DISTRIBUTED SYSTEMS

UNIT I

Characterization of Distributed Systems: Introduction, Examples of Distributed systems, Resource sharing and web, challenges.
System Models: Introduction, Architectural and Fundamental models.

Examples of Distributed Systems–Trends in Distributed Systems – Focus on resource sharing – Challenges. **Case study:** World Wide Web.

Introduction

A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal.

Distributed systems Principles

A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

Centralised System Characteristics

- One component with non-autonomous parts
- Component shared by users all the time
- All resources accessible
- Software runs in a single process
- Single Point of control
- Single Point of failure

Distributed System Characteristics

- Multiple autonomous components
- Components are not shared by all users
- Resources may not be accessible
- Software runs in concurrent processes on different processors
- Multiple Points of control
- Multiple Points of failure

Examples of distributed systems and applications of distributed computing include the following:

- telecommunication networks:
- telephone networks and cellular networks,
- computer networks such as the Internet,
- wireless sensor networks,
- routing algorithms;

- network applications:
- World wide web and peer-to-peer networks,
- massively multiplayer online games and virtual reality communities,
- distributed databases and distributed database management systems,
- network file systems,
- distributed information processing systems such as banking systems and airline reservation systems;

- real-time process control:
  - aircraft control systems,
  - industrial control systems;
- parallel computation:
  - scientific computing, including cluster computing and grid computing and various volunteer computing projects (see the list of distributed computing projects),
  - distributed rendering in computer graphics.

Common Characteristics

Certain common characteristics can be used to assess distributed systems

- Resource Sharing
- Openness
- Concurrency
- Scalability
- Fault Tolerance
- Transparency

Resource Sharing

- Ability to use any hardware, software or data anywhere in the system.
- Resource manager controls access, provides naming scheme and controls concurrency.
- Resource sharing model (e.g. client/server or object-based) describing how
  - resources are provided,
  - they are used and
  - provider and user interact with each other.

Openness

- Openness is concerned with extensions and improvements of distributed systems.
- Detailed interfaces of components need to be published.
- New components have to be integrated with existing components.
- Differences in data representation of interface types on different processors (of different vendors) have to be resolved.

Concurrency

Components in distributed systems are executed in concurrent processes.

- Components access and update shared resources (e.g. variables, databases, device drivers).
- Integrity of the system may be violated if concurrent updates are not coordinated.
  - Lost updates
  - Inconsistent analysis

Scalability

- Adaption of distributed systems to
  - accomodate more users
  - respond faster (this is the hard one)
- Usually done by adding more and/or faster processors.
- Components should not need to be changed when scale of a system increases.
- Design components to be scalable

Fault Tolerance

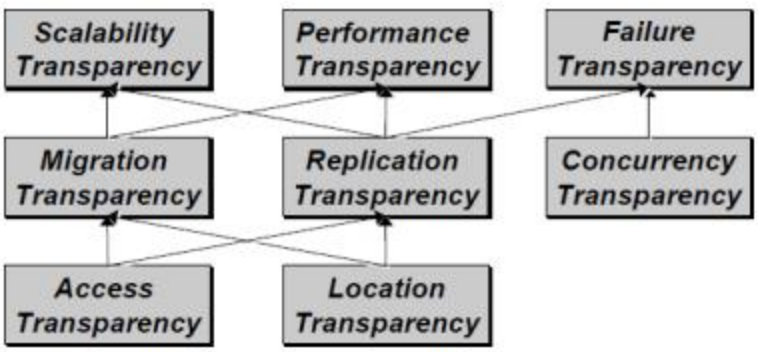
Hardware, software and networks fail!

- Distributed systems must maintain availability even at low levels of hardware/software/network reliability.
- Fault tolerance is achieved by
  - recovery
  - redundancy

Transparency

Distributed systems should be perceived by users and application programmers as a whole rather than as a collection of cooperating components.

- Transparency has different dimensions that were identified by ANSA.
- These represent various properties that distributed systems should have.



Access Transparency

Enables local and remote information objects to be accessed using identical operations.

- Example: File system operations in NFS.
- Example: Navigation in the Web.
- Example: SQL Queries

Location Transparency
Enables information objects to be accessed without knowledge of their location.

- Example: File system operations in NFS
- Example: Pages in the Web
- Example: Tables in distributed databases

Concurrency Transparency

Enables several processes to operate concurrently using shared information objects without interference between them.

- Example: NFS
- Example: Automatic teller machine network
- Example: Database management system

Replication Transparency

Enables multiple instances of information objects to be used to increase reliability and performance without knowledge of the replicas by users or application programs

- Example: Distributed DBMS
- Example: Mirroring Web Pages.

Failure Transparency

- Enables the concealment of faults
- Allows users and applications to complete their tasks despite the failure of other components.
- Example: Database Management System

Migration Transparency

Allows the movement of information objects within a system without affecting the operations of users or application programs

- Example: NFS
- Example: Web Pages

Performance Transparency
Allows the system to be reconfigured to improve performance as loads vary.

- Example: Distributed make.

Scaling Transparency

Allows the system and applications to expand in scale without change to the system structure or the application algortithms.

- Example: World-Wide-Web
- Example: Distributed Database

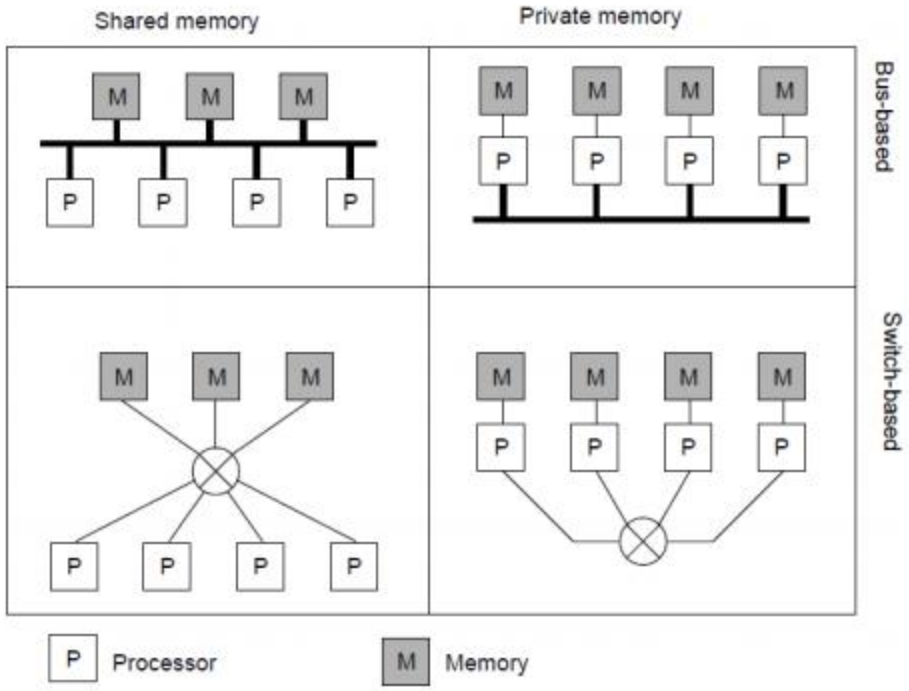
Distributed Systems: Hardware Concepts

- Multiprocessors
- Multicomputers

Networks of Computers

Multiprocessors and Multicomputers
Distinguishing features:

- Private versus shared memory
- Bus versus switched interconnection



Networks of Computers

High degree of node heterogeneity:

- High-performance parallel systems (multiprocessors as well as multicomputers)
- High-end PCs and workstations (servers)
- Simple network computers (offer users only network access)
- Mobile computers (palmtops, laptops)
- Multimedia workstations

High degree of network heterogeneity:

- Local-area gigabit networks
- Wireless connections
- Long-haul, high-latency connections
- Wide-area switched megabit connections

Distributed Systems: Software Concepts

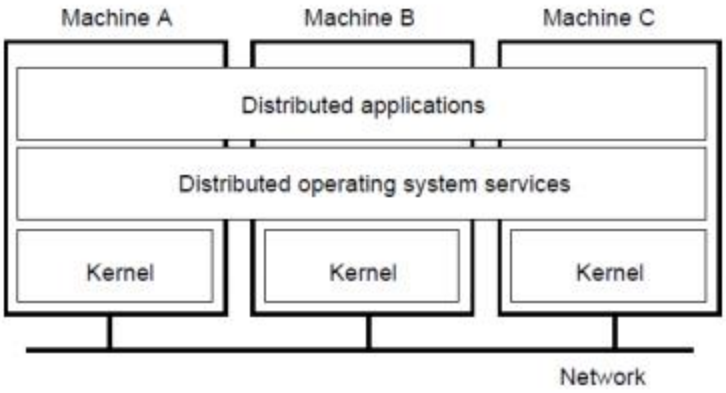
Distributed operating system
_ Network operating system
_ Middleware

System	Description	Main goal
DOS	Tightly-coupled OS for multiprocessors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled OS for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middle-ware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

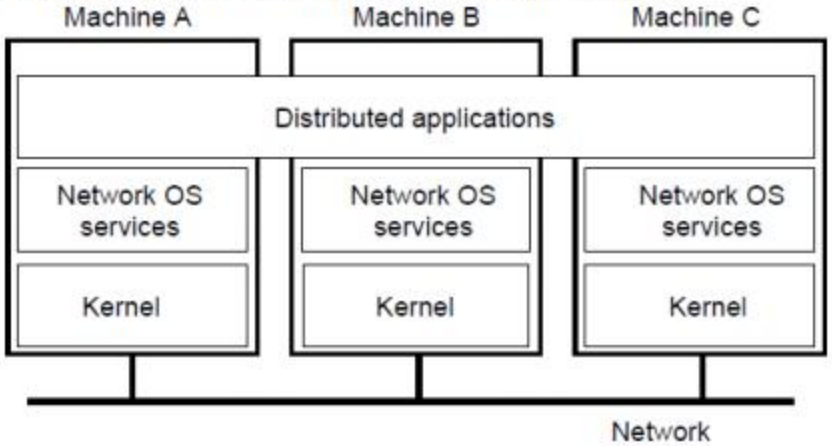
Distributed Operating System

Some characteristics:

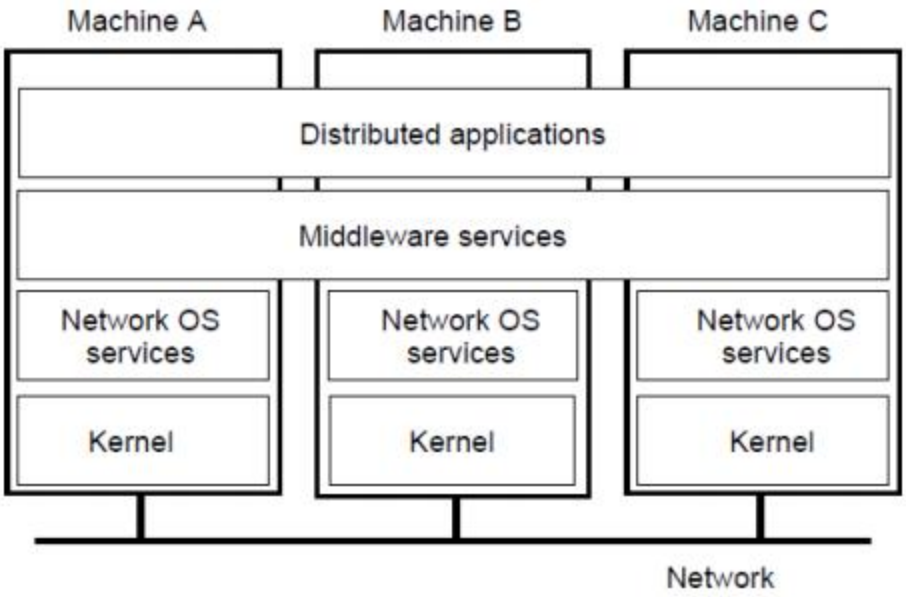
_ OS on each computer knows about the other computers
_ OS on different computers generally the same
_ Services are generally (transparently) distributed across computers



Network Operating System
Some characteristics:
_ Each computer has its own operating system with networking facilities
_ Computers work independently (i.e., they may even have different operating systems)
_ Services are tied to individual nodes (ftp, telnet, WWW)
_ Highly file oriented (basically, processors share *only* files)



Distributed System (Middleware)
Some characteristics:
_ OS on each computer need not know about the other computers
_ OS on different computers need not generally be the same
_ Services are generally (transparently) distributed across computers



Need for Middleware

Motivation: Too many networked applications were hard or difficult to integrate:

