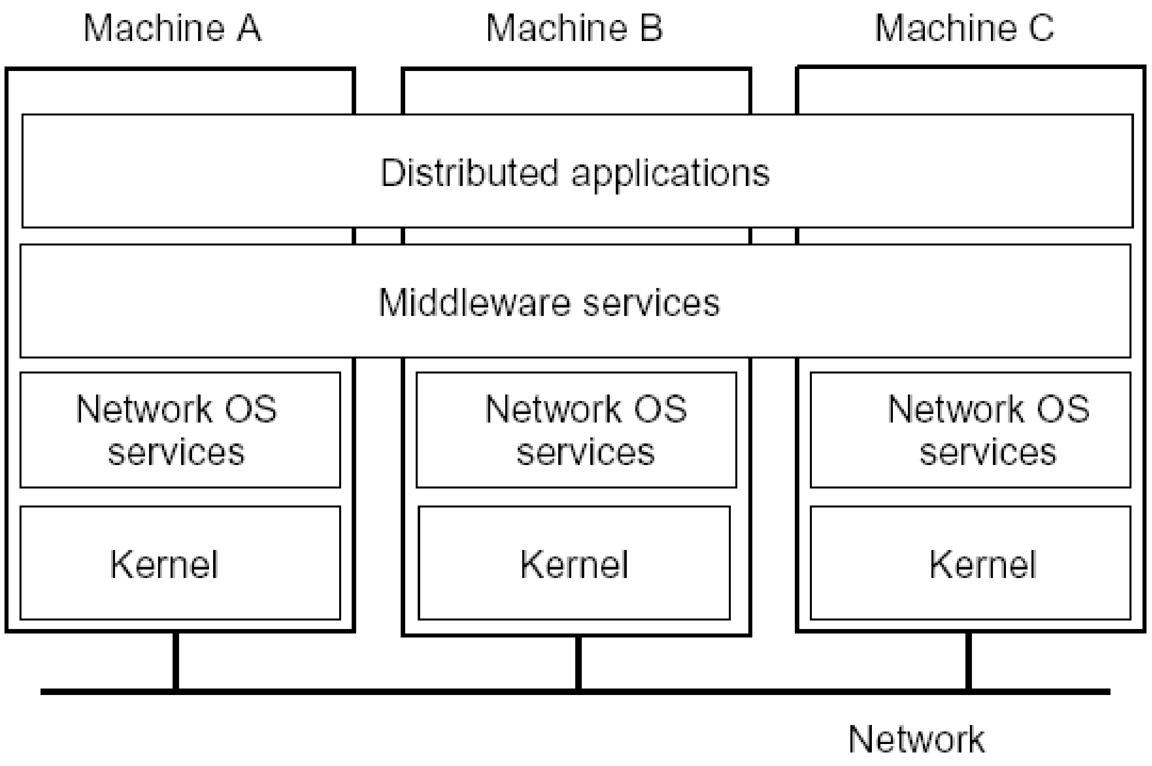
Network Operating System



- Can provide better services with shared servers.
- Example: Two clients and a file server in a network operating system.

Positional Middleware

- Neither a distributed OS or a network OS qualifies as a distributed system according to our definition.
- Solution: Structure a distributed system as middleware.



Middleware Models

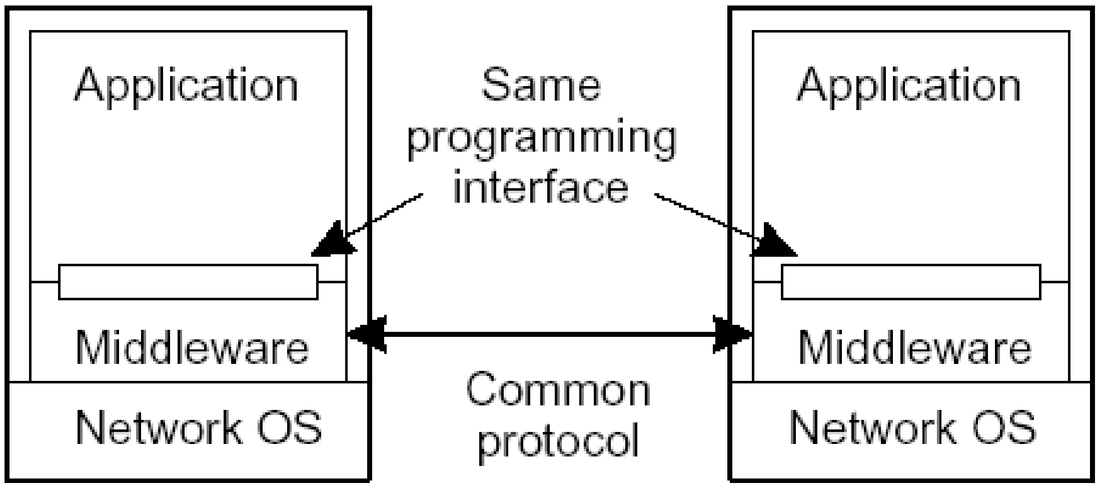
- Middleware models describe the way of distribution and communication.
- Treating everything as a file (from UNIX)
- Based on distributed file systems
- Based on Remote Procedure Calls (RPCs)
- Based on distributed objects
- Based on distributed documents

Middleware Services

- High-level communication facilities
- Naming
- Persistence
- Distributed transactions
- Security
- Replication
- Location management
- Data management

Middleware and Openness

- In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.



Comparison between Systems

Item	Distributed OS		Network OS	Middleware-based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

Distributed Systems Architectures

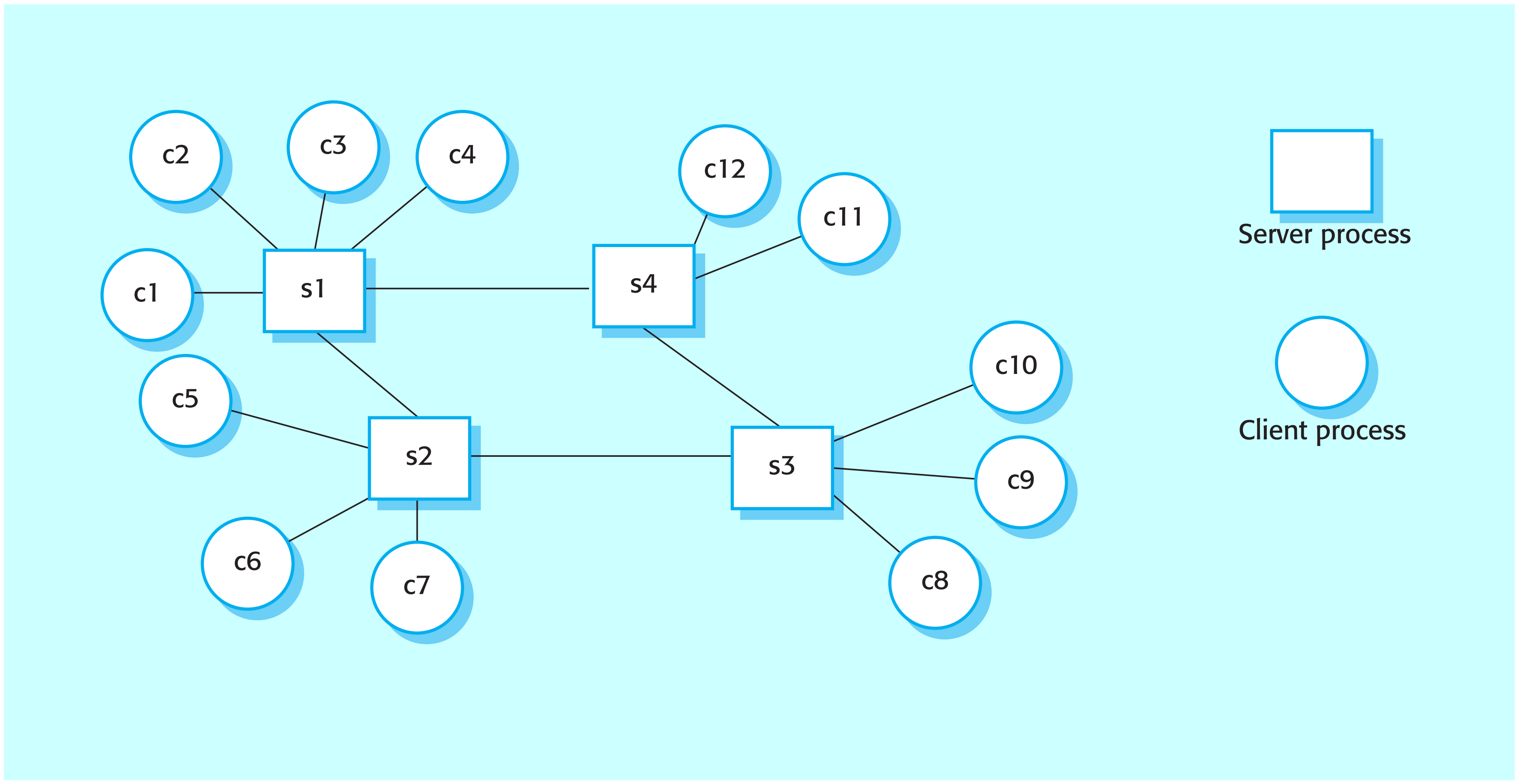
Architectural Models

- It is very important that we understand the Architectural Models for Distributed Systems presented in the following slides…

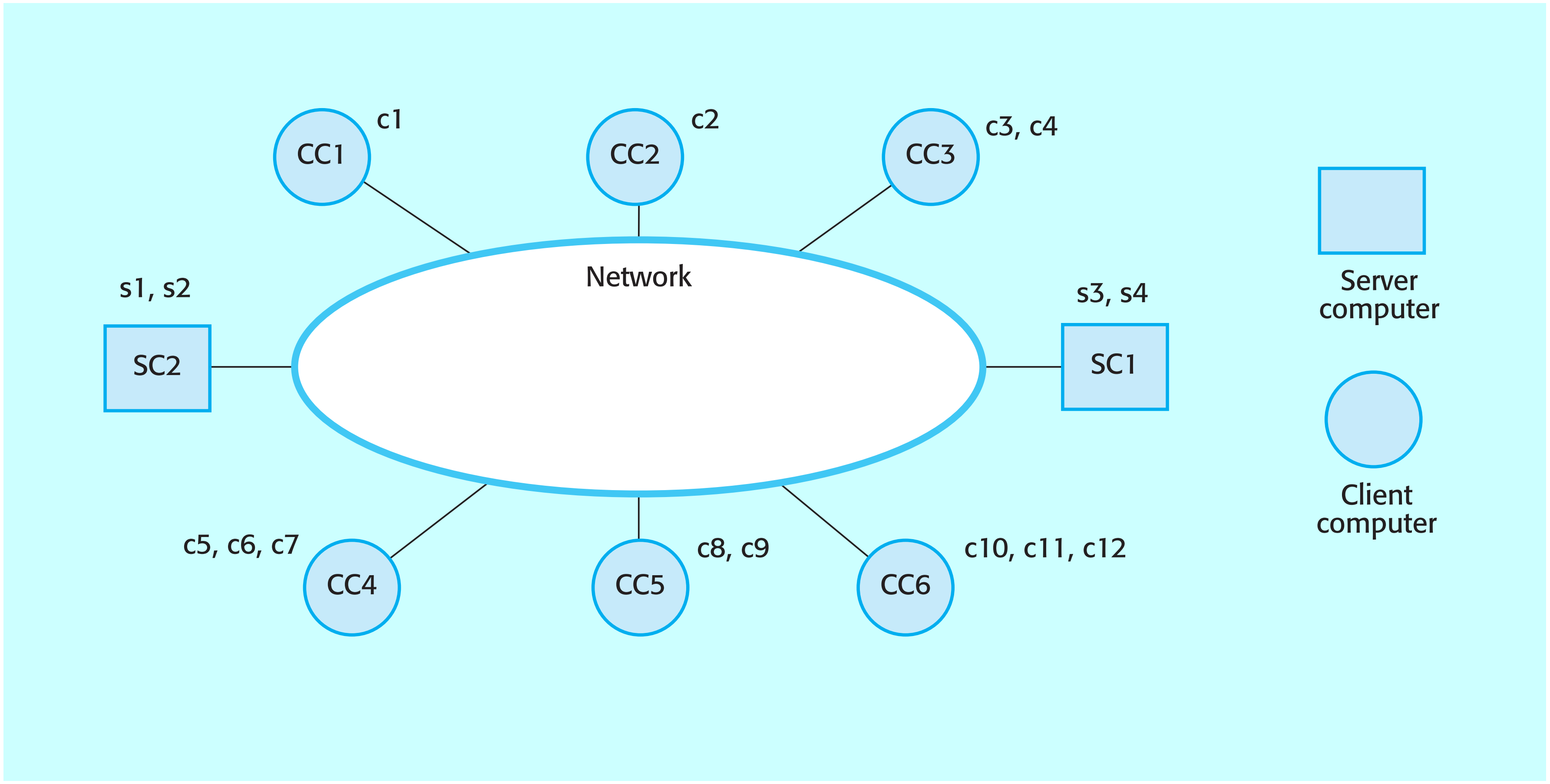
Client-server architectures

- The application is modelled as a set of services that are provided by servers and a set of clients that use these services.
- Clients know of servers but servers need not know of clients.
- Clients and servers are logical processes
- The mapping of processors to processes is not necessarily 1 : 1.

A client-server system

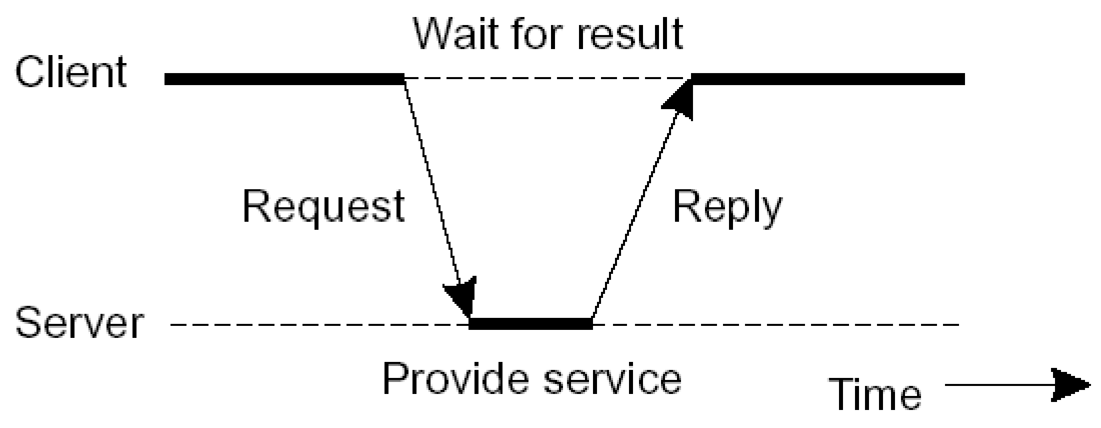


Computers in a C/S network



Client and Servers

- General interaction between a client and a server.



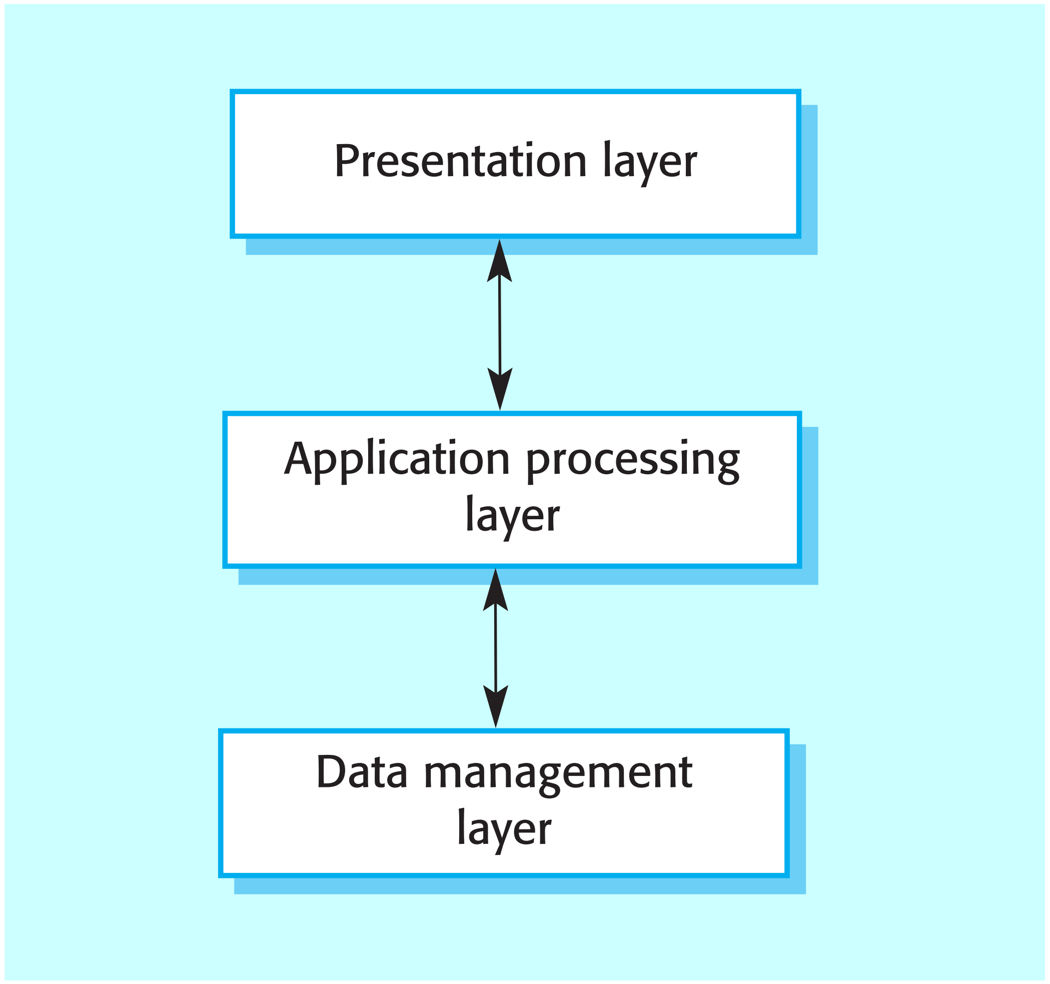
Application Layering

- Client-server applications are usually constructed with a distinction between three levels:
  - User-interface level
  - Processing level
  - Data level
- Clients implement the user-interface level.
- Servers implement the rest.

Layered application architecture

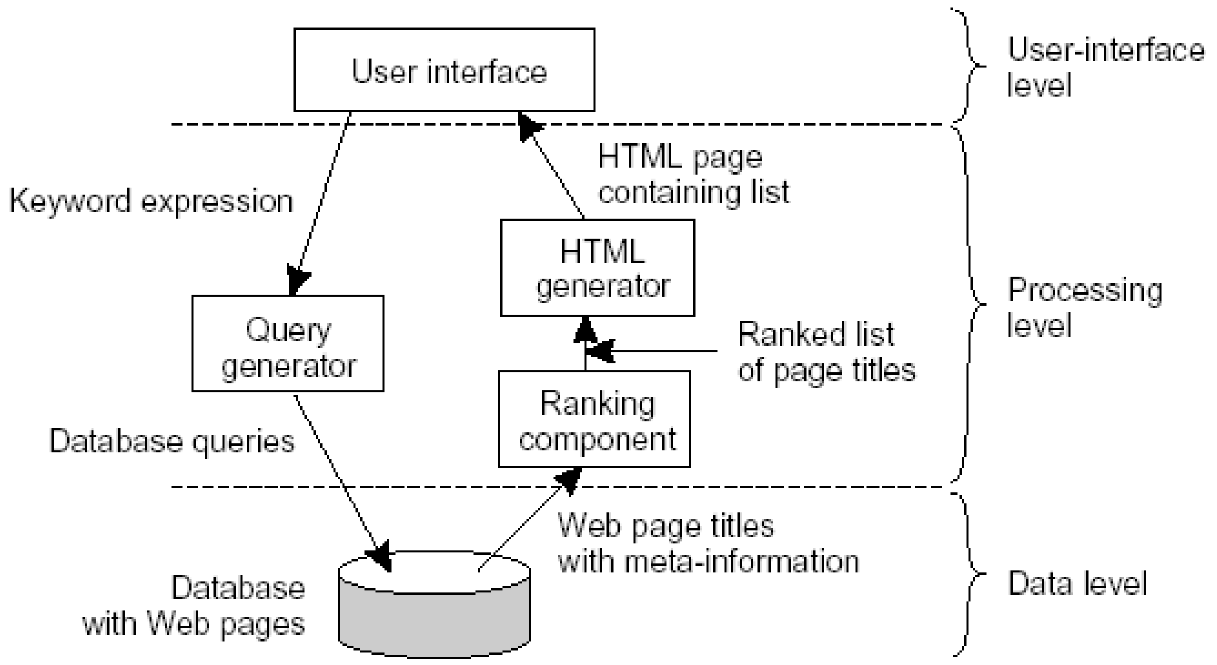
- Presentation layer
  - Concerned with presenting the results of a computation to system users and with collecting user inputs.
- Application processing layer
  - Concerned with providing application specific functionality e.g., in a banking system, banking functions such as open account, close account, etc.
- Data management layer
  - Concerned with managing the system databases.

Application layers

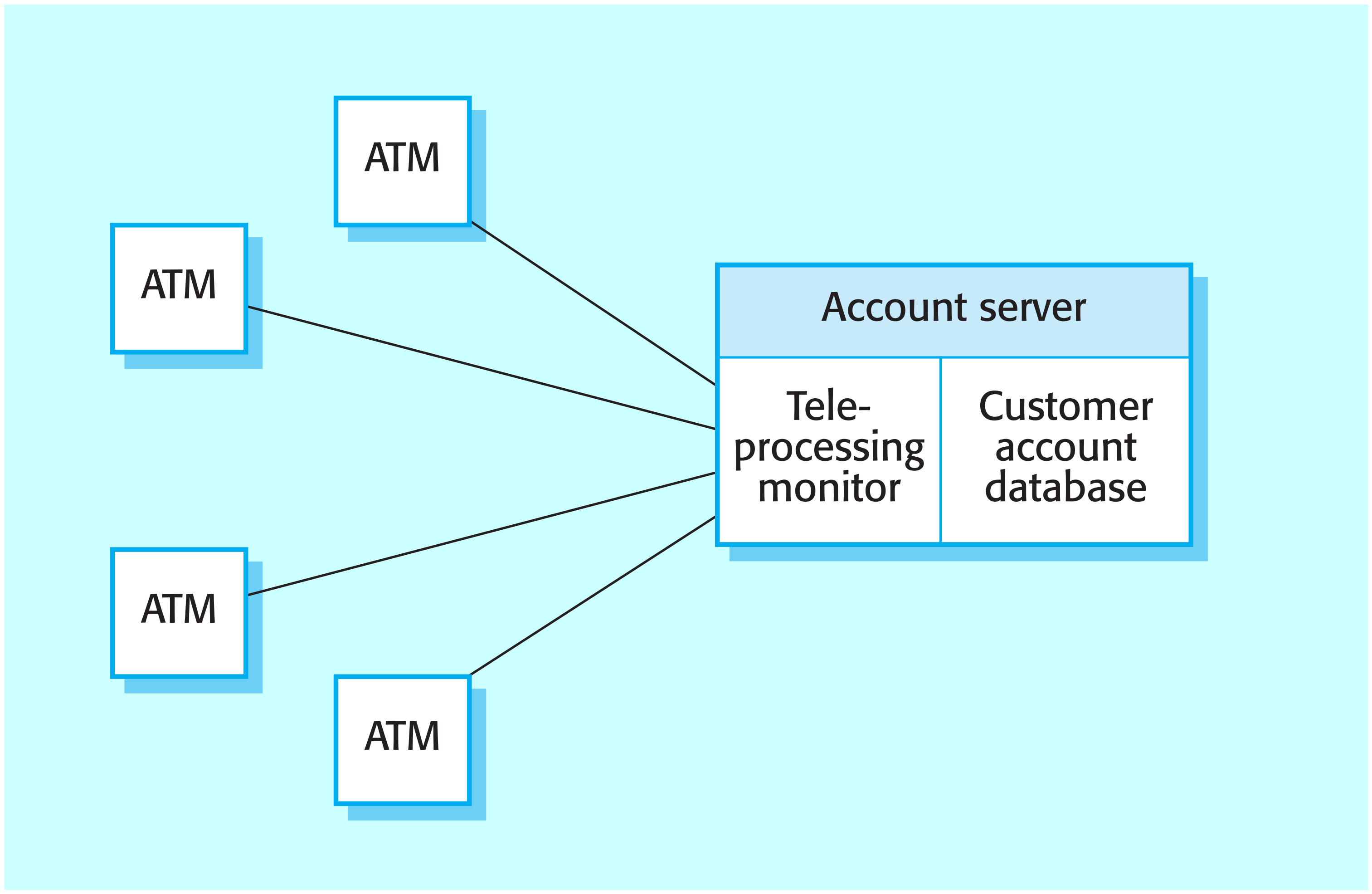


Application Layering

- The general organization of an Internet search engine into three different layers



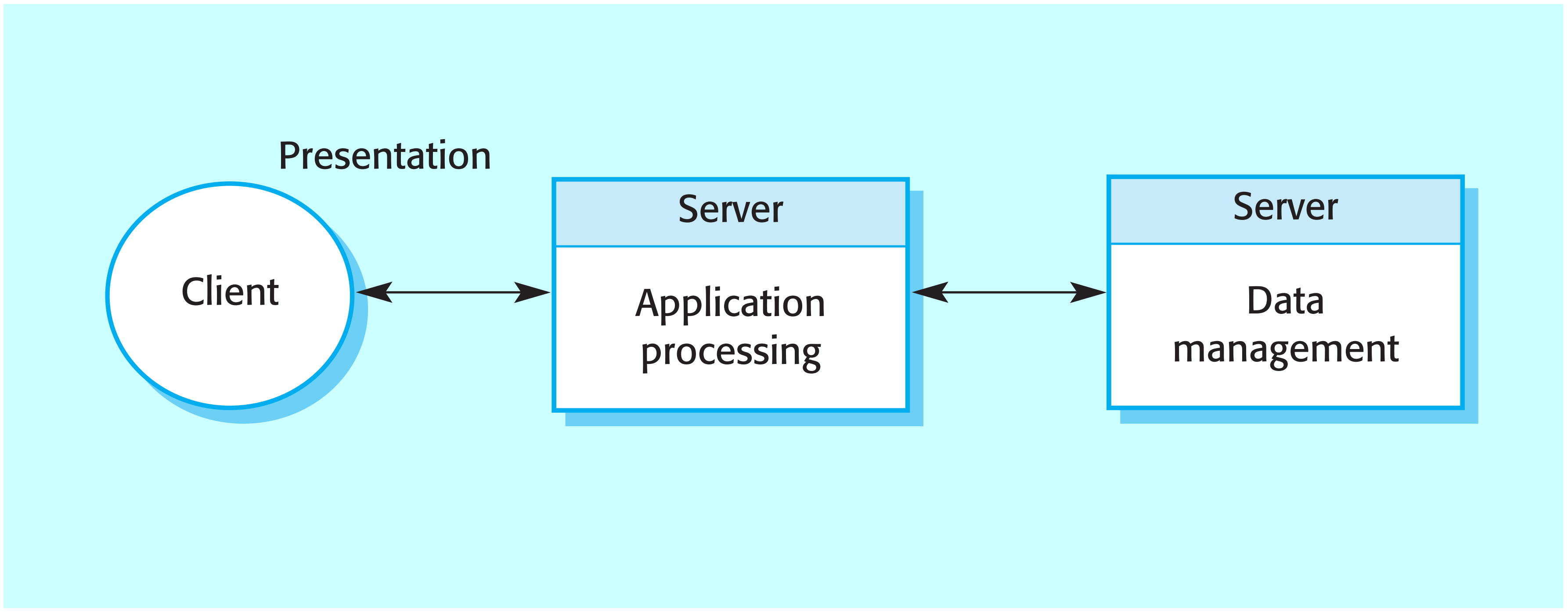
A client-server ATM system



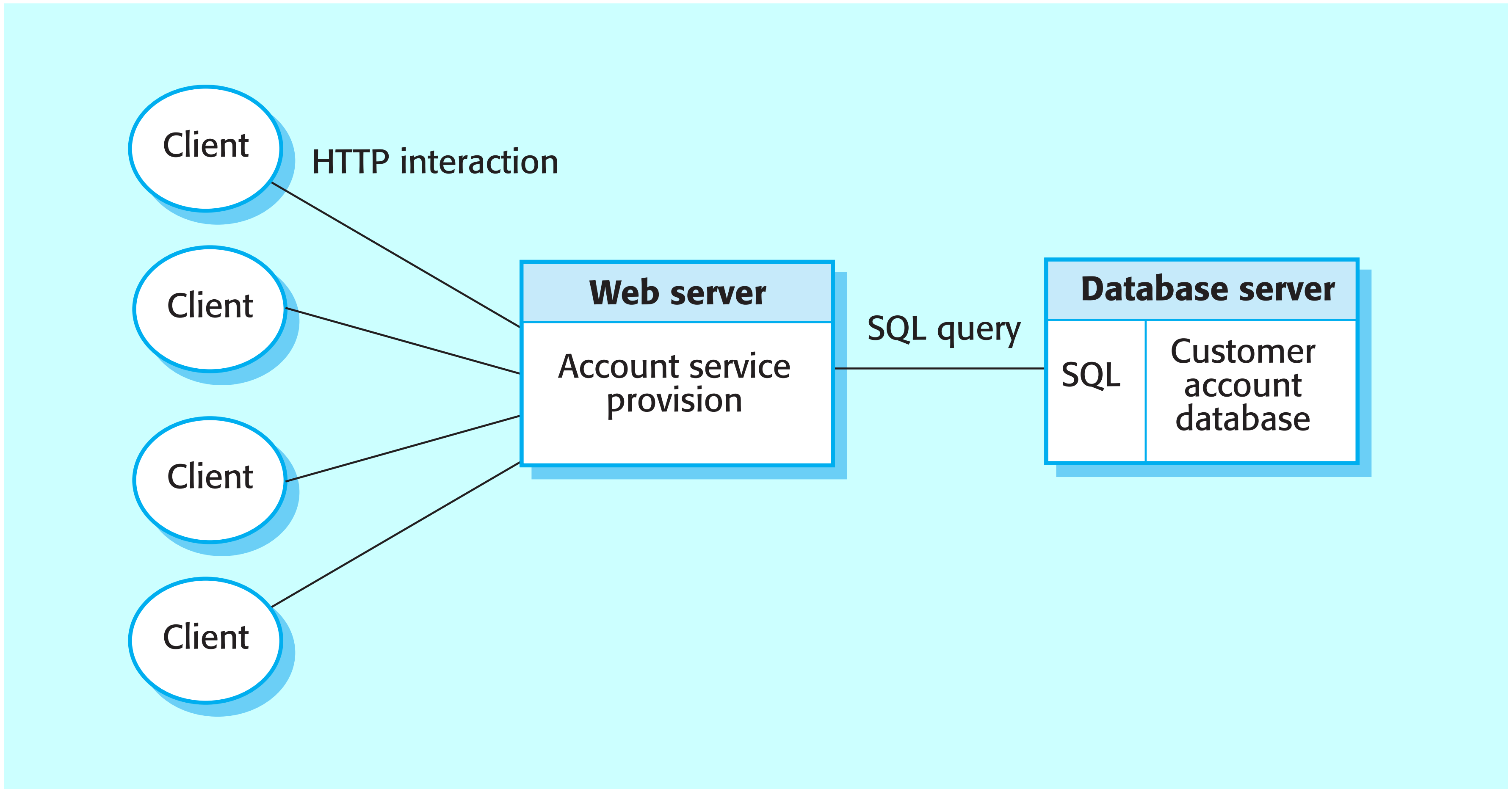
Three-tier architectures

- In a three-tier architecture, each of the application architecture layers may execute on a separate processor.
- Allows for better performance than a thin-client approach and is simpler to manage than a fat-client approach.
- A more scalable architecture - as demands increase, extra servers can be added.

A 3-tier C/S architecture

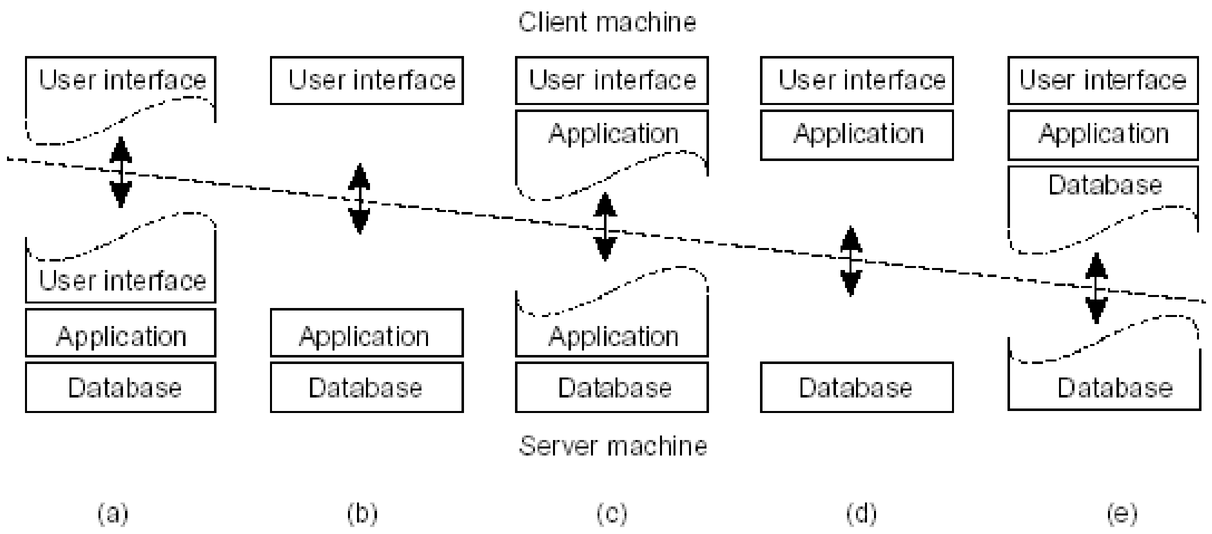


An internet banking system



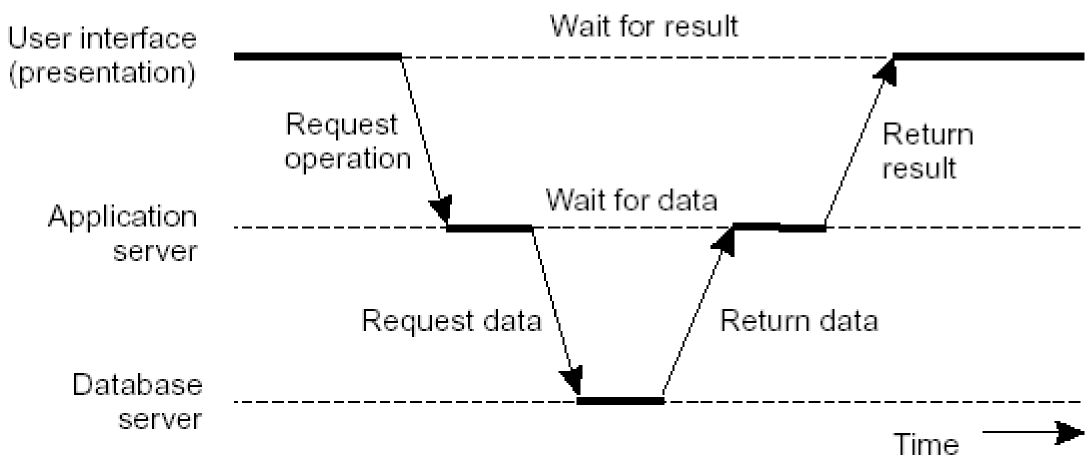
Multi-tiered Architecture

- Alternative client-server organizations (a) – (e).



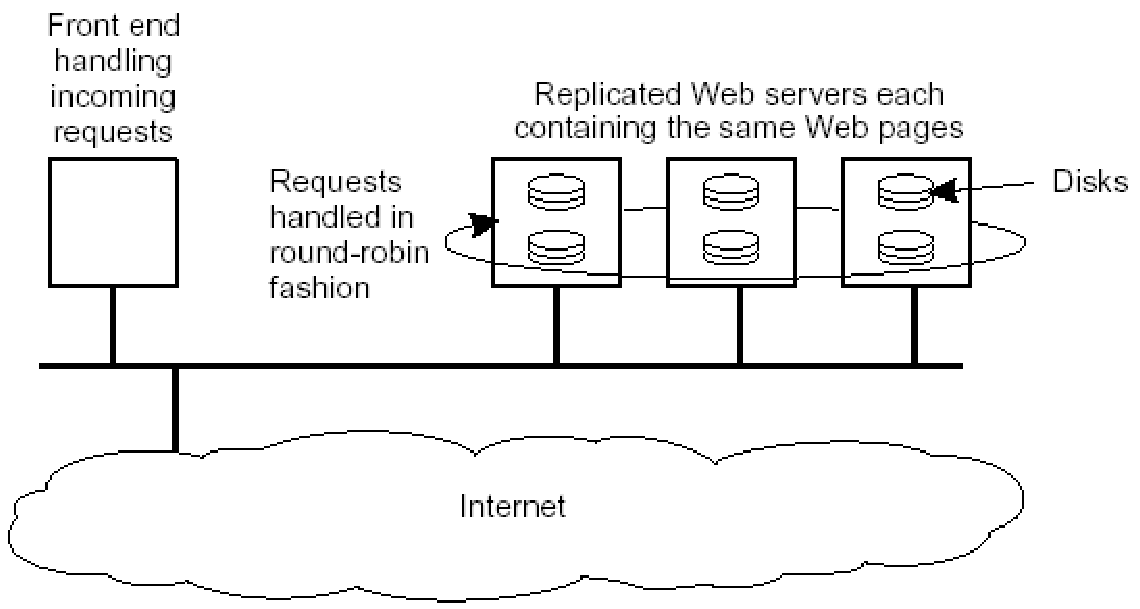
Multi-tiered Architecture

- An example of a three-tiered architecture.
- Note that the application server acts as client when requesting to the database server.



Modern Architectures

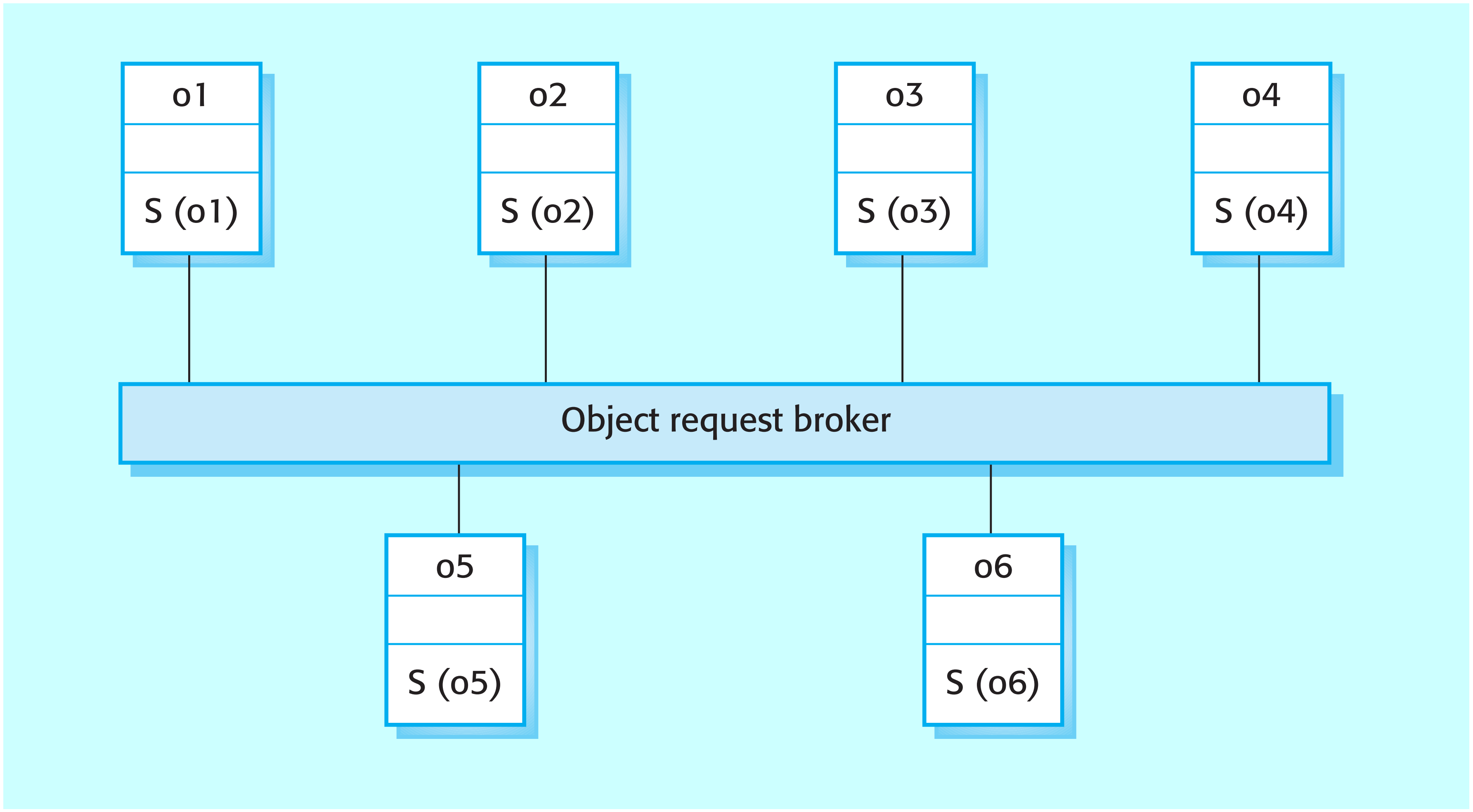
- Multitiered architectures use vertical distribution.
- Modern systems use horizontal distribution.
- An example of horizontal distribution of a Web server.