_ Departments are running different NOSs
_ Integration and interoperability only at level of primitive NOS services
_ Need for federated information systems:
– Combining different databases, but providing a single view to applications
– Setting up enterprise-wide Internet services, making use of existing information systems
– Allow transactions across different databases
– Allow extensibility for future services (e.g., mobility, teleworking, collaborative applications)
_ Constraint: use the existing operating systems, and treat them as the underlying environment (they provided the basic functionality anyway)

Communication services: Abandon primitive socket based message passing in favor of:

_ Procedure calls across networks
_ Remote-object method invocation
_ Message-queuing systems
_ Advanced communication streams
_ Event notification service

Information system services: Services that help manage data in a distributed system:

_ Large-scale, system wide naming services
_ Advanced directory services (search engines)
_ Location services for tracking mobile objects
_ Persistent storage facilities
_ Data caching and replication

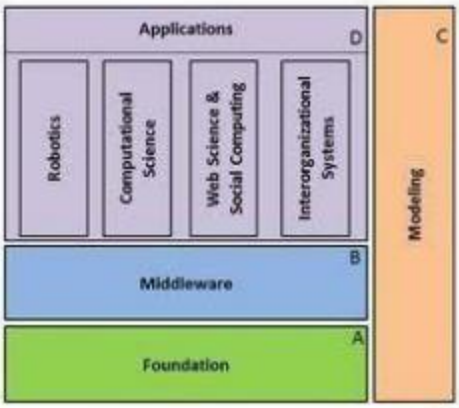
Control services: Services giving applications control over when, where, and how they access data:

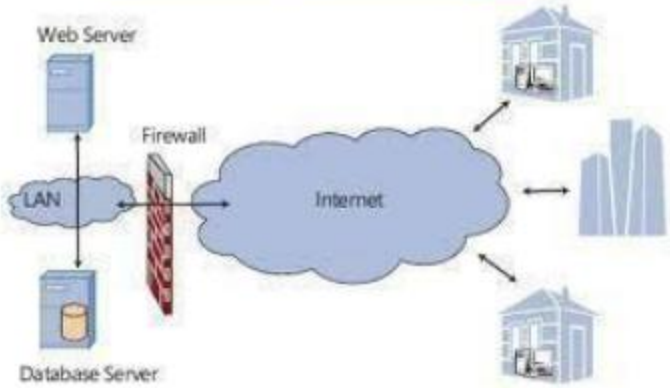
_ Distributed transaction processing
_ Code migration

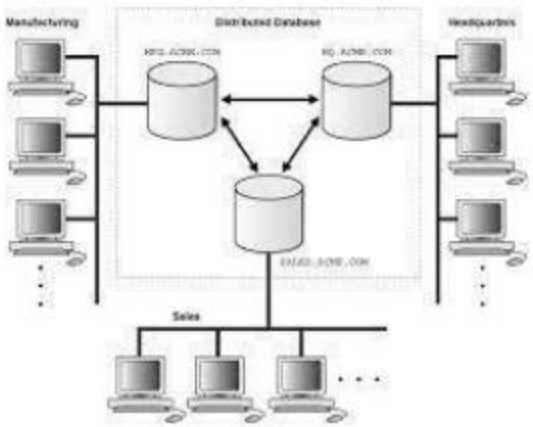
Security services: Services for secure processing and communication:
_ Authentication and authorization services
_ Simple encryption services
_ Auditing service

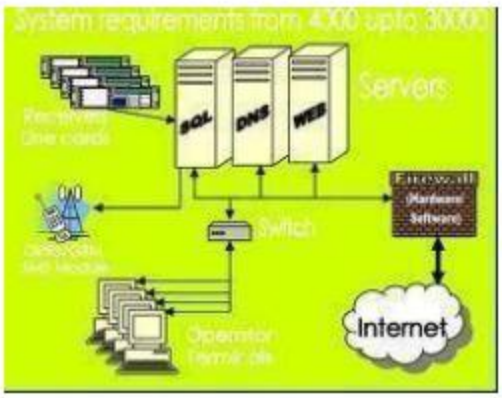
Comparison of DOS, NOS, and Middleware

Item	Distributed OS		Network OS	Middle-ware DS
	multiproc.	multicomp.		
1	Very High	High	Low	High
2	Yes	Yes	No	No
3	1	N	N	N
4	Shared memory	Messages	Files	Model specific
5	Global, central	Global, distributed	Per node	Per node
6	No	Moderately	Yes	Varies
7	Closed	Closed	Open	Open


Applications
D
C
Robotics
Computational Science
Web Science & Social Computing
Interorganizational Systems
Modeling
B
Middleware
A
Foundation


Web Server
Firewall
LAN
Internet
Database Server


Manufacturing
Distributed Database
Headquarters
Sales


Servers
SQL
DNS
WEB
Switch
Firewall
(Hardware/
Software)
Internet

Networks of computers are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks – all of these, both separately and in combination, share the essential characteristics that make them relevant subjects for study under the heading *distributed systems*.

Distributed systems has the following significant consequences:

Concurrency: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network. We will describe ways in which this extra capacity can be usefully deployed at many points in this book. The coordination of concurrently executing programs that share resources is also an important and recurring topic.

No global clock: When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network.

Independent failures: All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact, the programs
on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a *crash*), is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running.

TRENDS IN DISTRIBUTED SYSTEMS

Distributed systems are undergoing a period of significant change and this can be traced back to

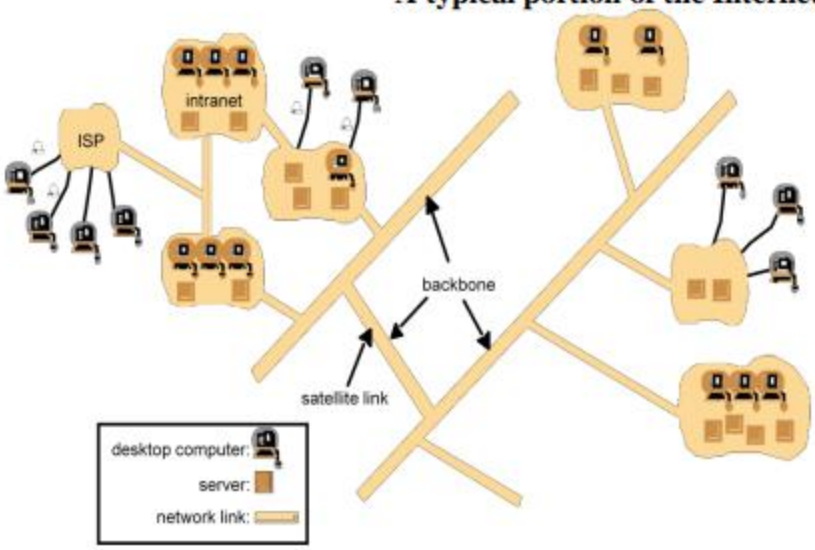
a number of influential trends:

- the emergence of pervasive networking technology;
- the emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems;
- the increasing demand for multimedia services;
- the view of distributed systems as a utility.

Internet

The modern Internet is a vast interconnected collection of computer networks of many different types, with the range of types increasing all the time and now including, for example, a wide range of wireless communication technologies such as WiFi, WiMAX, Bluetooth and third-generation mobile phone networks. The net result is that networking has become a pervasive resource and devices can be connected (if desired) at any time and in any place.

A typical portion of the Internet



The Internet is also a very large distributed system. It enables users, wherever they are, to make use of services such as the World Wide Web, email and file transfer. (Indeed, the Web is sometimes incorrectly equated with the Internet.) The set of services is open-ended – it can be extended by the addition of server computers and new types of service. The figure shows a collection of intranets – subnetworks operated by companies and other organizations and typically protected by firewalls. The role of a *firewall* is to protect an intranet by preventing unauthorized messages from leaving or entering. A firewall is implemented by filtering incoming and outgoing messages. Filtering might be done by source or destination, or a firewall might allow only those messages related to email and web access to pass into or out of the intranet that it protects. Internet Service Providers (ISPs) are companies that provide broadband links and other types of connection to individual users and small organizations, enabling them to access services anywhere in the Internet as well as providing local services such as email and web hosting. The intranets are linked together by backbones. A *backbone* is a network link with a high transmission capacity, employing satellite connections, fibre optic cables and other high-bandwidth circuits

Date	*Computers*	*Web servers*
1979, Dec.	188	0
1989, July	130,000	0
1999, July	56,218,000	5,560,866
2003, Jan.	171,638,297	35,424,956

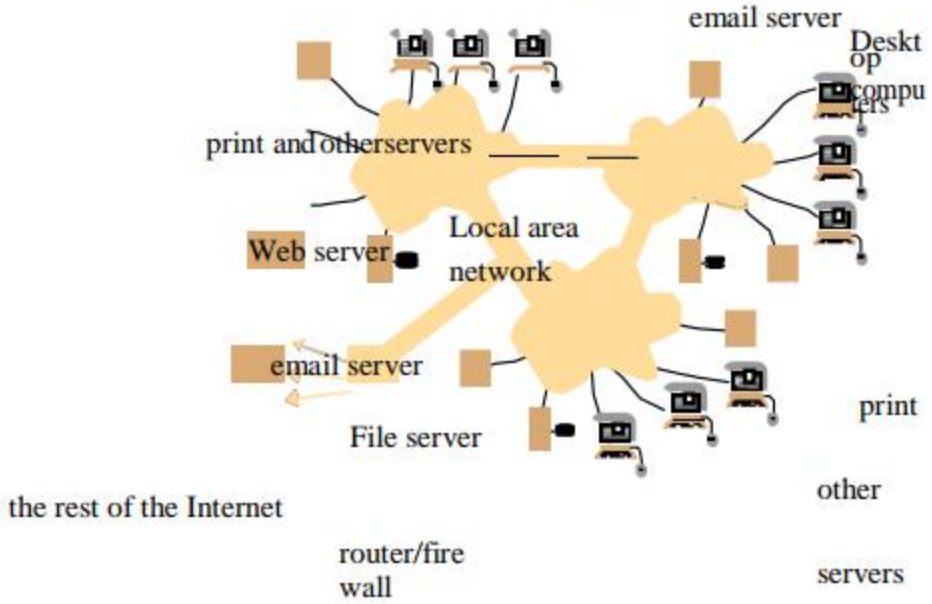
Computers vs. Web servers in the Internet

Date	*Computers*	*Web servers*	*Percentage*
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
		42,298,371	

Intranet

- A portion of the Internet that is separately administered and has a boundary that can be configured to enforce local security policies
- Composed of several LANs linked by backbone connections
- Be connected to the Internet via a router

A typical intranet



Main issues in the design of components for the use in intranet

- File services
- Firewall
- The cost of software installation and support

Mobile and ubiquitous computing

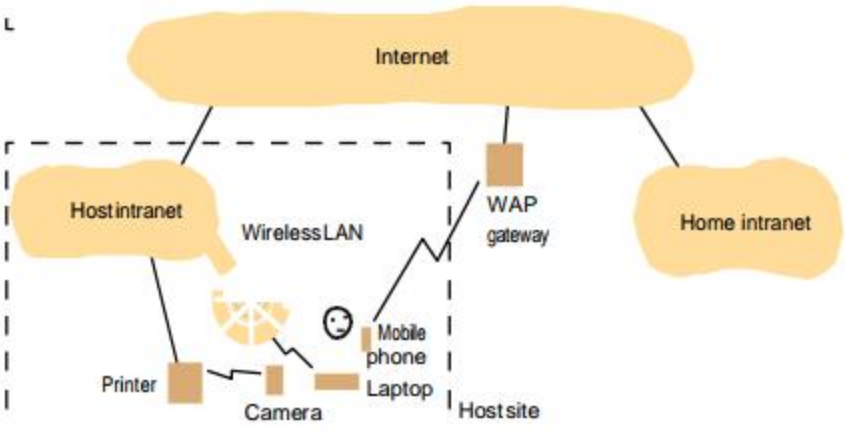
Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

- Laptop computers.
- Handheld devices, including mobile phones, smart phones, GPS-enabled devices, pagers, personal digital assistants (PDAs), video cameras and digital cameras.
- Wearable devices, such as smart watches with functionality similar to a PDA.
- Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes *mobile computing* possible. Mobile computing is the

performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their 'home' intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them. They can continue to access the Internet; they can continue to access resources in their home intranet; and there is increasing provision for users to utilize resources such as printers or even sales points that are conveniently nearby as they move around. The latter is also known as *location-aware* or *context-aware computing*. Mobility introduces a number of challenges for distributed systems, including the need to deal with variable connectivity and indeed disconnection, and the need to maintain operation in the face of device mobility.