Front end handling incoming requests
Requests handled in round-robin fashion
Replicated Web servers each containing the same Web pages
Disks
Internet

Distributed object architectures

- There is no distinction in a distributed object architectures between clients and servers.
- Each distributable entity is an object that provides services to other objects and receives services from other objects.
- Object communication is through a middleware system called an object request broker.
- However, distributed object architectures are more complex to design than C/S systems.

Distributed object architecture



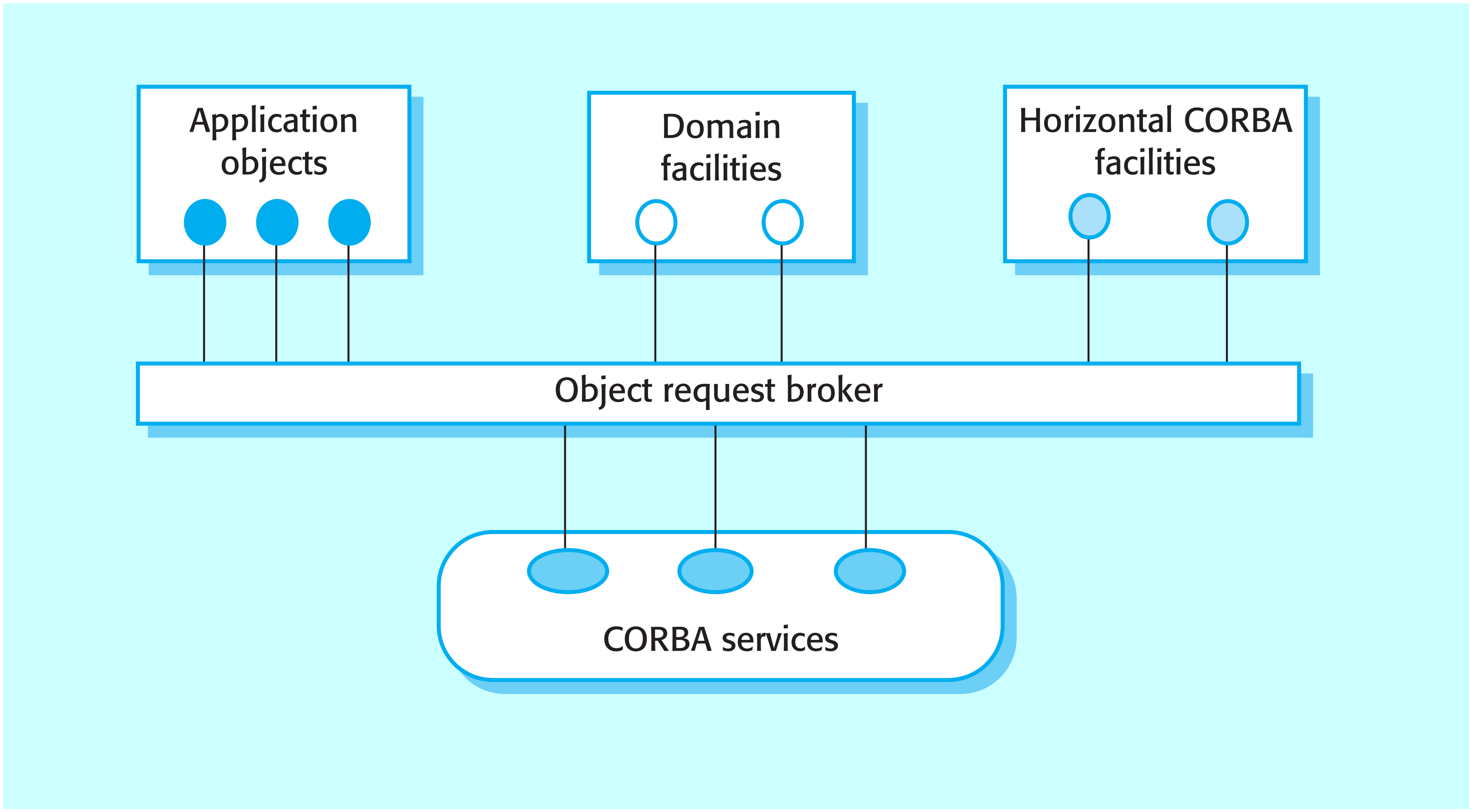
Advantages of distributed object architecture

- It allows the system designer to delay decisions on where and how services should be provided.
- It is a very open system architecture that allows new resources to be added to it as required.
- The system is flexible and scaleable.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required.

Uses of distributed object architecture

- As a logical model that allows you to structure and organise the system. In this case, you think about how to provide application functionality solely in terms of services and combinations of services.
- As a flexible approach to the implementation of client-server systems. The logical model of the system is a client-server model but both clients and servers are realised as distributed objects communicating through a common communication framework.

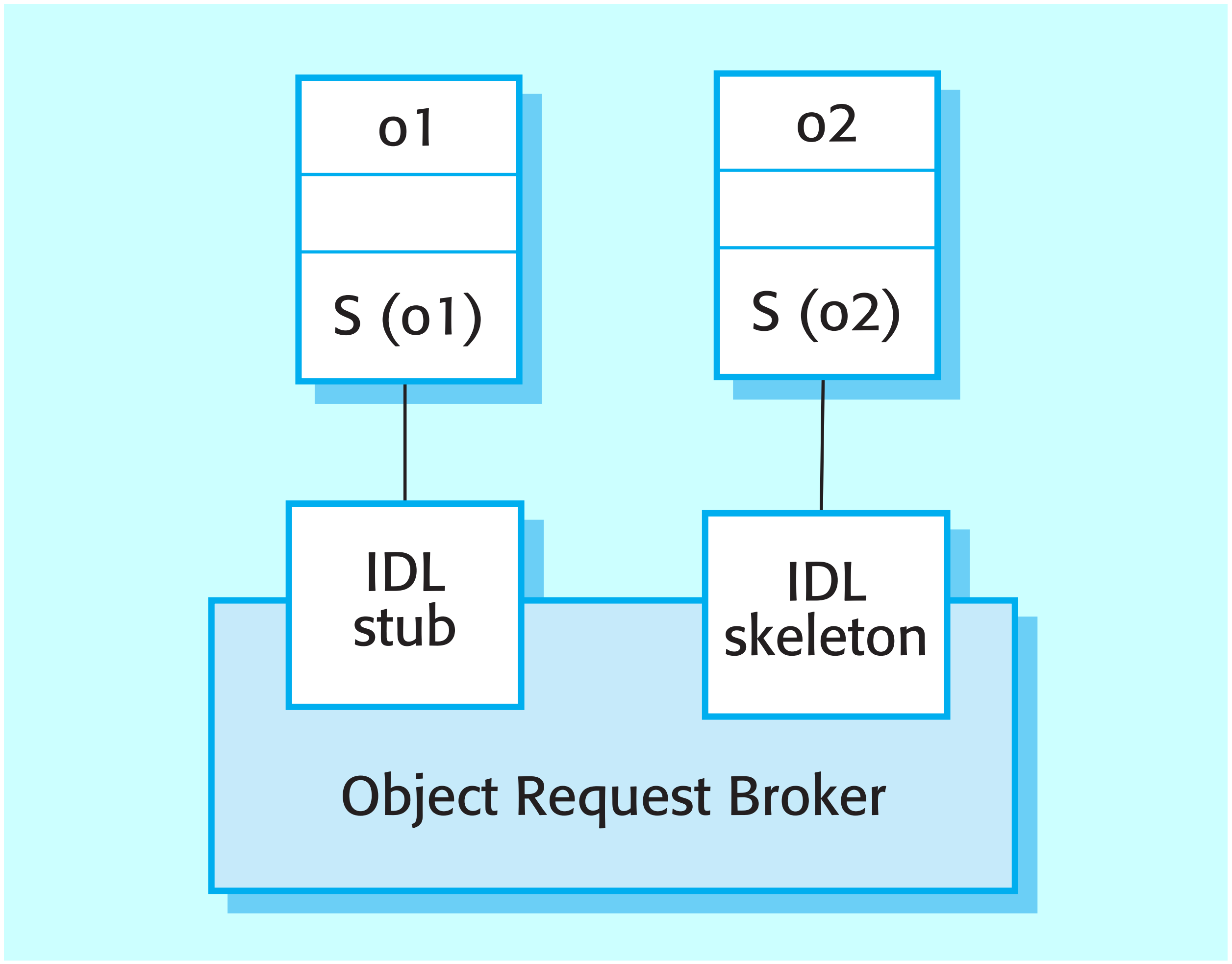
CORBA application structure



Object request broker (ORB)

- The ORB handles object communications. It knows of all objects in the system and their interfaces.
- Using an ORB, the calling object binds an IDL stub that defines the interface of the called object.
- Calling this stub results in calls to the ORB which then calls the required object through a published IDL skeleton that links the interface to the service implementation.

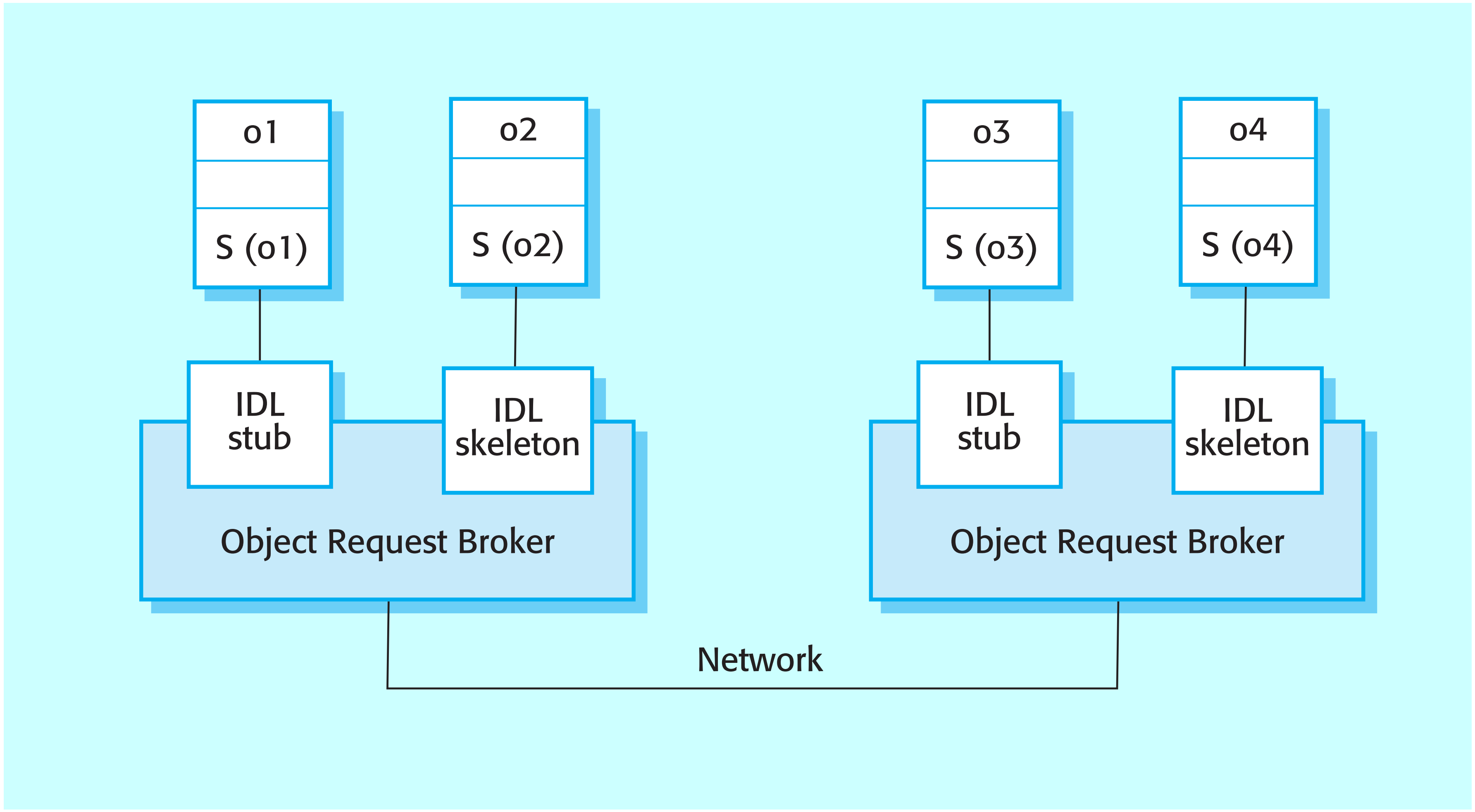
ORB-based object communications



Inter-ORB communications

- ORBs are not usually separate programs but are a set of objects in a library that are linked with an application when it is developed.
- ORBs handle communications between objects executing on the sane machine.
- Several ORBS may be available and each computer in a distributed system will have its own ORB.
- Inter-ORB communications are used for distributed object calls.

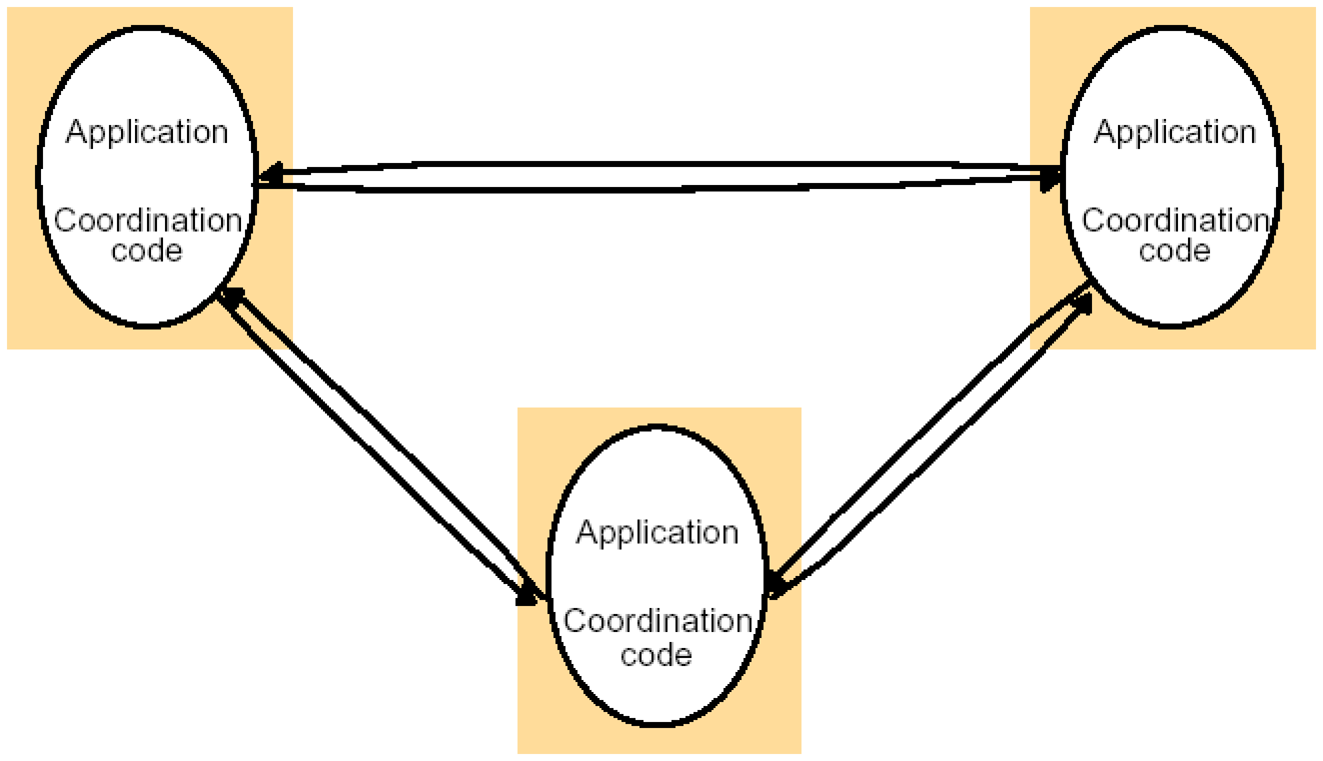
Inter-ORB communications



Peer-to-peer architectures

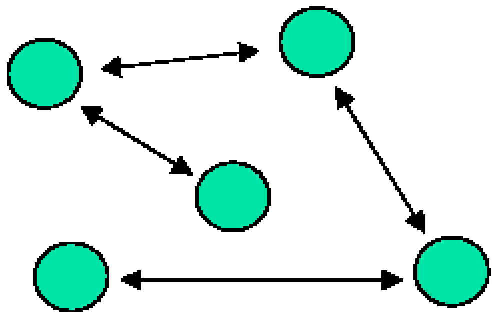
- Peer to peer (p2p) systems are decentralised systems where computations may be carried out by any node in the network.
- The overall system is designed to take advantage of the computational power and storage of a large number of networked computers.
- Most p2p systems have been personal systems but there is increasing business use of this technology.

Peer Process Model



Peer to Peer Architecture

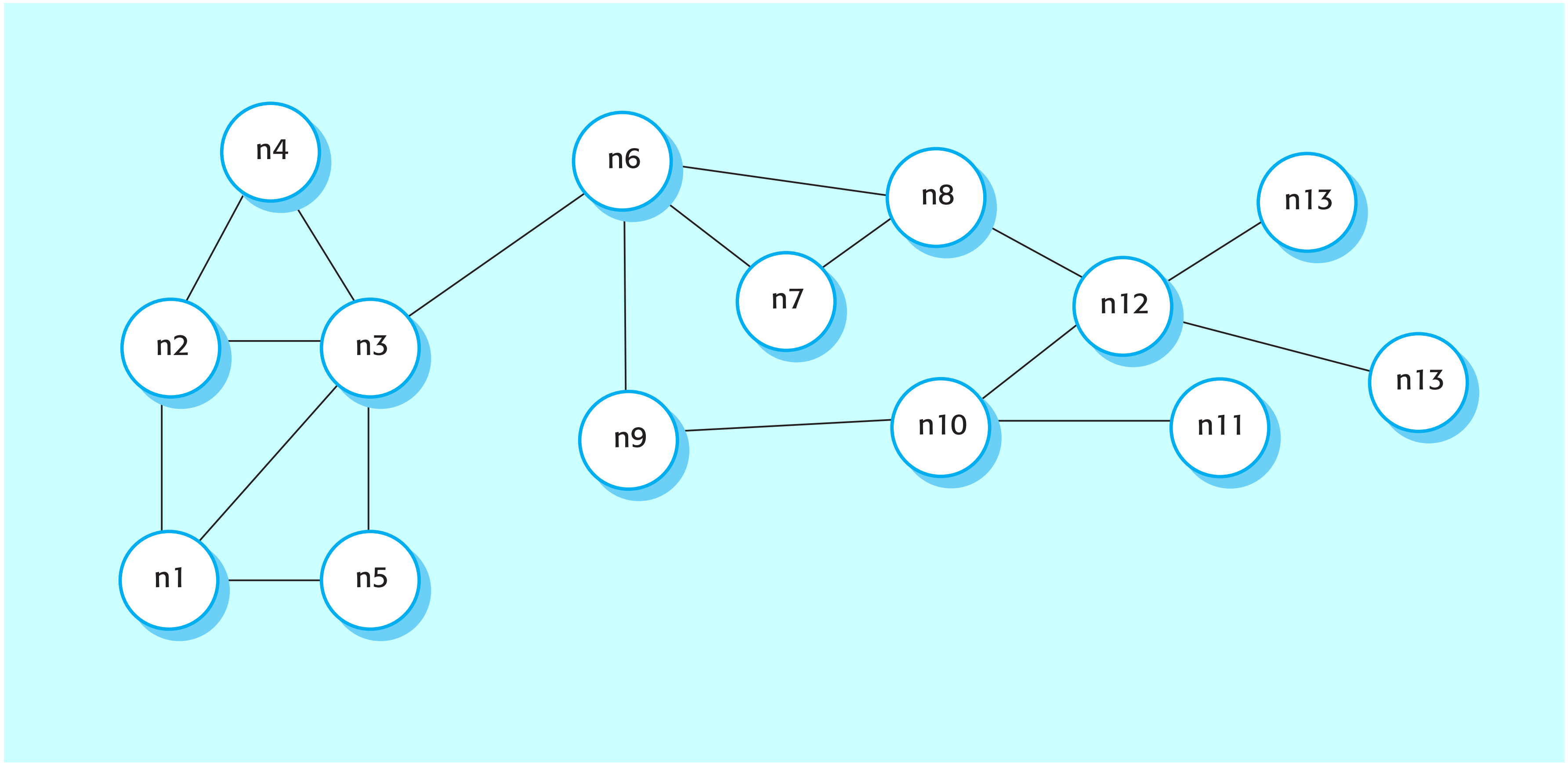
- Nodes have roles and responsibilities.
- May have servers or brokers for locating peers and communication initiation.
- Examples: Gnutella network, IRC, …



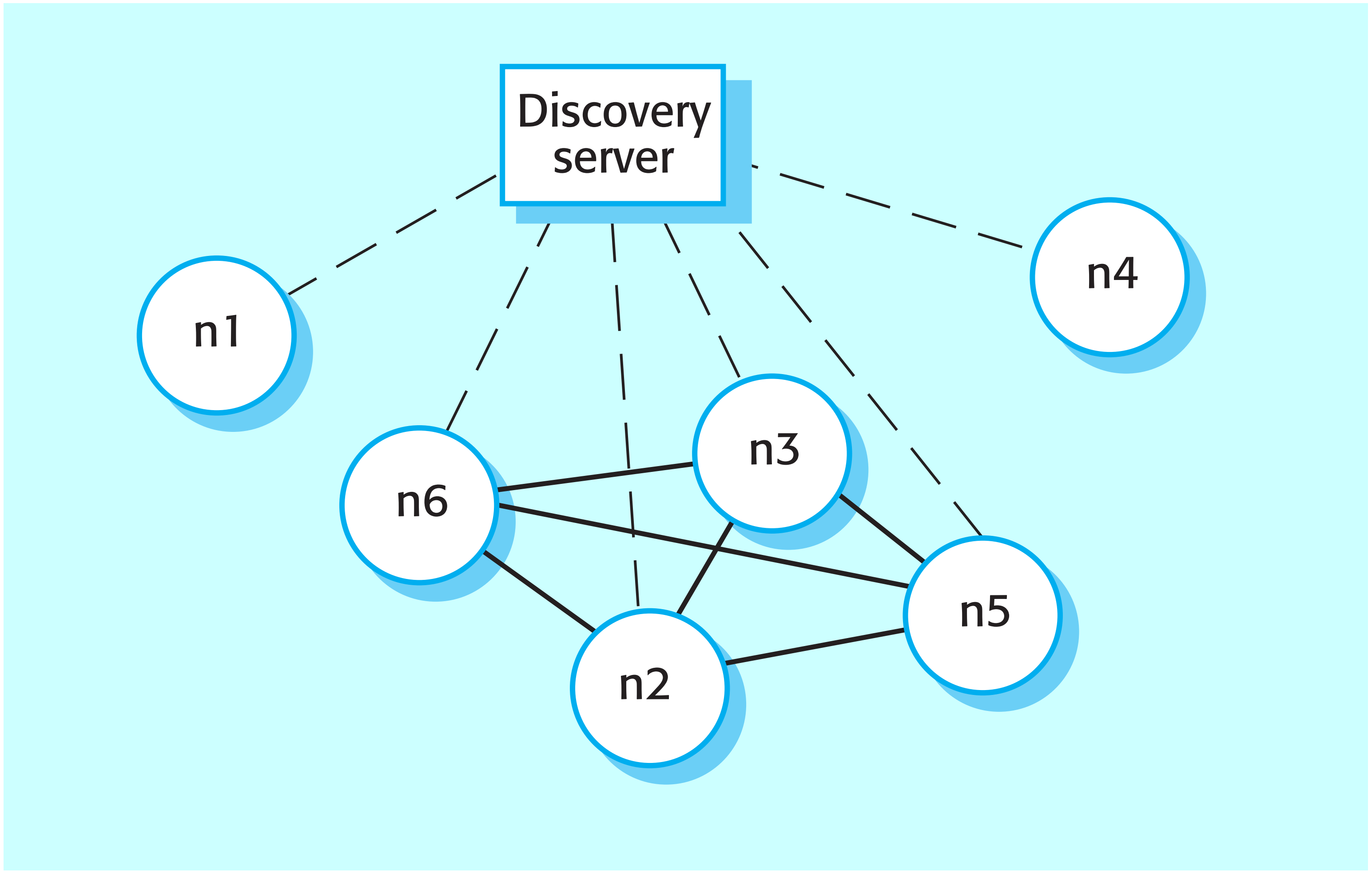
P2P architectural models

- The logical network architecture
  - Decentralised architectures;
  - Semi-centralised architectures.
- Application architecture
  - The generic organisation of components making up a p2p application.
- Focus here on network architectures.

Decentralised p2p architecture



Semi-centralised p2p architecture



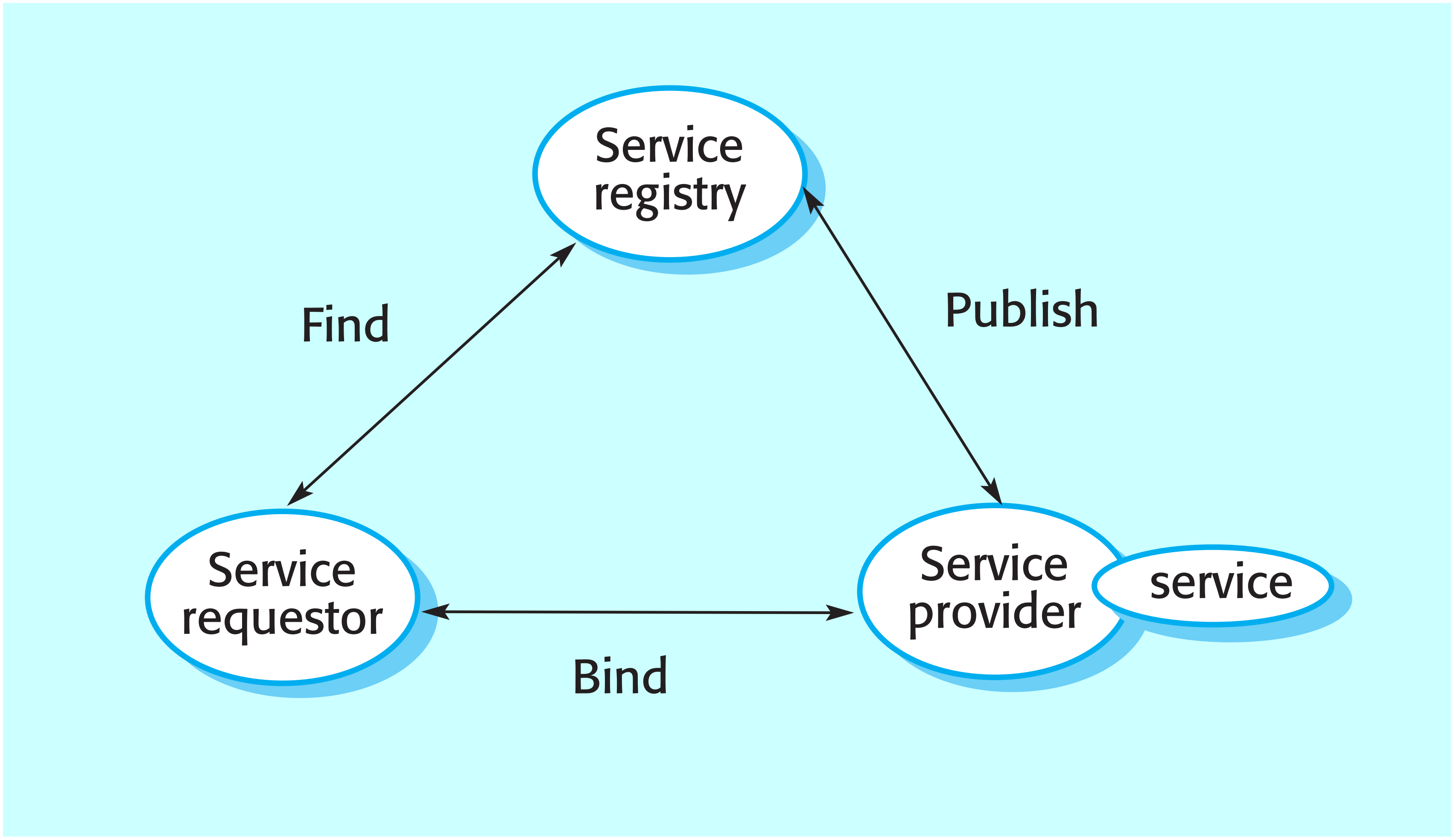
Service-oriented architectures

- Based around the notion of externally provided services **<u>(web services).</u>**
- A web service is a standard approach to making a reusable component available and accessible across the web
  - A tax filing service could provide support for users to fill in their tax forms and submit these to the tax authorities.

A generic service

- *An act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production.*
- Service provision is therefore independent of the application using the service.

Web services



Services and distributed objects

- Provider independence.
- Public advertising of service availability.
- Potentially, run-time service binding.
- Opportunistic construction of new services through composition.
- Pay for use of services.
- Smaller, more compact applications.
- Reactive and adaptive applications.

Web Services Infrastructure

Language and platform independent infrastructure for *loosely-coupled*, *inter-operable*, app2app communication *over the Internet*.

Web Services (Contd.)

Language and platform independent =>

separation of specification and implementation

Loosely coupled =>

message based, synchronous and asynchronous interactions.

Over the Internet =>

No centralized control, use of established protocols, security considerations.

Inter-operable =>

Standards based.

Services standards

- Services are based on agreed, XML-based standards so can be provided on any platform and written in any programming language.
- Key standards
  - SOAP - Simple Object Access Protocol;
  - WSDL - Web Services Description Language;
  - UDDI - Universal Description, Discovery and Integration.

App2App Interaction -- the Web Services Way

- *Transport protocol*
  - HTTP/HTTPS
- *Data Encoding*
  - SOAP (**Simple Object Access Protocol**), XML Schema
- *Interface Description*
  - WSDL (**Web Services Description Language**)
- *Service Description and Discovery*
  - UDDI (**Universal Description, Discovery and Integration**)
- *Security*
  - WS-Security, XML-Signature, XML-Encryption, ...

The Web Services Way

Web Services standards (SOAP, WSDL, …): based on widely accepted Internet friendly technologies (HTTP/HTTPS, XML, …), are mostly orthogonal to each other and enjoy broad support from vendors

Web Services: Network accessible programs, expose functionality by receiving/sending SOAP messages over HTTP/HTTPS, and describe this interface as WSDL descriptions.

Additional Web Services Infrastructure Components

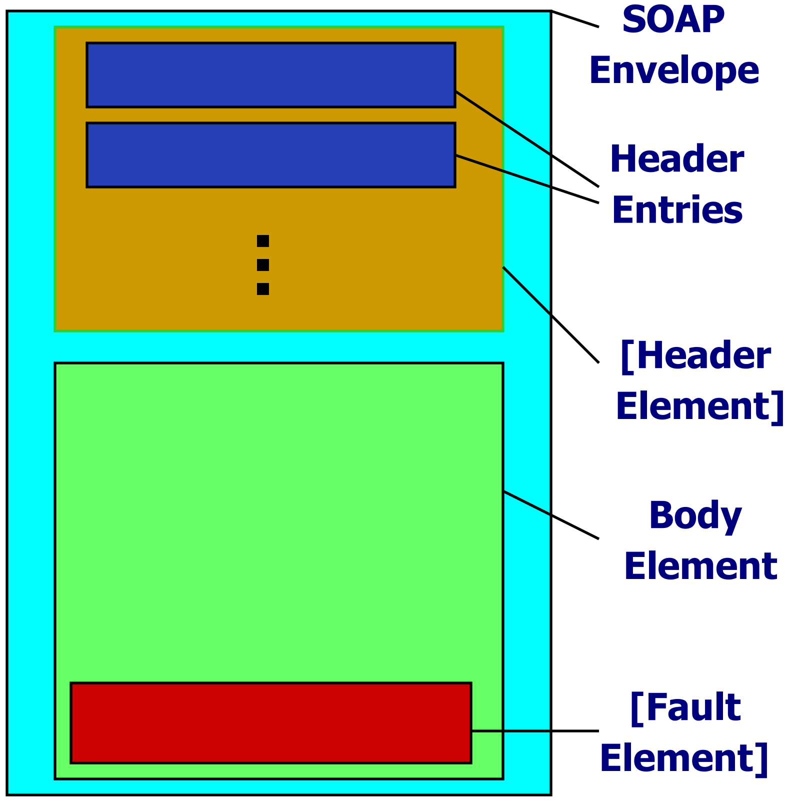
- *Key Management (Security)*
  - XKMS
- *Web Services Management*
  - OMI (**Open Management Interface**)
- ...

Interesting thing to note is that these are Web Services in themselves

SOAP In One Slide

- XML based protocol for exchange of information
  - Encoding rules for datatype instances
  - Convention for representing RPC invocations
- Designed for loosely-coupled distributed computing
  - No remote references
- Used with XML Schema
- Transport independent
- *SOAP with Attachments* allow *arbitrary* data to be packaged.

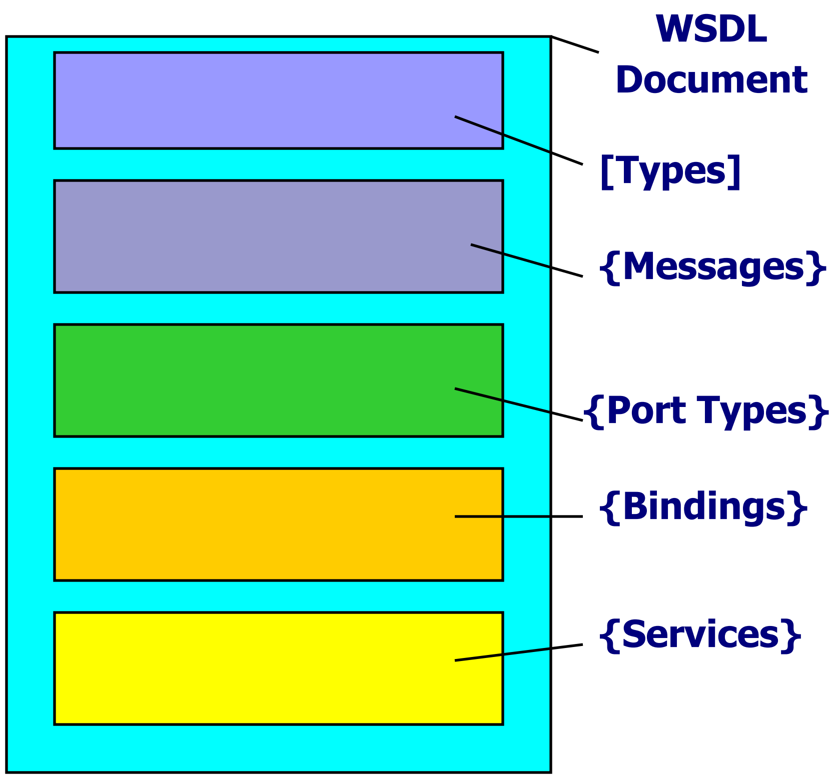
SOAP1.1 Message Structure

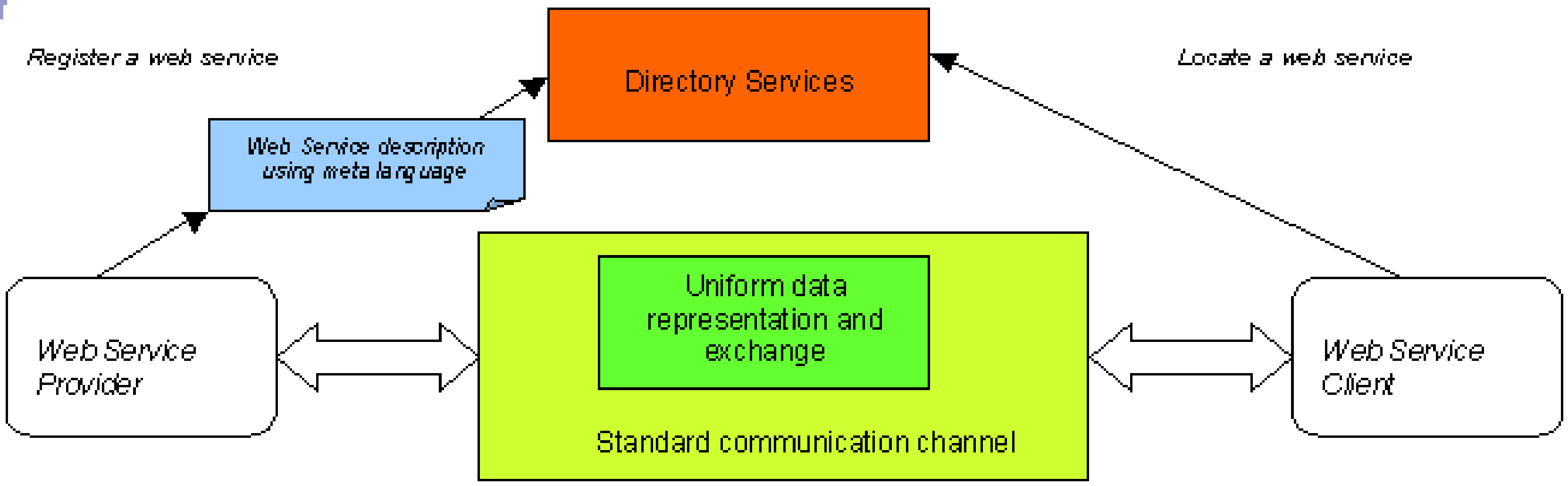


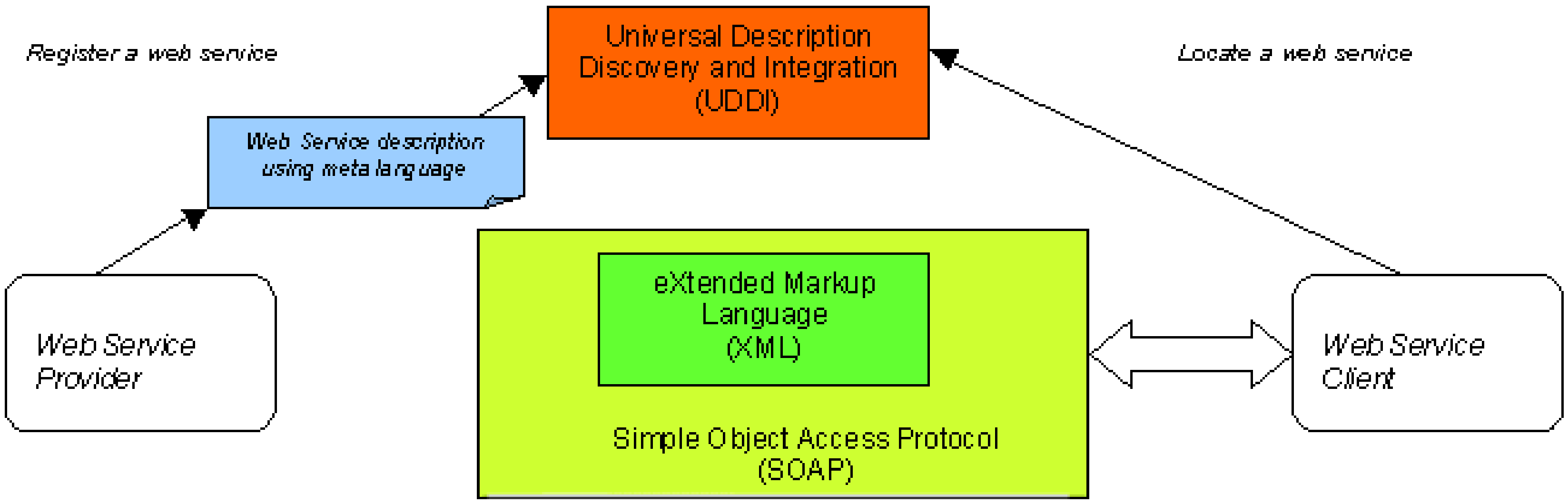
WSDL in One Slide

- A WSDL document describes
  - *What* the service can do
  - *Where* it resides
  - *How* to invoke it
- WSDL are like IDL but lot more flexible and extensible
- Defines binding for SOAP1.1, HTTP GET/POST and MIME
- WSDL descriptions can be made available from an UDDI registry

WSDL1.1 Document Structure




Register a web service
Directory Services
Locate a web service
Web Service description using meta language
Web Service Provider
Uniform data representation and exchange
Standard communication channel
Web Service Client


Register a web service
Universal Description Discovery and Integration (UDDI)
Locate a web service
Web Service description using meta language
Web Service Provider
eXtended Markup Language (XML)
Simple Object Access Protocol (SOAP)
Web Service Client

Uniform Data Representation: XML

- Extended Markup Language (XML) is a meta language that has a well-defined syntax and semantics. The syntax and semantics "self describing" features of XML make it a simple, yet powerful, mechanism for capturing and exchanging data between different applications.XML is a tried and tested way for exchanging data and has been extensively used in B2B applications. Hence, XML is used in the Web Services architecture as the format for transferring information/data between a Web Services provider application and a Web Services client application. The data embedded in an XML file is generated as well as processed using a powerful set of XML parser APIs viz. Simple API for XML (SAX) and Document Object Model (DOM) parser API.