Portable and handheld devices in a distributed system



Ubiquitous computing is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and even natural settings. The term 'ubiquitous' is intended to suggest that small computing devices will eventually

become so pervasive in everyday objects that they are scarcely noticed. That is, their computational behaviour will be transparently and intimately tied up with their physicalfunction.

The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it may be convenient for users to control their washing machine or their entertainment system from their phone or a 'universal remote control' device in the home. Equally, the washing machine could notify the user via a smart badge or phone when the washing is done.

Ubiquitous and mobile computing overlap, since the mobile user can in principle benefit from computers that are everywhere. But they are distinct, in general. Ubiquitous computing could benefit users while they remain in a single environment such as the home or a hospital. Similarly, mobile computing has advantages even if it involves only conventional, discrete computers and devices such as laptops and printers.

RESOURCE SHARING

- Is the primary motivation of distributed computing
- Resources types
  - Hardware, e.g. printer, scanner, camera
  - Data, e.g. file, database, web page
  - More specific functionality, e.g. search engine, file
- *Service*
  - manage a collection of related resources and present their functionalities to users and applications
- *Server*
  - a process on networked computer that accepts requests from processes on other computers to perform a *service* and responds appropriately
- *Client*
  - the requesting process
- *Remote invocation*

A complete interaction between *client* and *server*, from the point when the *client* sends its request to when it receives the server's response

- Motivation of WWW
  - Documents sharing between physicists of CERN
  - Web is an open system: it can be extended and implemented in new ways without disturbing its existing functionality.
  - Its operation is based on communication standards and document standards
  - Respect to the types of 'resource' that can be published and shared on it.
- HyperText Markup Language
  - A language for specifying the contents and layout of pages
- Uniform Resource Locators
  - Identify documents and other resources
- A client-server architecture with HTTP
  - By with browsers and other clients fetch documents and other resources from web servers

HTML

HTML text is stored in a file of a web server.

```
<IMG SRC = http://www.cdk3.net/WebExample/Images/earth.jpg>
<P>
Welcome to Earth! Visitors may also be interested in taking a look at
the
<A HREF = "http://www.cdk3.net/WebExample/moon.html>Moon</A>.
<P>
(etcetera)
```

- A browser retrieves the contents of this file from a web server.

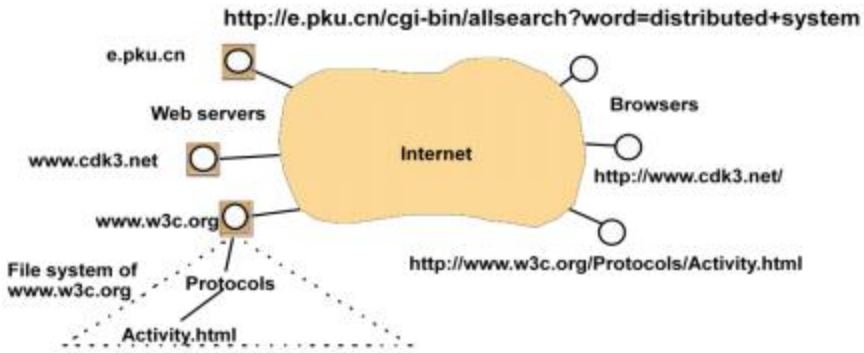
-The browser interprets the HTML text

-The server can infer the content type from the filename extension.

URL

- HTTP URLs are the most widely used
- An HTTP URL has two main jobs to do:
  - To identify which web server maintains the resource
  - To identify which of the resources at that server

Web servers and web browsers



HTTP URLs

- http://servername[:port]//pathNameOnServer][?arguments]
- e.g.

http://www.cdk3.net/
http://www.w3c.org/Protocols/Activity.html
http://e.pku.cn/cgi-bin/allsearch?word=distributed+system

Server DNS name	Pathname on server	Arguments
www.cdk3.net	(default)	(none)
www.w3c.org	Protocols/Activity.html	(none)
e.pku.cn	cgi-bin/allsearch	word=distributed+system

- Publish a resource remains unwieldy

HTTP

- Defines the ways in which browsers and any other types of client interact with web servers (RFC2616)
- Main features
  - Request-replay interaction
  - Content types. The strings that denote the type of content are called MIME (RFC2045,2046)
  - One resource per request. HTTP version 1.0
  - Simple access control

More features-services and dynamic pages

- Dynamic content
  - Common Gateway Interface: a program that web servers run to generate content for their clients
- Downloaded code
  - JavaScript
  - Applet

Discussion of Web

- Dangling: a resource is deleted or moved, but links to it may still remain
- Find information easily: e.g. Resource Description Framework which standardize the format of *metadata* about web resources
- Exchange information easily: e.g. XML – a *self describing* language
- Scalability: heavy load on popular web servers
- More applets or many images in pages increase in the download time

THE CHALLENGES IN DISTRIBUTED SYSTEM:

Heterogeneity

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- networks;
- computer hardware;
- operating systems;
- programming languages;
- implementations by different developers

Although the Internet consists of many different sorts of network, their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network.
Data types such as integers may be represented in different ways on different sorts of hardware – for example, there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware. Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another. Programs written by different developers cannot communicate with one another
unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols.

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. The Common Object Request Broker (CORBA), is an example. Some middleware, such as Java Remote Method Invocation (RMI), supports only a single programming language. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems and hardware.

Heterogeneity and mobile code • The term *mobile code* is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.
The *virtual machine* approach provides a way of making code executable on a variety of host computers: the compiler for a particular language generates code for a virtual machine instead of

a particular hardware order code. For example, the Java compiler produces code for a Java virtual machine, which executes it by interpretation.
The Java virtual machine needs to be implemented once for each type of computer to enable Java programs to run.
Today, the most commonly used form of mobile code is the inclusion Javascript programs in some web pages loaded into client browsers.

Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways. The openness of distributed systems 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. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures,
which are usually cumbersome and slow-moving. However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people. The designers of the Internet protocols introduced a series of documents called 'Requests For Comments', or RFCs, each of which is known by a number. The specifications of the Internet communication protocols were published in this series in the early 1980s, followed by specifications for applications that run over them, such as file transfer, email and telnet by the mid-1980s.

Systems that are designed to support resource sharing in this way are termed *open distributed systems* to emphasize the fact that they are extensible. They may be extended at the hardware level by the addition of computers to the network and at the software level by the introduction of new services and the reimplementation of old ones, enabling
application programs to share resources.

To summarize:
• 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. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

Security

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components: confidentiality (protection against disclosure to unauthorized individuals), integrity

(protection against alteration or corruption), and availability (protection against interference with the means to access the resources).

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network. For example:
1. A doctor might request access to hospital patient data or send additions to that data.
2. In electronic commerce and banking, users send their credit card numbers across the Internet.

In both examples, the challenge is to send sensitive information in a message over a network in a secure manner. But security is not just a matter of concealing the contents of messages – it also involves knowing for sure the identity of the user or other agent on whose behalf a message was sent.

However, the following two security challenges have not yet been fully met:

Denial of service attacks: Another security problem is that a user may wish to disrupt a service for some reason. This can be achieved by bombarding the service with such a large number of pointless requests that the serious users are unable to use it. This is called a *denial of service* attack. There have been several denial of service attacks on well-known web services. Currently such attacks are countered by attempting to catch and punish the perpetrators after the event, but that is not a general solution to the problem.

Security of mobile code: Mobile code needs to be handled with care. Consider someone who receives an executable program as an electronic mail attachment: the possible effects of running the program are unpredictable; for example, it may seem to display an interesting picture but in reality it may access local resources, or perhaps be part of a denial of service attack.

Scalability

Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The number of computers and servers in the Internet has increased dramatically. Figure 1.6 shows the increasing number of computers and web servers during the 12-year history of the Web up to 2005 [zakon.org]. It is interesting to note the significant growth in both computers and web servers in this period, but also that the relative percentage is flattening out – a trend that is explained by the growth of fixed and mobile personal computing. One web server may also increasingly be hosted on multiple computers.

The design of scalable distributed systems presents the following challenges:

Controlling the cost of physical resources: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it. For example, the frequency with which files are accessed in an intranet is likely to grow as the number of users and computers increases. It must be possible to add server computers to avoid the performance bottleneck that would arise if a single file server had to handle all file access requests. In general, for a system with n users to be scalable, the quantity of physical resources required to support them should be

at most O(n) – that is, proportional to n. For example, if a single file server can support 20 users, then two such servers should be able to support 40 users.

Controlling the performance loss: Consider the management of a set of data whose size is proportional to the number of users or resources in the system – for example, the table with the correspondence between the domain names of computers and their Internet addresses held by the Domain Name System, which is used mainly to look
up DNS names such as www.amazon.com. Algorithms that use hierarchic structures scale better than those that use linear structures. But even with hierarchic structures an increase in size will result in some loss in performance: the time taken to access hierarchically structured data is O($log\ n$), where n is the size of the set of data. For a
system to be scalable, the maximum performance loss should be no worse than this.

Preventing software resources running out: An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet). In the late 1970s, it was decided to use 32 bits for this purpose, but as will be explained in Chapter 3, the supply of available Internet addresses is running out. For this reason, a new version of the protocol with 128-bit Internet addresses is being adopted, and this will require modifications to many software components.

Growth of the Internet (computers and web servers)

Date	*Computers*	*Web servers*	*Percentage*
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
2003, July	~200,000,000	42,298,371	21
2005, July	353,284,187	67,571,581	19

Avoiding performance bottlenecks: In general, algorithms should be decentralized to avoid having performance bottlenecks. We illustrate this point with reference to the predecessor of the Domain Name System, in which the name table was kept in a single master file that could be downloaded to any computers that needed it. That was
fine when there were only a few hundred computers in the Internet, but it soon became a serious performance and administrative bottleneck.

Failure handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. Failures in a distributed system are partial – that is, some components fail while others continue to function. Therefore the handling of failures is particularly difficult.

Detecting failures: Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file. It is difficult or even impossible to detect some other
failures, such as a remote crashed server in the Internet. 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. Two examples of hiding failures:
1. Messages can be retransmitted when they fail to arrive.
2. File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

Tolerating failures: Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components. Their clients can be designed to tolerate failures, which generally involves the users tolerating them as well. For example, when a web browser cannot contact a web server, it does not make the user wait for ever while it keeps on trying – it informs the user about the problem, leaving them free to try again later. Services that tolerate failures are discussed in the paragraph on redundancy below.
Recovery from failures: Recovery involves the design of software so that the state of 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 (files and other material stored
in permanent storage) may not be in a consistent state.

Redundancy: Services can be made to tolerate failures by the use of redundant components. Consider the following examples:
1. There should always be at least two different routes between any two routers in the Internet.
2. In the Domain Name System, every name table is replicated in at least two different servers.
3. A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server; the servers can be designed to detect faults in their peers; when a fault is detected in one server, clients are redirected to the remaining servers.

Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. The process that manages a shared resource could take one client request at a time. But that approach limits throughput. Therefore services and applications generally allow multiple client requests to be processed concurrently. To make this more concrete, suppose that each resource is encapsulated as an object and that invocations are executed in concurrent threads. In this case it is possible that several threads may be executing concurrently within an object, in which case their operations on the object may conflict with one another and produce inconsistent results.

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 as a collection of independent components. The implications of transparency are a major influence on the design of the system software.

Access transparency enables local and remote resources to be accessed using identical operations.
Location transparency enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.
Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.
Failure transparency enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.
Mobility transparency allows the movement of resources and clients within a system without affecting the operation of users or programs.
Performance transparency allows the system to be reconfigured to improve performance as loads vary.
Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

Quality of service

Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are *reliability*, *security* and *performance*.
Adaptability to meet changing system configurations and resource availability has been recognized as a further important aspect of service quality.

Some applications, including multimedia applications, handle *time-critical data* – streams of data that are required to be processed or transferred from one process to another at a fixed rate. For example, a movie service might consist of a client program that is retrieving a film from a video server and presenting it on the user's screen. For a satisfactory result the successive frames of video need to be displayed to the user within some specified time limits.
In fact, the abbreviation QoS has effectively been commandeered to refer to the ability of systems to meet such deadlines. Its achievement depends upon the availability of the necessary computing and network resources at the appropriate times. This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time (for example, the task of displaying a frame of video).

INTRODUCTION TO SYSTEM MODELS

Systems that are intended for use in real-world environments should be designed to function correctly in the widest possible range of circumstances and in the face of many possible difficulties and threats .