Standard Communication Channel: SOAP

- The Simple Object Access Protocol (SOAP) is the channel used for communication between a Web Services provider application and a client application. The simplicity of SOAP is that it does not define any new transport protocol; instead, it re-uses the Hyper Text Transfer Protocol (HTTP) for transporting data as messages. This use of HTTP as the underlying protocol ensures that Web Services provider applications and client applications can communicate using the Internet as the backbone. It is the use of SOAP that multiplies the capabilities of Web Services and make it all the more exciting!

Standard Meta Language to describe the service offered: WSDL

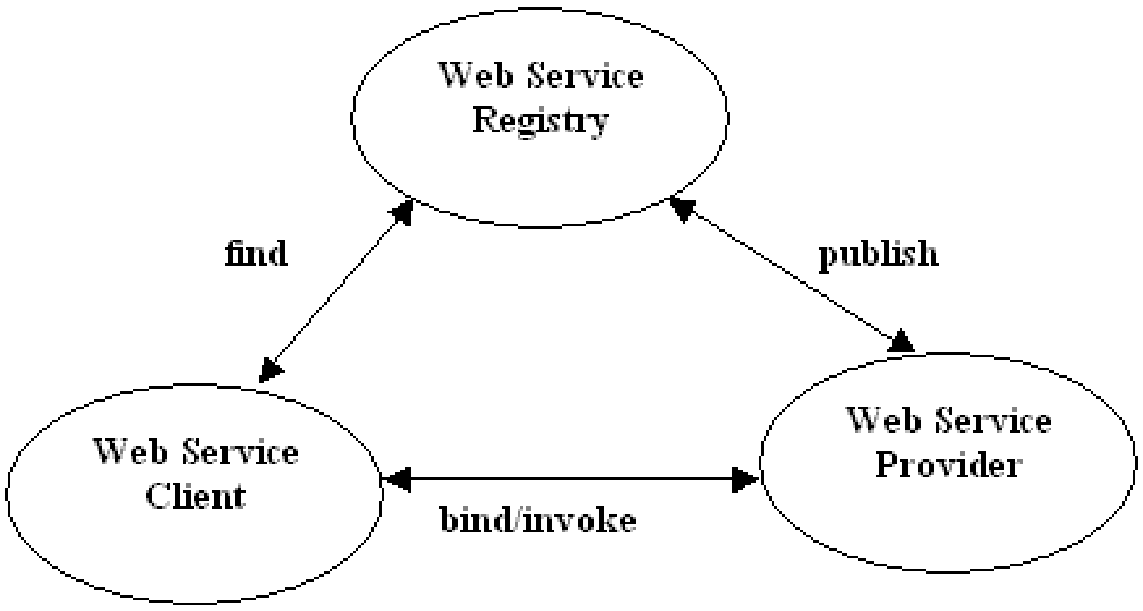
- Web Services provider applications advertise the different services they provide using a standard meta language called the Web Services Description Language (WSDL). Interestingly, WSDL is based on XML and uses a special set of tags to describe a Web service, services provided, where to locate it, and so forth. Client applications obtain information about a Web service prior to accessing and using a Web service of a Web Service provider.

Registering and Locating web services: UDDI

- The "yellow pages" of Web Services is the Universal Description Discovery and Integration (UDDI). Web Services application providers are listed in a registry of service providers using UDDI. Similarly, client applications locate Web Services application providers using UDDI. Like in the case of WSDL, UDDI also is based on XML.

Web Services Application

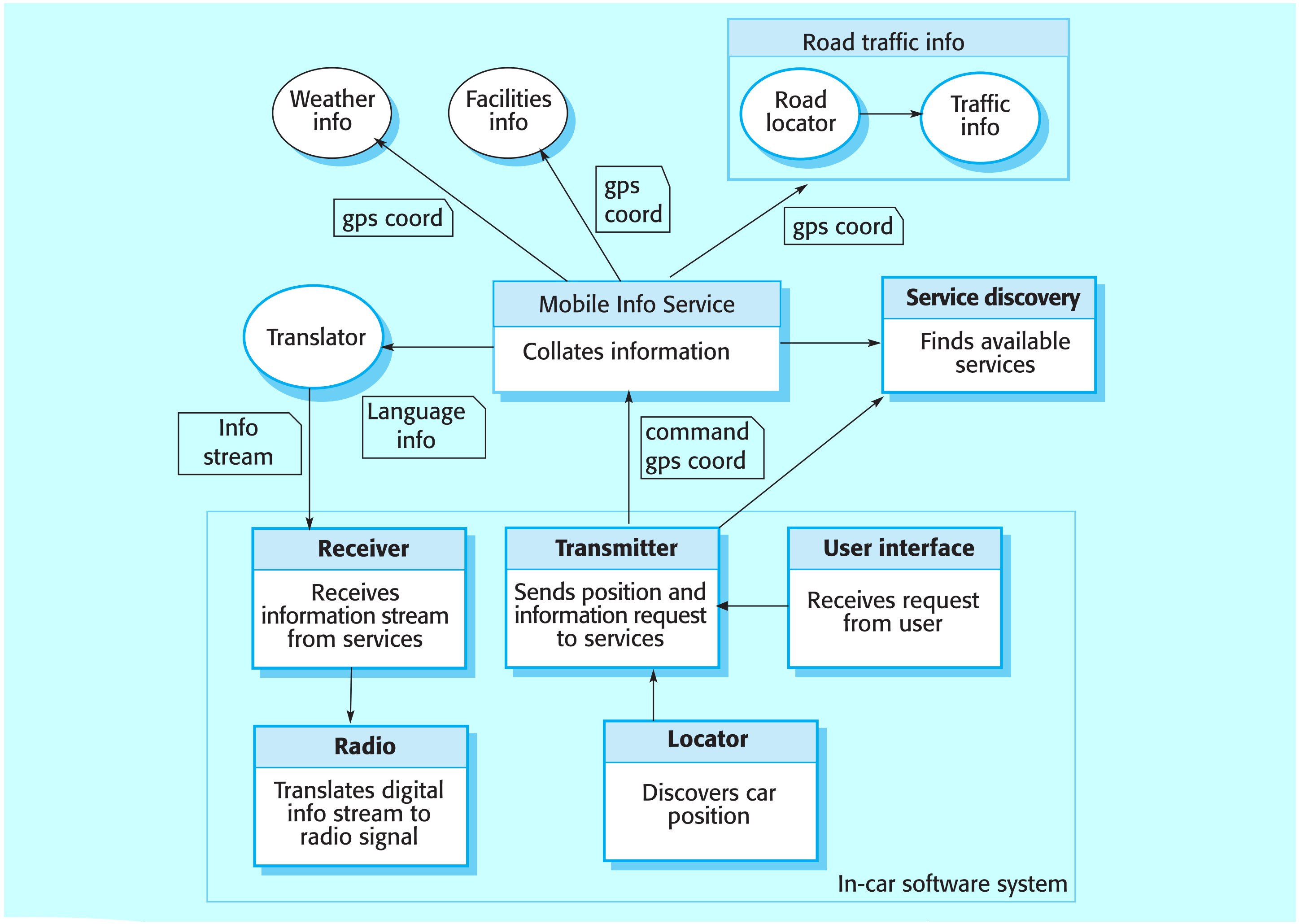
Service Provider	**Service User**
Define the services that will be provided	Identify the services that will be required
Implement the functionality behind the services	Locate the Web service by querying a directory service
Deploy the service provider application	Send the request to the service
Publish the Web services with a directory service	Receive the response from the service
Wait for processing client requests	


Web Service Registry
find
publish
Web Service Client
Web Service Provider
bind/invoke

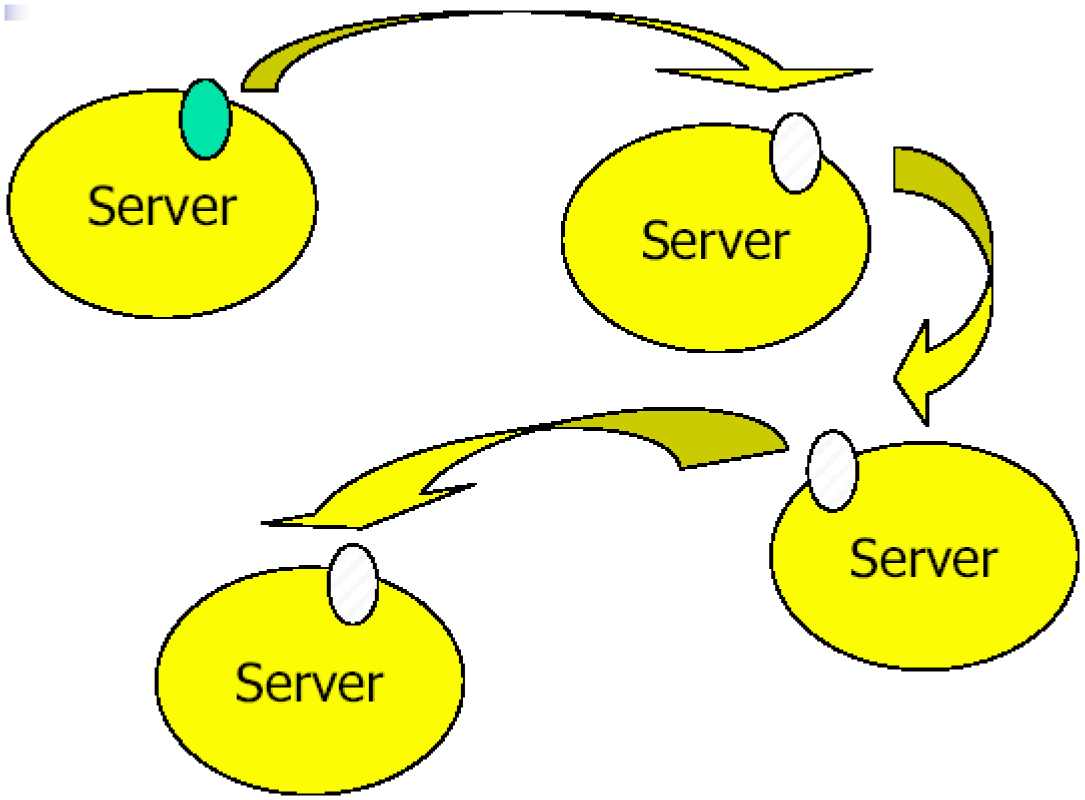
Services scenario

- An in-car information system provides drivers with information on weather, road traffic conditions, local information etc. This is linked to car radio so that information is delivered as a signal on a specific radio channel.
- The car is equipped with GPS receiver to discover its position and, based on that position, the system accesses a range of information services. Information may be delivered in the driver's specified language.

Automotive system



Mobile Code and Mobile Agents



Mobile Code and Mobile Agents

- Mobile Code
  - Code that is sent to a client process to carry out a specific task
  - Examples
    - Applets
    - Active Messages (containing communications protocol code)
- Mobile Agents
  - executing program (code + data), migrating amongst processes, carrying out of an autonomous task, usually on behalf of some other process
  - advantages: flexibility, savings in communications cost
  - virtual markets, worm programs

Few Key points

- Distributed systems support resource sharing, openness, concurrency, scalability, fault tolerance and transparency.
- Client-server architectures involve services being delivered by servers to programs operating on clients.
- User interface software always runs on the client and data management on the server. Application functionality may be on the client or the server.
- In a distributed object architecture, there is no distinction between clients and servers.

Few Key points

- Distributed object systems require middleware to handle object communications and to add and remove system objects.
- The CORBA standards are a set of middleware standards that support distributed object architectures.
- Peer to peer architectures are decentralised architectures where there is no distinction between clients and servers.
- Service-oriented systems are created by linking software services provided by different service suppliers.

Next Topic
Interprocess Communication

Notes !!!

Distributed Systems

BSc IV Computer Science

Dr. M-Waseem Akhtar

Communication

Just a Reminder ..Text Books

- **Distributed Systems Principles and Paradigms** by Andrew S. Tanenbaum and Maarten van Steen
- **Distributed Systems Concepts and Design** by George Coulouris, Jean Dolimore, Tim Kindberg

Communication

- Interprocess communication is at the heart of all distributed systems. Communication in distributed systems is always based on low level message passing as offered by the underlying network.
- Modern distributed systems often consist of thousands or even millions of processes scattered across an unreliable network such as internet.
- Unless the primitive communication facilities of computer networks are replaced by something else, development of large scale distributed applications is extremely difficult.

Communication

- In this lecture, we will study the rules that communicating processes must adhere to, known as protocols and concentrate on structuring those protocols.
- We then look at different widely-used models for communications: Remote Procedure Call (RPC), Remote Method Invocation (RMI), Message Oriented Middleware (MOM), Sockets and Streams.
- But first, let us have a look at some basic concepts and definitions !!!

Communication Subsystem

- collection of hardware and software components that facilitates communication between hosts in a distributed system
- ***Question:*** What are the communication requirements of a distributed system?

Issues and Requirements

- Performance
  - latency
  - data transfer rate
  - bandwidth
- Scalability
- Reliability
- Security
- Mobility
- Quality of Service
- Multicasting (one-to-many comm)

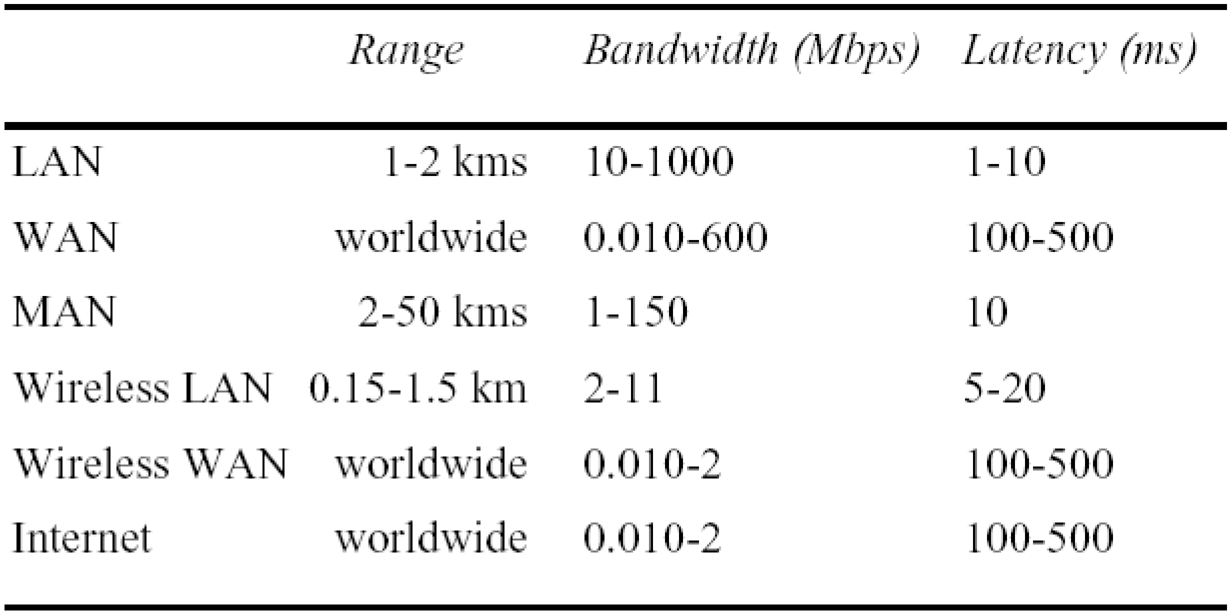
message transmission time =
latency + length/data trans. rate

Network

- A **network** is a collection of computers and other devices that can send and receive data between each other.
- Each machine on a network is called a **node**.
- Nodes that are fully functional computers are also called **hosts**.
- Every network node has an **address** which is a series of bytes that uniquely identify it.
- Addresses are assigned differently on different kinds of networks.

Recap:
The concepts you should already know, if not, please revise

Network Types

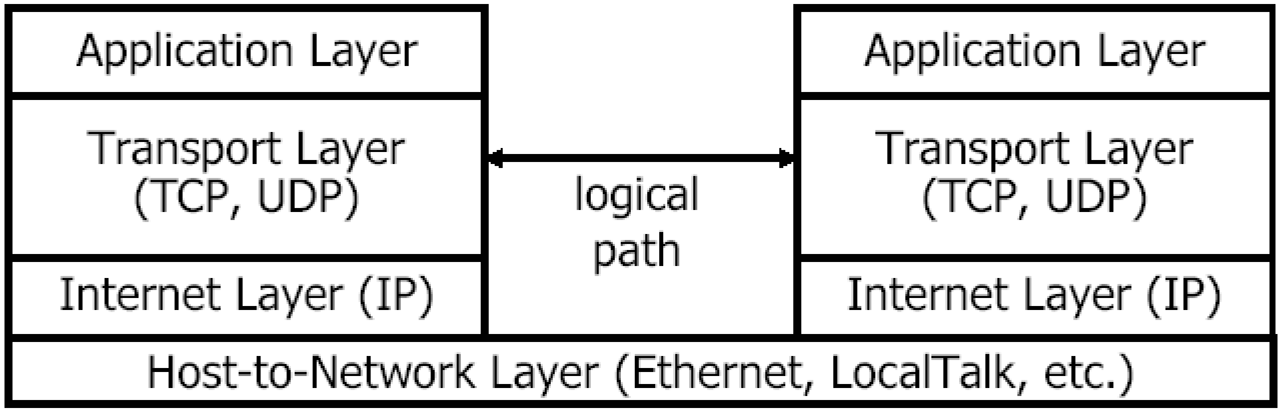


	Range	*Bandwidth (Mbps)*	*Latency (ms)*
LAN	1-2 kms	10-1000	1-10
WAN	worldwide	0.010-600	100-500
MAN	2-50 kms	1-150	10
Wireless LAN	0.15-1.5 km	2-11	5-20
Wireless WAN	worldwide	0.010-2	100-500
Internet	worldwide	0.010-2	100-500

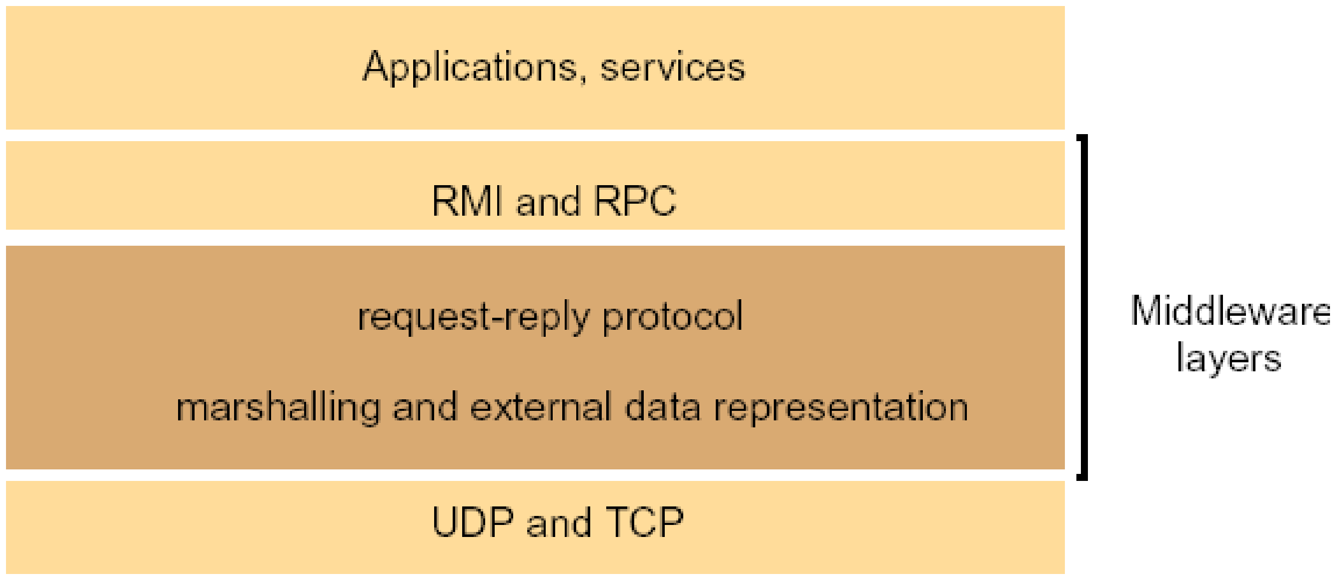
Recap:
The concepts you should already know, if not, please revise

Internet Protocol Layers

- The Internet connection can be simplified into a four-layer model. Each layer only talks to the layers immediately above and below it.
- Layers model reduces complexity and increase modularity.


Application Layer
Transport Layer (TCP, UDP)
Internet Layer (IP)
logical path
Application Layer
Transport Layer (TCP, UDP)
Internet Layer (IP)
Host-to-Network Layer (Ethernet, LocalTalk, etc.)

Middleware Layers



Message Synchronization

- **Asynchronous Communication**
  - Non-blocking send: sender process released after message copied into sender's kernel
- **Synchronous Communication**
  - Blocking send: sender process released after message transmitted to network
  - Reliable blocking send: sender process released after messages received by receiver's kernel
  - Explicit blocking send: sender process released after message received by receiver process
  - Request and Reply: sender process released after message been processed by receiver and response returned to sender

Message Destinations

- **IP (address, port)**
  - A port has exactly one receiver but can have many senders.
  - Processes may use multiple ports to receive messages.
- To provide **location transparency**
  - refer to services by name and use a name server or binder for name to server location translation
  - the OS provides location independent identifiers and handles the identifiers to lower level address mapping

Message Delivery

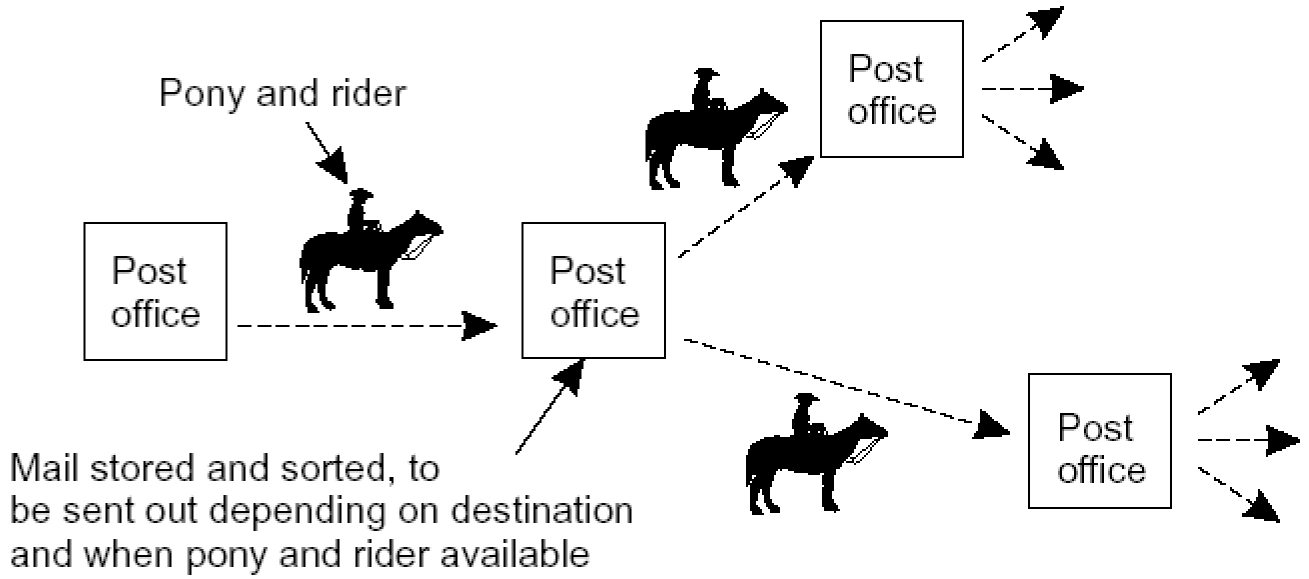
- Reliability
  - **validity** - a point-to-point message service is reliable if messages are guaranteed to be delivered despite a 'reasonable' number of packets lost
  - **integrity** - messages must arrive uncorrupted and without duplication
- Ordering
  - messages be delivered in sender order

Messaging Characteristics

- Message communication can be
  - persistent – a submitted message is stored by the communication system as long as it takes to deliver.
  - transient – a message is stored only as the sending and receiving application are executing.
- Message communication can also be
  - asynchronous – sender continues immediately after it has submitted its message
  - synchronous – sender is blocked until its message is stored in a local buffer at the receiving host, or actually delivered

Persistence and Synchronicity

- Persistent communication of letters back in the days of the Pony Express.

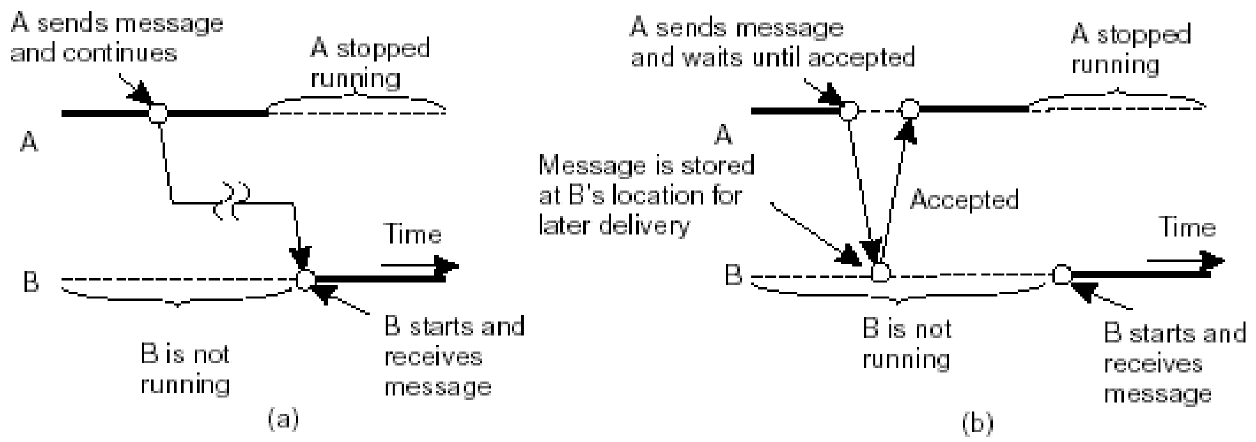

Pony and rider
Post office
Post office
Post office
Post office
Mail stored and sorted, to be sent out depending on destination and when pony and rider available

Sending a letter stars with depositing it at the local post office.
The post office is responsible for sorting letters for the next post office on the route to their respective final destinations.
Letter were kept in the bags, until there was a pony and rider available

Persistence and Synchronicity

a) Persistent asynchronous communication

b) Persistent synchronous communication

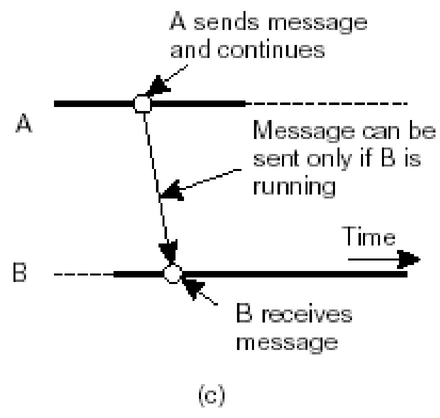


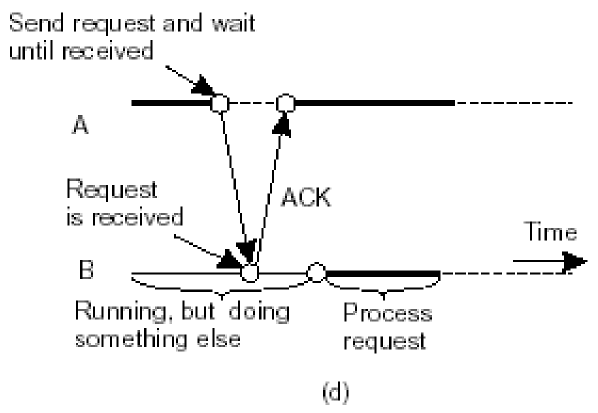
A sends message and continues
A stopped running
A
B
Time
B is not running
B starts and receives message
(a)
A sends message and waits until accepted
A stopped running
A
Message is stored at B's location for later delivery
Accepted
Time
B
B is not running
B starts and receives message
(b)

Persistence and Synchronicity

c) Transient asynchronous communication

d) Receipt-based transient synchronous communication

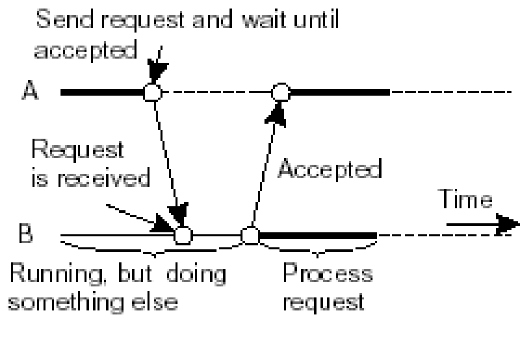

A sends message and continues
A
Message can be sent only if B is running
Time
B
B receives message
(c)


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

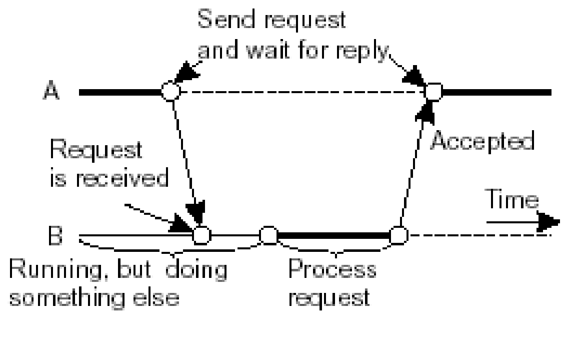
Persistence and Synchronicity

e) Delivery-based transient synchronous communication at message delivery

f) Response-based transient synchronous communication


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

(e)


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

(f)

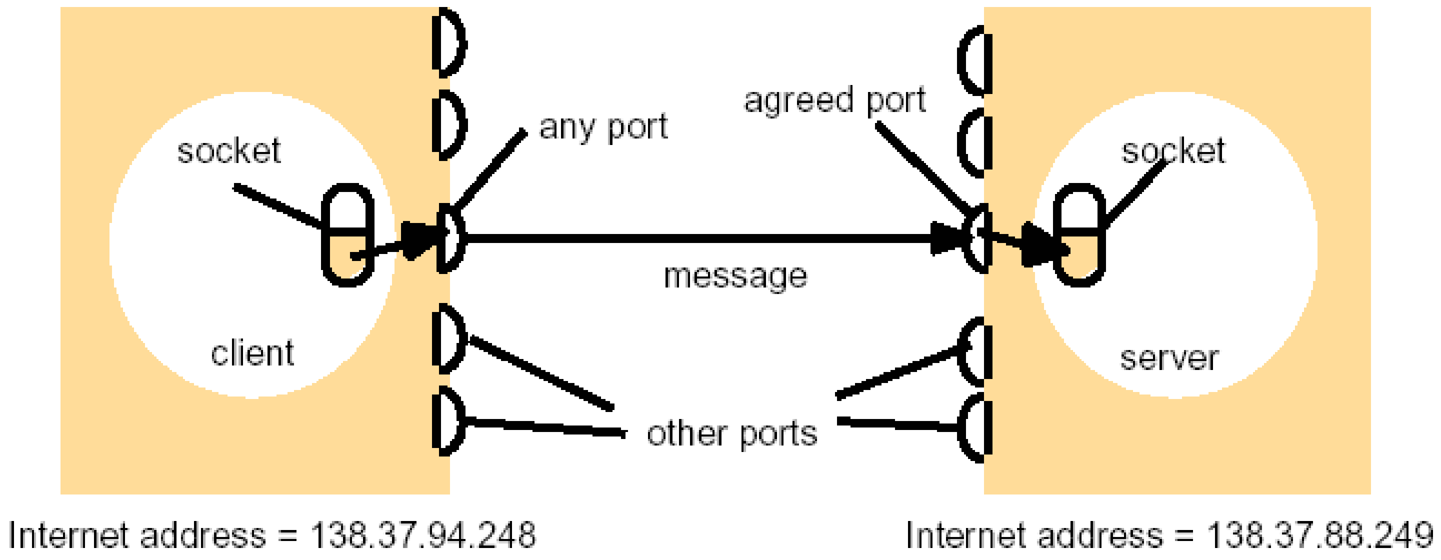
Socket Based Communication

Sockets and Ports

- Many message-oriented systems are built on top of the transport layer messaging through sockets.
- Socket abstraction provides an endpoint for communication between processes.
- Messages are transmitted between a socket in one process and a socket in another.
- To receive messages, a socket must be bound to a local port and IP address.
- Messages sent to a (address, port) can be received only by a process whose socket is associated with that (addres, port).
- A process can not share ports with others.
- May use the same socket for sending and receiving.
- Each socket is associated with a protocol.

An Example of Message Orientated Transient Communication

Sockets and Ports


socket
client
any port
agreed port
message
other ports
socket
server
Internet address = 138.37.94.248
Internet address = 138.37.88.249

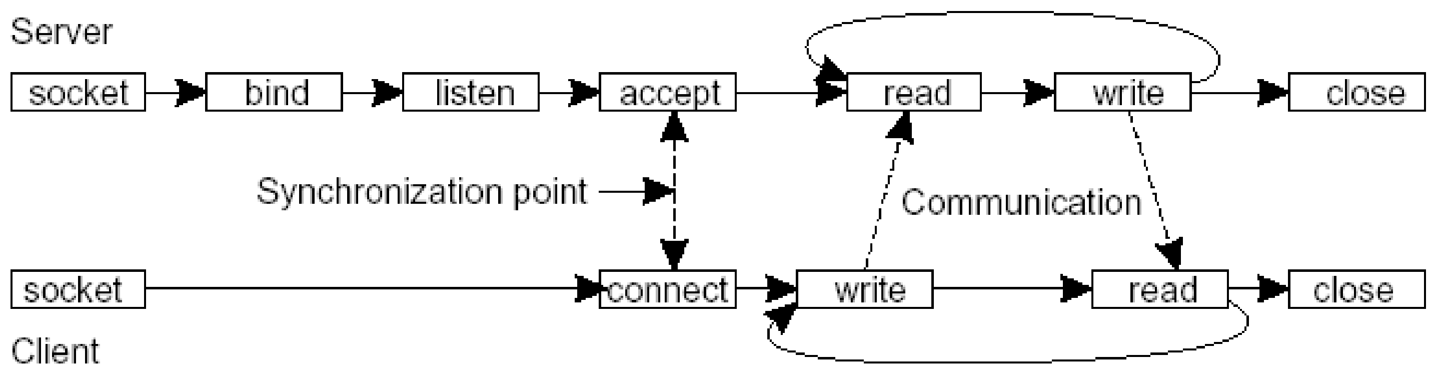
Berkeley Sockets

- Socket primitives for TCP/IP.

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

Berkeley Sockets

- Connection-oriented communication pattern using sockets.



We will come back to sockets again, in the stream communication section

Java API for IP Addresses

- Java provides the `InetAddress` class to represent Internet addresses.

```
InetAddress aHost =
    InetAddress.getByName("www.csie.ndhu.edu.tw");
```

- The method can throw an `UnknownHostException`.
- The class allows us to access Internet hosts by their DNS names instead of numeric IP address.
- We will discuss Java network programming along the way.

UDP Datagram Communication

- no acknowledgement or retries
- **Message buffering and size**
  - must use buffer (array of bytes) to receive
  - if the message is too big, it is truncated
  - IP allows 2^{16} bytes message length
  - 8K is the most commonly used message size
- **Blocking** - non-blocking sends, blocking receives
- **Timeouts** - can be set on sockets
- **Receive from any**
  - a receive gets a message addressed to its socket from any origin
  - the IP address and port can be checked

UDP Failure Model

- Omission failures
  - use checksum to detect corrupted message
  - both send-omission and receive-omission are treated as omission failures in the channel
- Ordering
  - message can be delivered out of order
- Applications using UDP must provide their own checks for reliable communication.
- This can be achieved by using acknowledgements.