Each type of model is intended to provide an abstract, simplified but consistent description of a relevant aspect of distributed system design:
Physical models are the most explicit way in which to describe a system; they capture the hardware composition of a system in terms of the computers (and other devices, such as mobile phones) and their interconnecting networks.

Architectural models describe a system in terms of the computational and communication tasks performed by its computational elements; the computational elements being individual computers or aggregates of them supported by appropriate network interconnections.

Fundamental models take an abstract perspective in order to examine individual aspects of a distributed system. The fundamental models that examine three important aspects of distributed systems: *interaction models*, which consider the structure and sequencing of the communication between the elements of the system; *failure models*, which consider the ways in which a system may fail to operate correctly and; *security models*, which consider how the system is protected against attempts to interfere with its correct operation or to steal its data.

Architectural models

The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it. Major concerns are to make the system reliable, manageable, adaptable and cost-effective. The architectural design of a building has similar aspects – it determines not only its appearance but also its general structure and architectural style (gothic, neo-classical, modern) and provides a consistent frame of reference for the design.

Software layers

The concept of layering is a familiar one and is closely related to abstraction. In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details, or indeed of any other layers beneath them.

In terms of distributed systems, this equates to a vertical organization of services into service layers. A distributed service can be provided by one or more server processes, interacting with each other and with client processes in order to maintain a consistent system-wide view of the service's resources. For example, a network time service is implemented on the Internet based on the Network Time Protocol (NTP) by server processes running on hosts throughout the Internet that supply the current time to any client that requests it and adjust their version of the current time as a result of interactions with each other. Given the complexity of distributed systems, it is often helpful to organize such services into layers. the important terms *platform* and *middleware*, which define as follows:

The important terms *platform* and *middleware*, which is defined as follows:

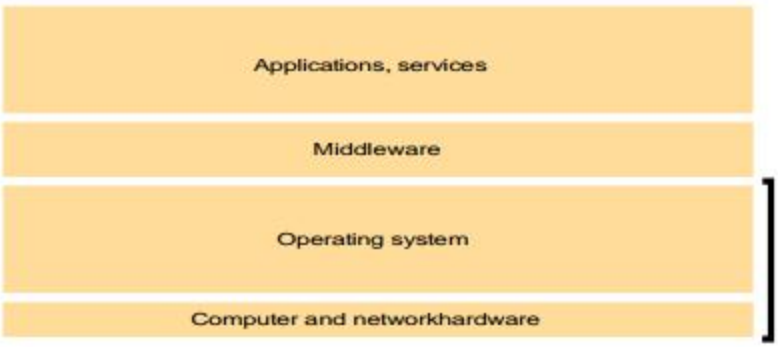
A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates communication and coordination between processes. Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM/Symbian are major examples.

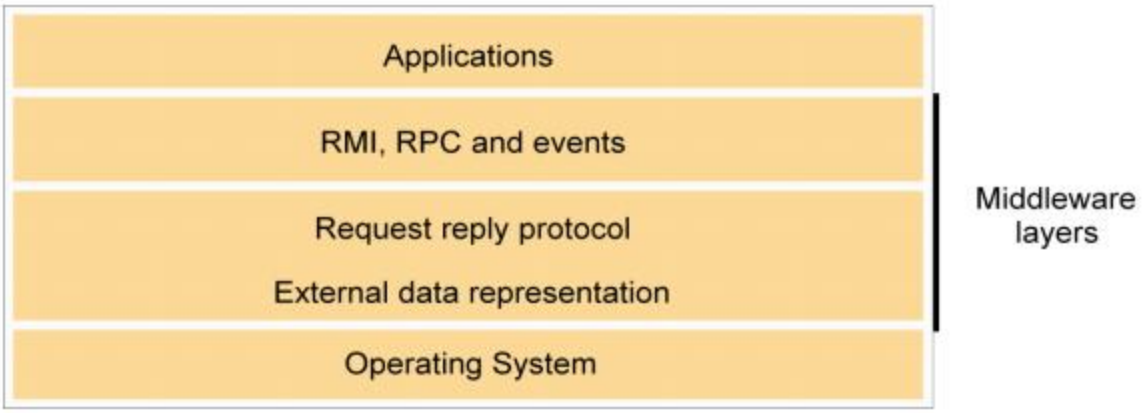
– Remote Procedure Calls – Client programs call procedures in server programs
– Remote Method Invocation – Objects invoke methods of objects on distributed hosts
– Event-based Programming Model – Objects receive notice of events in other objects in which they have interest

Middleware

• Middleware: software that allows a level of programming beyond processes and message

passing
– Uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events
– Supports location transparency
– Usually uses an interface definition language (IDL) to define interfaces





Interfaces in Programming Languages

– Current PL allow programs to be developed as a set of modules that communicate with each other. Permitted interact ions between modules are defined by interfaces

– A specified interface can be implemented by different modules without the need to modify other modules using the interface

• Interfaces in Distributed Systems

– When modules are in different processes or on different hosts there are limitations on the interactions that can occur. Only actions with parameters that are fully specified and understood can communicate effectively to request or provide services to modules in another process

– A service interface allows a client to request and a server to provide particular services

– A remote interface allows objects to be passed as arguments to and results from distributed modules

• Object Interfaces

– An interface defines the signatures of a set of methods, including arguments, argument types, return values and exceptions. Implementation details are not included in an interface.

A class may implement an interface by specifying behavior for each method in the interface.

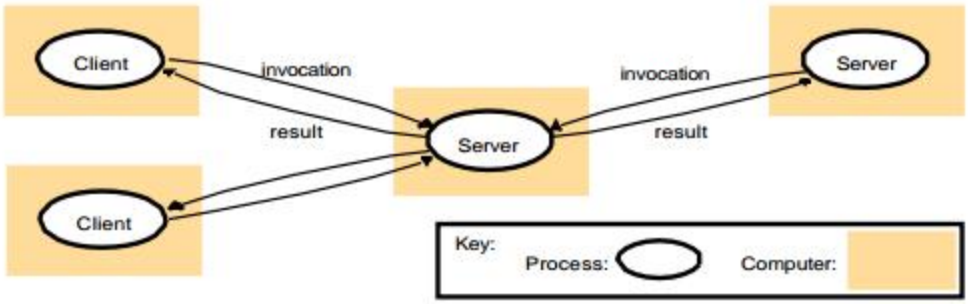
Interfaces do not have constructors.

System architectures

Client-server: This is the architecture that is most often cited when distributed systems are discussed. It is historically the most important and remains the most widely employed. Figure 2.3 illustrates the simple structure in which processes take on the roles of being clients or servers. In particular, client processes interact with individual server processes in potentially separate host computers in order to access the shared resources that they manage.

Servers may in turn be clients of other servers, as the figure indicates. For example, a web server is often a client of a local file server that manages the files in which the web pages are stored. Web servers and most other Internet services are clients of the DNS service, which translates Internet domain names to network addresses.

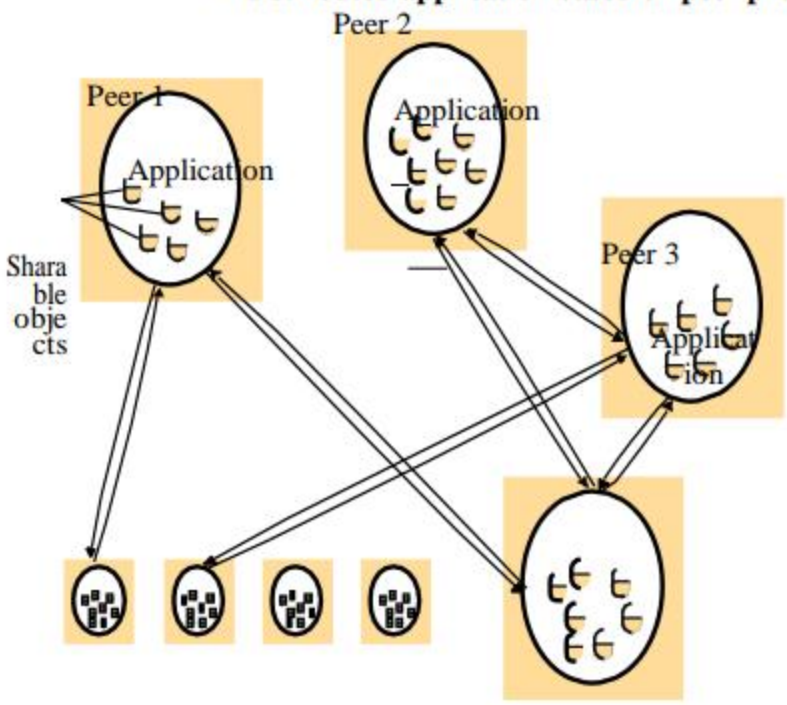
Clients invoke individual servers



Another web-related example concerns *search engines*, which enable users to look up summaries of information available on web pages at sites throughout the Internet. These summaries are made by programs called *web crawlers*, which run in the background at a search engine site using HTTP requests to access web servers throughout the Internet. Thus a search engine is both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers. In this example, the server tasks (responding to user queries) and the crawler tasks (making requests to other web servers) are entirely independent; there is little need to synchronize them and they may run concurrently. In fact, a typical search engine would normally include many
concurrent threads of execution, some serving its clients and others running web crawlers. In Exercise 2.5, the reader is invited to consider the only synchronization issue that does arise for a concurrent search engine of the type outlined here.

Peer-to-peer: In this architecture all of the processes involved in a task or activity play similar roles, interacting cooperatively as *peers* without any distinction between client and server processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each
other. While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly.

A distributed application based on peer processes



A number of placement strategies have evolved in response to this problem, but none of them addresses the fundamental issue – the need to distribute shared resources much more widely in order to share the computing and communication loads incurred in accessing them amongst a much larger number of computers and network links. The key insight that led to the development of peer-to-peer systems is that the network and computing resources owned by the users of a service could also be put to use to support that service. This has the useful consequence that the resources available to run the service grow with the number of users.

Models of systems share some fundamental properties. In particular, all of them are composed of processes that communicate with one another by sending messages over a computer network. All of the models share the design requirements of achieving the performance and reliability characteristics of processes and networks and ensuring the security of the resources in the system.

About their characteristics and the failures and security risks they might exhibit. In general, such a fundamental model should contain only the essential ingredients that need to consider in order to understand and reason about some aspects of a system's behaviour. The purpose of such a model is:

• To make explicit all the relevant assumptions about the systems we are modelling.
• To make generalizations concerning what is possible or impossible, given those assumptions. The generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed. The guarantees are
dependent on logical analysis and, where appropriate, mathematical proof.

The aspects of distributed systems that we wish to capture in our fundamental models are intended to help us to discuss and reason about:

Interaction: Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes. In the analysis and design of distributed systems we are concerned especially with these interactions. The interaction model must reflect the facts that communication takes place with delays that are often of considerable duration, and that the accuracy with which independent processes can be coordinated is limited by these delays and by the difficulty of maintaining the same notion of time across all the computers in a distributed system.

Failure: The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs (including software faults) or in the network that connects them. Our model defines and classifies the faults. This provides a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.

Security: The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents. Our security model defines and classifies the forms that such

attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

Interaction model

Fundamentally distributed systems are composed of many processes, interacting in complex ways. For example:

- Multiple server processes may cooperate with one another to provide a service; the examples mentioned above were the Domain Name System, which partitions and replicates its data at servers throughout the Internet, and Sun's Network Information Service, which keeps replicated copies of password files at several servers in a local area network.
- A set of peer processes may cooperate with one another to achieve a common goal: for example, a voice conferencing system that distributes streams of audio data in a similar manner, but with strict real-time constraints.

Most programmers will be familiar with the concept of an *algorithm* – a sequence of steps to be taken in order to perform a desired computation. Simple programs are controlled by algorithms in which the steps are strictly sequential. The behaviour of the program and the state of the program's variables is determined by them. Such a program is executed as a single process. Distributed systems composed of multiple processes such as those outlined above are more complex. Their behaviour and state can be described by a *distributed algorithm* – a definition of the steps to be taken by each of the processes of which the system is composed, *including the transmission of messages between them*. Messages are transmitted between processes to transfer information between them and to coordinate their activity.

Two significant factors affecting interacting processes in a distributed system:
• Communication performance is often a limiting characteristic.
• It is impossible to maintain a single global notion of time.

Performance of communication channels • The communication channels in our model are realized in a variety of ways in distributed systems – for example, by an implementation of streams or by simple message passing over a computer network. Communication over a computer network has the following performance characteristics relating to latency, bandwidth and jitter:

The delay between the start of a message's transmission from one process and the beginning of its receipt by another is referred to as *latency*. The latency includes:

– The time taken for the first of a string of bits transmitted through a network to reach its destination. For example, the latency for the transmission of a message through a satellite link is the time for a radio signal to travel to the satellite and back.

– The delay in accessing the network, which increases significantly when the network is heavily loaded. For example, for Ethernet transmission the sending station waits for the network to be free of traffic.

– The time taken by the operating system communication services at both the sending and the receiving processes, which varies according to the current load on the operating systems.

• The *bandwidth* of a computer network is the total amount of information that can be transmitted over it in a given time. When a large number of communication channels are using the same network, they have to share the available bandwidth.

• *Jitter* is the variation in the time taken to deliver a series of messages. Jitter is relevant to multimedia data. For example, if consecutive samples of audio data are played with differing time intervals, the sound will be badly distorted.

Computer clocks and timing events • Each computer in a distributed system has its own internal clock, which can be used by local processes to obtain the value of the current time. Therefore two processes running on different computers can each associate timestamps with their events. However, even if the two processes read their clocks at the same time, their local clocks may supply different time values. This is because computer clocks drift from perfect time and, more importantly, their drift rates differ from one another. The term *clock drift rate* refers to the rate at which a computer clock deviates from a perfect reference clock. Even if the clocks on all the computers in a distributed system are set to the same time initially, their clocks will eventually vary quite significantly unless corrections are applied.

Two variants of the interaction model • In a distributed system it is hard to set limits on the time that can be taken for process execution, message delivery or clock drift. Two opposing extreme positions provide a pair of simple models – the first has a strong assumption of time and the second makes no assumptions about time:

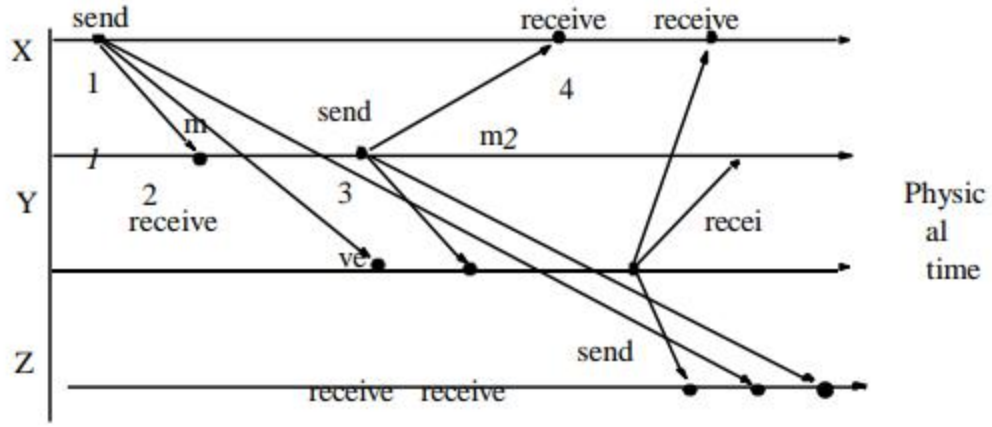
Synchronous distributed systems: Hadzilacos and Toueg define a synchronous distributed system to be one in which the following bounds are defined:
• The time to execute each step of a process has known lower and upper bounds.
• Each message transmitted over a channel is received within a known bounded time.
• Each process has a local clock whose drift rate from real time has a known bound.

Asynchronous distributed systems: Many distributed systems, such as the Internet, are very useful without being able to qualify as synchronous systems. Therefore we need an alternative model. An asynchronous distributed system is one in which there are no bounds on:
• Process execution speeds – for example, one process step may take only a picosecond and another a century; all that can be said is that each step may take an arbitrarily long time.
• Message transmission delays – for example, one message from process A to process B may be delivered in negligible time and another may take several years. In other words, a message may be received after an arbitrarily long time.
• Clock drift rates – again, the drift rate of a clock is arbitrary.

Event ordering • In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after or concurrently with another event at another process. The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks. For example, consider the following set of exchanges between a group of email users, X, Y, Z and A, on a mailing list:

1. User X sends a message with the subject *Meeting*.
2. Users Y and Z reply by sending a message with the subject *Re: Meeting*.

In real time, X's message is sent first, and Y reads it and replies; Z then reads both X's message and Y's reply and sends another reply, which references both X's and Y's messages. But due to the independent delays in message delivery, the messages may be delivered as shown in the following figure and some users may view these two messages in the wrong order.



A

m3 m1 m2
receive receive receive

t1 t2 t3

Failure model

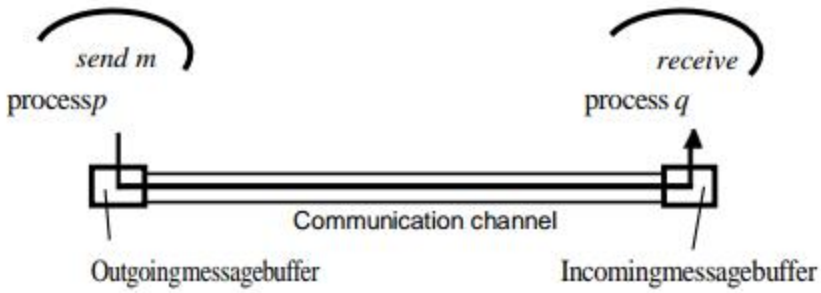
In a distributed system both processes and communication channels may fail – that is, they may depart from what is considered to be correct or desirable behaviour. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. Hadzilacos and Toueg provide a taxonomy that distinguishes between the failures of processes and communication channels. These are presented under the headings omission failures, arbitrary failures and timing failures.

Omission failures • The faults classified as *omission failures* refer to cases when a process or communication channel fails to perform actions that it is supposed to do.

Process omission failures: The chief omission failure of a process is to crash. When, say that a process has crashed we mean that it has halted and will not execute any further steps of its program ever. The design of services that can survive in the presence of faults can be simplified if it can be assumed that the services on which they depend crash cleanly – that is, their processes either function correctly or else stop. Other processes may be able to detect such a

crash by the fact that the process repeatedly fails to respond to invocation messages. However, this method of crash detection relies on the use of *timeouts* – that is, a method in which one process allows a fixed period of time for
something to occur. In an asynchronous system a timeout can indicate only that a process is not responding – it may have crashed or may be slow, or the messages may not have arrived.

Communication omission failures: Consider the communication primitives *send* and *receive*. A process *p* performs a *send* by inserting the message *m* in its outgoing message buffer. The communication channel transports *m* to *q*'s incoming message buffer. Process *q* performs a *receive* by taking *m* from its incoming message buffer and delivering it. The outgoing and incoming message buffers are typically provided by the operating system.



Arbitrary failures • The term *arbitrary* or *Byzantine* failure is used to describe the worst possible failure semantics, in which any type of error may occur. For example, a process may set wrong values in its data items, or it may return a wrong value in response to an invocation.
An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or

takes unintended processing steps. Arbitrary failures in processes cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply.

Communication channels can suffer from arbitrary failures; for example, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once. Arbitrary failures of communication channels are rare because the communication software is able to recognize them and reject the faulty
messages. For example, checksums are used to detect corrupted messages, and message sequence numbers can be used to detect nonexistent and duplicated messages.

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.

Timing failures • Timing failures are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate. Timing failures are listed in the following figure. Any one of these failures may result in responses being unavailable to clients within a specified time interval.
In an asynchronous distributed system, an overloaded server may respond too slowly, but we cannot say that it has a timing failure since no guarantee has been offered. Real-time operating systems are designed with a view to providing timing guarantees, but they are more complex to design and may require redundant hardware.
Most general-purpose operating systems such as UNIX do not have to meet real-time constraints.

Masking failures • Each component in a distributed system is generally constructed from a collection of other components. It is possible to construct reliable services from components that exhibit failures. For example, multiple servers that hold replicas of data can continue to provide a service when one of them crashes. A knowledge of the failure characteristics of a component can enable a new service to be designed to mask the failure of the components on which it depends. A service *masks* a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For an example of the latter, checksums are used to mask corrupted messages, effectively converting an arbitrary failure into an omission failure. The omission failures can be hidden by using a protocol that retransmits messages that do not arrive at their destination. Even process crashes may be masked, by replacing the process and restoring its memory from information stored on disk by its predecessor.

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

Reliability of one-to-one communication • Although a basic communication channel can exhibit the omission failures described above, it is possible to use it to build a communication service that masks some of those failures.

The term *reliable communication* is defined in terms of validity and integrity as follows:

Validity: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.
Integrity: The message received is identical to one sent, and no messages are delivered twice.

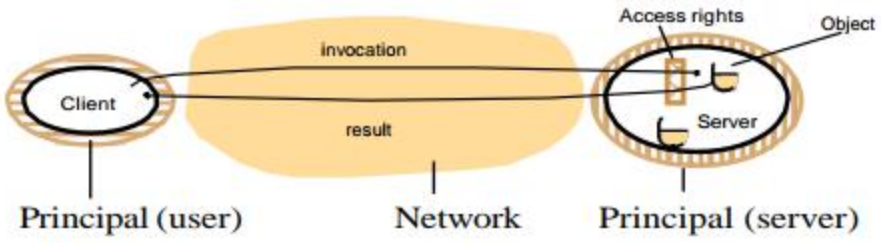
The threats to integrity come from two independent sources:
• Any protocol that retransmits messages but does not reject a message that arrives twice. Protocols can attach sequence numbers to messages so as to detect those that are delivered twice.

• Malicious users that may inject spurious messages, replay old messages or tamper with messages. Security measures can be taken to maintain the integrity property in the face of such attacks.

Security model
The sharing of resources as a motivating factor for distributed systems, and in Section 2.3 we described their architecture in terms of processes, potentially encapsulating higher-level abstractions such as objects, components or services, and providing access to them through interactions with other processes. That architectural model provides the basis for our security model:
the security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

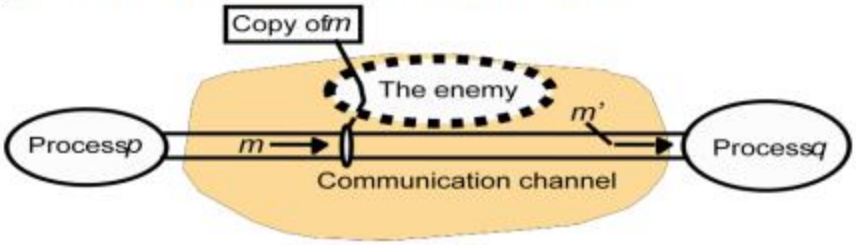
Protection is described in terms of objects, although the concepts apply equally well to resources of all types

Protecting objects :
Server that manages a collection of objects on behalf of some users. The users can run client programs that send invocations to the server to perform operations on the objects. The server carries out the operation specified in each invocation and sends the result to the client.
Objects are intended to be used in different ways by different users. For example, some objects may hold a user's private data, such as their mailbox, and other objects may hold shared data such as web pages. To support this, *access rights* specify who is allowed to perform the operations of an object – for example, who is allowed to read or to write its state.



Securing processes and their interactions • Processes interact by sending messages. The messages are exposed to attack because the network and the communication service that they use are open, to enable any pair of processes to interact. Servers and peer processes expose their interfaces, enabling invocations to be sent to them by any other process.

The enemy • To model security threats, we postulate an enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in the following figure. Such attacks can be made simply by using a computer connected to a network to run a program that reads network messages addressed to other computers on the network, or a program that generates messages that make false requests to services, purporting to come from authorized users. The attack may come from a computer that is legitimately connected to the network or from one that is connected in an unauthorized manner. The threats from a potential enemy include *threats to processes* and *threats to communication channels*.



Defeating security threats

Cryptography and shared secrets: Suppose that a pair of processes (for example, a particular client and a particular server) share a secret; that is, they both know the secret but no other process in the distributed system knows it. Then if a message exchanged by that pair of processes includes information that proves the sender's knowledge of the
shared secret, the recipient knows for sure that the sender was the other process in the pair. Of course, care must be taken to ensure that the shared secret is not revealed to an enemy.

Cryptography is the science of keeping messages secure, and *encryption* is the process of scrambling a message in such a way as to hide its contents. Modern cryptography is based on encryption algorithms that use secret keys – large numbers that are difficult to guess – to transform data in a manner that can only be reversed with knowledge of the corresponding decryption key.