Java API for UDP Datagrams

- Java API provides to classes for UDP.
- `DatagramPacket` is for constructing and receiving messages.

message buffer	length	IP address	port number

- `DatagramSocket` supports sockets for sending and receiving UDP datagrams.
  - `send, receive`
  - `setSoTimeout` // set timeout
  - `connect` // setting up connection

Java UDP Client Example

- UDP client sends a message and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient {
    public static void main(String args[]) {
        // args give message contents and server hostname
        try {
          DatagramSocket aSocket = new DatagramSocket();
          byte[] m = args[0].getBytes();  // the message
          InetAddress aHost = InetAddress.getByName(args[1]);
          int serverPort = 6789;
          DatagramPacket request = new DatagramPacket(
                              m,  args[0].length(), aHost, serverPort);
          aSocket.send(request);
```

Java UDP Client Example

```
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(
                                buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " +
                                     new String(reply.getData()));
            aSocket.close();
        } catch (SocketException e) {
            System.out.println("Socket: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IO: " + e.getMessage());}
    }
}
```

message buffer	length	IP address	port number

Java UDP Server Example

```
import java.net.*;
import java.io.*;
public class UDPServer {
    public static void main(String args[]) {
        try {
           DatagramSocket aSocket = new DatagramSocket(6789);
           byte[] buffer = new byte[1000];
           while(true) {
               DatagramPacket request = new DatagramPacket(
                      buffer, buffer.length);
               aSocket.receive(request);
               DatagramPacket reply = new DatagramPacket(
                      request.getData(), request.getLength(),
                      request.getAddress(), request.getPort());
               aSocket.send(reply);
           }
```

Java UDP Server Example

```
        } catch (SocketException e) {
          System.out.println("Socket: " + e.getMessage());
        } catch (IOException e) {
          System.out.println("IO: " + e.getMessage());
        }
    }
}
```

TCP Stream Communication

- Connection-oriented communication
- Need to set up a connection between client and server first.
- After setting up the connection, the two processes could be peers.
- Stream communication procedure:
  - client sends a connect request to server
  - server sends a accept request to client
  - a pair of streams, one in each direction

Stream Communication API

- **Client**: creates a stream socket and binds it to any available port.
- **Client**: makes a connect request to a server at its server port.
- **Server**: creates a listening socket bound to a server port.
- **Server**: accepts a connection
- **Server**: creates a new stream socket with a pair of streams for comm with current client
- **Server**: retains the listening socket for other connect requests

Java TCP Client Example

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
    // arguments supply message and hostname of destination
        try {
          int serverPort = 7896;
          Socket s = new Socket(args[1], serverPort);
          DataInputStream in =
                  new DataInputStream( s.getInputStream());
          DataOutputStream out =
                  new DataOutputStream( s.getOutputStream());
```

Java TCP Client Example

```
            out.writeUTF(args[0]);        // UTF is a string encoding
            String data = in.readUTF();
            System.out.println("Received: "+ data) ;
            s.close();
        } catch (UnknownHostException e) {
            System.out.println("Sock:"+e.getMessage());
        } catch (EOFException e) {
            System.out.println("EOF:"+e.getMessage());
        } catch (IOException e) {
            System.out.println("IO:"+e.getMessage());
        }
    }
}
```

Java TCP Server Example

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
       try {
          int serverPort = 7896;
          ServerSocket listenSocket = new ServerSocket(serverPort);
          while(true) {
             Socket clientSocket = listenSocket.accept();
             Connection c = new Connection(clientSocket); // next slide
          }
    } catch(IOException e) {
       System.out.println("Listen :"+e.getMessage());}
   }
}
```

Java TCP Server Example

```
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
       try {
          clientSocket = aClientSocket;
          in = new DataInputStream( clientSocket.getInputStream());
          out =new DataOutputStream( clientSocket.getOutputStream());
          this.start();
       } catch(IOException e) {
          System.out.println("Connection:"+e.getMessage());}
    }
```

Java TCP Server Example

```
  public void run(){
    try {                                       // an echo server
      String data = in.readUTF();
      out.writeUTF(data);
      clientSocket.close();
    } catch(EOFException e) {
      System.out.println("EOF:"+e.getMessage());
    } catch(IOException e) {
      System.out.println("IO:"+e.getMessage());
    }
  }
}
```

Remote Procedure Call
RPC

RPC

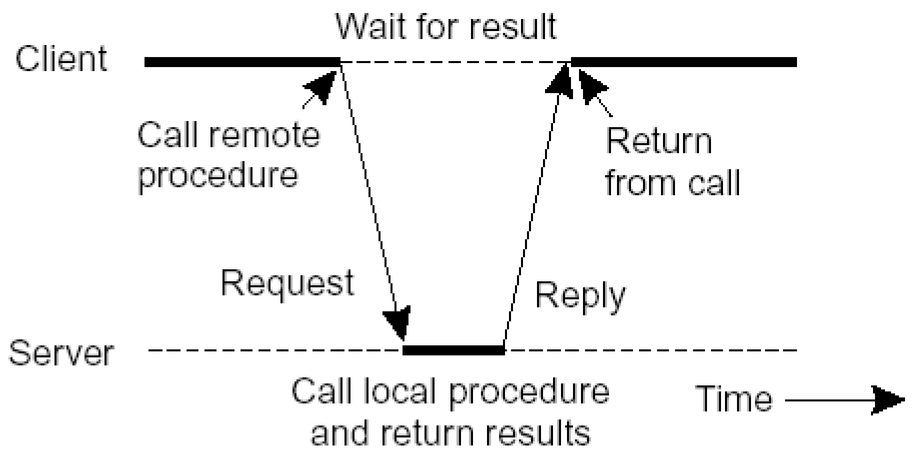
- RPC allows programs to call procedures located on other machines. When a process on machine "A" calls a procedure on machine "B", the calling process on "A" is suspended, and execution of the called procedure takes place on "B". Information can be transported from the caller to the callee, in the parameters, and can come back in the procedure results.
- A Server Process defines its service interface, which specifies the procedures that are available for calling remotely
- No message passing is visible to the programmer.
- This method is known as Remote Procedure Call or RPC.

RPC

- Because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications.
- Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical.
- Both machines can crash, and each of the possible failures causes different problems.
- Most of these can be dealt with, and RPC is a widely used technique that underlies many distributed systems.

Client and Server Interaction

- Principle of RPC between a client and server program.



Client & Server Stubs

- The idea behind RPC is to make a remote procedure call look as much as possible like a local one. (Transparent). Calling procedure should not be aware that the called procedure is executing on a different machine.
- RPC achieves its transparency by using client and server “Stubs”.

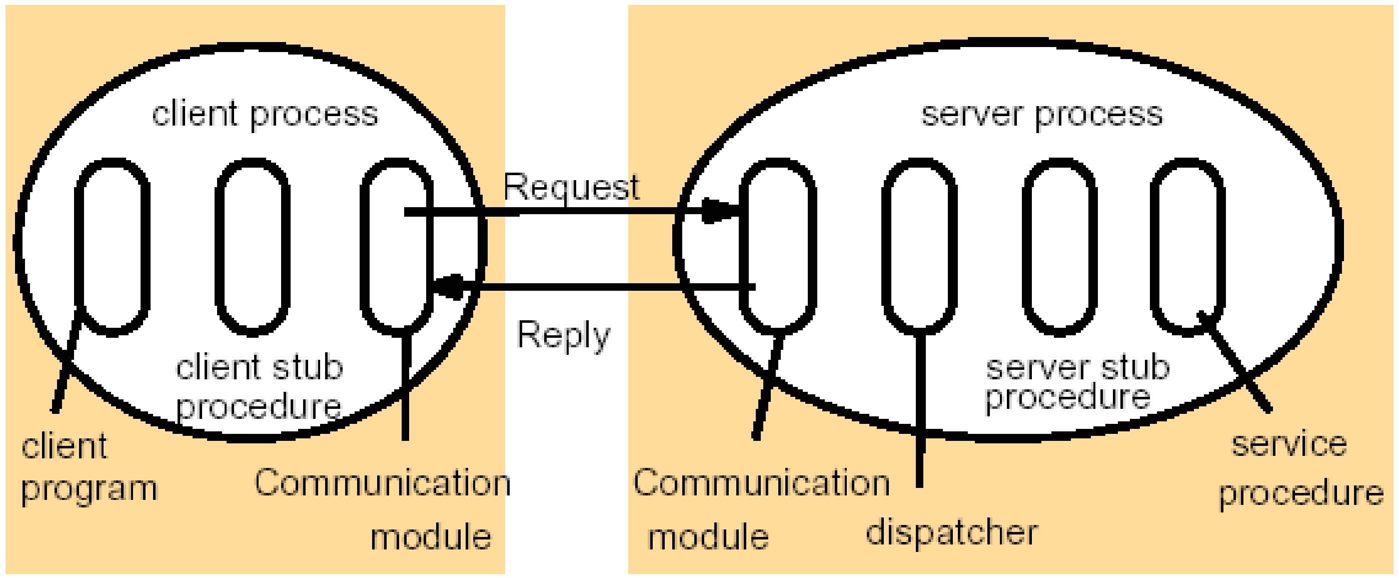
Client & Server Stubs

- When a procedure is actually a remote procedure, a different version of the procedure, on the client stub, is invoked.
- It packs the parameters into a message and requests that message to be sent to the server.
- When the message arrives at the server, the server's operating system passes it up to a server stub.
- A server stub is the server side equivalent of a client stub: it is a piece of code that transforms requests coming in over the network into local procedure calls.

Client & Server Stubs

- The server stub unpacks the parameters from the message and then calls the server procedure in the usual way.
- From server's point of view, it is as though it is being called directly by the client.
- The server procedure performs its work and then returns the results to the server stub.
- Server stub then packs the results in a message and calls a send to return it to the client.
- When the message returns back to the client machine, the clients operating system see that it is addressed to the client process, so it is passed on to it, the client stub inspects the message, unpacks the results, copies it to the caller. (client process)

Stub Procedures in RPC



Stub Procedures in RPC

- The **client stub**
  - similar to the proxy in RMI, behaves like a local procedure to the client
  - marshals the procedure id and arguments into request message
  - sends the request to server
  - wait for the reply message
  - unmarshals the results
- The **dispatcher**
  - receives client request message
  - selects proper server stub

RMI: Remote Method Invocation To be discussed later

Start noticing the similarities with RMI

Stub Procedures in RPC

- The **server stub**
  - acts like a skeleton method in RMI
  - unmarshals the arguments in the request message
  - calls the corresponding service procedure
  - marshals the return values for the reply message

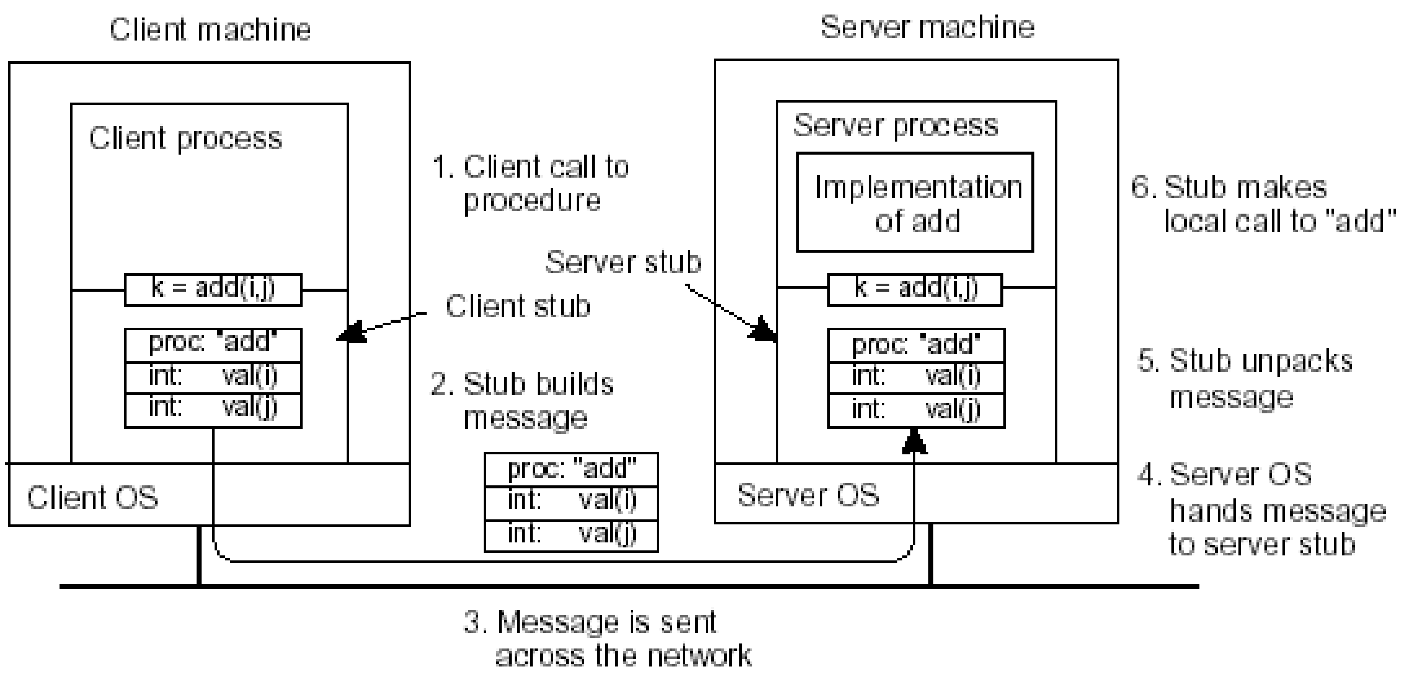
- The client and server stub procedures and the dispatcher can be automatically generated by an interface compiler.

Steps of a Remote Procedure Call

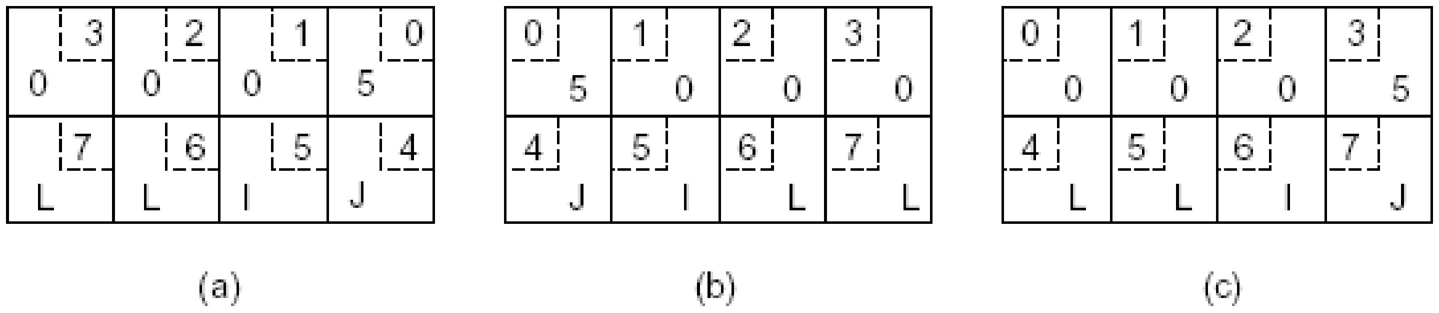
1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it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

Steps of a RPC

- Steps involved in doing remote computation through RPC



Passing Value Parameters



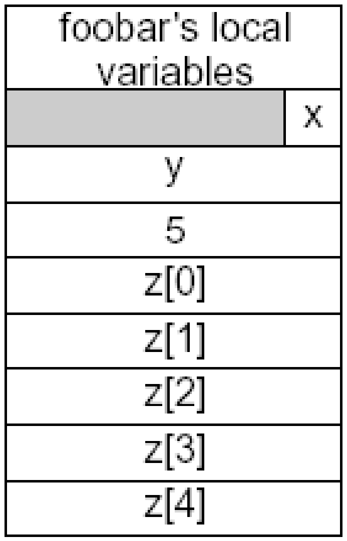
a) Original message on the Pentium (little endian)
b) The message after receipt on the SPARC (big endian)
c) The message after being inverted. The little numbers in boxes indicate the address of each byte

Parameter Spec and Stub Generation

a) A procedure
b) The corresponding message.

```
foobar( char x; float y; int z[5] )
{
   ....
}
```

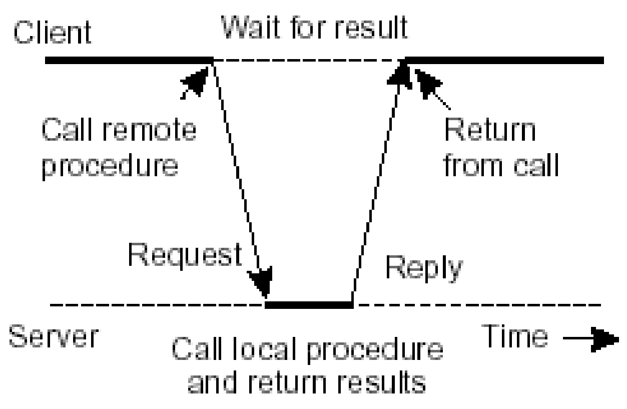
(a)



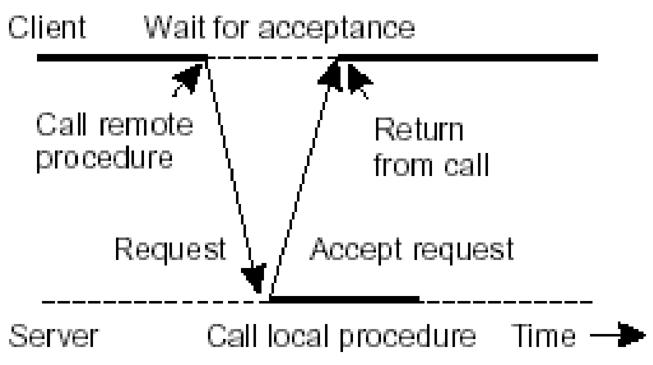
(b)

Asynchronous RPC

a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC



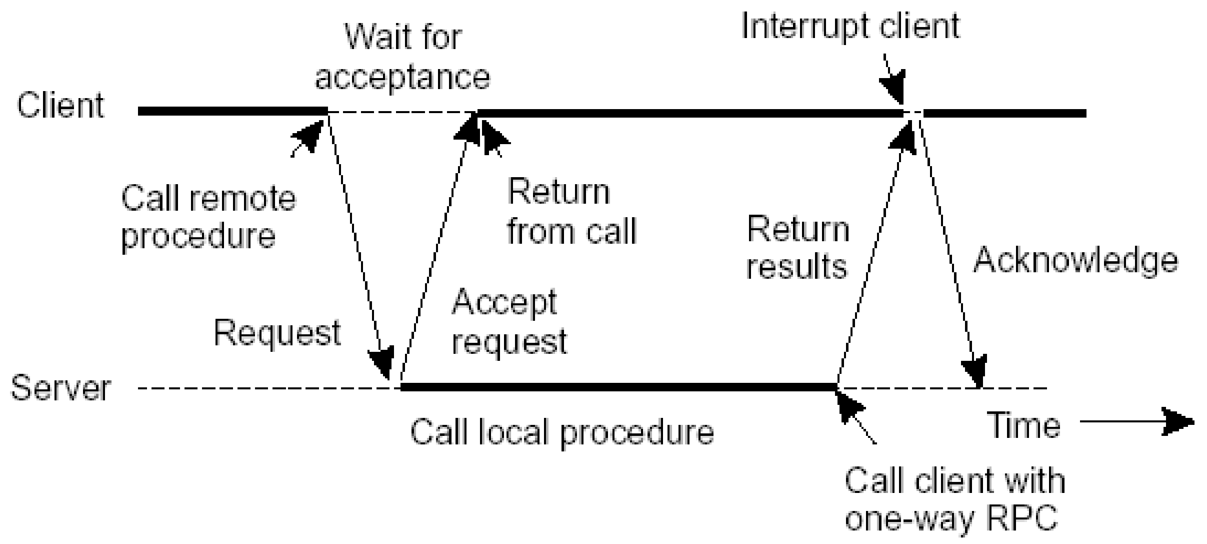
(a)



(b)

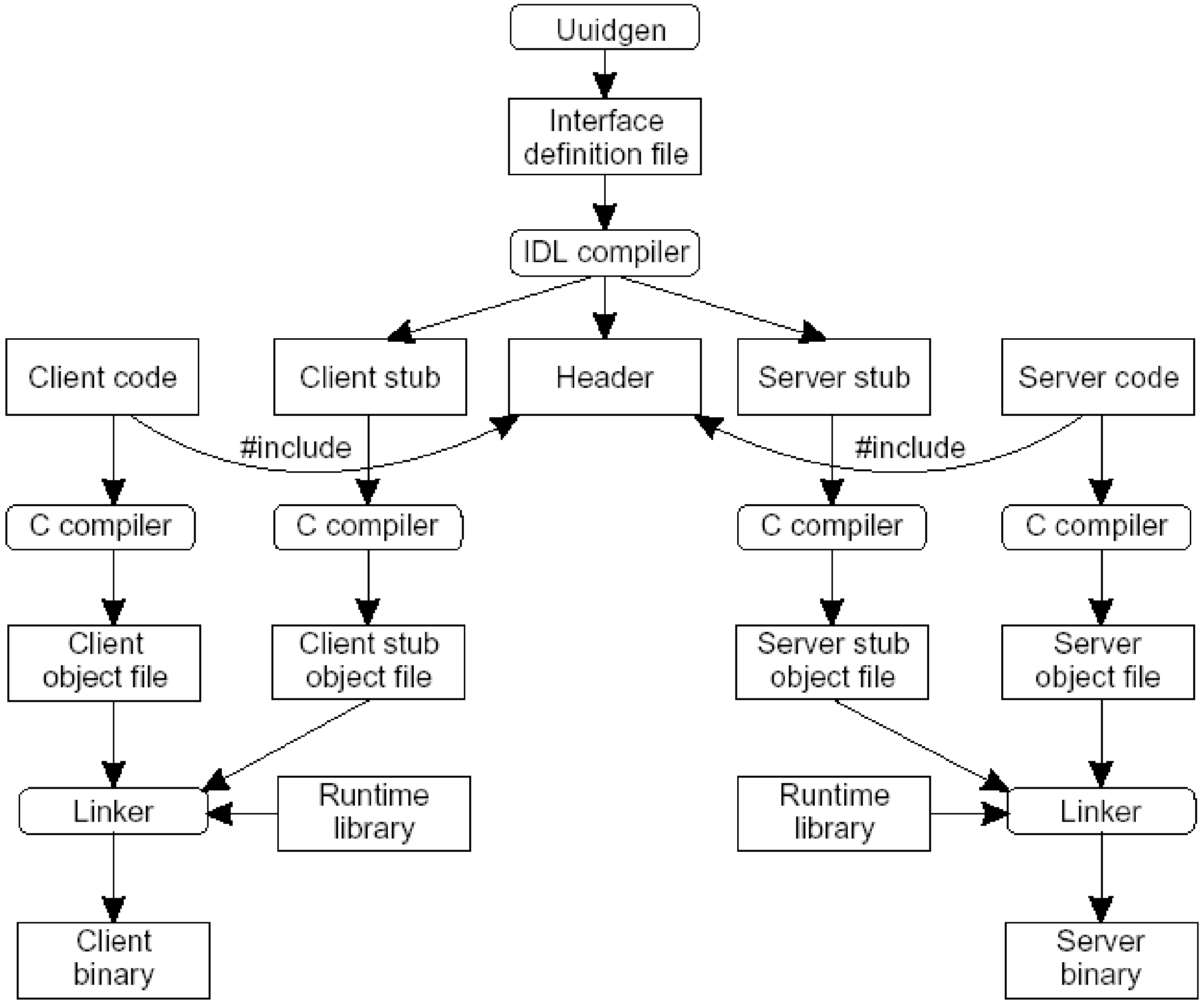
Asynchronous RPC

- A client and server interacting through two asynchronous RPCs



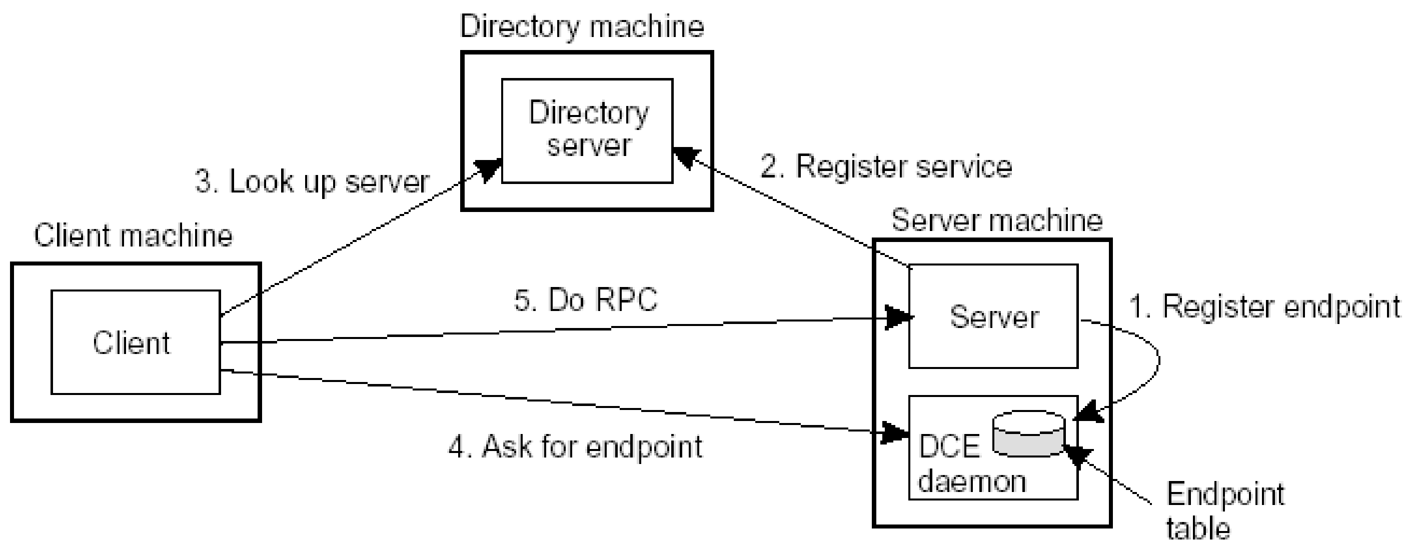
Writing a Client and a Server

- The steps in writing a client and a server in DCE RPC.



Binding a Client to a Server

- Client-to-server binding in DCE.



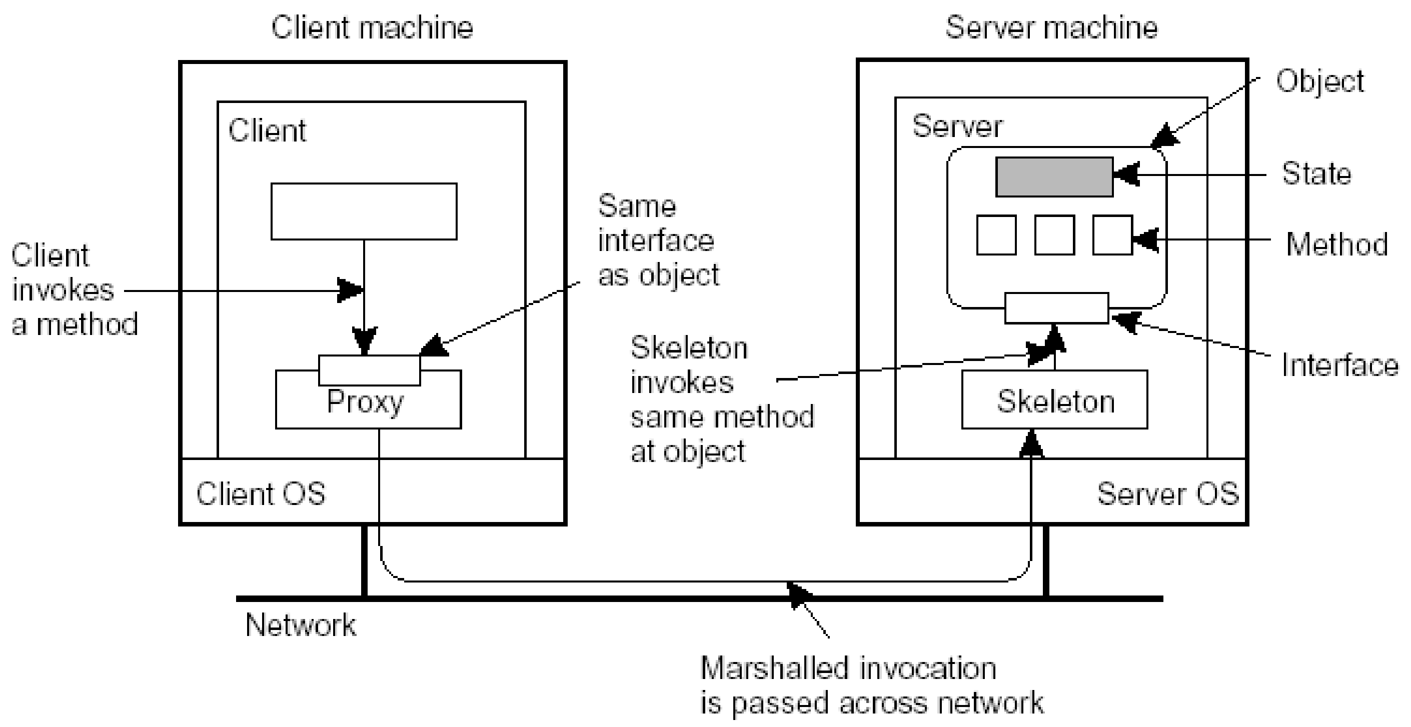
Distributed Object Model

Distributed Object Model

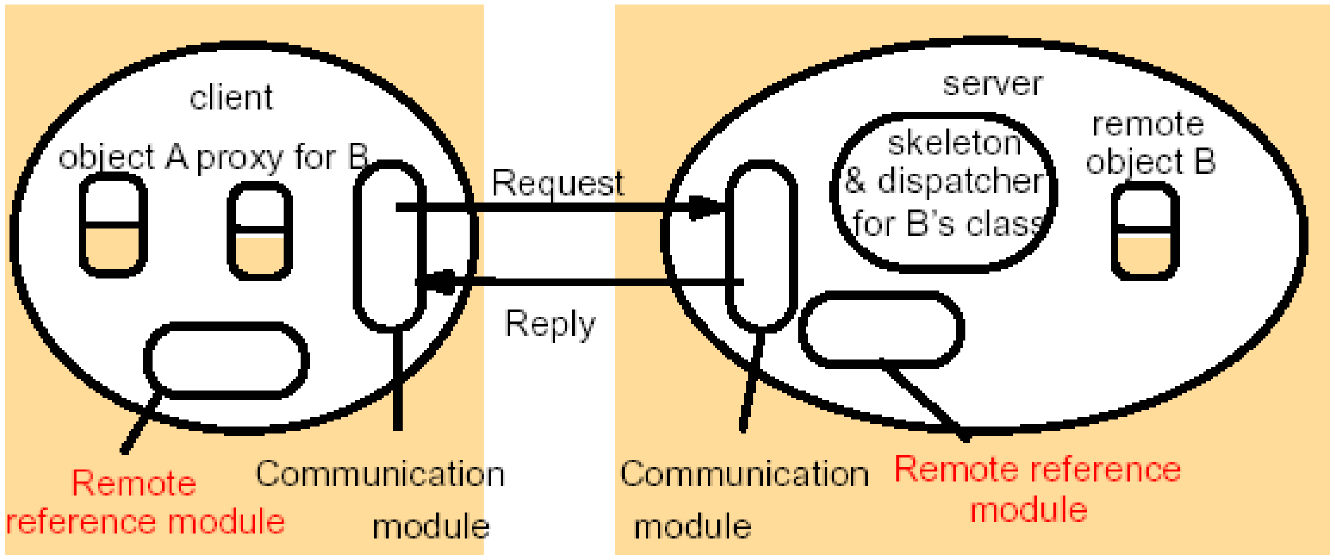
- Remote method invocations
  - method invocations between objects in different processes
- Remote objects
  - objects that can receive remote invocations
- Remote object reference
  - a unique global identifier to refer to an object with methods for remote invocation
- Remote interface
  - specifies which methods in an object can be invoked remotely

Distributed Objects

- A remote object with client-side proxy.



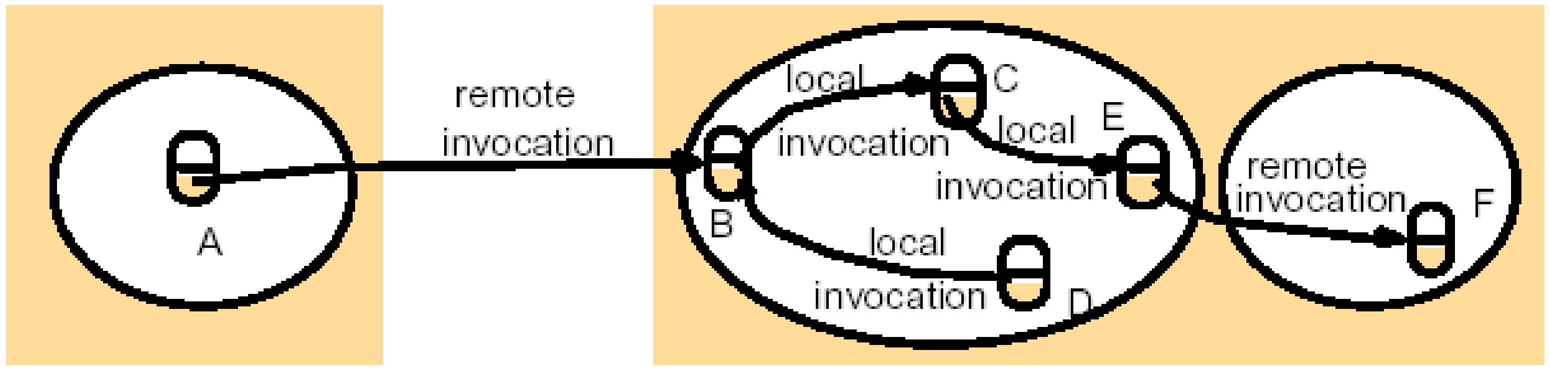
Proxy and Skeleton in RMI



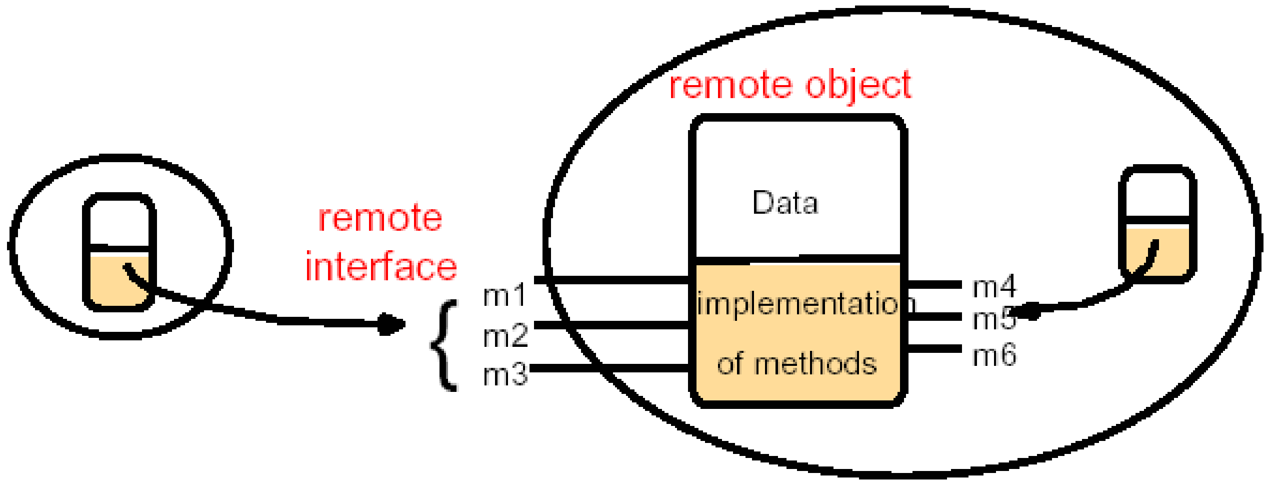
The RMI Software

- Proxy: one proxy for each remote object to make remote method invocation transparent to clients
- Skeleton
  - each remote object class has a skeleton that implements the methods in the remote interface
  - unmarshals the request message, invokes the corresponding remote object, waits for its completion, and marshals the result
- Dispatcher
  - one dispatcher and skeleton for each class representing a remote object
  - receives request message, selects appropriate method, passes on the request to the skeleton

Remote and Local Method Invocations



Remote Object and Interface



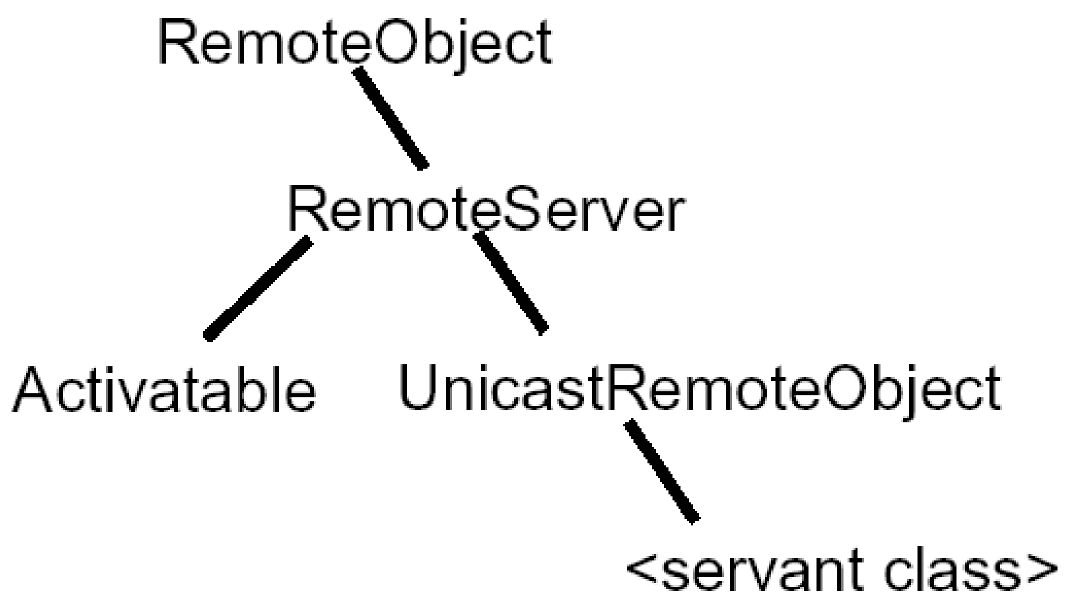
Java Distributed Object Model

- Java has built in support for distributed objects using the Java RMI.
- Strictly speaking, only remote objects are supported (i.e. an object's state always resides on a single machine, but whose interfaces can be made available to remote processes).
- All related classes are in the `java.rmi` package.

Java Remote Method Invocation

- Local and remote objects are almost the same at the language level.
- All serializable data (i.e. can be marshaled) can be passed as a parameter to an RMI.
- Local objects are passed by value whereas remote objects are passed by reference.
- A remote object is built from two different classes.
  - Server class. Contain the object's state and methods and the skeleton generated from interface specifications.
  - Client class. Contain the client code and the proxy also generated from interface specification.

Classes Supporting Java RMI



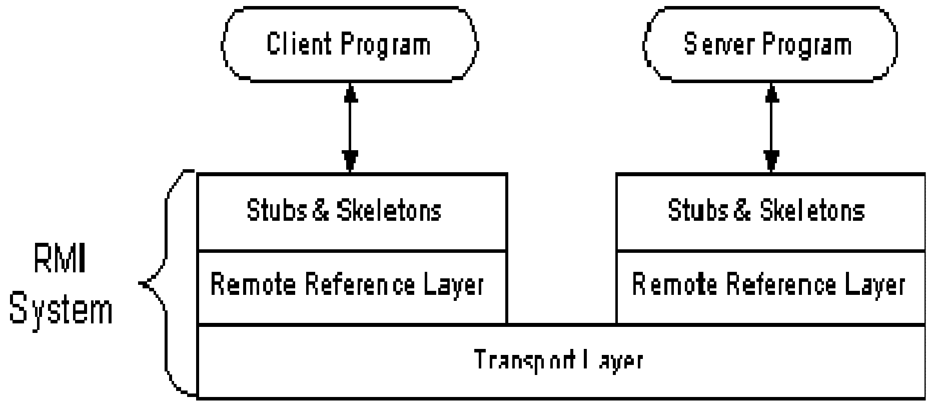
An RMI Example

Developing Remote Object Programs

- RMI Review
- How to Develop Remote Object Programs
- Compiling & Running the Code
- Demonstration
- Conclusion
- Questions & Answers

RMI Review

- Remote Method Invocation
  - Allows an object running on one Java Virtual Machine to invoke methods on an object running on a remote machine.
  - Uses a stub and skeleton layer over a remote reference layer.



How to Develop Remote Object Programs

- Remote object programs are made up of several parts:
  - Remote Interface
  - Implementation Code
  - Server Code
  - Client Code
- This client code could live on the same machine but is usually located on a different machine.

The Remote Interface

- Must extend java.rmi.Remote.
- Must be public.

```
public interface Calculator extends java.rmi.Remote
```

- Specifies methods that will be provided by a remote object.
- All methods must “throw RemoteException.”

Sample Remote Interface

```
public interface Calculator extends java.rmi.Remote {
            public long add(long a, long b) throws java.rmi.RemoteException;
            public long sub(long a, long b) throws java.rmi.RemoteException;
            public long mul(long a, long b) throws java.rmi.RemoteException;
            public long div(long a, long b) throws java.rmi.RemoteException;
}
```

The Implementation Code

- Implements the remote interface.
- Extends UnicastRemoteObject
  - This enables communication between a client and a single object residing on a server.

```
public class CalculatorImpl extends
java.rmi.server.UnicastRemoteObject implements Calculator
```

The Implementation Code (continued)

- Constructor
  - Calls the constructor of the super class.
- Declares methods.

```
public CalculatorImpl() throws java.rmi.RemoteException {
        super();
}
public long add(long a, long b) throws java.rmi.RemoteException{
        return a + b;
}
```

The Host Server

- Sets a Security Manager.
- Informs the RMI Registry that this object exists.

```
public CalculatorServer() {
        System.setSecurityManager(new RMISecurityManager());
        try {
                Calculator c = new CalculatorImpl();
                Naming.rebind("rmi://localhost:1099/CalculatorService", c);
        } catch (Exception e) {
                System.out.println("Trouble: " + e);
        }
}
public static void main(String args[]) {
        new CalculatorServer();
}
```

The Client Code

- Looks the object up in the RMI Registry.
- Must typecast the object.

```
try {
   Calculator c = (Calculator)
               Naming.lookup("rmi://localhost/CalculatorService");
   System.out.println( c.sub(4, 3) );
   System.out.println( c.add(4, 5) );
   System.out.println( c.mul(3, 6) );
   System.out.println( c.div(9, 3) );
}
```

Compiling & Running the Code

- Java Compiler
- RMI Compiler
- RMI Registry
- Code Execution

javac & rmic

- Use the regular java compiler to compile all code (interface, implementation, server, client).

```
javac *.java
```

- Use the RMI compiler (rmic) to generate the stubs and skeletons.

```
rmic CalculatorImpl
```

The RMI Registry

- Start the RMI Registry

```
rmiregistry [port number] &
```

- Default port: 1099

Code Execution

- Start the host server by executing the server code.

```
java CalculatorServer
```

- Run the client.