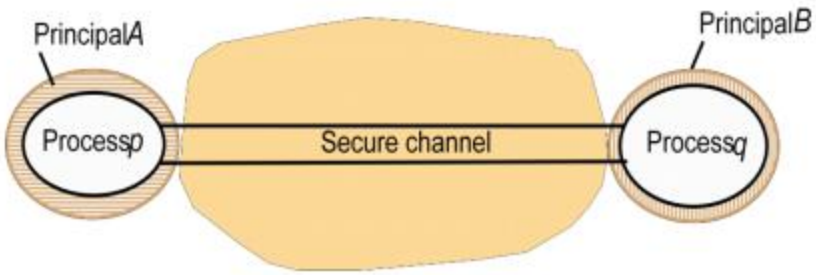
Authentication: The use of shared secrets and encryption provides the basis for the *authentication* of messages – proving the identities supplied by their senders. The basic authentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to guarantee its authenticity. The authentication portion of a request to a file server to read part of a file, for example, might include a representation of the requesting principal's identity, the identity of the file and the date and time of the request, all encrypted with

a secret key shared between the file server and the requesting process. The server would decrypt this and check that it corresponds to the unencrypted details specified in the request.

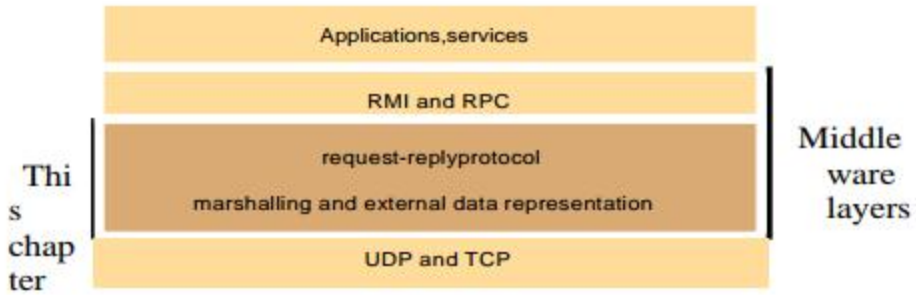
Secure channels: Encryption and authentication are used to build secure channels as a service layer on top of existing communication services. A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal, as shown in the following figure. A secure channel has the following properties:

• Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing. Therefore if a client and server communicate via a secure channel, the server knows the identity of the principal behind the invocations and can check their access rights before performing an operation. This enables the server to protect its objects correctly and allows the client to be sure that it is receiving results from a *bona fide* server.

• A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it.
• Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered.



Communication aspects of middleware, although the principles discussed are more widely applicable. This one is concerned with the design of the components shown in the darker layer in the following figure.



UNIT II

Time and Global States: Introduction, Clocks, Events and Process states, Synchronizing physical clocks, Logical time and Logical clocks, Global states, Distributed Debugging.
Coordination and Agreement: Introduction, Distributed mutual exclusion, Elections, Multicast Communication, Consensus and Related problems.

CLOCKS, EVENTS AND PROCESS STATES

Each process executes on a single processor, and the processors do not share memory (Chapter 6 briefly considered the case of processes that share memory). Each process *pi* in has a state *si* that, in general, it transforms as it executes. The process's state includes the values of all the variables within it. Its state may also include the values of any objects in its local operating system environment that it affects, such as files. We assume that processes cannot communicate with one another in any way except by sending messages through the network.

So, for example, if the processes operate robot arms connected to their respective nodes in the system, then they
are not allowed to communicate by shaking one another's robot hands! As each process *pi* executes it takes a series of actions, each of which is either amessage *send* or *receive* operation, or an operation that transforms *pi* 's state – one that
changes one or more of the values in *si*. In practice, we may choose to use a high-leveldescription of the actions, according to the application. For example, if the processes in are engaged in an eCommerce application, then the actions may be ones such as 'client dispatched order message' or 'merchant server recorded transaction to log'.
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. The sequence of events within a single process *pi* can be placed in a single, total ordering, which we denote by the relation *i* between the events.
That is, if and only if the event *e* occurs before *e* at *pi* . This ordering is well defined, whether or not the process is multithreaded,
since we have assumed that the process executes on a single processor. Now we can define the *history* of process *pi* to be the series of events that take place within it, ordered as we have described by the relation **Clocks** • We have seen how to order the events at a process, but not how to timestamp them – i.e., to assign to them a date and time of day. Computers each contain their own physical clocks. These clocks are electronic devices that count oscillations occurring in a crystal at a definite frequency, and 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, timeslicing can be implemented; however, we shall not concern ourselves with this aspect of clock operation.

The operating system reads the node's hardware clock value, *Hit* , scales it and adds an offset so as to produce a software clock *Cit* = *Hit* + that approximately measures real, physical time *t* for process *pi* . In other words, when the real time in an absolute frame of reference is *t*, *Cit* is the reading on the software clock. For example,
Cit could be the 64-bit value of the number of nanoseconds that have elapsed at time *t* since a convenient reference time. In general, the clock is not completely accurate, so *Cit* will differ from
t. Nonetheless, if *Ci* behaves sufficiently well (we shall examine the notion of clock correctness shortly), we can use its value to timestamp any event at *pi* . Note that successive events will

correspond to different timestamps only if the *clock resolution* – the period between updates of the clock value – is smaller than the time interval between successive events. The rate at which events occur depends on such factors as the length of the processor instruction cycle.

Clock skew and clock drift • Computer clocks, like any others, tend not to be in perfect agreement

Coordinated Universal Time • Computer clocks can be synchronized to external sources of highly accurate time. The most accurate physical clocks use atomic oscillators, whose
drift rate is about one part in 1013. The output of these atomic clocks is used as the standard second has been defined as 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of Caesium-133 (Cs133).
Seconds and years and other time units that we use are rooted in 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 the Earth's rotation about 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 step.

Coordinated Universal Time – abbreviated as UTC (from the French equivalent) – is an international standard for timekeeping. It is based on atomic time, but a so-called 'leap second' is inserted – or, more rarely, deleted – occasionally to keep it in step with astronomical time. UTC signals are synchronized and broadcast regularly from landbased
radio stations and satellites covering many parts of the world. For example, in the USA, the radio station WWV broadcasts time signals on several shortwave frequencies.
Satellite sources include the *Global Positioning System* (GPS).Receivers are available commercially. Compared with 'perfect' UTC, the signals received from land-based stations have an accuracy on the order of 0.1–10 milliseconds,
depending on the station used. Signals received from GPS satellites are accurate to about 1 microsecond. Computers with receivers attached can synchronize their clocks with these timing signals.

Synchronizing physical clocks
In order to know at what time of day events occur at the processes in our distributed system – for example, for accountancy purposes – it is necessary to synchronize the processes' clocks, Ci , with an authoritative, external source of time. This is *external synchronization*. And if the clocks Ci are synchronized with one another to a known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks, even though they are not necessarily synchronized to an external source of time. This is *internal synchronization*.We define these two modes of synchronization more closely as follows, over an interval
of real time I:

External synchronization: For a synchronization bound D 0 , and for a source S of UTC time, $St - Cit < D$, for $i = 1\ 2N$ and for all real times t in I. Another way of saying this is that the clocks Ci are *accurate* to within the bound D.
Internal synchronization: For a synchronization bound D 0 , $Cit - Cjt$ D for $i\ j = 1\ 2N$, and for all real times t in I. Another way of saying this is that he clocks *Ci agree* within the bound D. Clocks that are internally synchronized are not necessarily externally synchronized, since they may drift collectively from an external source of time even though they agree with one another. However, it follows from the definitions that if the system is externally synchronized with a bound D then the same system is internally synchronized with a bound of $2D$. Various notions of *correctness* for clocks

have been suggested. It is common to define a hardware clock H to be correct
if its drift rate falls within a known bound (a value derived from one supplied by the manufacturer, such as 10–6 seconds/second).
This means that the error in measuring the interval between real times t and t (t t) is bounded:
$1 - t - t$ $Ht - Ht$ $1 + t - t$
This condition forbids jumps in the value of hardware clocks (during normal operation). Sometimes we also require our software clocks to obey the condition but a weaker condition of *monotonicity* may suffice. Monotonicity is the condition that a clock C only ever advances: t t Ct Ct For example, the UNIX *make* facility is a tool that is used to compile only those source files that have been modified since they were last compiled. The modification dates of each corresponding pair of source and object files are compared to determine this condition. If a computer whose clock was running fast set its clock back after compiling a source file but before the file was changed, the source file might appear to have been modified prior to the compilation. Erroneously, *make* will not recompile the source file.
We can achieve monotonicity despite the fact that a clock is found to be running fast. We need only change the rate at which updates are made to the time as given to applications. This can be achieved in software without changing the rate at which the underlying hardware clock ticks – recall that $Cit = Hit +$, where we are free to
choose the values of and . A hybrid correctness condition that is sometimes applied is to require that a clock
obeys the monotonicity condition, and that its drift rate is bounded between synchronization points, but to allow the clock value to jump ahead at synchronization points.
A clock that does not keep to whatever correctness conditions apply is defined to be *faulty*. A clock's *crash failure* is said to occur when the clock stops ticking altogether;
any other clock failure is an *arbitrary failure*. A historical example of an arbitrary failure is that of a clock with the 'Y2K bug', which broke the monotonicity condition by registering the date after 31 December 1999 as 1 January 1900 instead of 2000; another example is a clock whose batteries are very low and whose drift rate suddenly becomes
very large.
Note that clocks do not have to be accurate to be correct, according to the definitions. Since the goal may be internal rather than external synchronization, the criteria for correctness are only concerned with the proper functioning of the clock's 'mechanism', not its absolute setting. We now describe algorithms for external synchronization and for internal
synchronization.
Logical time and logical clocks
From the point of view of any single process, events are ordered uniquely by times shown on the local clock. However, as Lamport [1978] pointed out, since we cannot synchronize clocks perfectly across a distributed system, we cannot in general use physical time to find 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. This ordering is based on two simple and intuitively obvious points: • If two events occurred at the same process pi $i = 1$ 2 N , then they occurred in the order in which pi observes them – this is the order i that we defined above.• Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.
Lamport called the partial ordering obtained by generalizing these two relationships the *happened-before* relation. It is also sometimes known as the relation of *causal ordering* or *potential causal ordering*.
We can define the happened-before relation, denoted by , as follows: HB1: If processpi : e i e', then e e .
HB2: For any message m, *send*(m) *receive*(m) – where *send*(m) is the event of sending the message, and *receive*(m)

is the event of receiving it. HB3: If *e*, *e* and *e* are events such that *e e* and *e e* , then *e*
e .

Totally ordered logical clocks • Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps. However, we can create a total order on the set of events – that is, one for which all pairs of distinct events are ordered – by taking into account the identifiers of the processes at which events occur. If *e* is an event occurring at *pi* with local timestamp *Ti* , and *e* is an event occurring at *pj* with local timestamp *Tj* , we define the global logical timestamps for these events to be *Ti i* and *Tj j* , respectively. And we define *Ti i Tj j* if and only if either *Ti Tj* , or *Ti* = *Tj* and *i j* . This ordering has no general physical significance
(because process identiiers are arbitrary), but it is sometimes useful. Lamport used it, for example, to order the entry of processes to a critical section.

Vector clocks • Mattern [1989] and Fidge [1991] developed vector clocks to overcome the shortcoming of Lamport's clocks: the fact that from *Le Le* we cannot conclude that *e e*.
. A vector clock for a system of *N* processes is an array of *N*
integers. Each process keeps its own vector clock, *Vi* , which it uses to timestamp local events. Like Lamport timestamps, processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:

VC1: Initially, *Vij* = 0 , for *i j* = 1 2 *N* .
VC2: Just before *pi* timestamps an event, it sets *Vii* :=*Vii* + 1. VC3:
pi includes the value *t* = *Vi* in every message it sends.
VC4: When *pi* receives a timestamp *t* in a message, it sets *Vij* := *max*Vij *tj* , for *j* = 1 2 *N* . Taking the componentwise maximum of two vector timestamps in this way is known as a *merge* operation.For a vector clock *Vi* , *Vii* is the number of events that *pi* has timestamped, and *Vij j i* is the number of events that have occurred at *pj* that have potentially affected *pi* . (Process *pj* may have timestamped more events by this point, but no information has flowed to *pi* about them in messages as yet.)