```
java CalculatorClient
```

Demonstration

- Remote Interface: Calculator.java
- Implementation: CalculatorImpl.java
- Host Server: CalculatorServer.java
- Client: CalculatorClient.java

Conclusion

- Developing remote object programs requires the following:
  - A remote Interface that specifies the methods that can be invoked on a remote object
  - A server running an object that extends java.rmi.Remote
  - Client that needs to invoke a method on a remote object.
  - A naming mechanism or registry.

References

- Sun Microsystems, Java Remote Method Invocation -- http://java.sun.com/products/jdk/rmi/
- Sun Microsystems, Java API Documentation -- http://java.sun.com/j2se/1.3/docs/api/
- Henry, Kevin. “Distributed Computing with Java Remote Method Invocation.” http://www.acm.org/crossroads/xrds6-5/ovp65.html

Message Oriented Communication

Message Oriented Communication

- RPCs and RMIs are not always appropriate, especially when it cannot be assumed that the receiving side is executing at the time a request is issued.
- The synchronous nature of RPCs and RMIs may be a serious drawback.
- We need an alternative way such that processes can exchange information even if the other party is not executing at the time communication is initiated.
- That alternative is messaging.

Message Oriented Communication

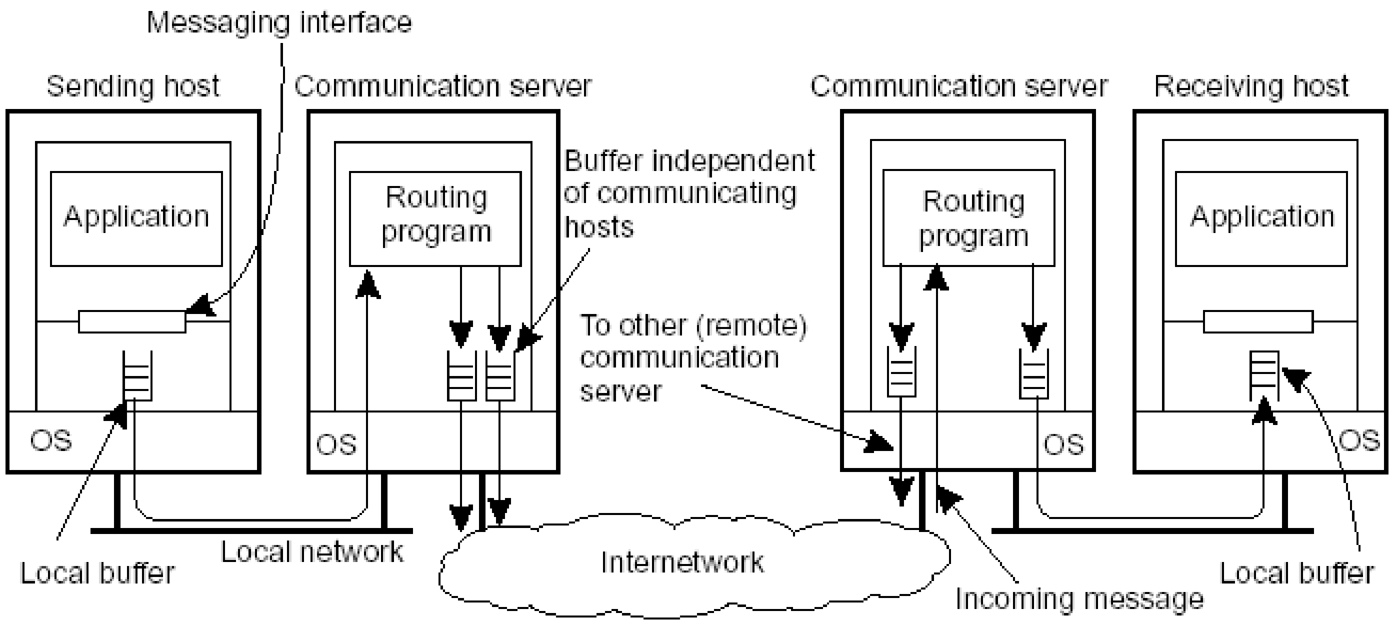
- Applications are always executed on hosts, where each host offers an interface to the communication system through which messages can be submitted for the transmission. The hosts are connected through a network of communication servers, which are responsible for passing (and routing) messages between hosts. Usually each host is connected to exactly one communication server.
- Buffers (for temporary storage) can be put on the hosts or on the servers or on both, depending upon the design, architecture of the system

Message Oriented Communication

- Consider an electronic mail system, a host runs a user agent: an application to compose, send, receive and read messages.
- Each host is connected to one e mail server. The interface at the user’s host allows the user agent to send messages to a specific destination.
- When a user’s agent submits a message at its host, the host generally forwards the message to its local mail server, where it is temporarily stored in an output buffer.
- A mail server removes a message from it output buffer and looks up the destination, this look up will return the transport level address of the mail server to which the message should be sent.
- The mail sets up a connection and passes the message to the target mail server.
- The receiving server stores the message in an output buffer for the designated receiver, also called the receiver’s mail box.
- If the target server is (temporarily) unreachable, the local mail server will continue to store the message.
- The interface at the receiving host offers a service to the receiver’s user agent by the it can regularly check for incoming mail. An receives the message, (copies the message to a local buffer on the host).
- Communication is persistent !!

Messaging Architecture

- General organization of a communication system in which hosts are connected through a network



Message Queuing Systems

- The middleware services to provide message oriented communication, also known as Message-Oriented Middleware (MOM).
- Provide intermediate-term storage capacity for messages.
- Provide extensive support for persistent asynchronous communication.
- Target message transfers that take minutes instead of seconds or milliseconds.

An Example of Message Orientated Persistent Communication

Message Queuing Model

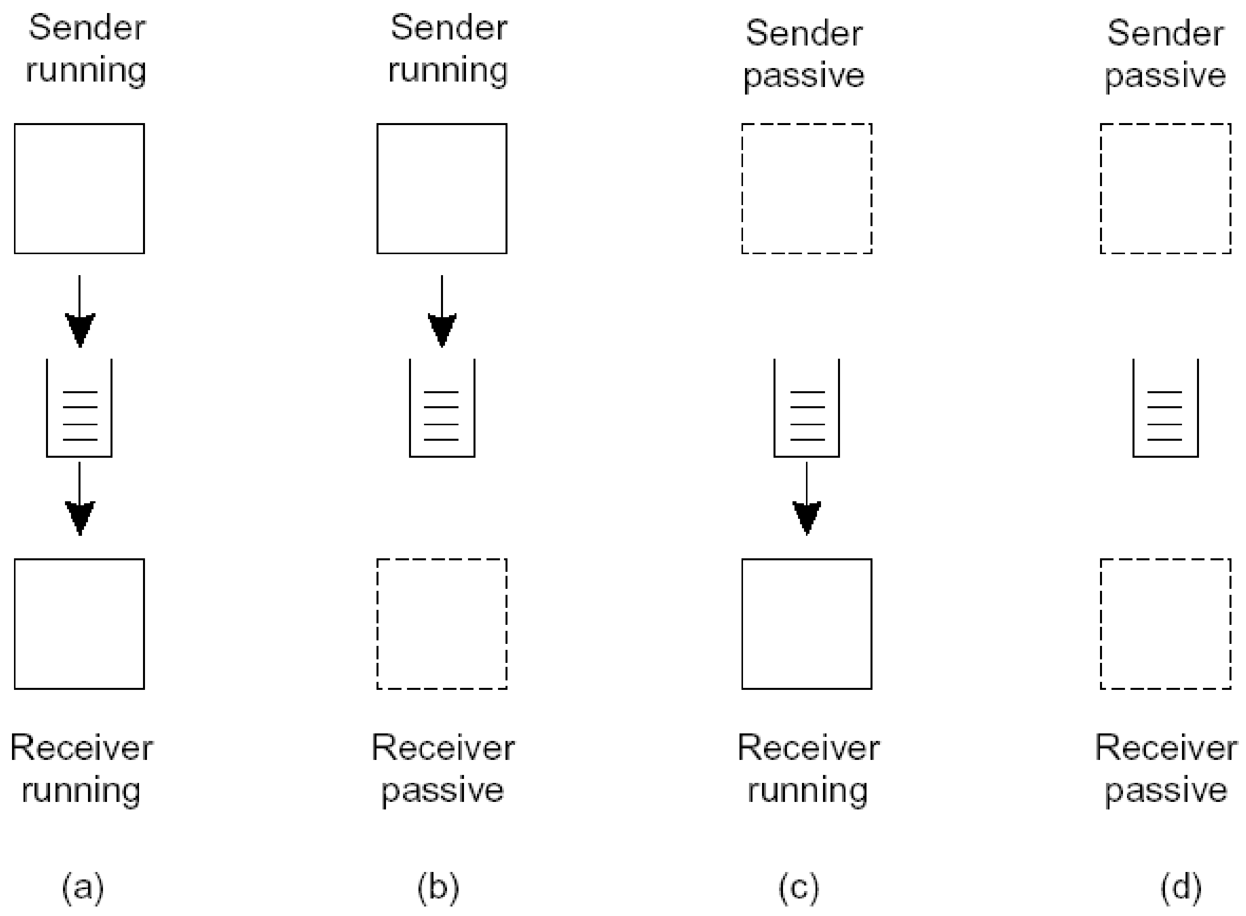
- Applications communicate by inserting messages in specific queues. These messages are forwarded over a series of communication servers and are eventually delivered to the destination, even if it was down when the message was sent.
- A message is generally transferred directly to a destination server.
- Each application has its own private queue, to which other applications can send messages.
- A queue can be read only by its associated application, but it is also possible for multiple applications to share a single queue.
- Sender is generally given only the guarantees that its message will eventually be inserted in the recipient’s queue, No guarantees are given about when!!!

Message Queuing Model

- These semantics permit loosely-coupled communication. The sender and receiver can execute completely independently of each other. Once a message has been deposited in a queue, it will remain there until it is removed, irrespective of whether its sender or receiver is executing. This give us the four combinations …see the figure

Message Queuing Model

- Four combinations for loosely-coupled communications using queues.



Message Queuing Model

- Message can contain any data. Addressing is done by providing a systemwide unique name of the destination queue.
- The "put" primitive is called by a sender to pass a message to the underlying system that is to be appended to the specified queue. This is a non blocking call.
- The "get" primitive is a blocking call by which an authorized process can remove the longest pending message in the speicified queue. The process is blocked only if the queue is empty. Variation of this call can allow to search for specific message in the queue. The non blocking variant is given by the "poll" primitive, if the queue is non empty, or if a specific message could not be found, the calling process simply continues.

Message Queuing Model

- Finally most queuing systems allow a process to install a handler as a “callback function”, which is automatically invoked whenever a message is put into the queue. Callbacks can also be used to automatically start a process, that will fetch messages from the queue if no process is currently executing. This approach is often implemented by means of a daemon on the receiver’s side, that continuously monitors the queue, for incoming messages and handles accordingly.

Message Queuing Model

- Basic interface to a queue in a message-queuing system.

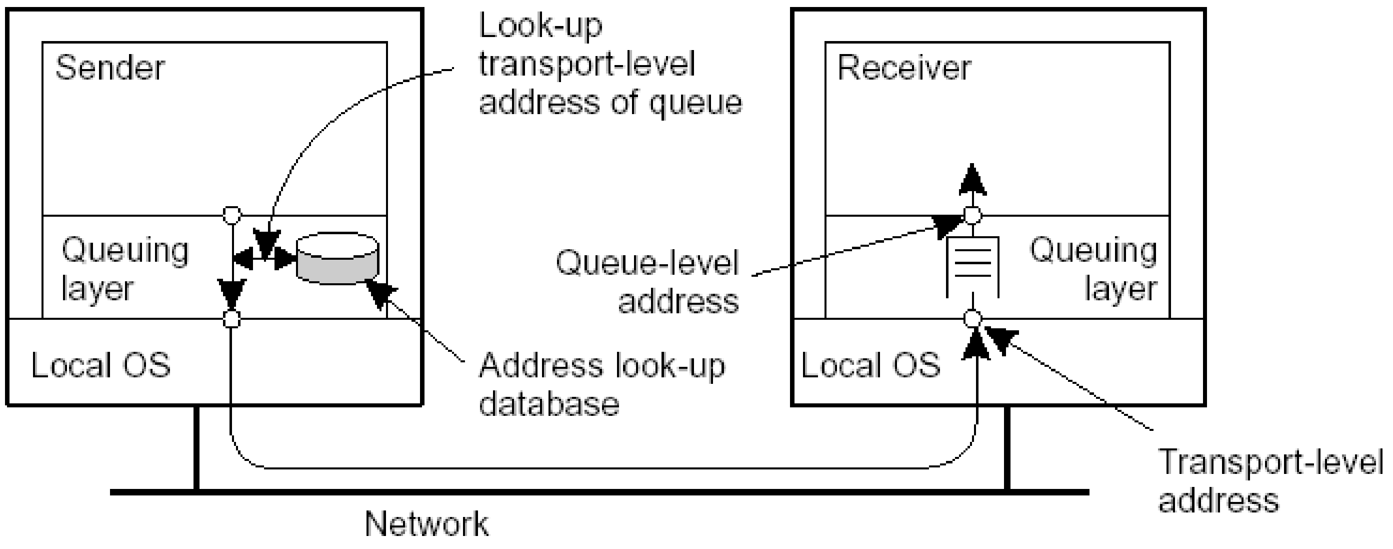
Primitive	**Meaning**
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block.
Notify	Install a handler to be called when a message is put into the specified queue.

Message Queuing System Architecture

- Queue are distributed across multiple machines. So a message queuing system should maintain a mapping of queues to network locations. In other words, it should maintain a (possibly distributed) database of queue names to the network locations (see fig next slide).
- Such mapping is similar to the DNS for e mail in the Internet. For example, when sending mail to the logical mail address abc@yahoo.com , the mailing system will query DNS to find the network address (i.e., the IP address) of the recipient's mails server to use for the actual message transfer.

Message Queuing System Architecture

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

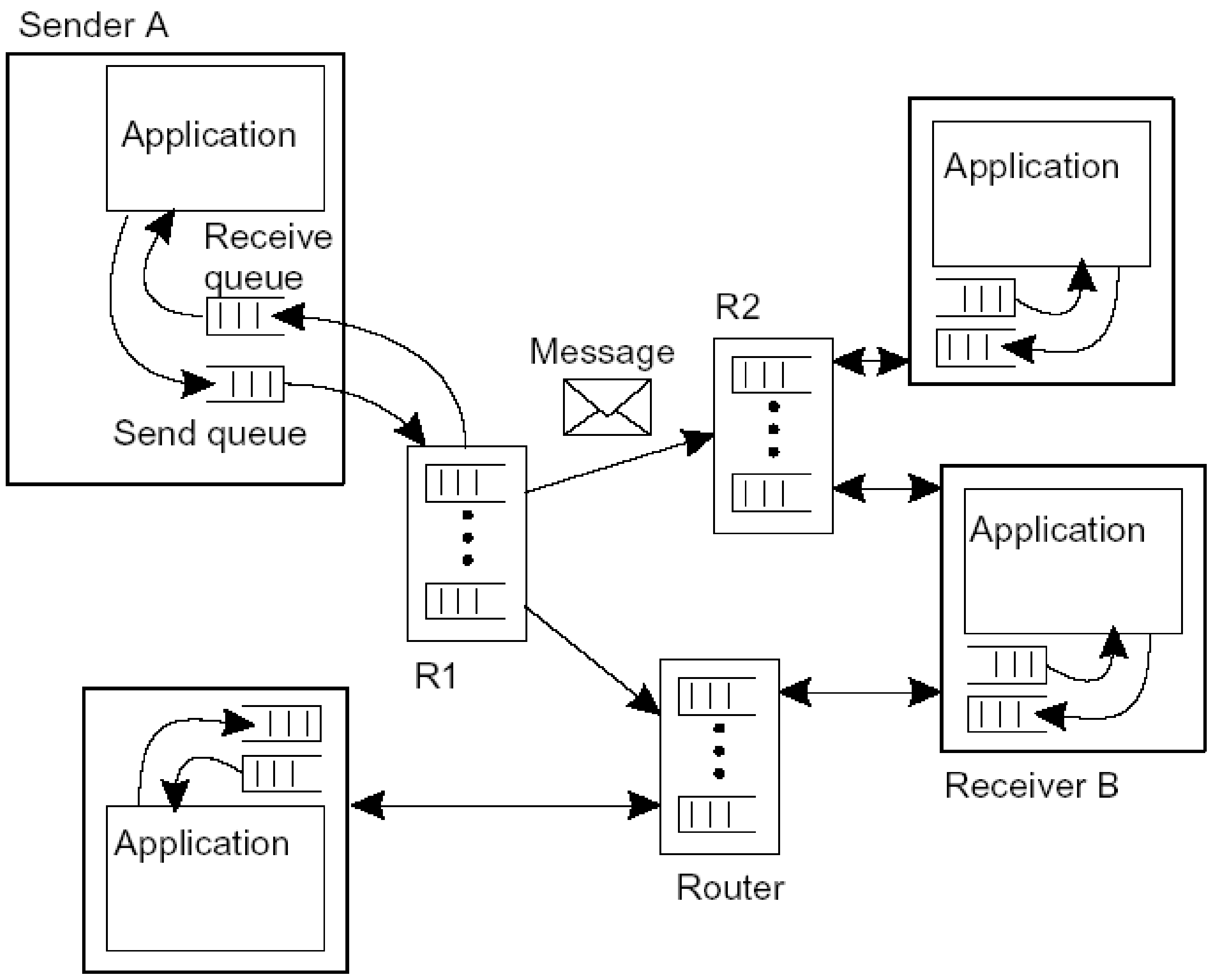


Message Queuing System Architecture

- Queues are managed by the queue managers, which directly interacts with the application that is sending or receiving a message.
- There are special queue managers that operate as routers, or relays, they forward incoming messages to other queues managers.
- In this way, a message queuing system may gradually grow into a complete, application level, overlay network on the top of an existing computer network.

Message Queuing System Architecture

- The general organization of a message-queuing system with routers.



Message Brokers

- An important application area of message queuing system is integrating existing and new applications into single coherent distributed system. Integration requires that applications can understand the messages, they receive. This requires the sender to have its outgoing messages in the same format as that of the receiver.
- The problem with this approach is, each time a new application is added to the system that requires a separate message format, each potential receiver will have to be adjusted.
- So existing system have to learn to live with the different formats, and try to provide the means to make conversions as simple as possible.
- In message queuing systems, conversions are handled by special nodes, known as message brokers.

Message Brokers

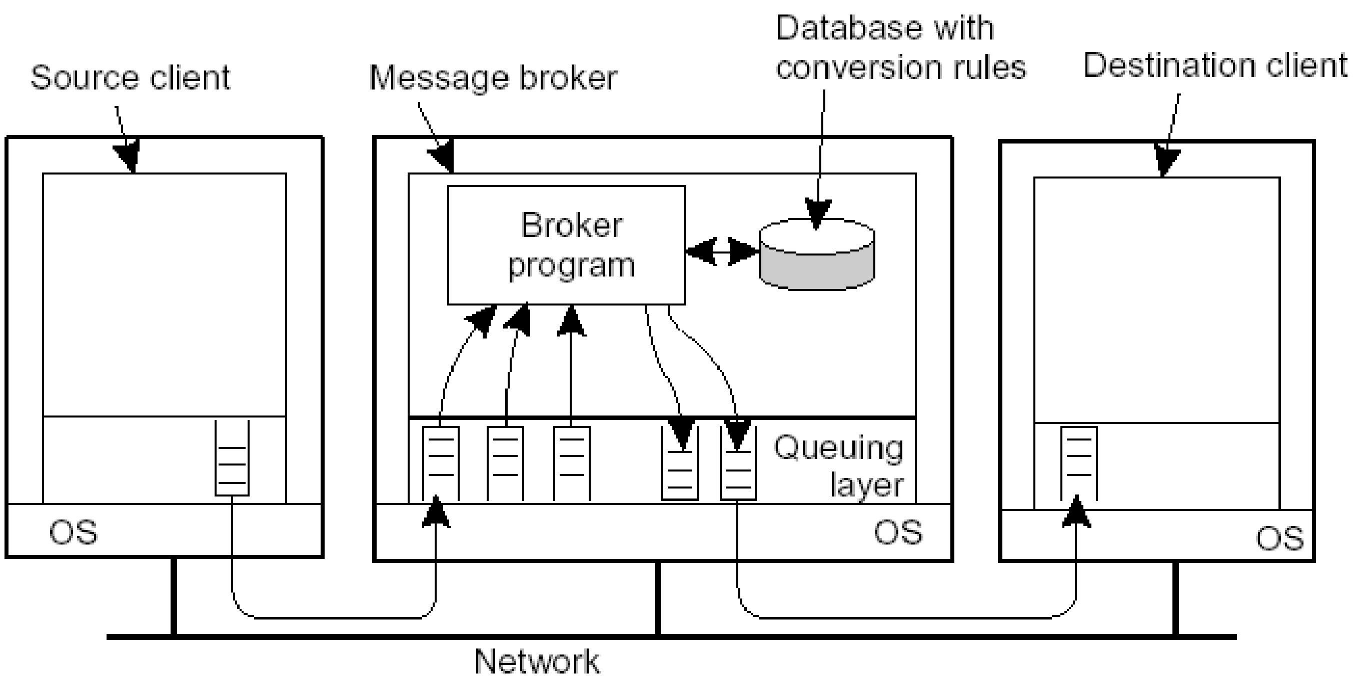
- A message broker acts as an application level gateway, its main purpose is to convert incoming messages to a format that can be understood by the destination application.
- A message broker is an added application/feature, it is not considered an integral part of the message queuing system.
- A message broker can be as simple as reformatter for messages. (Changing the break sequences of database queries, escape characters formats etc etc), or it can be more advanced, such as one that handles the conversions between X.400 and Internet e mail messages. In such cases, sometimes, it cannot be guarenteed that all the information in the incoming messages can actually be transformed into something appropriate for the outgoing messages. In other words, it may be necessary to accept a certain loss of information during the transformation.

Message Brokers

- At the heart of a message broker lies a database of rules that specify how a message in format “x” can be converted to a message in format “y”.
- Many message broker products come with sophisticated rule development tools, but the bottom line is still that rules are to be manually entered into the database.
- Rules can be formulated in a special conversion language or it can be done using normal programming languages.
- Setting up a message broker is thus generally a highly laborious task.

Message Brokers

- The general organization of a message broker in a message-queuing system.



Message Queuing Systems

- General Message Queuing Systems are not aimed at supporting only end users (E- mail). An important point is that they are setup to enable persistent communication between processes, regardless of whether a process is running a user application, handling database access, performing computations and so on. This approach leads to different set of requirements for message queuing systems that pure e mail systems, such as message priorities, logging facilities, efficient multicasting, load balancing, fault tolerance and so on.

For Reference

- Have a look at Java Messaging API
- IBM MQSeries (Tanenbaum Book)

- Have a look at Java RMI / and Java Socket tutorial on www.javaworld.com
- http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html
- http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html

How do we find the entities / services in a distributed system?

Naming, Directory and Discovery

- How entities are identified in a distributed environments?
- Naming: Identification
- Directory: Finding entities
- Discovery: Discovering the entities, they might be available, might not be available,
    - Domain Name system,
    - Java Naming and Directory Interface (JNDI)
    - UDDI, Registry Services

Synchronization (I)

Distributed Systems

BSc IV Computer Science

Dr. M- Waseem Akhtar

Time & Clock Synchronization

Introduction

- Time is an important practical issue. For example we require computers around the world to timestamp electronic commerce transactions consistently.
- Time is also an important theoretical construct in understanding how distributed executions unfold.
- But time is problematic in distributed systems. Each computer may have its own physical clock, but the clocks typically deviate, and we cannot synchronize them perfectly.

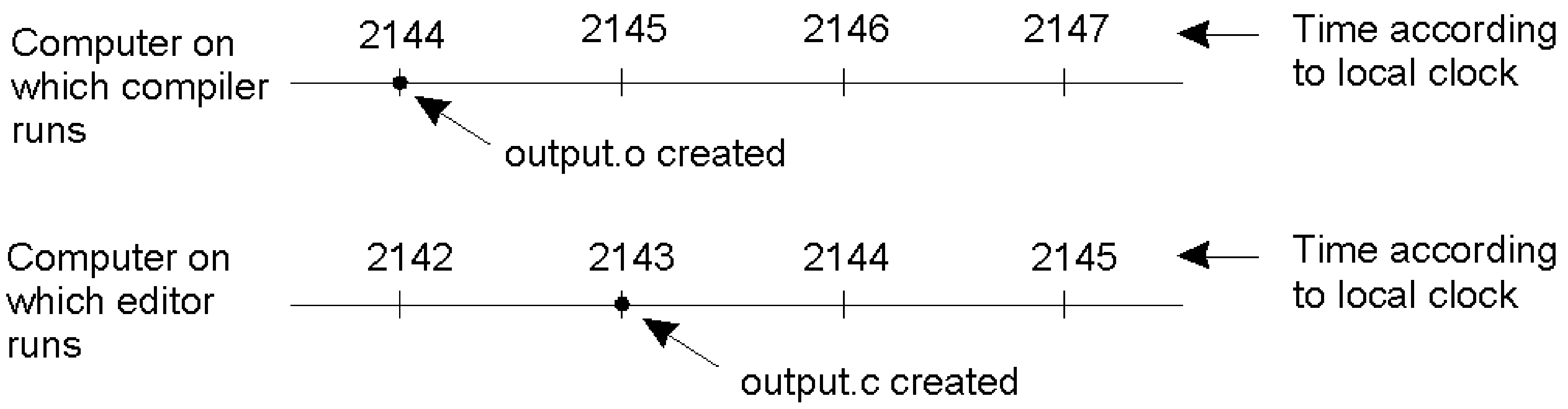
Introduction

- We shall examine algorithms for **synchronizing physical clocks** approximately, and then go on to explain **logical clocks**, including **vector clocks**, which are a tool for ordering events without knowing precisely when they occurred.
- The absence of global physical time makes it difficult to find out the state of our distributed programs as they execute.
- we often need to know that state process A is in when process B is in certain state, but we cannot rely on physical clocks to know what is true at the same time.
- We examine (Next Lecture) algorithms to determine **global states** of distributed computations despite the lack of global time.

The issue of Time in distributed systems

- A quantity that we often have to measure <u>accurately</u>
  - necessary to synchronize a node’s clock with an authoritative external source of time
    - Eg: timestamps for electronic transactions
      - both at merchant’s & bank’s computers
      - auditing
- An important theoretical construct in understanding how distributed executions unfold
  - Algorithms for several problems depend upon <u>clock synchronization</u>
    - timestamp-based serialization of transactions for consistent updates of distributed data
    - Kerberos authentication protocol
    - elimination of duplicate updates

Clock Synchronization


Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created

- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Fundamental limits

The notion of physical time is problematic in distributed systems:
- limitations in our ability to timestamp events at different nodes sufficiently accurately to know the order in which any pair of events occurred, or whether they occurred simultaneously.

Distributed System Model

- We take a distributed system to consist of a collection of N processes p_i, i=1,2,3.. N.
- Each process executes on a single processor, and processes do not share memory.
- Each process p_i, has a state s_i, which in general, it transforms as it executes.
- The process state includes the values of all the variables, values of any objects.
- Processes communicate with each other by sending messages through the network.
- As each process executes, it takes a series of actions, each of which is either a Message Send, a Message Receive or an operation that transforms p_i, state- (one or mare values change in s_i,
- We define an event to be the occurrence of a single action that a process carries out as it executes, a communication action or a state-transforming action.

History of Process p_i

- The sequence of events within a single process can be placed in a single total ordering, which we shall denote by the relation $\rightarrow$ between the events
- $e \rightarrow_i e'$
  - total ordering of events at process
    - Assuming that process executes on a single processor
- $\text{history}(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$
  - series of events that take place within p_i

Clocks

- In a distributed system computers contain their own physical clock. These clocks are electronic devices that count oscillations occurring in a crystal at a definite frequency, and that typically divide this count and store the result in a counter register. Clock devices can be programmed to generate interrupts at regular intervals in order that, for example, time slicing can be implemented.
- Operating system reads the nodes hardware clock $H_i(t)$
- Scales it and add an offset to produce a software clock $C_i(t)$ that approximately measures real, physical time t for the process p.
- $H_i(t)$: hardware clock value (by oscillator)
- $C_i(t)$: software clock value (generated by OS)
  - $C_i(t) = a\ H_i(t) + \beta$
    - Eg: # nsec's elapsed at time t since a reference time
  - clock resolution: period between updates of $C_i(t)$
    - limit on determining order of events

Clock Skew & Drift

- **Skew**: instantaneous difference between readings of any clocks is called their skew
- **Drift**: different rates of counting time
  - physical variations of underlying oscillators
  - variance with temperature
  - Even extremely small differences accumulate over a large number of oscillations
    - leading to observable difference in the counters
  - **drift rate**: difference in reading between a clock and a nominal “perfect clock” per unit of time measured by the reference clock
    - 10^{-6} seconds/sec for quartz crystals
    - 10^{-7} - 10^{-8} seconds/sec for high precision quartz crystals

International Atomic Time & Astronomical time.

- Atomic oscillators:
  - The most accurate physical clocks use atomic oscillators
  - drift rate ~ 10^{-13} seconds/second
  - International Atomic Time (since 1967)
    - 1 standard sec = 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of Caesium-133 (Cs^{133})
- **Astronomical Time**: Seconds and years and other time units are rooted in the astronomical time. They were originally defined in terms of the rotation of the earth on its axis and its rotation about the Sun. However the period of earth's rotation abut its axis is gradually getting longer. Primarily because of tidal friction; atmospheric effects and convection currents within the earth's core also cause short term increases and decreases in the period. So astronomical time and atomic time have a tendency to get out of set.

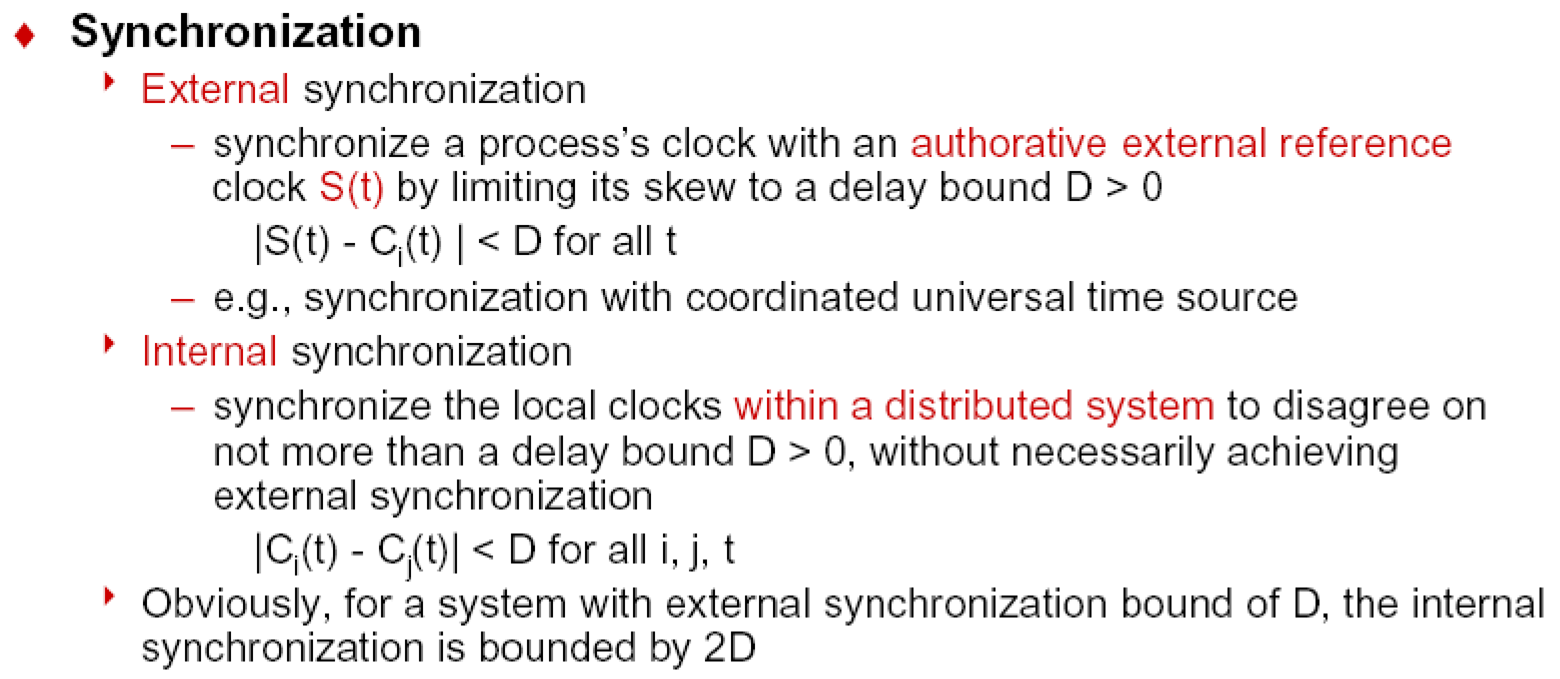
UTC: Coordinated Universal Time

- UTC is an international standard for timekeeping. It is based on the atomic time, but so called leap second is inserted or more rarely deleted.
- UTC: 1 leap sec is occasionally inserted, or more rarely deleted, to keep in step with Astronomical Time
  - time signals broadcasted from land-based radio stations (WWV) and satellites (GPS)
    - accuracy: 0.1-10 millisec (land-based), 1 microsec (GPS)

Synchronisation of Clocks

Two types of clock synchronisation:

1. External Synchronisation
2. Internal Synchronisation



- **Synchronization**
  - External synchronization
    - synchronize a process's clock with an authorative external reference clock S(t) by limiting its skew to a delay bound D > 0

      $|S(t) - C_i(t)| < D$ for all t
    - e.g., synchronization with coordinated universal time source
  - Internal synchronization
    - synchronize the local clocks within a distributed system to disagree on not more than a delay bound D > 0, without necessarily achieving external synchronization

      $|C_i(t) - C_j(t)| < D$ for all i, j, t
  - Obviously, for a system with external synchronization bound of D, the internal synchronization is bounded by 2D

Synchronisation of Clocks

- D: synchronization bound
- S: source of UTC time,
- External synchronization:
  - $|S(t) - C_i(t)| < D$
  - Clocks are accurate within the bound D
- Internal synchronization:
  - $|C_i(t) - C_j(t)| < D$
  - Clocks agree within the bound D
- (From)external sync $\underset{\not\Leftarrow}{\Rightarrow}$ (to) internal sync

Correctness of clocks

- Hardware correctness:
  - There can be no jumps in the value of H/W clocks
- Monotonicity:
  - $t' > t \Rightarrow C(t') > C(t)$
  - A clock only ever advances
  - Even if a clock is running fast, we only need to change at which updates are made to the time given to apps
    - can be achieved in software: $C_i(t) = a\, H_i(t) + b$
- Hybrid:
  - monotonicity + drift rate bounded bet. sync. points (where clock value can jump ahead)

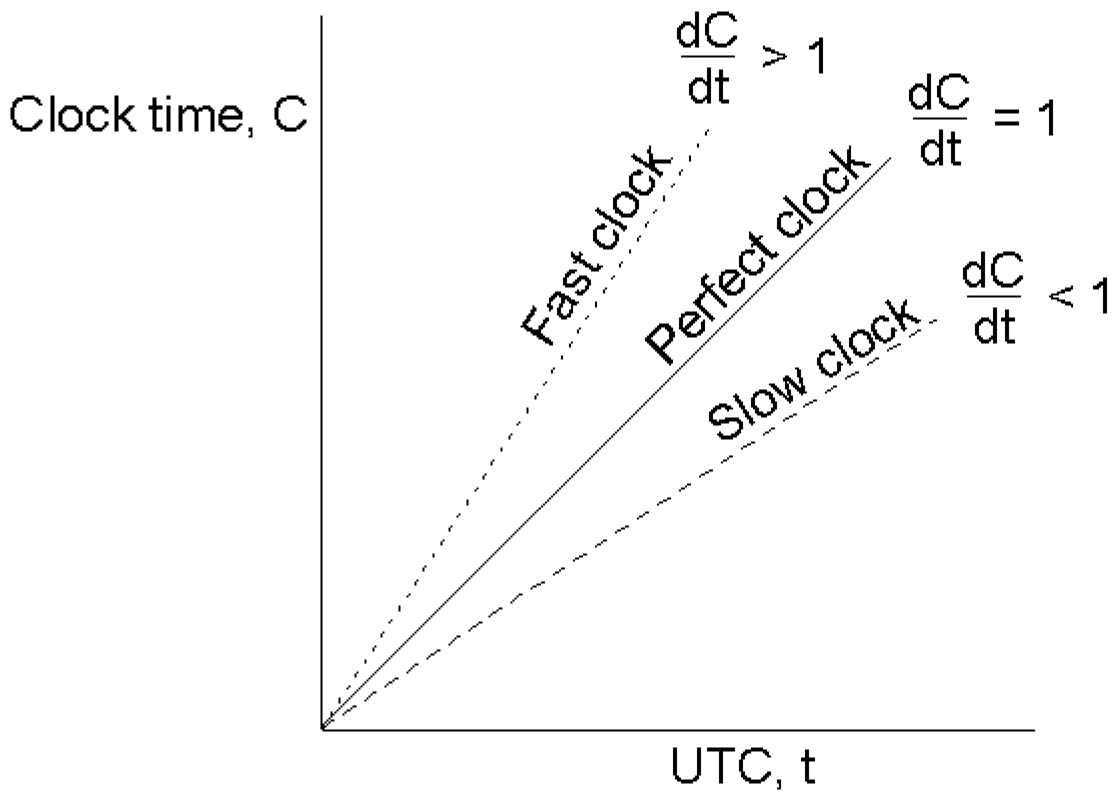
Synchronous systems

- P1 sends its local clock value t to P2
  - P2 can set its clock value to $(t + T_{transmit})$
  - $T_{transmit}$ can be variable or unknown
    - resource competition bet. processes
    - network congestion
- u = (max - min)
  - uncertainty in $T_{transmit}$
    - obtained if P2 sets its clock to (t + min) or (t + max)
    - In both above cases skew can be as large as u
  - If P2 sets its clock value to t + (max+min)/2, then skew <= u/2

In asynchronous systems:
$T_{transmit} = \min + x$, where $x \geq 0$
Only the distribution of x may be measurable, for a given installation

Clock Synchronization Algorithms

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



Clock Synchronization Algorithms

- Christian’s Algorithm
- Berkeley’s Algorithm
- Averaging Algorithm

Time servers: Christian’s algorithm

- Christian suggested the use of a time server, connected to a device that receives signal from a UTC source to synchronize computers externally.
- Upon request server process S supplies time according to its clock.
- Christian observed that while there is no upper bound on message transmission delays in an asynchronous system, the round trip times for messages exchanged between pairs of processes are often reasonable short, a small fraction of a second.
- He describes the algorithm as probabilistic: the method achieves synchronization only if the observed round trip times between client and server are sufficiently short as compared with the required accuracy.

Time servers: Christian’s algorithm

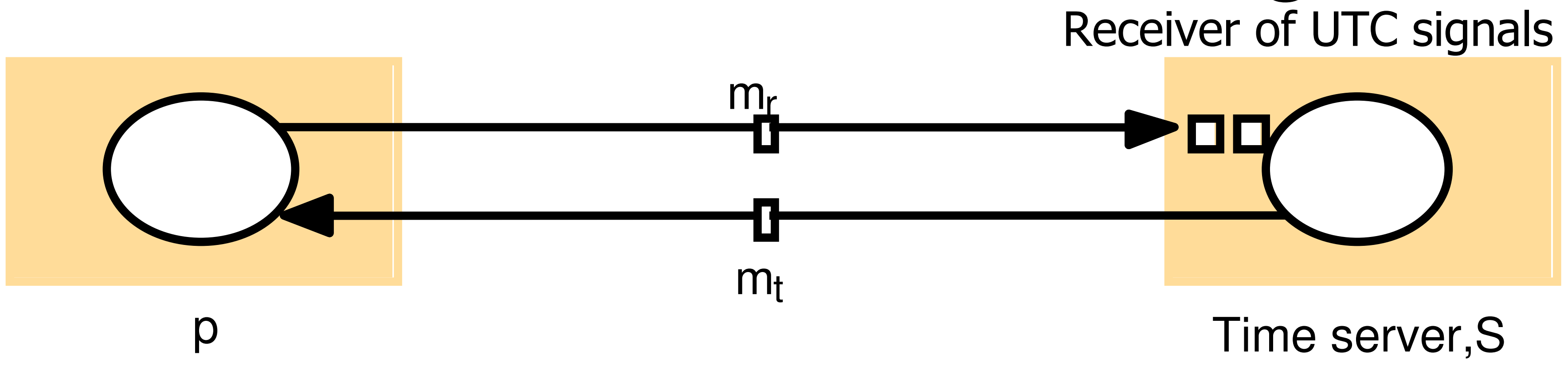
- A process p requests the time in a message m_r and receives the time value t in a message m_t. Time is inserted in the message at the last possible point before transmission from the Server’s computer.
- Process p records the total round trip time T_round taken to send the request and receive the reply.
- It can measure this time with reasonable accuracy if its rate of clock drift is small.
- A simple estimate of the time to which p should set its clock is t+T_round /2, which assumes that elapsed time is split equally before and after S placed time in the message.
- If the value of minimum transmission time is known or can be conservatively estimated, then we can determine the accuracy of this result as follows.

Time servers: Christian's algorithm

- The earliest point at which server could placed the time in reply message, was <u>*min*</u> after p dispatched request.
- The latest point at which it could have done this was <u>*min*</u> before reply arrived at requesting process p.
- The time by Server's clock when the reply message arrives is therefore in the rang [t+min, t+**Tround**-min]
- The width of this range is Tround – 2min, so the accuracy is

$$\boxed{\text{Accuracy:} \quad \pm \quad (\mathbf{T_{round}/2 - min})}$$

Time servers: Christian's algorithm



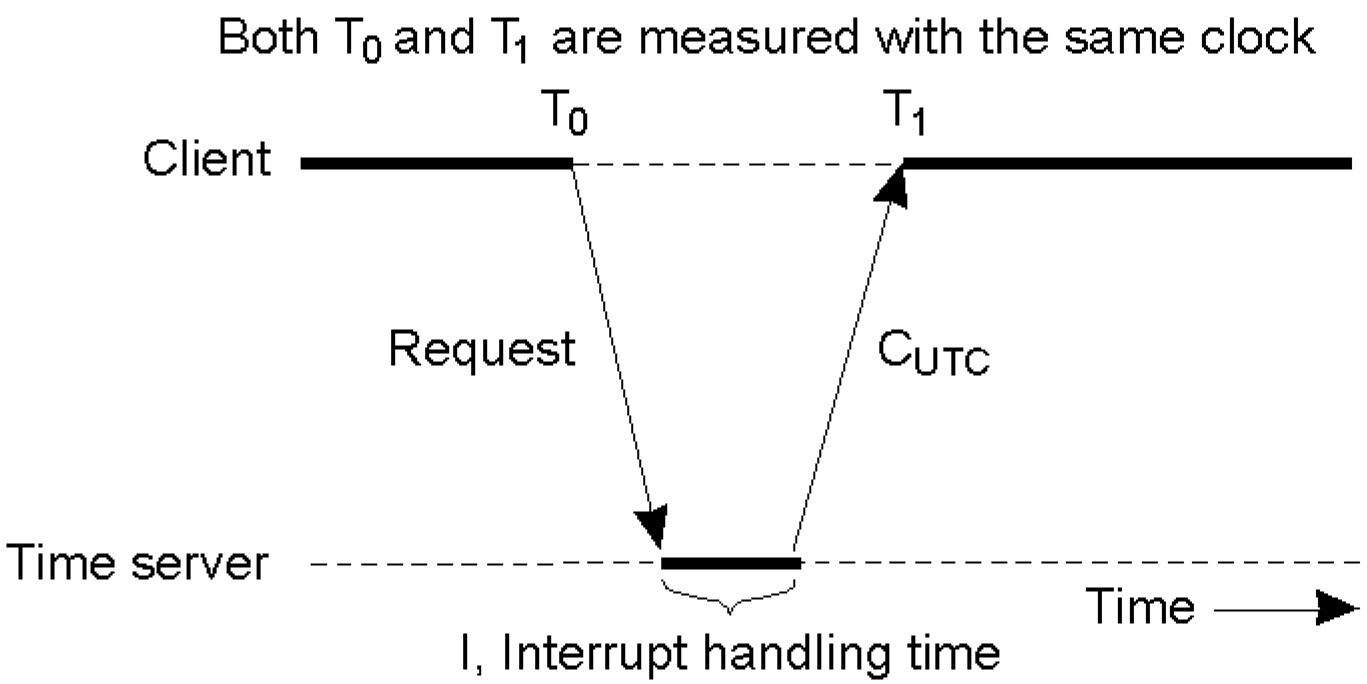
T_{round} := total round-trip time
t := time value in message m_t
estimate := (t + T_{round} /2)

Time by S's clock when reply msg arrives $\in$ [t+min, t+$\mathbf{T_{round}}$-min]

Accuracy: $\pm$ (**T_{round}/2 - min**)

Cristian's Algorithm

- Getting the current time from a time server.



Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

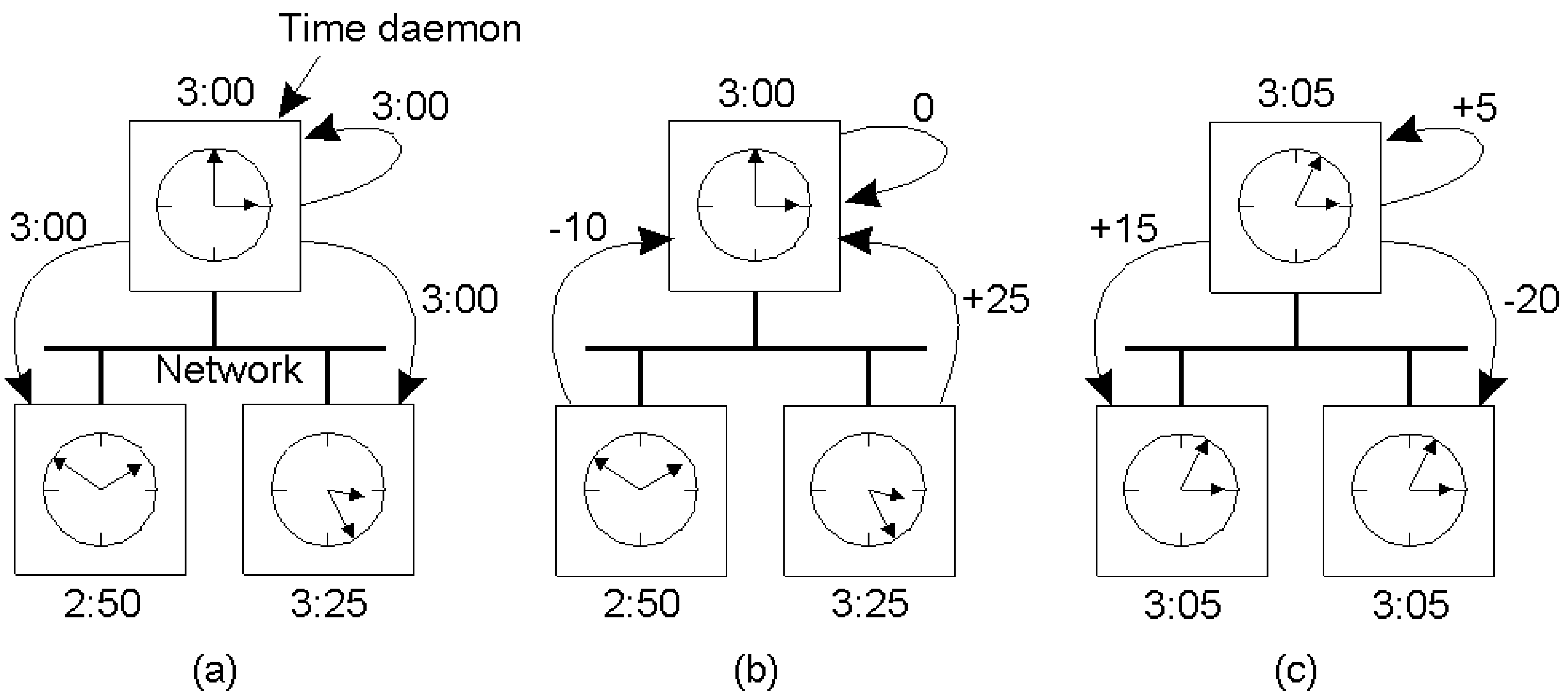
Limitations of Cristian's algorithm

- Variability in estimate of $\mathbf{T_{round}}$
  - can be reduced by repeated requests to S & taking the minimum value of $\mathbf{T_{round}}$
- Single point of failure
  - group of synchronized time servers
    - multicast request & use only 1st reply obtained
- Faulty clocks:
  - f: #faulty clocks, N: #servers
    - N > 3f, Number of servers should be at least more than three times of the faulty clocks for the correct clocks to achieve agreement (observation)
- Malicious interference
  - Protection by authentication techniques

The Berkeley algorithm (I)

- Gusella & Zatti (1989)
  - A Co-ordinator is chosen to act as the master.
  - Co-ordinator (master) periodically polls slaves whose clocks are to be synchronized.
    - estimates each slave's local clock based on round trip time (RTT)
    - averages the values obtained (incl. its own clock value)
    - ignores any occasional readings with RTT higher than *max*
  - Slaves are notified of the adjustment required
    - This amount can be positive or negative
    - Sending the updated current time would introduce further uncertainty, due to message transmit delay- Thus the master sends the amount by which each individual slave's clock requires adjustment.
  - Elimination of faulty clocks
    - averaging over clocks that do not differ from one another more than a specified amount
  - Election of new master, in case of failure
    - no guarantee for election to complete in bounded time

The Berkeley Algorithm (II)



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

Averaging algorithms

- Both of the methods described above are highly centralized with the usual disadvantages.
- One class of decentralized algorithms works by dividing time into fixed length resynchronization intervals. This ith interval starts at T0+iR and runs until T0+(i+1)R, where T0 is an agreed upon moment in the past, and R is a system parameter.
- At the beginning of each interval, every machine broadcasts the current time according to its clock.
- Because the clocks on the different machines do not run exactly the same speed, these broadcasts will not happen precisely simultaneously.
- After a machine broadcasts its time, it starts a local timer to collect all other broadcasts that arrive during some interval S.
- When all broadcasts arrive, an algorithm is run to compute a new time from them. The simplest algorithm is just to average the values from all the other machines.
- A slight variation on this theme is first to discard the m highest and m lowest values, and average the rest. Discarding the extreme values can be regarded as a self defence against up to m faulty clocks sending out nonsense.

Averaging algorithms

- Divide time into fixed-length re-synchronization intervals: $[T_0 + iR, T_0 + (i+1)R]$
  - At the beginning of an interval, each machine broadcasts the current time according to its clock
    - … and starts a local timer to collect all incoming broadcasts during a time interval S
  - When the broadcasts have been received, a new time value is computed
    - Average
    - Average after discarding the m lowest and the m highest values
      - … tolerate up to m faulty machines
    - May also correct each value based on estimate of propagation time from the source machine

Network Time Protocol

NTP Introduction

- Network Time Protocol (NTP) synchronizes clocks of hosts and routers in the Internet.
- NIST estimates 10-20 million NTP servers and clients deployed in the Internet and its tributaries all over the world. Every Windows/XP has an NTP client.
- NTP provides nominal accuracies of low tens of milliseconds on WANs, submilliseconds on LANs, and submicroseconds using a precision time source such as a cesium oscillator or GPS receiver.
- NTP software has been ported to almost every workstation and server platform available today - from PCs to Crays - Unix, Windows, VMS and embedded systems, even home routers and battery backup systems.
- The NTP architecture, protocol and algorithms have been evolved over the last two decades to the latest NTP Version 4 software distributions.

NTP Introduction

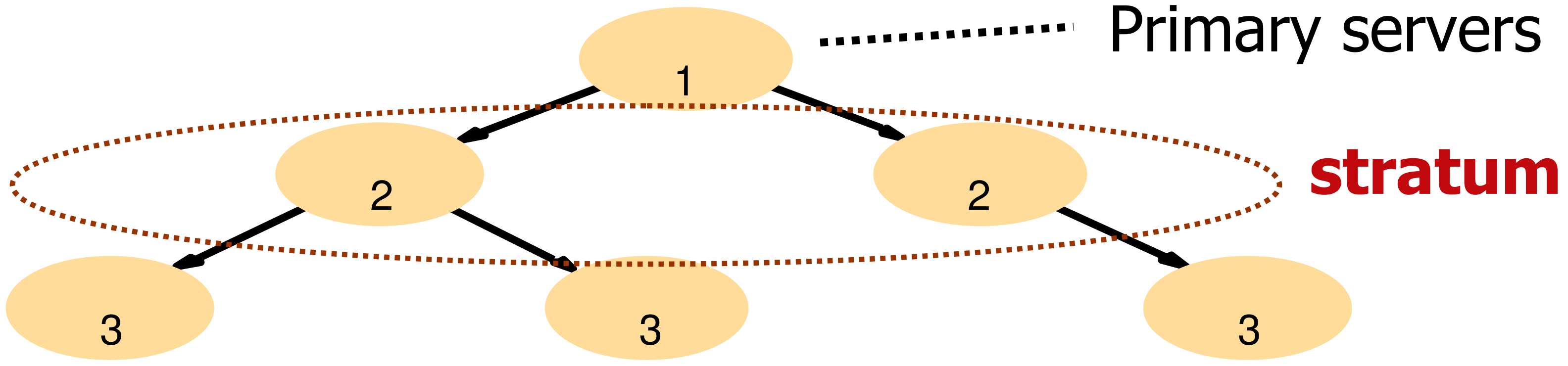
- Primary (stratum 1) servers synchronize to national time standards via radio, satellite and modem.
- Secondary (stratum 2, ...) servers and clients synchronize to primary servers via hierarchical subnet.
- Clients and servers operate in master/slave, symmetric and multicast modes with or without cryptographic authentication.
- Reliability assured by redundant servers and diverse network paths.
- Engineered algorithms reduce jitter, mitigate multiple sources and avoid improperly operating servers.
- The system clock is disciplined in time and frequency using an adaptive algorithm responsive to network time jitter and clock oscillator frequency wander.

NTP: An Internet-scale time protocol

- Statistical filtering of timing data
  - discrimination based on quality of data from different servers
- Re-configurable inter-server connections
  - logical hierarchy
- Scalable for both clients & servers
  - Clients can re-sync. frequently to offset drift
- Authentication of trusted servers
  - … and also validation of return addresses

<u>Sync. Accuracy</u>: ~10s of milliseconds over Internet paths
~ 1 millisecond on LANs

NTP Synchronization Subnets



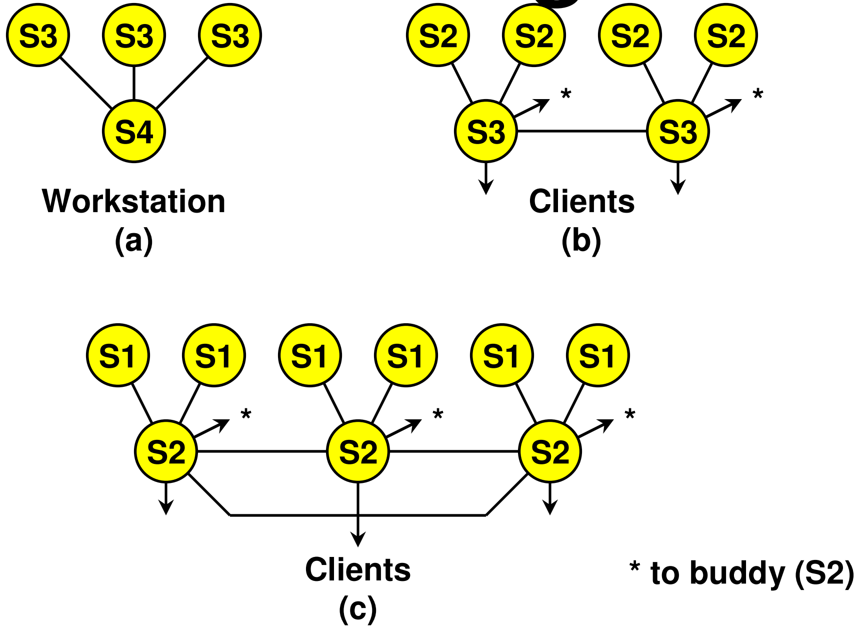
High stratum # → server more liable to be less accurate

Node → root RTT as a quality criterion

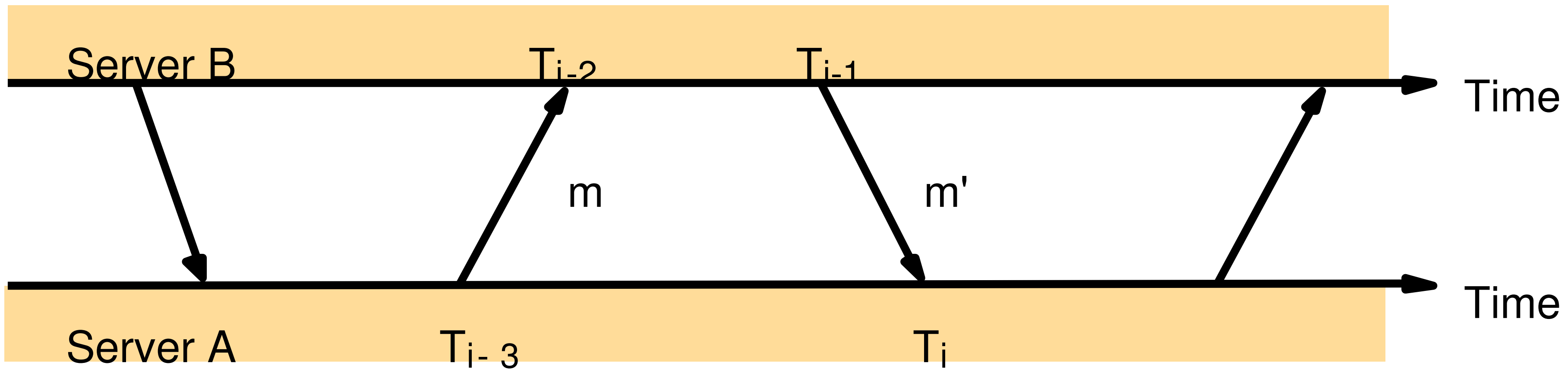
3 modes of synchronization:
- multicast: acceptable for high-speed LAN
- procedure-call: similar to Cristian's algorithm
- symmetric: between a pair of servers

All modes rely on UDP messages.

NTP subnet configurations



- (a) Workstations use multicast mode with multiple department servers.
- (b) Department servers use client/server modes with multiple campus servers and symmetric modes with each other.
- (c) Campus servers use client/server modes with up to six different external primary servers and symmetric modes with each other and external secondary (buddy) servers.



Each message contains the local times when the previous message was sent & received, and the local time when the current message was sent.

- There can be a non-negligible delay between the arrival of one message & the dispatch of the next.
- Messages may be lost

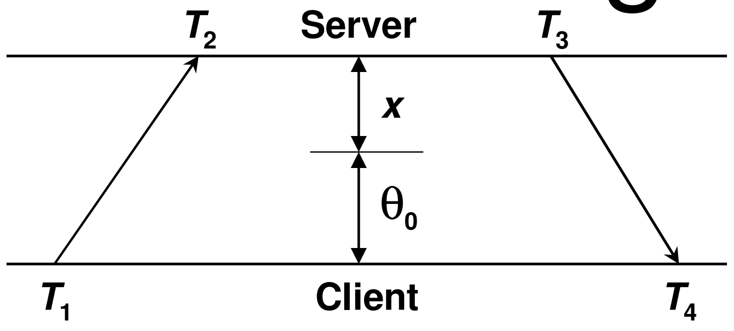
Offset o_i : estimate of the actual offset between two clocks, as computed from a pair of messages

Delay d_i : total transmission time for the message pair

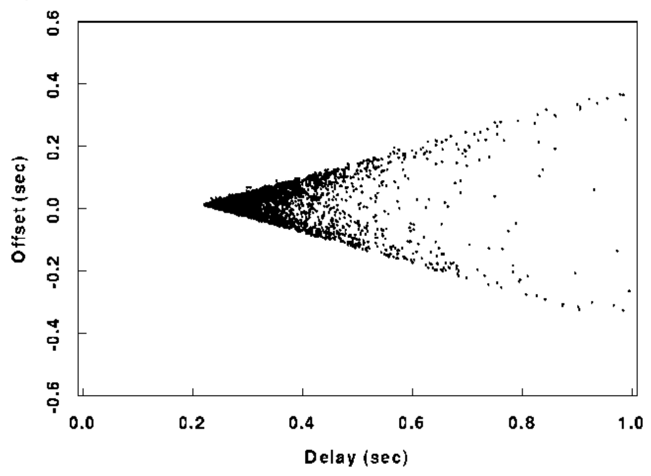
NTP Timestamp Calculation

- let $T1$ be the client timestamp on the request message, $T2$ the server timestamp upon arrival, $T3$ the server timestamp on departure of the reply message and $T4$ the client timestamp upon arrival. NTP calculates the clock offset
  - $offset = [(T2 - T1) + (T3 - T4)] / 2$
- and roundtrip delay
  - $delay = (T4 - T1) - (T3 - T2)$.

Clock filter algorithm



$$\theta = \tfrac{1}{2}[(T_2 - T_1) + (T_3 - T_4)]$$
$$\delta = (T_4 - T_1) - (T_3 - T_2)$$

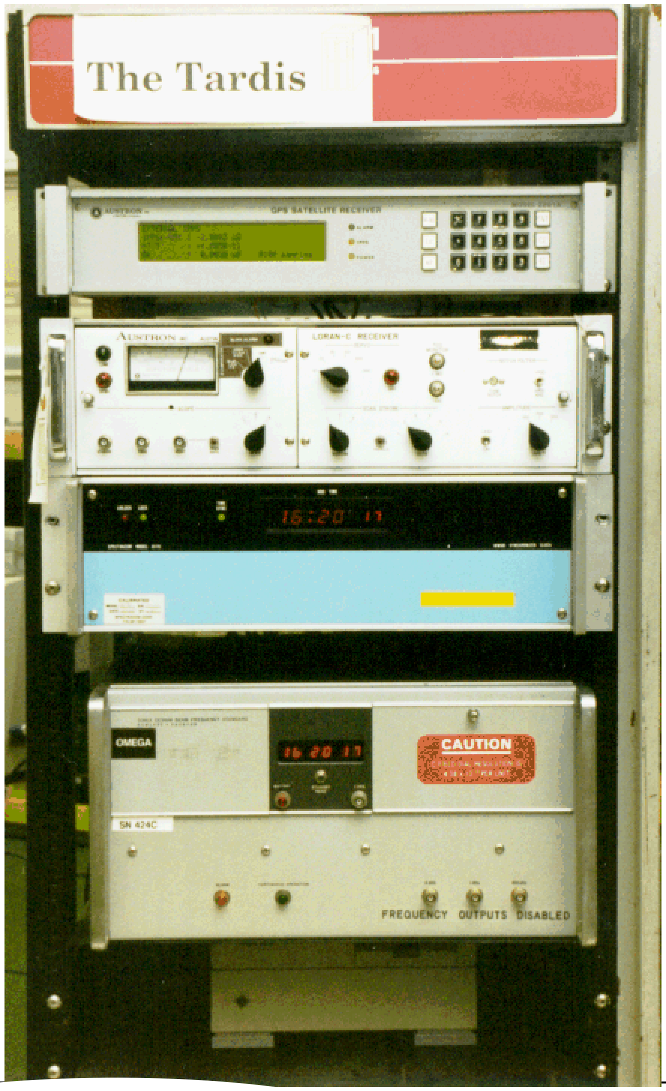


- The most accurate offset θ_0 is measured at the lowest delay δ_0 (apex of the wedge scattergram).
- The correct time θ must lie within the wedge $\theta_0 \pm (\delta - \delta_0)/2$.
- The δ_0 is estimated as the minimum of the last eight delay measurements and (θ_0, δ_0) becomes the peer update.
- Each peer update can be used only once and must be more recent than the previous update.

NTP data filtering & peer selection

- Retain 8 most recent $<o_i, d_i>$ pairs
  - compute “filter dispersion” metric
    - higher values → less reliable data
    - The estimate of offset with min. delay is chosen
- Examine values from several peers
  - look for relatively unreliable values
- May switch the peer used primarily for sync.
- Peers with low stratum # are more favored
  - “closer” to primary time sources
- Also favored are peers with lowest sync. dispersion:
  - sum of filter dispersions bet. peer & root of sync. subnet
- May modify local clock update frequency wrt observed drift rate

Precision timekeeping equipment (Before 2000)



Austron 2200A GPS Receiver

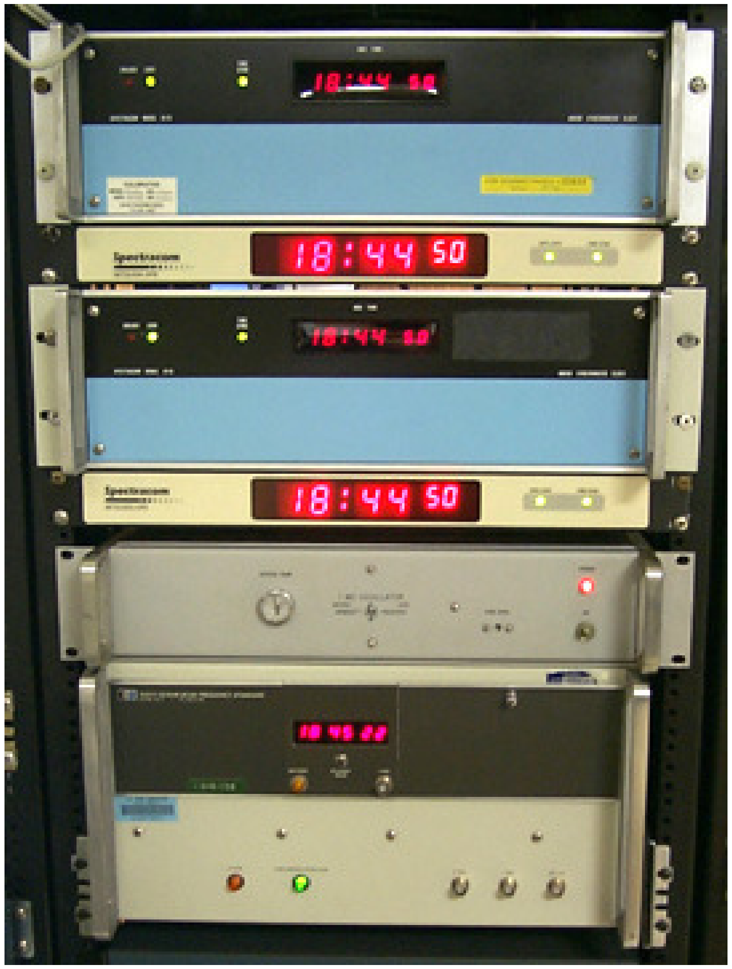
Austron 2000 LORAN-C Receiver

Spectracom 8170 WWVB Reciver

Hewlett Packard 5061A Cesium Beam Frequency Standard

NTP primary time server *rackety*

Udel Master Time Facility (MTF) (from January 2000)



Spectracom 8170 WWVB Receiver

Spectracom 8183 GPS Receiver

Spectracom 8170 WWVB Receiver

Spectracom 8183 GPS Receiver

Hewlett Packard 105A Quartz Frequency Standard

Hewlett Packard 5061A Cesium Beam Frequency Standard

NTP primary time servers *rackety*

Logical time and Logical Clock

Logical time and Logical Clock

- From the point of view of any single process, events are ordered uniquely by times shown on the local clock.
- However, as Lamport pointed out, since we cannot clocks perfectly across a distributed system, we cannot in general use physical time to out the order of any arbitrary pair of events occurring within it.
- In general we can use a scheme that is similar to physical causality, but that applies in distributed systems, to order some of the events that occur at different processes.

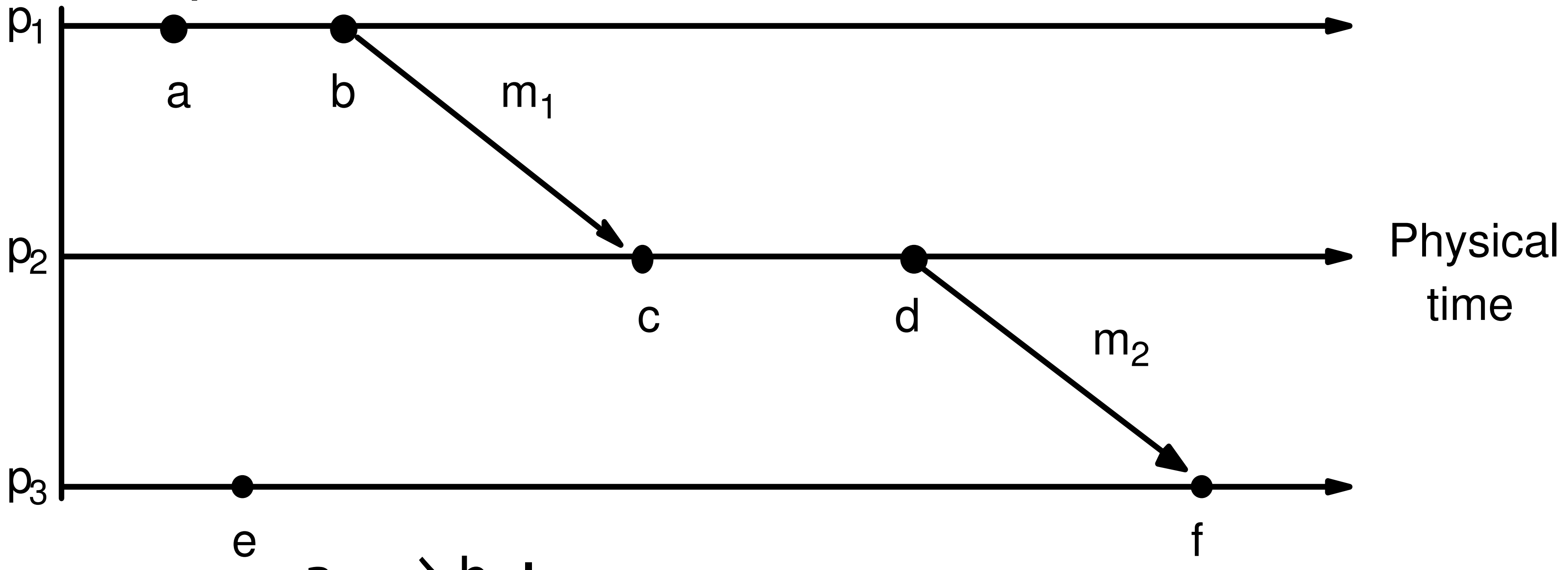
Lamport's notion of logical time

- For many purposes, it is sufficient that all machines agree on the same time
  - … Emphasis on internal consistency
- If two processes do not interact, lack of synchronization will not be observable
  - … and thus will not cause problems
- Ordering of events is needed to avoid ambiguities
- Lamport pointed out that what usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order in which events occur.
- To synchronize logical clocks, lamport defined a relation called happens-before.

The "happened-before" relation

- We cannot synchronize clocks perfectly across a distributed system
  - cannot use physical time to find out event order
  - Lamport, 1978: "happened-before" partial order
    - (potential) causal ordering
    - $e \rightarrow_i e'$, for process $P_i \Rightarrow e \rightarrow e'$
    - send(m) $\rightarrow$ receive(m), for any message m
    - $e \rightarrow e'$ and $e' \rightarrow e'' \Rightarrow e \rightarrow e''$
    - concurrent events: a // b
      - occur at different processes which simply mean that nothing can be (or need be) said about when the events happened or which event happened first.

Space-Time diagram representation of a distributed computation



$$\left.\begin{array}{l} a \rightarrow b \\ c \rightarrow d \\ b \rightarrow c \\ d \rightarrow f \end{array}\right| \Longrightarrow a \rightarrow f$$

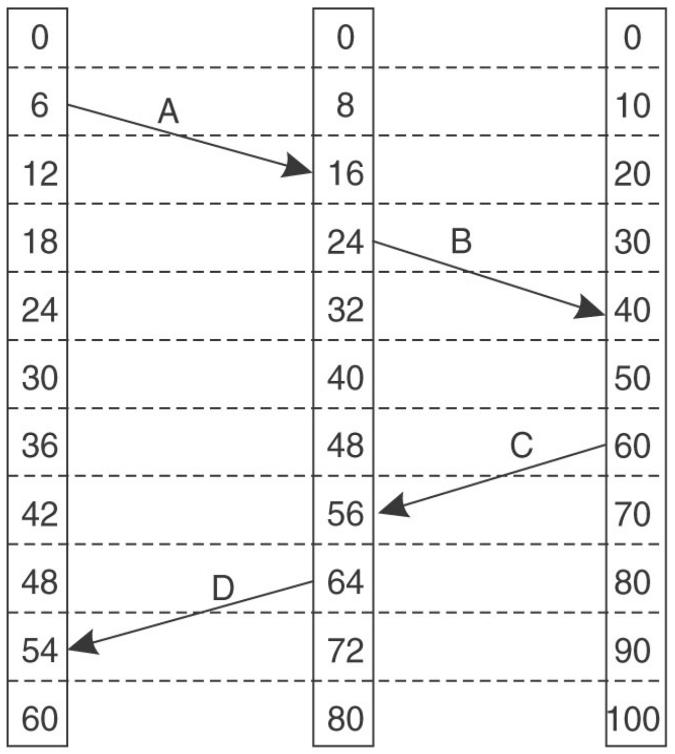
Lamport Timestamps

- Consider the three processes shown in the next slide. These processes run at different machines, each with its own clock running at its own , speed. Each clock runs at constant rate, but the rates are different due to difference in the crystals.
- At time 6, process 0 sends a message A to processes 1. how long this message takes to arrives depends on whose clock you believe. In any event, the clock in process 1 reads 16 when it arrives. If the message carries the starting time, 6, in it, process 1 will conclude that it took 10 ticks to make the journey. This value is certainly possible. According to this reasoning, message B from 1 to 2, takes 16 ticks, again a plausible value.

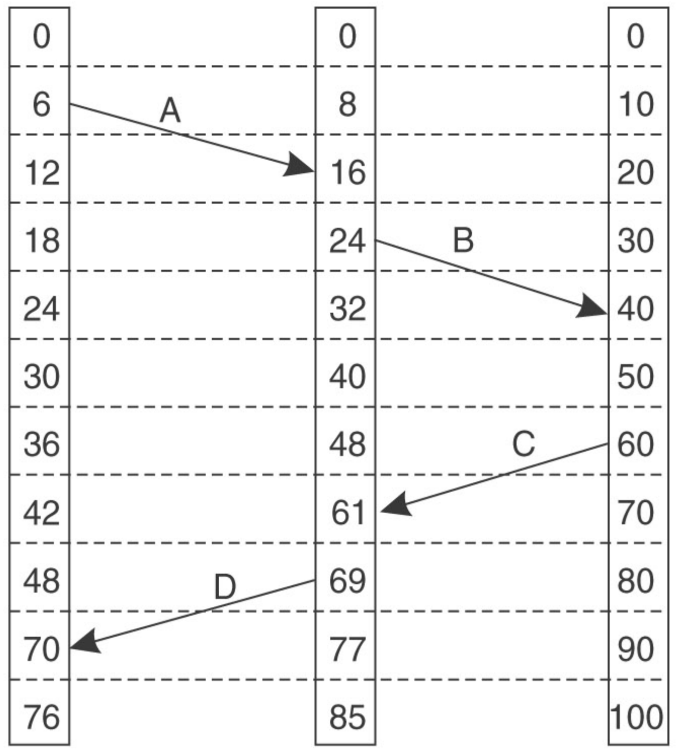
Lamport Timestamps

- Next, message C from 2 to 1 leaves at 60 and arrives at 56. Similarly, message D from 1 to 0 leaves at 64 and arrives at 54. these values are clearly impossible, it is this situation that must be prevented.
- Lamport's solution follows directly from the happens-before relation. Since C left at 60, it must arrive at 61 or later. Therefore each message carries the sending time according to sender's clock. When message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than sending time.
- In fig b, (next slide) we see that C now arrives at 61. Similarly, D arrives at 70.

Lamport Timestamps



(a)



(b)

- 3 processes, each with its own clock. The clocks run at different rates.
- Lamport's algorithm corrects the clocks.

Lamport Timestamps

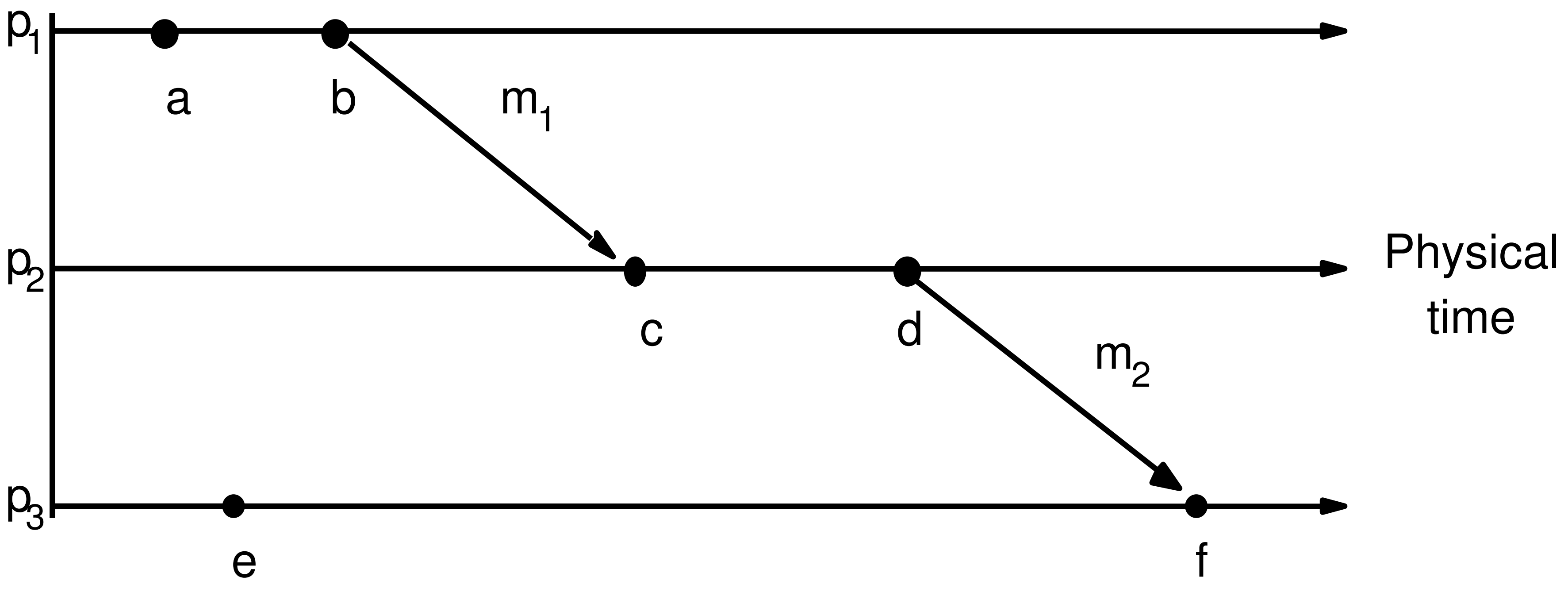
- with one small addition, this algorithm meets our requirements for the global time. The addition is that between every two events, the clock must tick at least once.
- In some situations, an additional requirement is desirable: no two events ever occur at exactly that same time. To achiever this goal, we can attach the number of the processes in which the event occurs to the low-order end of the time, separated by a decimal point. thus, if events happen in processes 1 and 2, both with time 40, the former becomes 40.1 and the latter becomes 40.2
- Using this method we now have a way to assign time to all the events in a distributed system.

Lamport's Logical Clocks (I) Revisited

- Per-process monotonically increasing counters
  - $L_i := L_i + 1$, before each event is recorded at P_i
  - Clock value, t, is piggy-backed with messages
  - Upon receiving <m ,t>, P_j updates its clock:
    - L_j :=max $\{L_{j,\ t}\}$, $L_j := L_j + 1$
- Total order by taking into account process ID:
  - $(T_i, i) < (T_j, j)$ iff $(T_i < T_j$ or $(T_i = T_j$ and $i < j)$)

$$\boxed{e \longrightarrow e' \Rightarrow L(e) < L(e')}$$

Lamport's Logical Clocks (II) Revisited



$\boxed{L(b) > L(e), \text{ but } b \mathbin{//} e}$

Vector Clocks

- A vector clock for a system of N processors is any array of N integers.
- Each process keeps its own vector clock Vi, which it uses to timestamp local events. Like Lamport’s, processes piggyback vector timestamps on the messages they send to one another

Vector Clocks

- **Vector Clocks**
  - Overcome weakness of Lamport's logical clocks:
    - array of N integers
    - each process i keeps own vector clock V_i [1, ..N]
    - piggybacking of timestamps as in Lamport's protocol
    - clock update rules

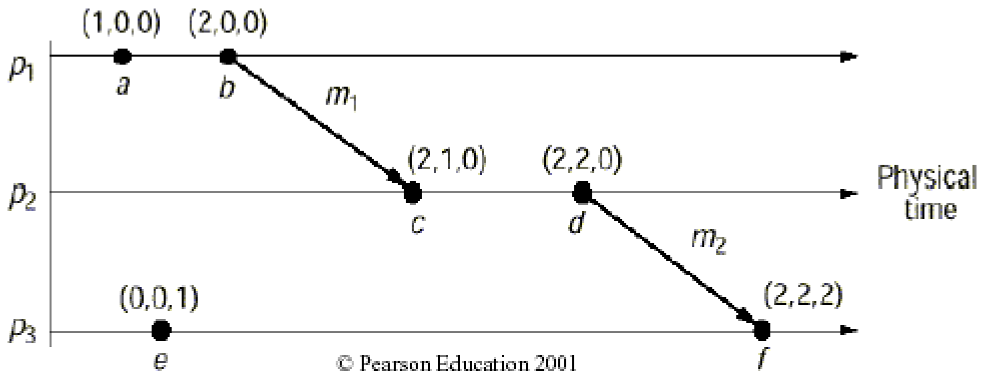
      VC1: Initially, all clocks are 0 on all components

      VC2: i sets $V_i[i] := V_i[i] + 1$ just before timestamping an event

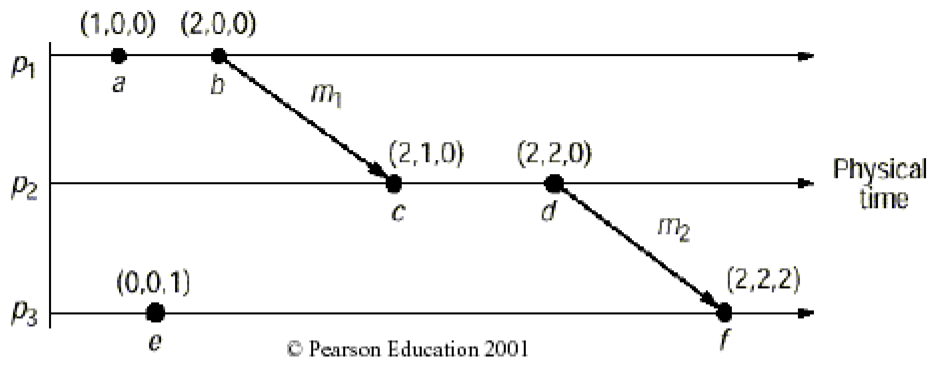
      VC3: i includes t= V_i in every message (piggybacking)

      VC4: i receives a timestamp t, then

      $V_i[j] := \max(V_i[j], t[j])$, $\forall j=1,..,N$ ("merge")



Vector Clocks



- **Vector Clocks**
  - Observation
    - $V_i[j]$ is the number of events in j that i has potentially been affected by.
  - Vector timestamp comparison:
    V = V' iff V[j] = V'[j] $\forall$j=1,..,N
    V $\leq$ V' iff V[j] $\leq$ V'[j] $\forall$j=1,..,N
    V < V' iff V $\leq$ V' and V $\neq$ V'
  - In example
    - V(b) < V(d)
    - V(e) unordered to V(d), i.e., e || d
  - Theorem
    e $\rightarrow$ e' $\Leftrightarrow$ V(e) < V(e')
  - Critique
    - storage and message overhead proportional to N
    - matrix clocks: reduced message overhead through partial vector transmission and local clock estimation

Reference

Vector Clocks: Revisited

- Mattern, 1989 & Fidge, 1991:
  - clock := vector of N numbers (one per process)
  - V_i [i] := V_i [i] + 1, before P_i timestamps an event
  - Clock vector is piggybacked with messages
  - When P_i receives <m ,t> :
    - V_i [j] := max{ t[j], V_i [j] }, for j=1, …, N
  - V_i [j], $j \neq i$: #events that have occurred at P_j and has a (potential) effect on P_i
  - V_i [i]: #events that P_i has timestamped

$$\boxed{e \longrightarrow e' \Leftrightarrow V(e) < V(e')}$$

Notes !!!

Synchronization (II)

Distributed Systems

BScIV Computer Science

Dr. M- Waseem Akhtar

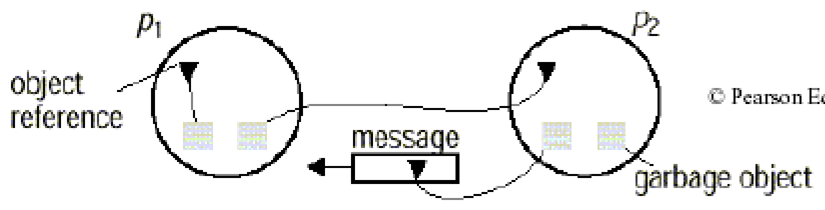
Global States & Coordination

Global States.

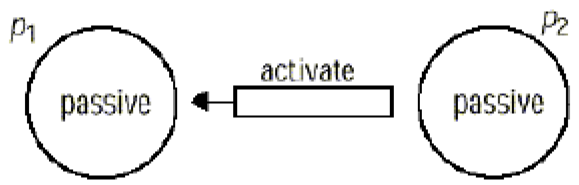
- Consider the problem of finding out whether a particular property is true of a distributed system as it executes. Such as, **Garbage Collection**, **Deadlock Detection**, **Termination Detection** and **Debugging**.

Global States

- **Problems that would require the view on a global state**
  - Distributed garbage collection: is there any reference to an object left?



  - Distributed termination detection: is there either an active process left or is any process activation message in transit?

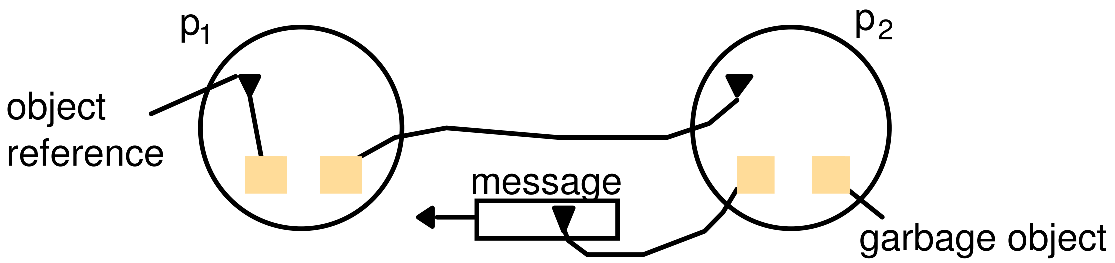


Detecting global properties

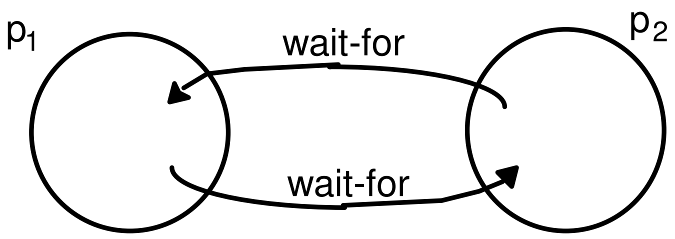
- Evaluation of predicates of system's state
  - stable predicates:
    - distributed garbage collection
    - deadlock detection
    - termination detection
  - non-stable (transient) predicates:
    - distributed debugging
  - safety properties:
    - "Nothing bad ever happens"
      - eg: mutual exclusion
  - liveness properties:
    - "Something good eventually happens"
      - eg: fair scheduling

Evaluation of Stable Global Predicates

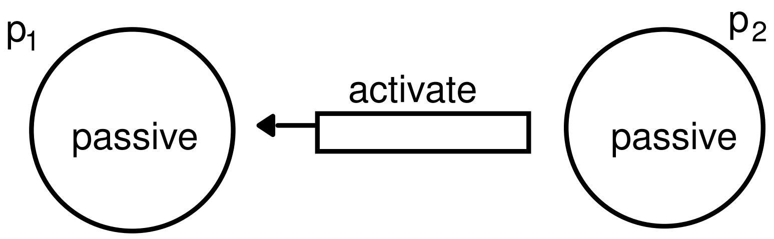
a. Garbage collection



b. Deadlock



c. Termination



Global States

- It is possible to observer the succession states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes – is much harder to address.
- The essential problem is the absence of global time.
- So the question is whether we can assemble a meaningful global state from local states recorded at different times.
- The answer is a qualified “yes”.

Global State Predicates

- Detecting a condition such as deadlock or termination amounts to evaluating a global sate predicate.
- A global state predicate is a function that maps from the set of global states of processes in the system to [true, false].
- One of the useful characteristics of the predicate associated with the state of an object being garbage, of the system being deadlock or the system being terminated is that they are all stable: once the system enters a state in which the predicate is True, it remains True for all future states reachable from that state.
- By contrast, when we monitor or debug an application we are often interested in non-stable predicates, even if the application reaches a such state, it need not stay in that state.

Global States

- A process can record its own state and the messages it sends and receives; it can record nothing else. To determine a global system state, a process p must enlist the cooperation of other processes that must record their local states and send the recorded local states to p.
- All processes cannot record their local states at precisely the same instant unless they have access to a common clock. We assume that processes do not share clocks or memory. The problem is to devise algorithms by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state.
- The global-state-detection algorithm is to be superimposed on the underlying computation: it must run concurrently with, but not alter, this underlying computation.

Distributed Snapshots

- The state-detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds-a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to define "meaningful" and then to determine how the photographs should be taken.

Distributed snapshot

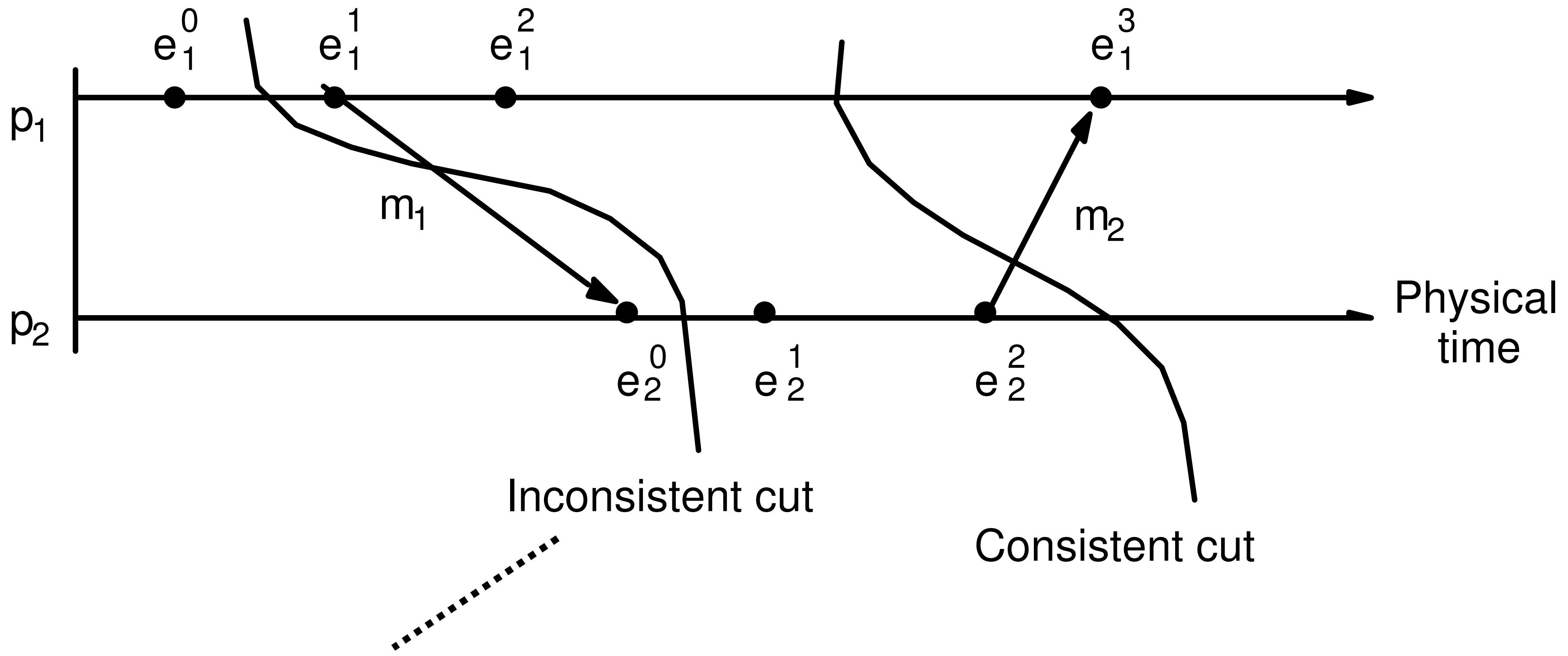
… reflects a state in which the system might have been

This state is consistent.

A global state can be graphically represented by a cut:

- represents the last event recorded for each process

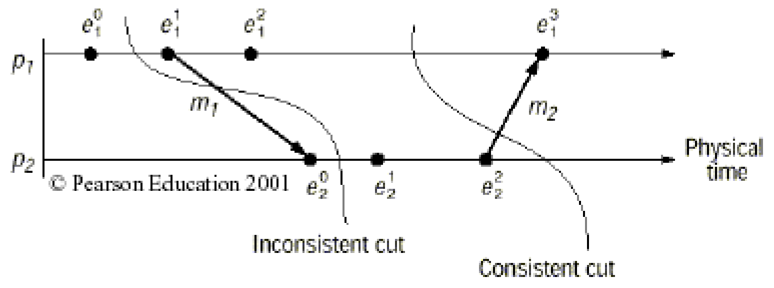
Consistent vs Inconsistent cuts



Includes the receipt by P2 of message m1 from P1, while P1 has yet no record of sending !

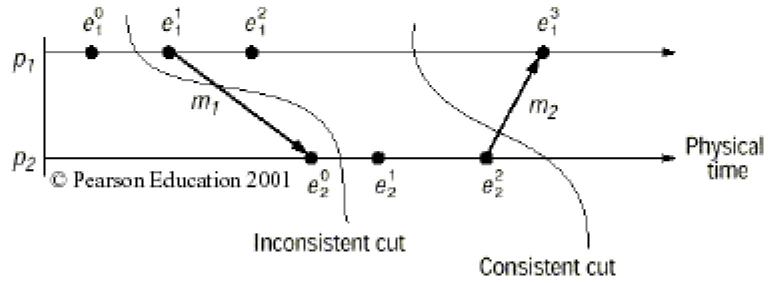
"effect without a cause"

Global States



- **Consistent global states**
  - state of s_i in cut C is that of p_i immediately succeeding the last event processed by p_i in C (i.e., e_i^{ci})
  - global system state S = (s_1, s_2, .., s_N)
  - a consistent global system state is one that corresponds to a consistent cut

Global States



- **Global state sequences**
  - consider a system as evolving in a sequence of global state transitions

    $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$

    – precisely one process performs a local transition in every step of the sequence

    – describes a partial order of all events in the system

    – concurrent events can be thought of as having happened in some total order (linearization) that is consistent with the partial order described through $\rightarrow$

    – S' is reachable from a state S if there is a linearization that passes through S and then S'

    – possible to further formalize these concepts

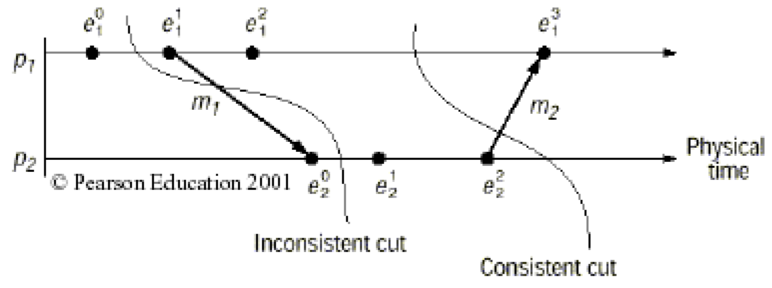
Predicate Functions defined on Global States

- We now describe an important class of problems that can be solved with the global-state-detection algorithm. Let y be a predicate function defined on the global states of a distributed system D; that is, y(S) is true or false for a global state S of D. The predicate y is said to be a stable property of D if y(S) implies y(S') for all global states S' of D reachable from global state S of D. In other words, if y is a stable property and y is true at a point in a computation of D, then y is true at all later points in that computation. Examples of stable properties are "computation has terminated, " "the system is deadlocked," and "all tokens in a token ring have disappeared."

Predicate Functions defined on Global States

- Several distributed-system problems can be formulated as the general problem of devising an algorithm by which a process in a distributed system can determine whether a stable property y of the system holds. Deadlock detection and termination detection are special cases of the stable-property detection problem. The basic idea of the algorithm is that a global state S of the system is determined and y(S) is computed to see if the stable property y holds.

Global States



- **State Properties**
  - liveness properties: "Eventually something good will happen", e.g.
    - the system will eventually make progress
    - the system will eventually terminate
    - every sent message will eventually be received (leadsto)
    - if a process requests access to the critical section infinitely often, it will be granted access infinitely often (strong fairness)

Global state

- Initial prefix of process history:
  - $h_i^k = \langle e_i^0, e_i^1, e_i^2, \ldots, e_i^k \rangle$
  - s_i^k: state of P_i immediately before the kth event
- Cut of system's execution:
  - $C = h_1^{c1} \cup \cdots \cup h_N^{cN}$
  - $\{e_i^{ci} \mid i=1, \ldots, N\}$ - frontier of cut C
- Consistent cut:
  - $f \longrightarrow e$ and $e \in C \Rightarrow f \in C$

A consistent cut corresponds to a consistent global state.

Model of A Distributed System

- A distributed system consists of a finite set of processes and a finite set of channels. It is described by a labeled, directed graph in which the vertices represent processes and the edges represent channels. Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order sent. (The infinite buffer assumption is made for ease of exposition: bounded buffers may be assumed provided there exists a proof that no process attempts to add a message to a full buffer.) The delay experienced by a message in a channel is arbitrary but finite. The sequence of messages received along a channel is an initial subsequence of the sequence of messages sent along the channel. The state of a channel is the sequence of messages sent along the channel, excluding the messages received along the channel.

Model of A Distributed System

- A process is defined by a set of states, an initial state (from this set), and a set of events. An event e in a process p is an atomic action that may change the state of p itself and the state of at most one channel c incident on p: the state of c may be changed by the sending of a message along c (if c is directed away from p) or the receipt of a message along c (if c is directed towards p). An event e is defined by (1) the process p in which the event occurs, (2) the state s of p immediately before the event, (3) the state s' of p immediately after the event, (4) the channel c (if any) whose state is altered by the event, and (5) the message M, if any, sent along c (if c is a channel directed away from p) or received along c (if c is directed towards p). We define e by the 5-tuple (p, s, s', M, c), where M and c are a special symbol, null, if the occurrence of e does not change the state of any channel.

Model of A Distributed System

- A global state of a distributed system is a set of component process and channel states: the initial global state is one in which the state of each process is its initial state and the state of each channel is the empty sequence. The occurrence of an event may change the global state.

The Algorithm

- The global-state recording algorithm works as follows: Each process records its own state, and the two processes that a channel is incident on cooperate in recording the channel state. We cannot ensure that the states of all processes and channels will be recorded at the same instant because there is no global clock; however, we require that the recorded process and channel states form a "meaningful" global system state.

Global System States

- Snapshot algorithm
- Monitor Process

Please have a look at the handout

Snapshots (I)

- Chandy & Lamport, 1985
  - Algorithm to select a <u>consistent cut</u>
  - Any process may initiate a snapshot at any time
  - Assumes no failures of processes & channels
  - Assumes string connectivity
    - At least one path between each process pair
  - Assumes unidirectional, FIFO channels
  - Assumes reliable delivery of messages
  - Records snapshot state locally at <u>all</u> processes
    - No direct method for collecting state at a designated "collector" process

Snapshots (II)

- Application of 2 rules at each process:
  - Marker sending rule:
    - After P_i has recorded its state, it sends a “marker” message over each of its outgoing channels (before sending any other message)
  - Marker receipt rule: (c := the incoming channel)
    - If P_i has not yet recorded its state:
      - P_i records its state & records the state of channel c as “$\emptyset$”
      - P_i turns on recording of messages arriving over incoming channels
    - Else:
      - P_i records the state of channel c as the set of messages that it has received over c since it saved its state

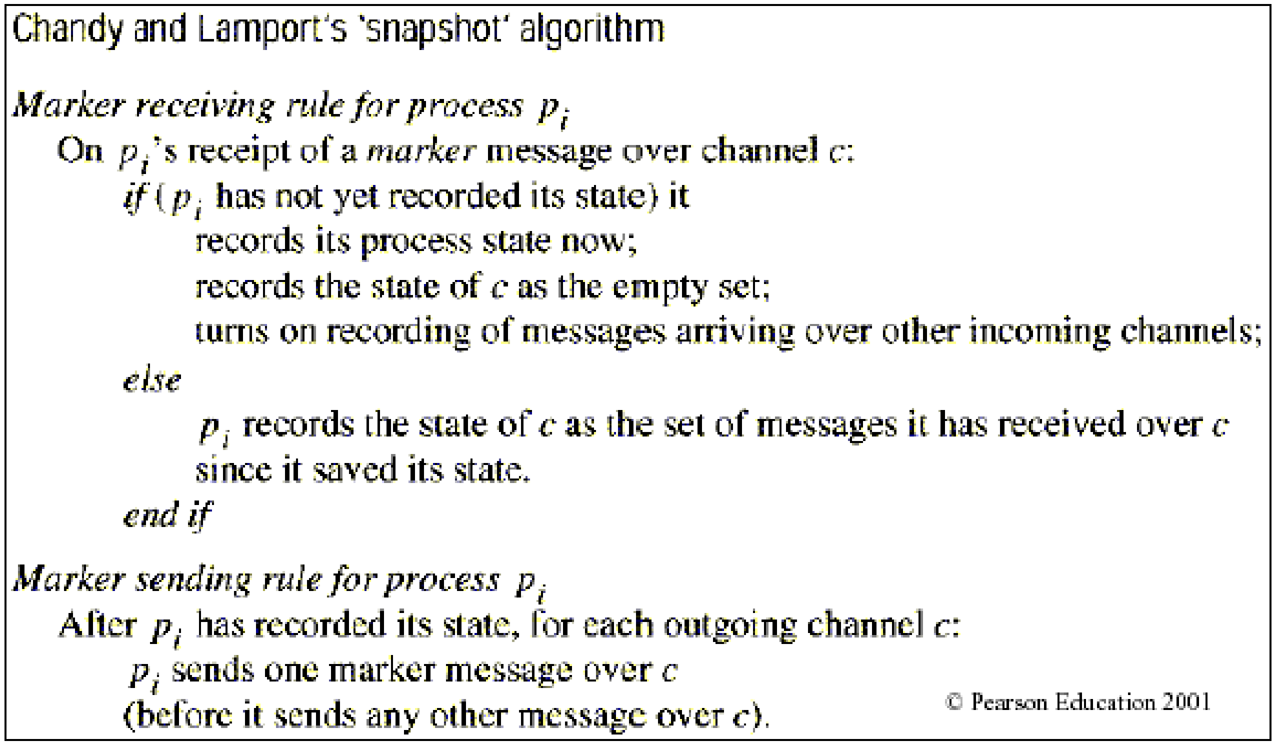
Snapshots (III)

In-transit messages are accounted for as belonging to the state of a channel between process.

- A process that has received a “marker” message, records its state in finite time & relays the “marker” over its outgoing channels in finite time.
- Strongly connected network
- The “marker” traverses each channel, exactly once.
- When a process has received a “marker” over all its incoming channels, its contribution to the snapshot protocol is complete.

Snapshots

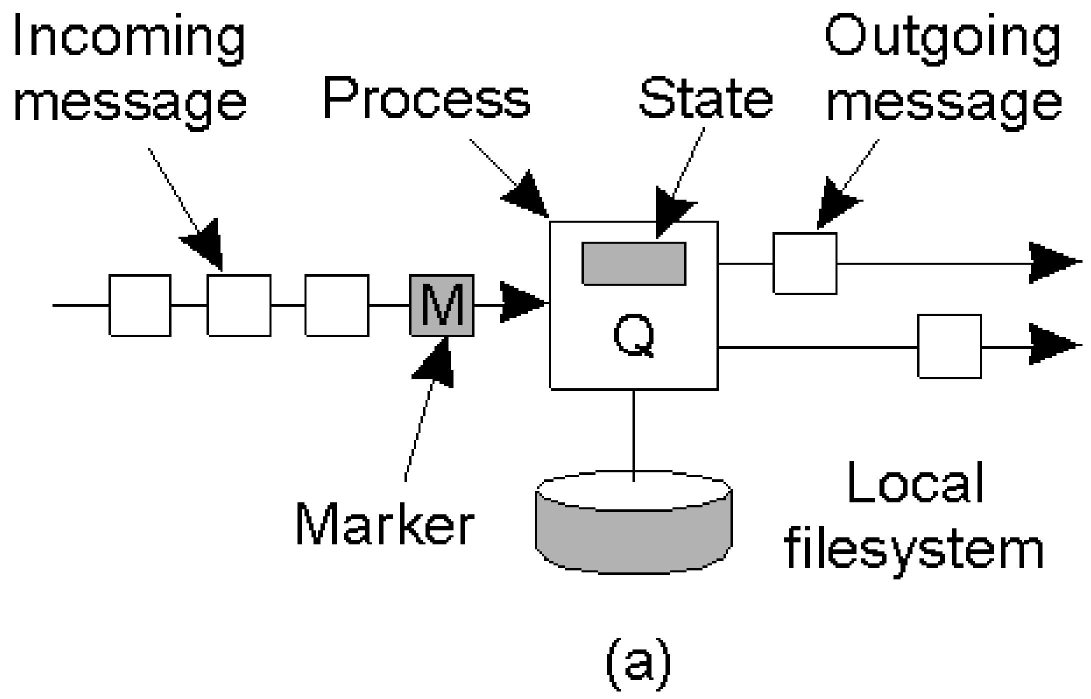
- **Chandy-Lamport Algorithm for the determination of consistent global states**



Chandy and Lamport's 'snapshot' algorithm

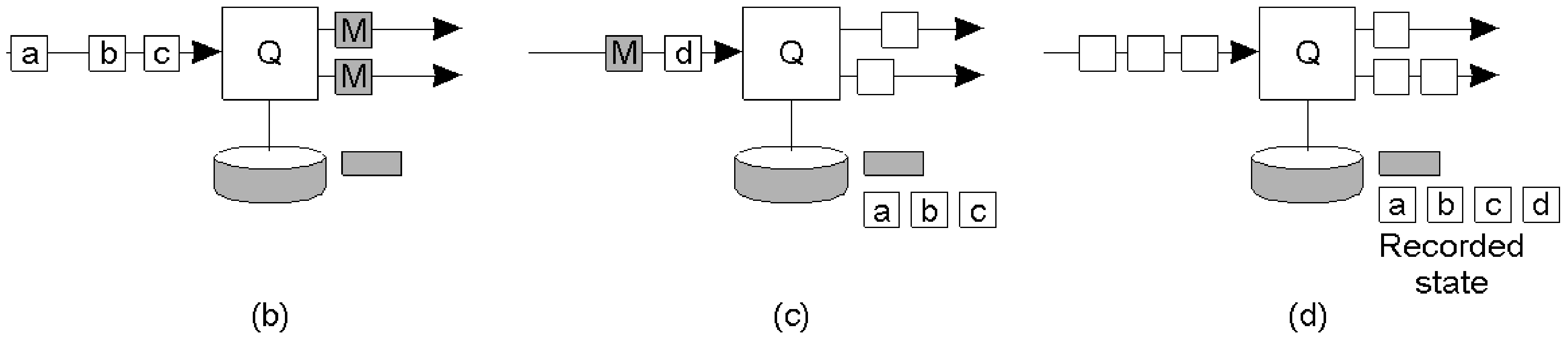
```
Marker receiving rule for process $p_i$
  On $p_i$'s receipt of a marker message over channel $c$:
    if ($p_i$ has not yet recorded its state) it
      records its process state now;
      records the state of $c$ as the empty set;
      turns on recording of messages arriving over other incoming channels;
    else
      $p_i$ records the state of $c$ as the set of messages it has received over $c$
      since it saved its state.
    end if
Marker sending rule for process $p_i$
  After $p_i$ has recorded its state, for each outgoing channel $c$:
    $p_i$ sends one marker message over $c$
    (before it sends any other message over $c$).
```

Global State (I)



a) Organization of a process and channels for a distributed snapshot

Global State (II)



- b) Process Q receives a marker for the first time and records its local state
- c) Q records all incoming message
- *d)* *Q* receives a marker for its incoming channel and finishes recording the state of the incoming channel

Global State (III)

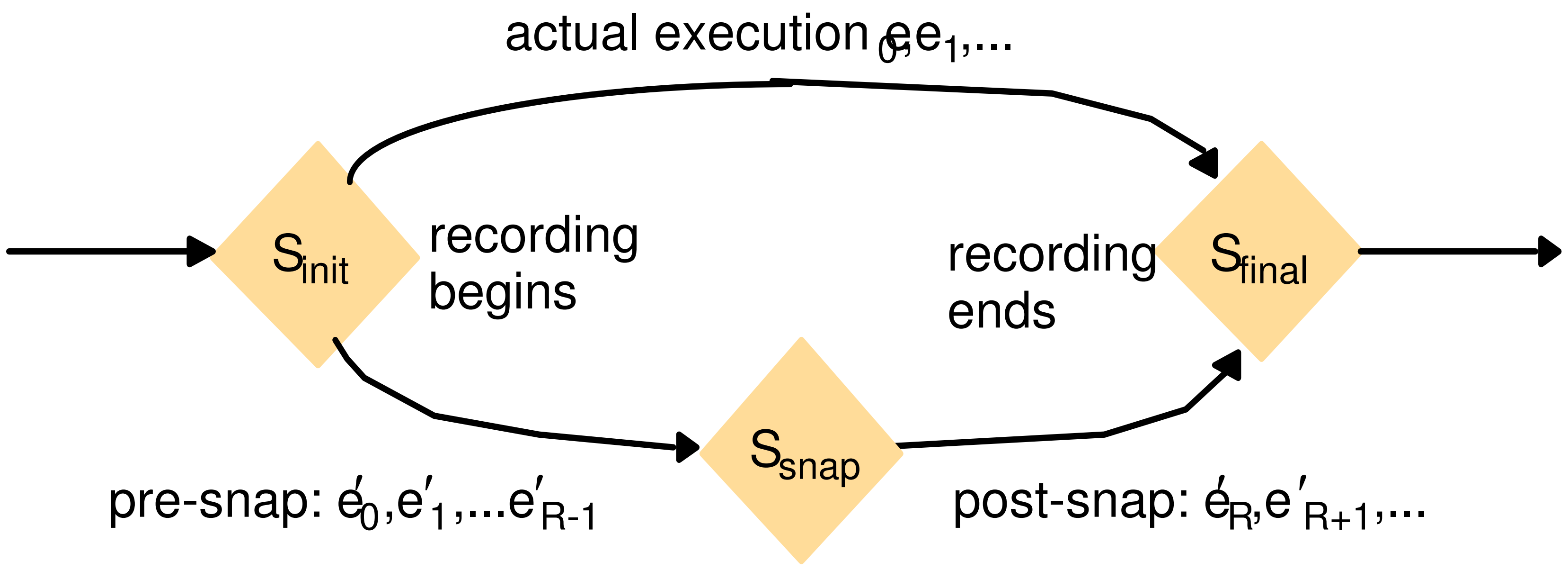
- The construction of multiple snapshots may be in progress at the same time
  - The marker must be tagged with an ID (and possibly also a version #) of the initiator.

Example application: termination detection

- Each process considers the process from which it first received the marker as its predecessor.
- We need a snapshot in which all channels are empty.
- When a process finishes its part in the snapshot, it either returns a DONE or a CONTINUE message to its predecessor.

It returns DONE only when all of its successors have returned DONE, and it has not received any msg between the point it recorded its state & the point it had received the marker along its of its incoming channels.

Reachability between states in the snapshot algorithm


actual execution $e_0, e_1, \ldots$
S_{init}
recording begins
recording ends
S_{final}
S_{snap}
pre-snap: $e'_0, e'_1, \ldots e'_{R-1}$
post-snap: $e'_R, e'_{R+1}, \ldots$

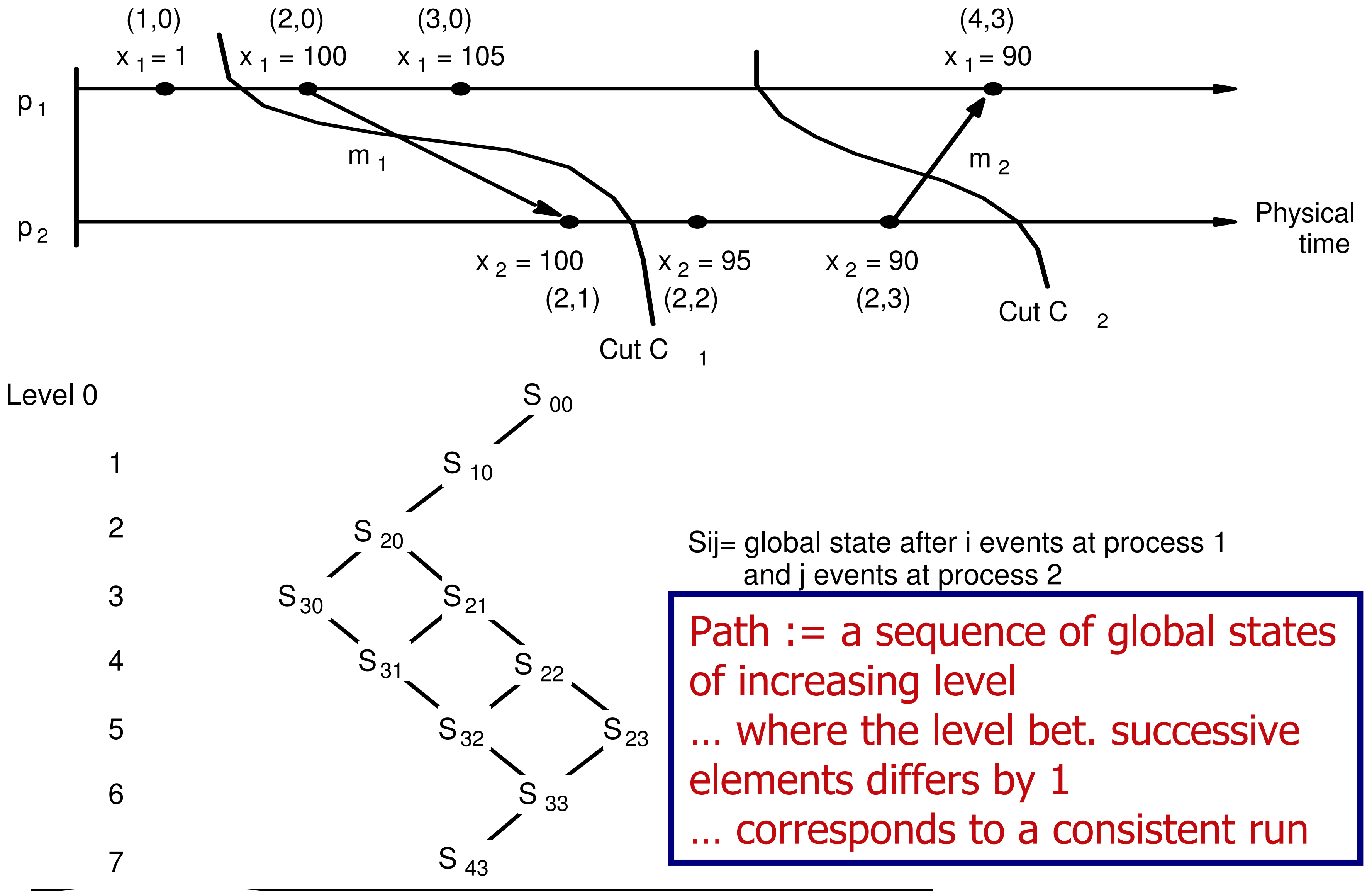
At-Most-Once Message Delivery

- Assign each message a unique number
- Receiver remembers all message numbers received
- Receiver discards messages with numbers already seen

What if receiver crashes & loses message number tables?
When can message tables be purged?

•Assign a timestamp and a connection number (chosen by sender).
•Receiver records most recent time stamp for that connection
  •Messages with a lower timestamp are discarded
•After time G = CurrentTime - (MaxLifeTime + MaxClockSkew), discard entry, where MaxLifeTime is how long message can live in the network
•Write this table to disk periodically
•After crash, reload table & update G, rejecting any message generated before crash.

Lattice of global states



Sij= global state after i events at process 1
and j events at process 2

Path := a sequence of global states of increasing level
… where the level bet. successive elements differs by 1
… corresponds to a consistent run

Transient Global Predicates

Reference Slide

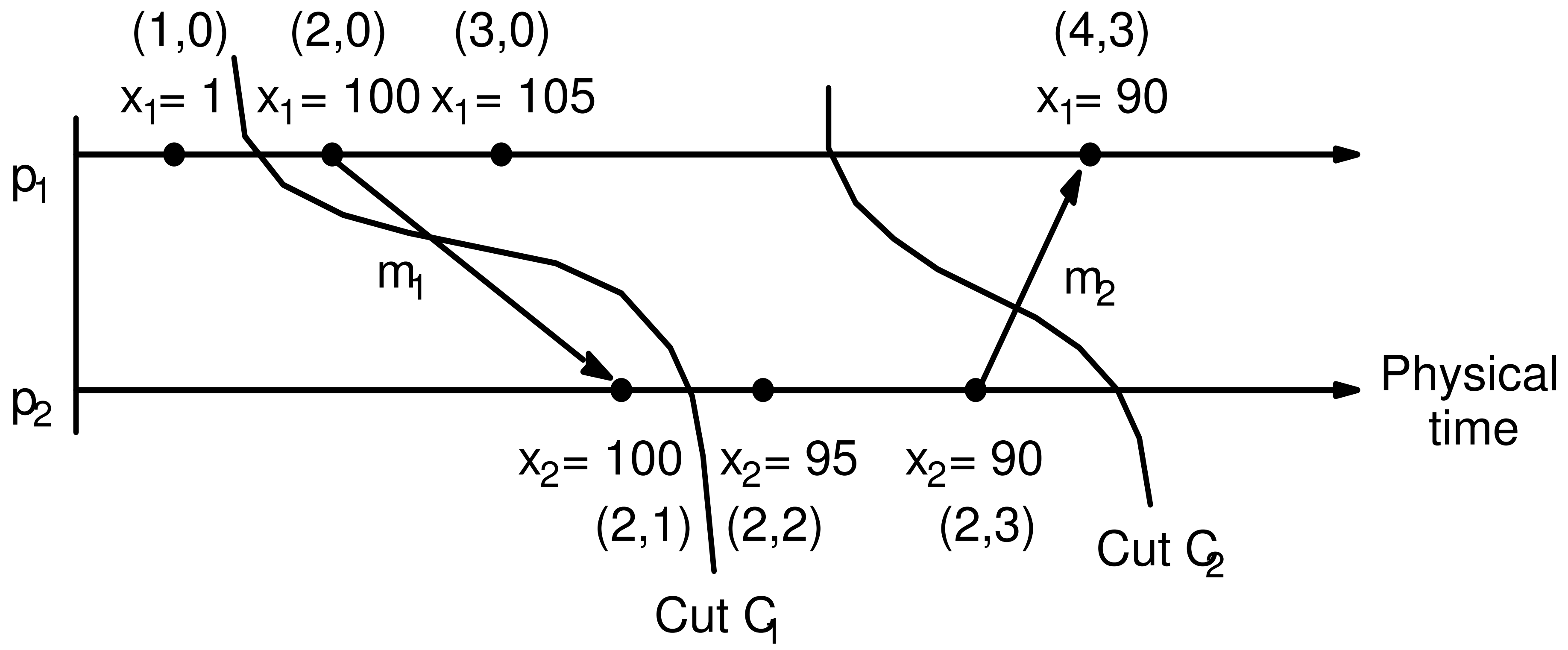
- Distributed debugging:
  - Is $|x_i - x_j| <= D$?
- Distributed control:
  - Are all pressure valves open at the same time ?
- Key challenge:
  - Capture “trace” rather than a single snapshot
  - Establish **post hoc** whether the required safety condition was or may have been violated
- Monitor algorithm - Marzullo & Neiger, 1991
- Φ(s): global state predicate
  - possibly Φ:
    - There is a consistent global state S through which a consistent run passes s.t. Φ(S) = TRUE
  - definitely Φ:
    - For all consistent runs:
      - There is a consistent global state S s.t. Φ(s) = TRUE

$$\boxed{\neg \text{possibly } \Phi \;\; \overset{\Rightarrow}{\not\Leftarrow} \;\; \text{definitely}(\neg \Phi)}$$

Collecting the state

- M: monitor process
  - maintains queue Q_i for each process
  - requires consistent global state S to evaluate Φ
- Process P_i sends its state to M
  - periodically
  - … or only upon changes
  - may send only “relevant” part of state
  - may send state only when Φ can change
  - State messages include vector clock $V(s_i)$
    - M keeps Q_i ordered by <u>sending order</u>
      - order indicated by ith component of vector timestamp

Consistent vs Inconsistent states



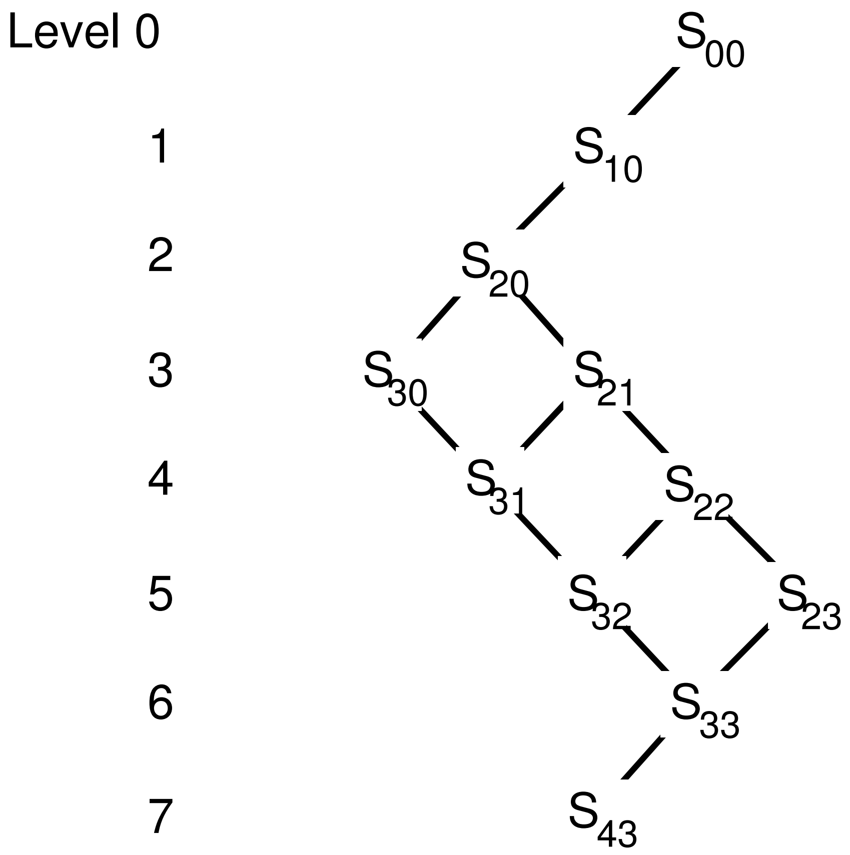
Integrity constraint: |x1 - x2| < 50

For cut C1, the integrity constraint would be seen as violated … although this has never happened !

The Consistent GS Condition

- S: consistent global state iff
  - $V(s_i)[i] >= V(s_j)[i]$, for all i,j=1, …, N
- If the state of P_i depends upon that of P_j (according to the relation $\rightarrow$) then the global state must include the state of P_j
- Lattice of global states
  - S_{ij}: state after i events at P1 & j events at P2
  - arranged in levels: S_{ij} -> level #(i+j)
- Consistent run:
  - traverses the lattice one level at a time
    - In each step some process experiences one event

Lattice of global states



To evaluate "possibly Φ":
M starts at initial state & steps through all consistent states reachable, evaluating Φ at each stage.
- M stops when Φ becomes TRUE.

To evaluate "definitely Φ":
M tries to find a set of states through which all consistent runs must pass, and at which Φ evaluates to TRUE.

S' reachable from S iff
$V(s_j)[i] >= V(s'_j)[j]$,
for all $j \neq i$

Evaluation of Φ level-by-level
… rather than depth-first !

Make Sure that you know !!!

- Distributed Computations, Events, History of a Process, FIFO Channels, Vector Time Stamps
- The Need / Concept of Global states, Consistent and Inconsistent Global States.
- Cut, consistent and inconsistent cut.
- Global States Predicates, stable and unstable predicates.
- Use of global state predicates to evaluate distributes system properties, such as Distributed deadlock detection, termination detection, Garbage Collection, Distributed Debugging etc.
- Distributed Snapshots Algorithm, role and function of a Monitor process, monitor algorithm.
- Lattice of Global State, Consistent Run, Evaluation of *possibility* and *definitely* of a predicate function Φ.

IMPORTANT

Make Sure that you go through the following handout distributed in the class to fully understand the concepts mentioned in the previous slide.

Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms
by Ozlap Babaoguu & Keith Marzullo

Coordination & Agreement

Coordination & Agreement

- Now we introduce some topics and algorithms related to the issue of how processes coordinate their actions and agree on the shared values in distributed systems.
- We begin with the leader election algorithms.
- Then we look at the algorithms to achieve mutual exclusion among collection of processes, so as to coordinate their accesses to shared resources.

Election Algorithms

- An algorithm for choosing a unique process to play a particular role is called an election algorithm.
- Afterwards, if the process that plays the “leader role” wishes to retire, then another election is required to choose a replacement.
- It is assumed that a process calls the election if it takes an action that initiates a particular run of the election algorithm.
- At any given time a process is either a participant, meaning it is engaged in the election algorithm, or non-participant, meaning that it is not currently engaged in any election.
- An important requirement is for the choice of elected process to be unique, even if several processes call elections concurrently.
- Usually it is required that the elected process be chosen as the one with the largest identifier. The “identifier” may be any useful value, as long as the identifiers are unique.

Elections

- Choose a unique process to play a "role"
  - We require that the elected process be chosen as the one with the largest ID
    - Ids must be unique & totally ordered
      - Eg: ID := <1/load, i> : process with the lowest computational load
- Requirements:
  - safety:
    - A participant process has elected == P, where P is chosen as a non-crashed process with max. ID, or elected is undefined
  - liveness:
    - All processes participate & eventually set 'elected', or crash

The Bully algorithm

Garcia-Molina, 1982

Uses timeouts to detect failures:

$T = 2T_{trans} + T_{process}$

Assumes that each process knows which processes have higher IDs:

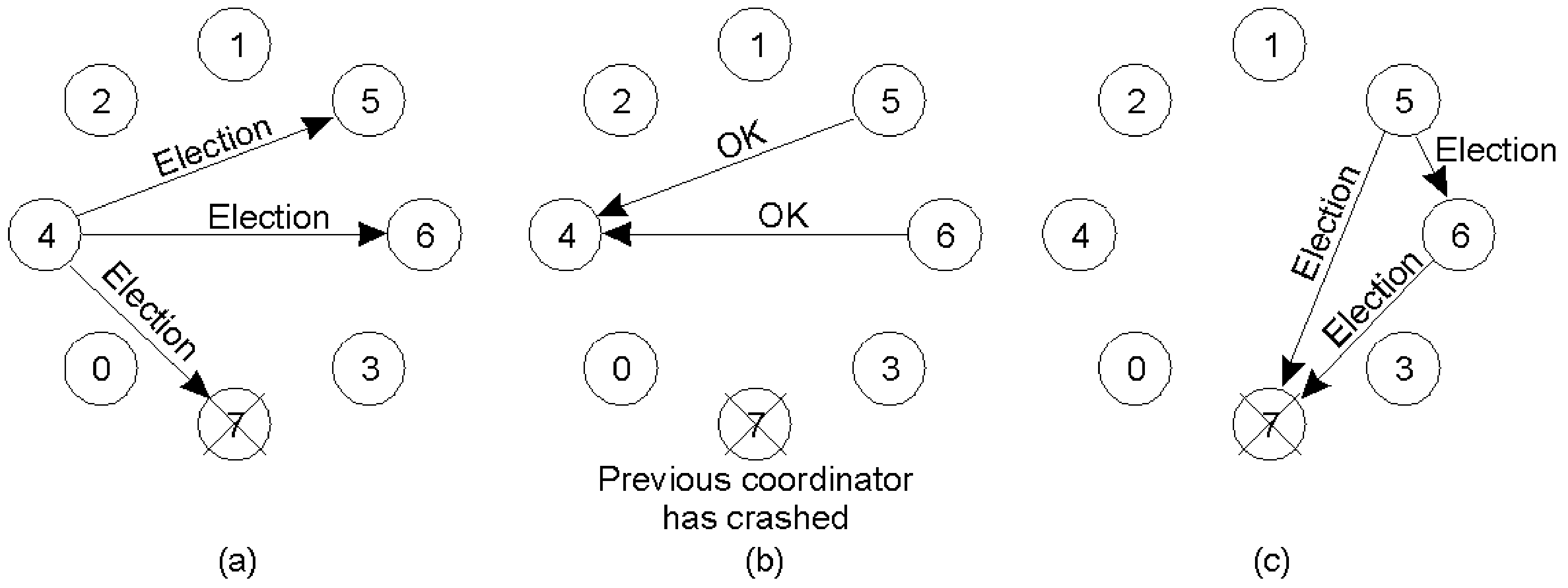
A process can elect itself

<u>3 msg types</u>:

- coordinator:
  - announcement to all processes with lower IDs
- election:
  - sent to processes with higher IDs
- answer:
  - answer to "election" - If not received within T, the sender of "election" sends "coordinator".
  - Otherwise, the process waits for T' to receive a "coordinator" msg. If no msg arrives, it begins a new election.

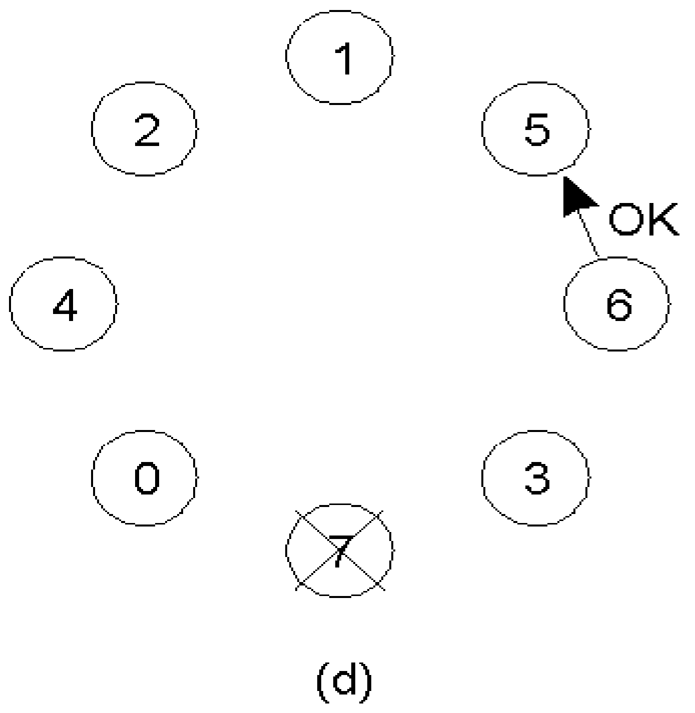
Not "safe" if the crashed processes are replaced with processes with the same ID !

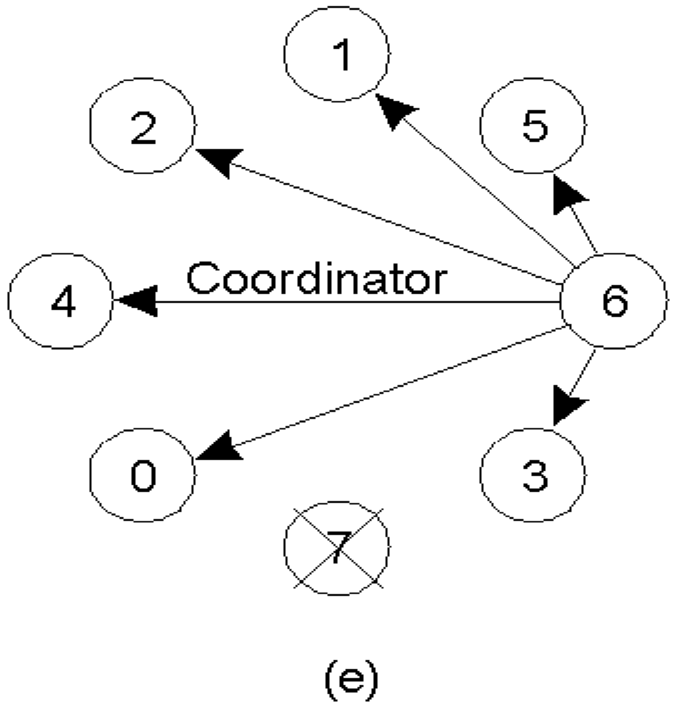
Election: The Bully Algorithm (I)



- The bully election algorithm
- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

Election: The Bully Algorithm (II)



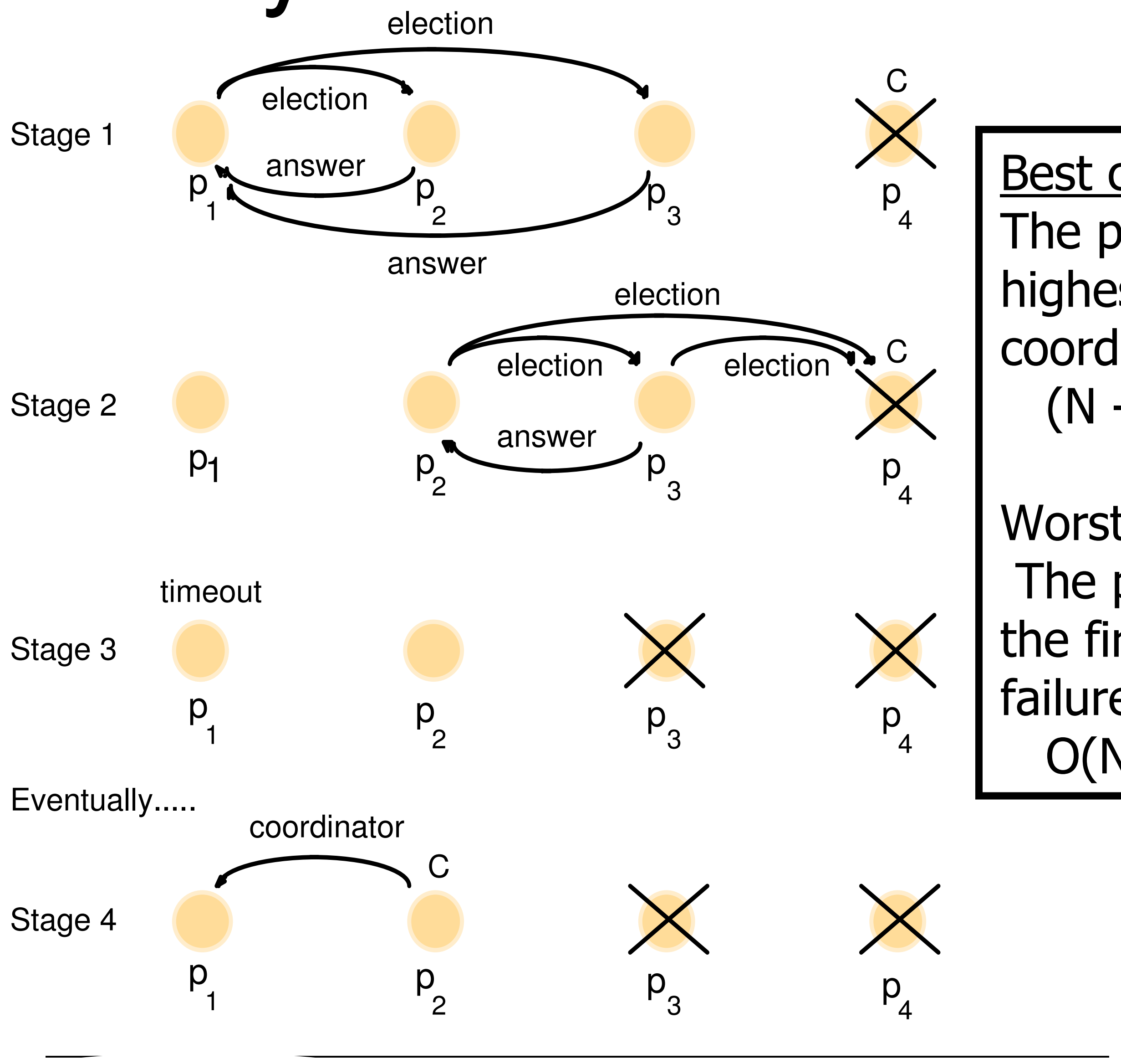


d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone

If process 7 is ever restarted, it will just send a COORDINATOR msg …

“Bully” election of coordinator



<u>Best case:</u>

The process with the 2nd highest ID is the first to detect coordinator’s failure:

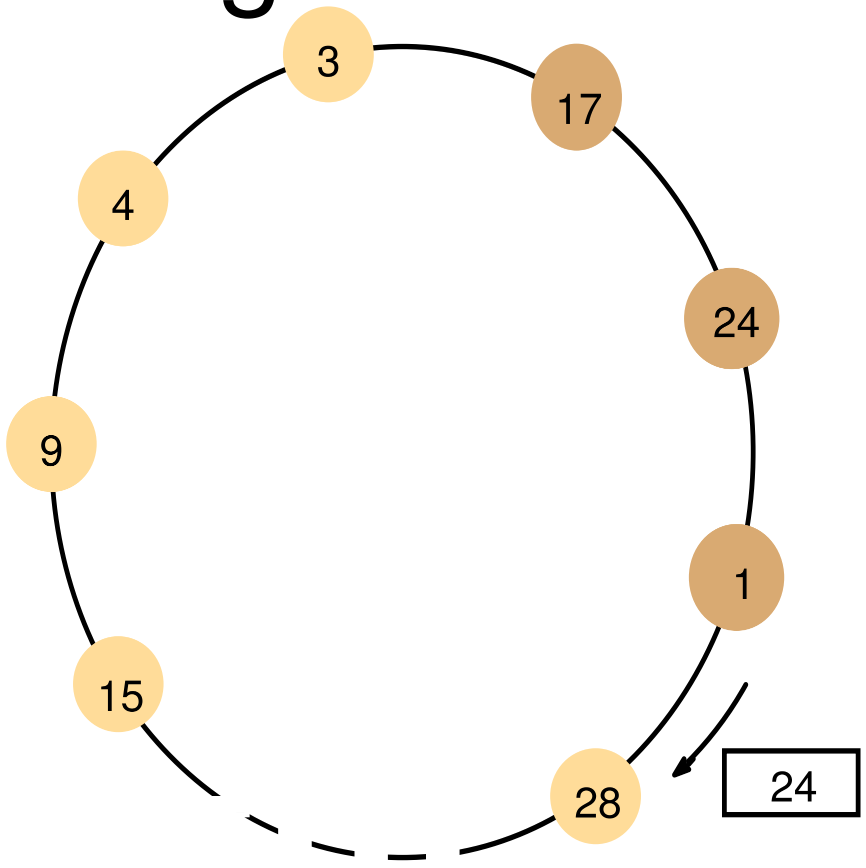
(N -2) msgs

Worst case:

The process with lowest ID is the first to detect coordinator’s failure:

O(N^2) msgs, in (N-1) elections

Ring-based election



Chang & Roberts, 1979

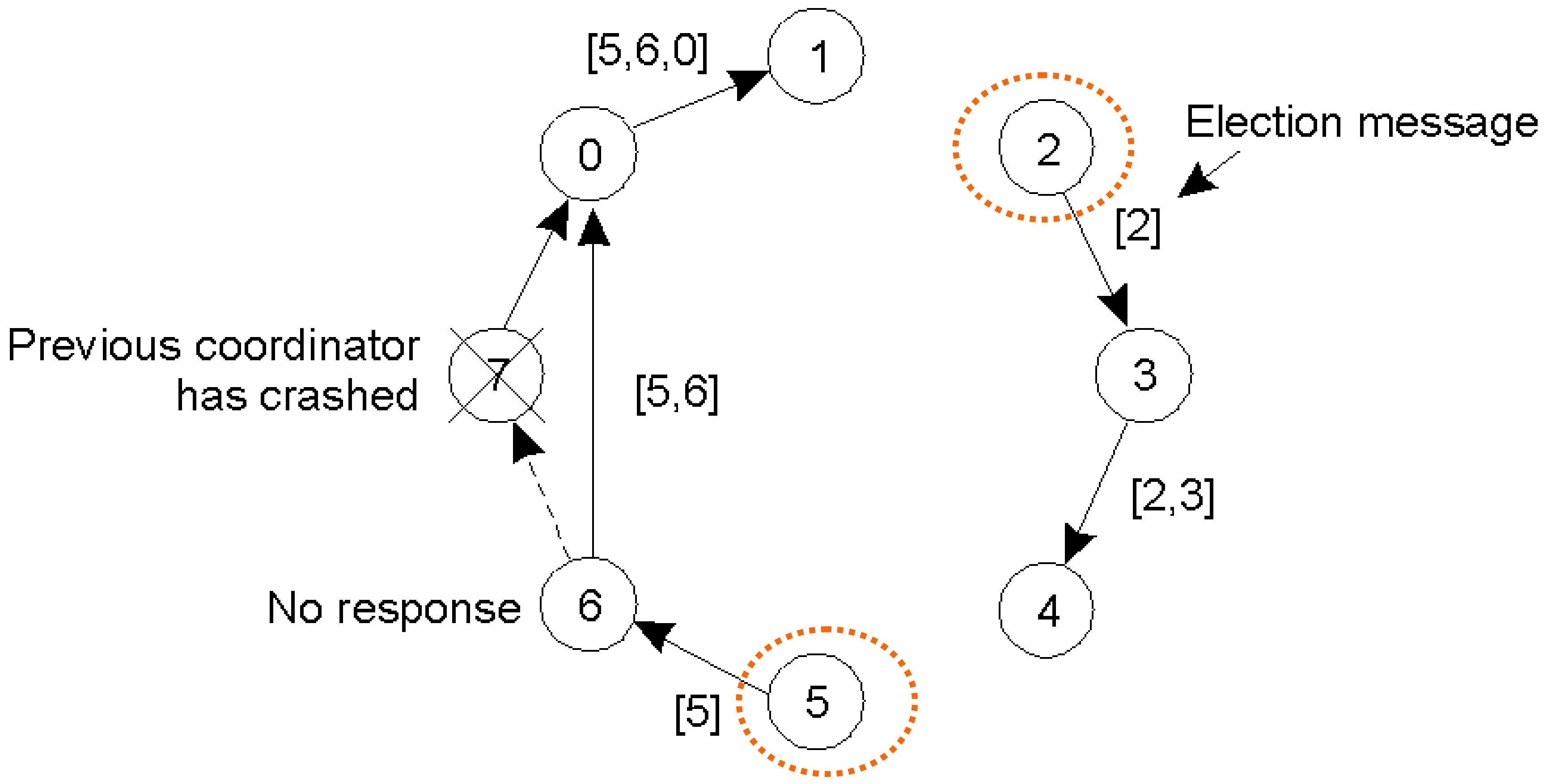
# msgs: (N -1) + N + N

Any process can begin an election, by marking itself as "participant" and then sending an "election" msg to its neighbor.