Clocks, Events and Process States

- A distributed system consists of a collection *P* of ***N*** processes ***pi***, *i = 1,2,... N*Each process *pi* has a state ***si*** consisting of its variables (which it transforms as it executes)
  Processes communicate only by messages (via a network)
- **Actions** of processes*: Send, Receive,* change own state
- ***Event****:* the occurrence of a single action that a process carries out as it executes
  – Events at a single process *pi*, can be placed in a total **ordering** denoted by the relation →*i* between the events. i.e.***e*** →***i*** ***e'*** if and only if event *e* occurs before event *e'* at process *pi*
- A history of process pi: is a series of events ordered by →*i*
  – ***history*****(pi)** = hi =<ei0, *ei*1, *ei*2, …>

Clocks

To timestamp events, use the computer's clock • At **real time, *t***, the OS reads the time on the computer's **hardware clock *Hi*(*t*)**
• It calculates the time on its **software clock *Ci*(*t*)=α*Hi*(*t*) + β**

– e.g. a 64 bit value giving nanoseconds since some base time

– **Clock resolution**: period between updates of the clock value

• In general, the clock is not completely accurate – but if *Ci* behaves well enough, it can be used to timestamp events at *pi*

Skew between computer clocks in a distributed system

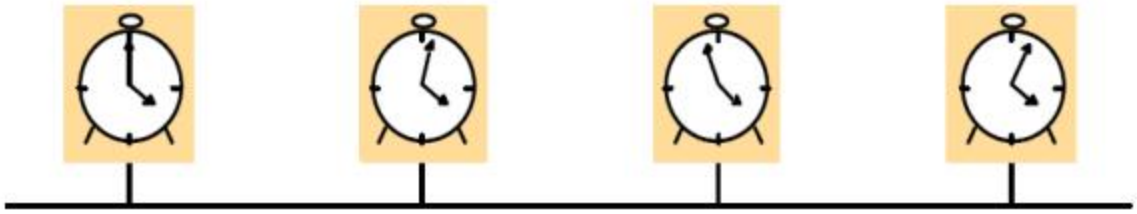


Figure 11.1

Computer clocks are not generally in perfect agreement
• ***Clock skew***: the difference between the times on two clocks (at any instant)
• Computer clocks use crystal-based clocks that are subject to physical variations
– ***Clock drift***: they count time at different rates and so diverge (frequencies of oscillation differ)
– ***Clock drift rate***: the difference per unit of time from some ideal reference clock
– Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10-6 secs/sec).
– High precision quartz clocks drift rate is about 10-7 or 10-8 secs/sec

Coordinated Universal Time (UTC)

• UTC is an international standard for time keeping
– It is based on atomic time, but occasionally adjusted to astronomical time
– International Atomic Time is based on very accurate physical clocks (drift rate 10-13)
• It is broadcast from radio stations on land and satellite (e.g.GPS)

• Computers with receivers can synchronize their clocks with these timing signals (by requesting time from GPS/UTC source)
– Signals from land-based stations are accurate to about 0.1-10 millisecond
– Signals from GPS are accurate to about 1 microsecond

Synchronizing physical clocks

Two models of synchronization
• External synchronization: a computer's clock *Ci* is synchronized with an external authoritative time source *S*, so that:
– $|S(t) - Ci(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, *I* of real time
– The clocks *Ci* are **accurate** to within the bound *D*.
• Internal synchronization: the clocks of a pair of computers are synchronized with one another so that:
– $| Ci(t) - Cj(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, *I* of real time
– The clocks *Ci* and *Cj* **agree** within the bound *D*.
Internally synchronized clocks are not necessarily externally synchronized, as they may drift collectively

– if the set of processes *P* is synchronized externally within a bound *D*, it is also internally synchronized within bound 2*D (worst case polarity)*

Clock correctness
• ***Correct clock***: a hardware clock *H* is said to be correct if its drift rate is within a bound $\rho > 0$ (e.g. 10-6 secs/ sec)
This means that the error in measuring the interval between real times *t* and *t'* is bounded:
– $(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$ (where $t' > t$) Which forbids jumps in time readings of hardware clocks

– ***Clock monotonicity***: weaker condition of correctness – $t' > t \Rightarrow C(t') > C(t)$ e.g. required by Unix *make*
– A hardware clock that runs fast can achieve monotonicity by adjusting the values of α and β such that $Ci(t) = \alpha Hi(t) + \beta$
– ***Faulty clock***: a clock not keeping its correctness condition *crash failure* - a clock stops ticking
• *arbitrary* failure - any other failure e.g. jumps in time; Y2K bug

Synchronization in a synchronous system

A synchronous distributed system is one in which the following bounds are defined

The time to execute each step of a process has known lower and upper bounds each message transmitted over a channel is received within a knownbounded time (min and max) each process has a local clock whose drift rate from real time has a known bound
Internal synchronization in a synchronous system

- One process *p*1 sends its local time *t* to process *p*2 in a message *m*
- *p*2 could set its clock to *t* + *T*trans where *T*trans is the time to transmit *m*
- *T*trans is unknown but *min* ≤ *T*trans ≤ *max*
- uncertainty *u* = *max*-*min*. Set clock to *t* + (*max* - *min*)/2 then skew ≤ *u*/2

Cristian's method for an asynchronous system

- A time server *S* receives signals from a UTC source
- Process *p* requests time in *mr* and receives *t* in *mt* from *S*
- *p* sets its clock to *t* + *T*round/2
- Accuracy ± (*T*round/2 - *min*) :
- because the earliest time *S* puts *t* in message *mt* is *min* after *p* sent *mr*
- the latest time was *min* before *mt* arrived at *p*
- the time by *S*'s clock when *mt* arrives is in the range [*t*+*min*, *t* + *T*round - *min*]
- the width of the range is *T*round + 2*min*

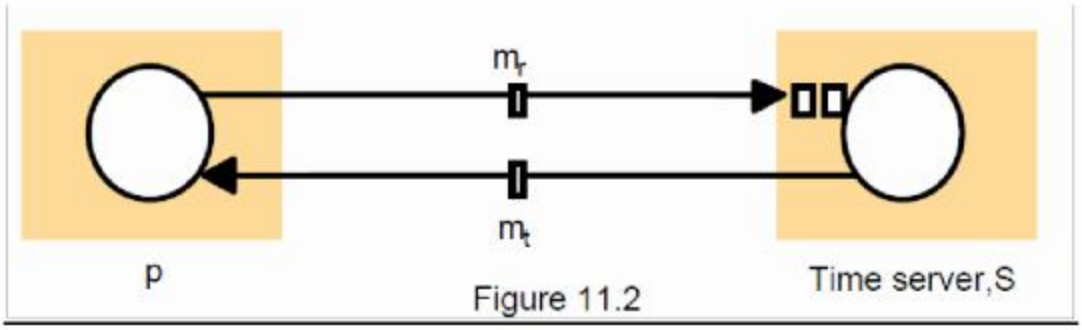


Figure 11.2

The Berkeley algorithm

- Problem with Cristian's algorithm
- a single time server might fail, so they suggest the use of a group of synchronized servers
- it does not deal with faulty servers
- Berkeley algorithm (also 1989)
- An algorithm for internal synchronization of a group of computers
- A *master* polls to collect clock values from the others (*slaves*)
- The master uses round trip times to estimate the slaves' clock values
- It takes an average (eliminating any above some average round trip time or with faulty clocks)
- It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
- Measurements
- 15 computers, clock synchronization 20-25 millisecs drift rate < 2x10-5
- If master fails, can elect a new master to take over (not in bounded time)

Network Time Protocol (NTP)

- A time service for the Internet - synchronizes clients to UTC Reliability from redundant paths, scalable, authenticates time sources Architecture
- Primary servers are connected to UTC sources
- Secondary servers are synchronized to primary servers
- Synchronization subnet - lowest level servers in users' computers
- strata: the hierarchy level

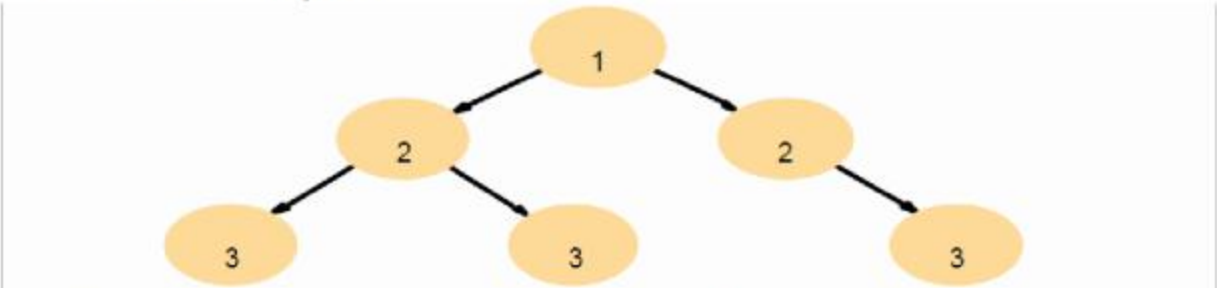


Figure 11.3 An example synchronization subnet in an NTP implementation

NTP - synchronization of servers

- The synchronization subnet can reconfigure if failures occur
- a primary that loses its UTC source can become a secondary
- a secondary that loses its primary can use another primary
- Modes of synchronization for NTP servers:
- Multicast
- A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
- Procedure call
- A server accepts requests from other computers (like Cristian's algorithm)

- Higher accuracy. Useful if no hardware multicast.

Messages exchanged between a pair of NTP peers

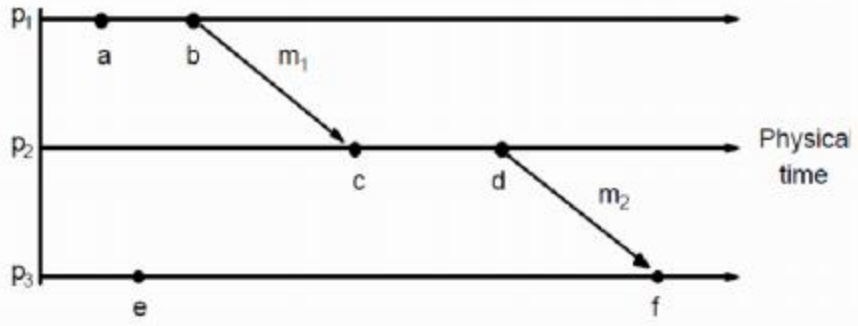
- All modes use UDP
- Each message bears timestamps of recent events:
- Local times of *Send* and *Receive* of previous message
- Local times of *Send* of current message
- Recipient notes the time of receipt *Ti* (we have Ti-3, Ti-2, Ti-1, Ti)
- Estimations of clock offset and message delay
- For each pair of messages between two servers, NTP estimates an offset ***oi*** (between the two clocks) and a delay ***di*** (total time for the two messages, which take *t* and *t'*)
- *Ti*-2 = *Ti-3* + *t* + *o* and *Ti* = *Ti-1* + *t'* - *o*
- This gives us (by adding the equations) : *di* = *t* + *t'* = *T*i-2 - *Ti-3* + *T*i - *Ti-1*
- Also (by subtracting the equations)
  - = *o*i + (*t'* - *t*)/2 where *oi* = (*T*i-2 - *Ti-3* + *Ti-1* - *Ti*)/2
- Using the fact that *t*, *t'*>0 it can be shown that
- *oi* - *di* /2 ≤ *o* ≤ *oi* + *di* /2 .
- Thus *oi* is an estimate of the offset and *di* is a measure of the accuracy
- Data filtering
- NTP servers filter pairs <*oi*, *di*>, estimating reliability from variation (dispersions), allowing them to select peers; and synchronization based on the lowest dispersion or min *di* ok
- A relatively high filter dispersion represents relatively unreliable data
- Accuracy of tens of milliseconds over Internet paths (1 ms on LANs)

Logical time and logical clocks

- Instead of synchronizing clocks, event ordering can be used
- If two events occurred at the same process *pi* (*i* = 1, 2, … *N*) then theyoccurred in the order observed by *pi*, that is order →***i***
- when a message, *m* is sent between two processes, *send(m)* happened before *receive(m)*

- Lamport[1978] generalized these two relationships into the **happened-before relation:** ***e*** →***i e'***
- HB1: if *e* →*i e'* in process *p*i, then *e* → e'
- HB2: for any message *m*, send(*m*) → receive(*m*)
- HB3: if *e* → *e'* and *e'* → *e*", then *e* → *e"*

Figure 11.5 Events occurring at three processes



- HB1: a→b, c→d, e→f
- HB2: b→c, d→f
- HB3: a→b→c→d→f
- a||e: a and e are concurrent (neither a→e nor e→a)

Lamport's logical clocks

- Each process *pi* has a logical clock ***Li***
  - a monotonically increasing software counter
  - not related to a physical clock
- Apply Lamport timestamps to events with happened-before relation
  - LC1: *Li* is incremented by 1 before each event at process *pi*
  - LC2:
  - when process *pi* sends message *m*, it piggybacks $t = Li$
  - when *pj* receives *(m,t)*, it sets $Lj := max(Lj, t)$ and applies LC1 before timestamping the event *receive* (*m*)
- $e \rightarrow e'$ implies $L(e) < L(e')$, but $L(e) < L(e')$ does not imply $e \rightarrow e'$