```
Upon receipt of an election msg:
   if (arrived ID < receiver's ID and the
         receiver is not a participant) {
      Receiver is marked as a participant;
      Substitute ID in msg & forward it;
   } else if(receiver's ID != arrived ID) {
      if(receiver is not a participant) {
         Receiver is marked as a participant;
         Forward msg;
      }
   } else {
      Receiver becomes coordinator;
      Send "elected" msg;
   }
```

Election: A Ring Algorithm

- Election algorithm using a ring.



Election: A Ring Algorithm

- If a single process starts an election, then the worst performance case is when its ani-clockwise neighbour has the highest identifier. A total of N-1 messages is then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further N messages.
- The elected message is then sent N times, making 3N-1 messages in all.
- So the total cost is 3N-1 communication steps, since these messages are sent sequentially.

$\boxed{\#\ \text{msgs: } (N - 1) + N + N}$

Mutual Exclusion

- Distributed processes often need to coordinate their activities .
- If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources. This is Critical Section problem, familiar in the domain of operating systems.
- But in case of distributed systems the requirement is, a solution to distributed mutual exclusion: one that is distributed and is based solely on message passing.

Mutual exclusion

- Critical sections
  - no shared variables
  - no support by single local kernel
- Examples:
  - NFS lockd server
  - peer access to a shared resource
    - Ethernet & IEEE 801.11 wireless networks
      - Only one node can (coherently) transmit at any time
  - update of shared state
- Primitives:
  - enter(), resourceAccesses(), exit()
- Safety & liveness requirements
- Desirable: ordering consistent with the relation $\rightarrow$

Safety, Liveness, Fairness

- *Safety*: At most one process may execute in the critical section (CS) at a time.
- *Liveness*: Request to enter and exit the critical section eventually succeed.
- *Fairness*: Sometimes it is not possible to order entry to the critical section by the times that the processes requested it, because of the absence of global clock, but a useful fairness requirement is to make use of the happened-before ordering between messages, that request entry to the critical section:($\rightarrow$ Ordering): if one request to enter the CS happened-before another, then entry to the CS is granted in that order.

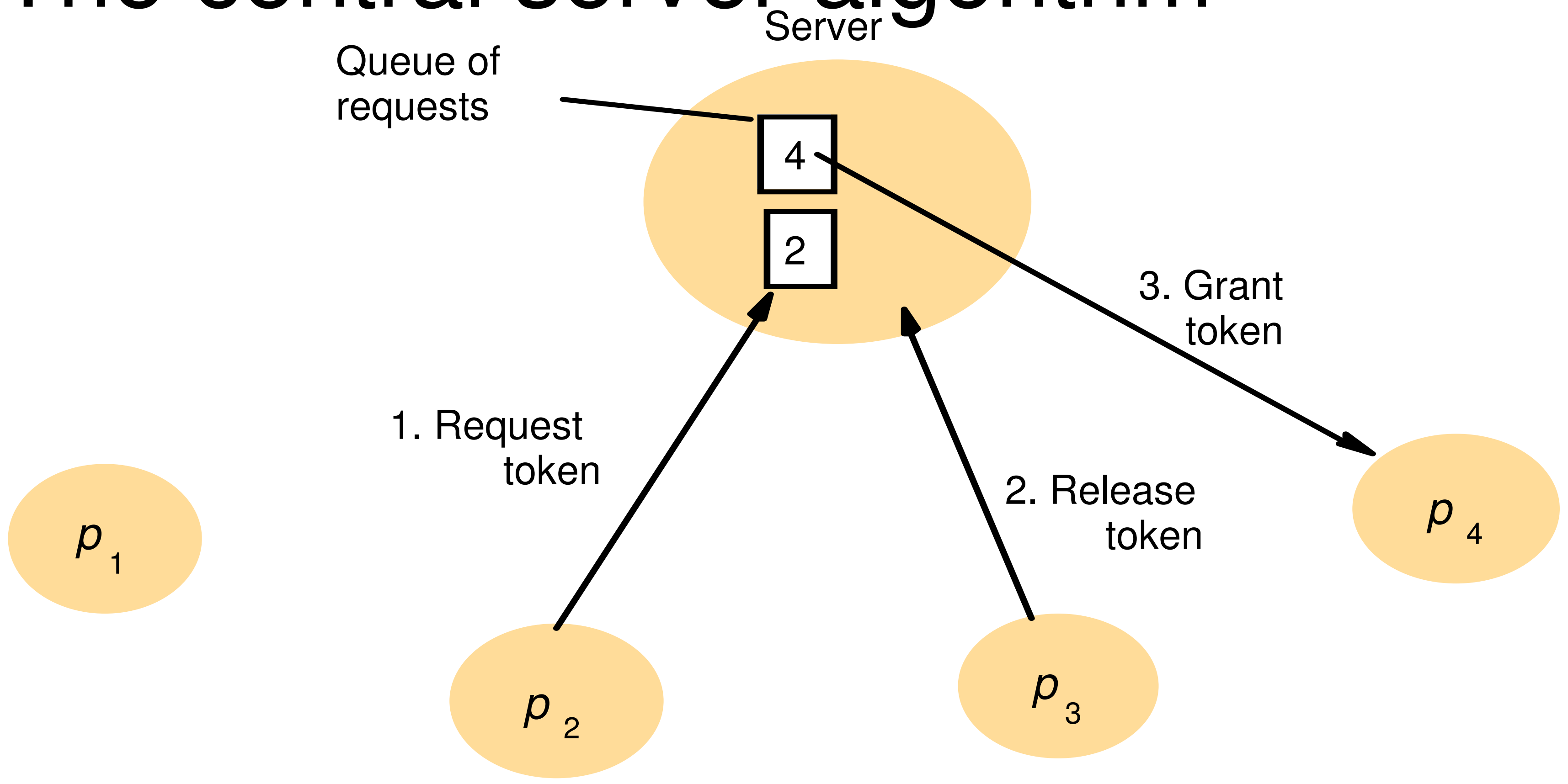
→ Ordering (Happened-before ordering)

- If solution grants entry to the critical section in happened before order, and if all requests are related by happened before, then it is not possible for a process to enter the critical section more than once while another process waits to enter.
- → ordering also allows processes to coordinate their access to the critical section.

Performance Evaluation

- *Bandwidth*: which is proportional to the number of messages sent in each entry and exit operation.
- *Client Delay*: Incurred by a process at each entry and exit operation.
- *Throughput*: This is the rate at which the collection of processes as a whole can access the critical section, given that some communication is necessary between successive processes. We measure the effect using the synchronization delay between one process exiting the CS and the next process entering it, the throughput is greater when the synchronization delay is short.

The central server algorithm



2 messages for enter(), even for a free resource

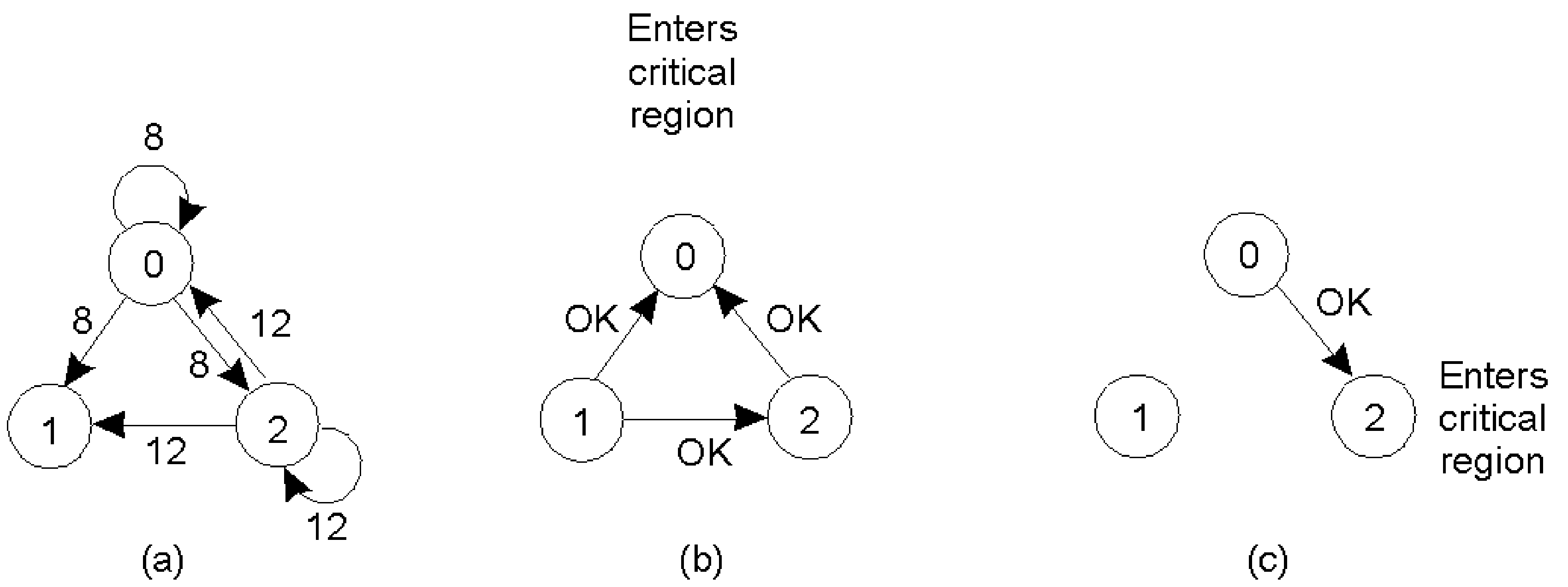
Does not preserve the relation →

Sync. Delay = RTT

Ricart & Agrawala (1981)

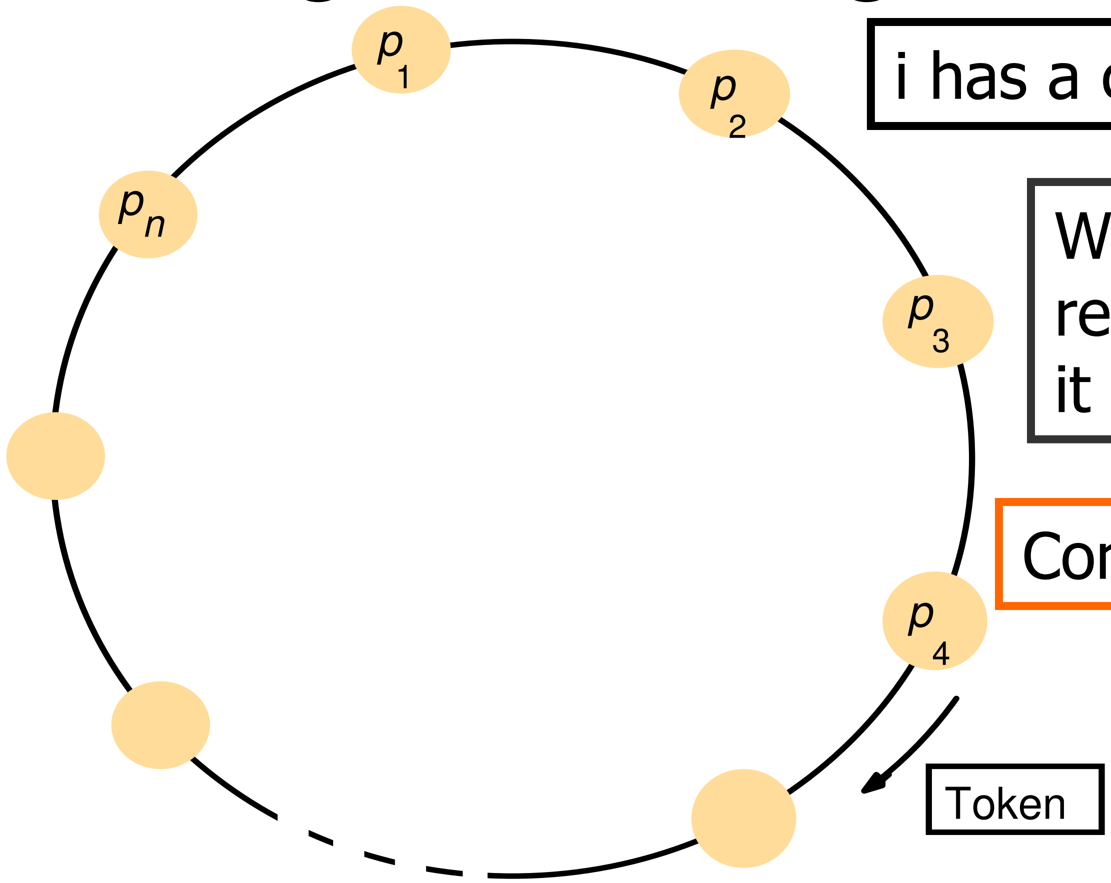
```
enter():
   state := WANTED'
   Multicast request to all peers;
   T := request's timestamp;
   Wait until (N - 1) responses are received;
   state := HELD;

On receipt of a request <T(i),  P(i)> at P(j), j ≠ i:
   if(state == HELD or (state == WANTED and
         (T, p(j)) < (T(i), p(i)) )   {
      Queue request without replying;
   } else {
      Reply to P(i);
   }

release():
   state := RELEASED;
   Respond to queued requests;
```

Mutual Exclusion: A Distributed Algorithm



a) Two processes want to enter the same critical region at the same moment.
b) Process 0 has the lowest timestamp, so it wins.
c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

A ring-based algorithm



i has a channel to (i +1) mod N

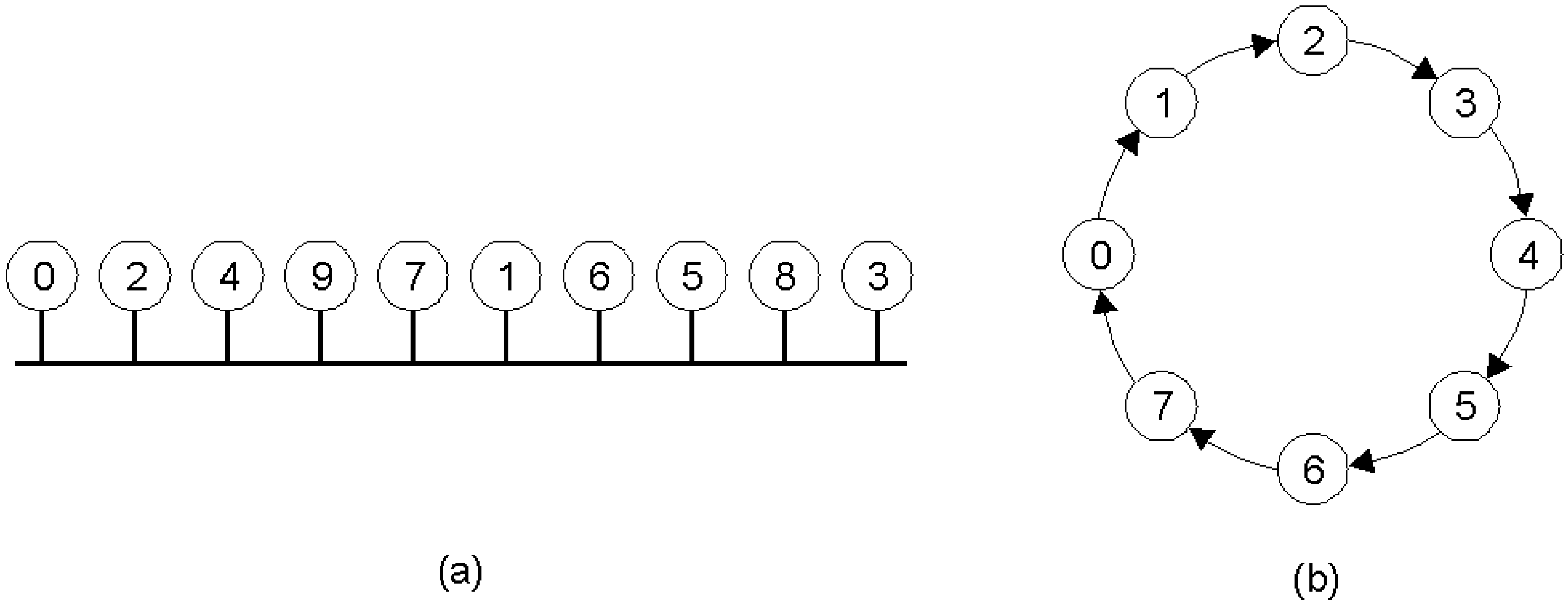
Wait for token to pass, retain it to enter(), release it to neighbor when done

Continuously consumes B/W

Sync. Delay: 1-N msg's

Does not preserve the relation →

Mutual Exclusion: A Token Ring Algorithm



a) An unordered group of processes on a network.

b) A logical ring constructed in software.

Comparison of Mutual Exclusion Algorithms

Algorithm	**Messages per entry/exit**	**Delay before entry (in message times)**	**Problems**
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Maekawa's voting algorithm (1985) (I)

Not necessary for all peers to grant access:
  Processes need only obtain permission from overlapping subsets of peers.

Voting sets { V(i) | i = 1, …, N }
- P(i) is in V(i)
- At least one common member for any 2 sets
- Sets of equal size, K
- Each process is in M of the sets

Optimal solution:
- K ~ $\sqrt{N}$
- M = K

A sample arrangement:
- $\sqrt{N}$ x $\sqrt{N}$ matrix
- V(i) := union of row & column containing P(i)

Maekawa’s voting algorithm (1985) (II)

```
enter():
   state := WANTED;
   Multicast request to processes in V(i) - { P(i) };
   Wait until (K - 1) responses are received;
   state := HELD;

On receipt of a request from P(I) at P(j), i ≠ j:
   if(state == HELD or voted == TRUE) {
      Queue request without responding;
   } else {
      Reply to P(i);
      voted := TRUE;
   }
```

2 $\sqrt{N}$ messages per entry

Maekawa's voting algorithm (1985) (III)

```
release():
   state :=RELEASED;
   Multicast release to processes in V(i) - { P(i) };

On receipt of a release msg from P(I) at P(j), i ≠ j:
   if(request queue == EMPTY) {
      voted := FALSE;
   } else {
      Remove head of queue, P(k);
      Reply to process P(k);
      voted := TRUE;
   }
```

At most 1 vote bet. successive receipts of a release msg

$\sqrt{N}$ messages per entry

IMPORTANT

- Make sure that you can describe the basic idea of all the distributed algorithms discussed in this section and you can describe the algorithm in detail.
- You should be able to write and explain the pseudo code of the algorithms (if given in the notes) discussed in this section
- It will be a good idea to have a look at the text books of the course to fully understand these algorithms

References

- L. Lamport, “Time, clocks, and the ordering of events in a distributed system”, CACM, 21(7), pp. 558-565, 1978.
- K. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems”, ACM Trans. Computer Systems, 3(1), pp. 63-75, 1985.
- D. Mills, “Improved Algorithms for Synchronizing Computer Network Clocks”, IEEE Trans. Networks, pp. 245-254, 1995.
- F. Mattern, “Virtual time and global states in distributed systems”, Proc. Workshop on Parallel and Distributed Algorithms, pp. 215-226, edited by M. Cosnard et al, North-Holland, 1989.
- O. Babaoglu and K. Marzullo, “Consistent global states of distributed systems: Fundamental concepts and Mechanisms”, in: “Distributed Systems (2nd Edition)”, edited by S. Mullender, ACM Press, 1993.

Notes !!!

Notes !!!