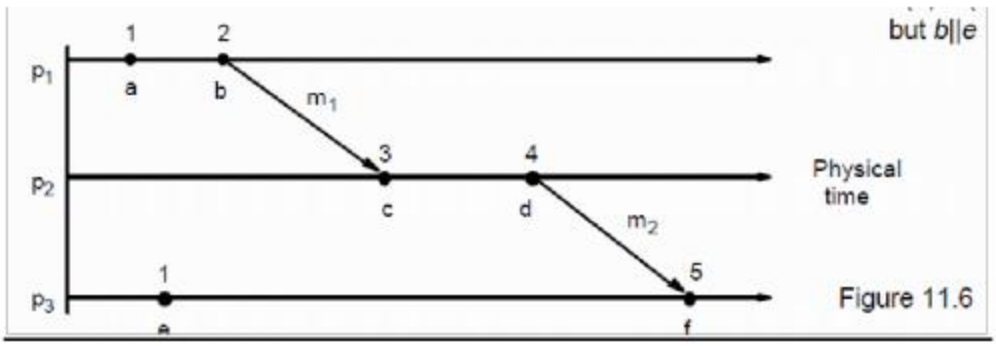


Figure 11.6

Totally ordered logical clocks

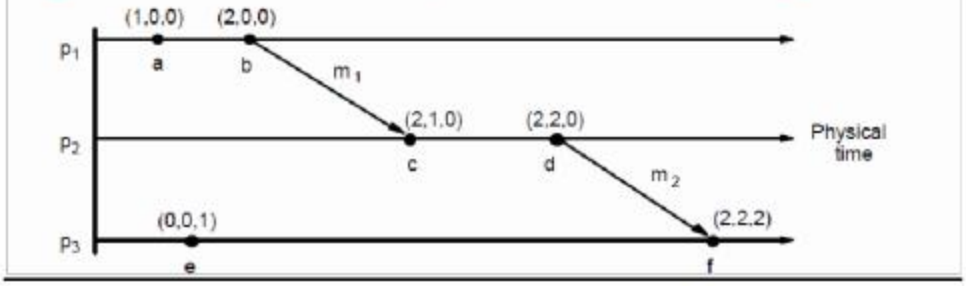
- Some pairs of distinct events, generated by different processes, may have numerically identical Lamport timestamps
- Different processes may have same Lamport time
- Totally ordered logical clocks
- If e is an event occurring at pi with local timestamp Ti, and if e' is an event occurring at pj with local timestamp Tj

- Define global logical timestamps for the events to be (Ti, i) and (Tj, j)
- Define $(Ti, i) < (Tj, j)$ iff
- $Ti < Tj$ or
- $Ti = Tj$ and $i < j$
- No general physical significance since process identifiers are arbitrary

Vector clocks

- Shortcoming of Lamport clocks:
- $L(e) < L(e')$ doesn't imply $e \rightarrow e'$
- Vector clock: an array of N integers for a system of N processes
- Each process keeps its own vector clock Vi to timestamp local events
- Piggyback vector timestamps on messages
- Rules for updating vector clocks:
- $Vi[i]]$ is the number of events that pi has timestamped
- $Vijj]$ ($j \neq i$) is the number of events at pj that pi has been affected by VC1: Initially, $Vi[\ j\ ] := 0$ for pi, j=1.. N (Nprocesses)
- VC2: before pi timestamps an event, $Vi[\ i\ ] := Vi[\ i\ ]$+1 VC3: pi piggybacks $t = Vi$ on every message it sends
- VC4: when pi receives a timestamp t, it sets $Vi[\ j\ ] := \max(Vi[\ j\ ], t[\ j\ ])$ for
- j=1..N (merge operation)

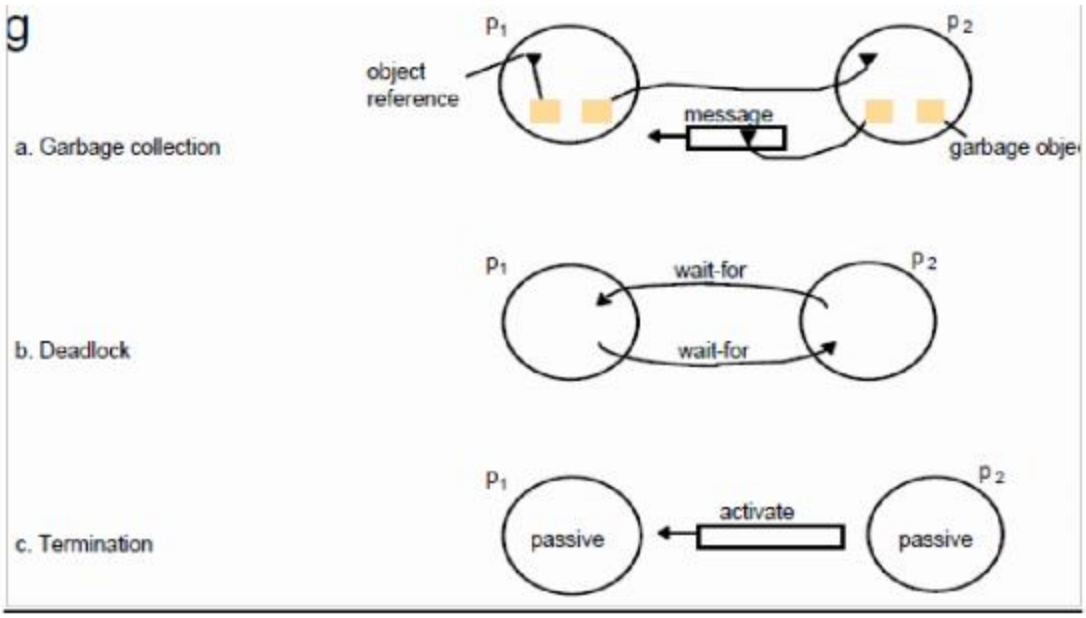
Figure 11.7 Vector timestamps for events shown in Figure 11.5



- Compare vector timestamps
- V=V‘ iff V[j] = V‘[j] for j=1..N
- V>=V‘ iff V[j] <= V‘[j] for j=1..N
- V<V‘ iff V<= V‘ ^ V!=V‘
- Figure 11.7 shows
- $a \rightarrow f$ since V(a) < V(f)
- c || e since neither V(c) <= V(e) nor V(e) <= V(c)

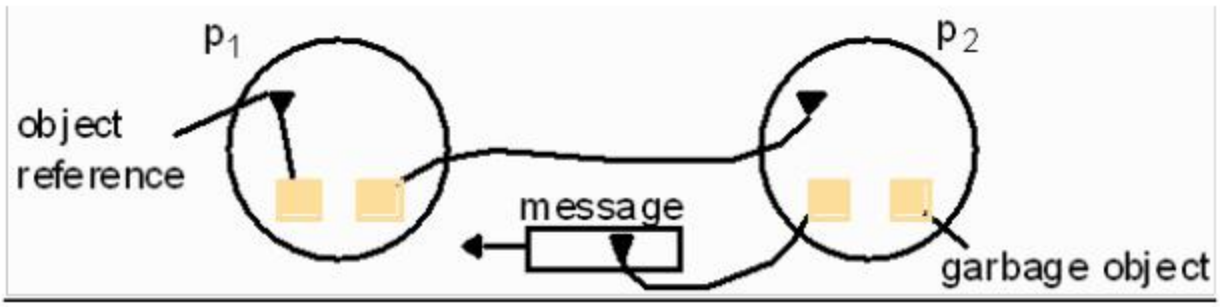
Global states

- How do we find out if a particular property is true in a distributed system? For examples, we will look at:
- Distributed Garbage Collection
- Deadlock Detection
- Termination Detection
- Debugging



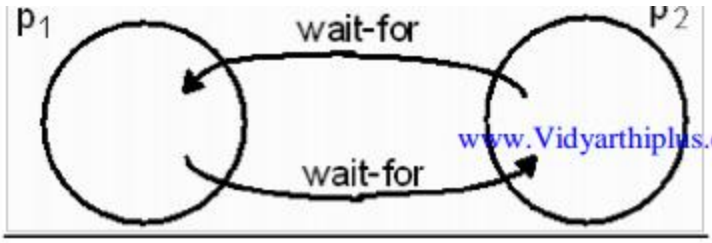
Distributed Garbage Collection

- Objects are identified as ***garbage*** when there are no longer any references to them in the system
- Garbage collection reclaims memory used by those objects
- In figure 11.8a, process p2 has two objects that do not have any references to other objects, but one object does have a reference to a message in transit. It is not garbage, but the other p2 object is
- Thus we must consider communication channels as well as object references to determine unreferenced objects



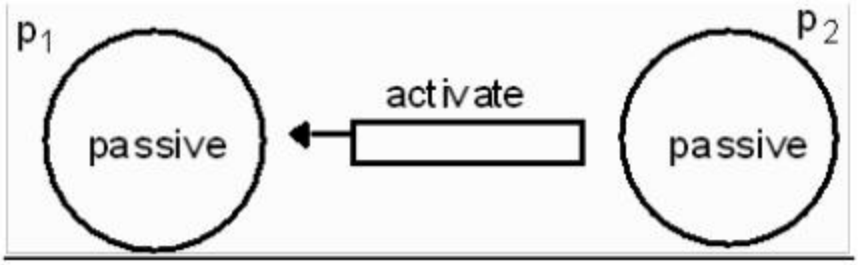
Deadlock Detection

- A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and there is a cycle in the graph of the ***waits-for*** relationship
- In figure 11.8b, both p1 and p2 wait for a message from the other, so both are blocked and the system cannot continue



Termination Detection

- It is difficult to tell whether a distributed algorithm has terminated. It is not enough to detect whether each process has halted
- In figure 11.8c, both processes are in passive mode, but there is an activation request in the network
- Termination detection examines multiple states like deadlock detection, except that a deadlock may affect only a portion of the processes involved, while *termination detection must ensure that all of the processes have completed*



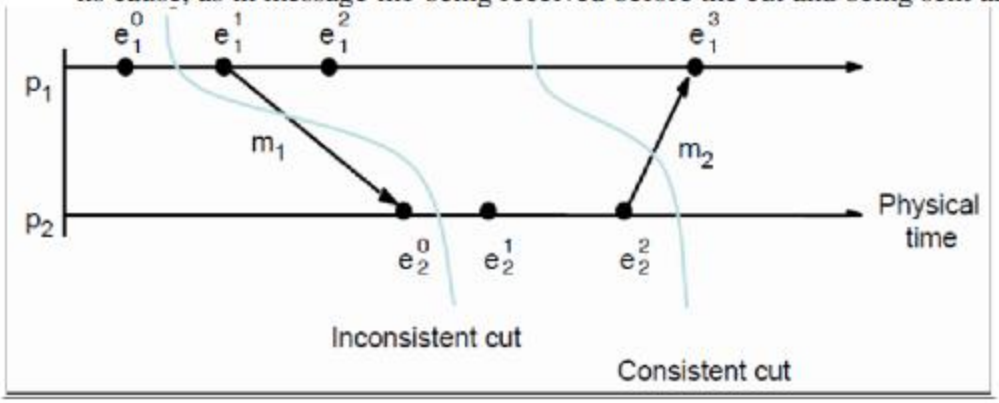
Distributed Debugging

- Distributed processes are complex to debug. One of many possible problems is that consistency restraints must be evaluated for simultaneous attribute values in multiple processes at different instants of time.
- All four of the distributed problems discussed in this section have particular solutions, but all of them also illustrate the need to observe global states. We will now look at a general approach to observing global states.
- Without global time identified by perfectly synchronized clocks, the ability to identify successive states in an individual process does not translate into the ability to identify successive states in distributed processes
- We can assemble meaningful global states from local states recorded at different local times in many circumstances, but must do so carefully and recognize limits to ourcapabilities
- A general system P of N processes pi (i=1..N)
- pi‘s history: history(pi)=hi=<ei0, ei1, ei2, …>
- finite prefix of pi‘s history:
  hi k= <ei0, ei1, ei2, …, eik>
- state of pi immediately before the kth event occurs: sik
- global history H=h1 U h2 U…U hN
- A cut of the system‘s execution is a subset of its global history that is a union of prefix of process histories C=h1c1 U h2c2 U…U hNcN

- The following figure gives an example of an inconsistent cutic and a consistent cutcc. The distinguishing characteristic is that
- cutic includes the receipt of message m1 but *not* the sending of it, while
- cutcc includes the sending *and* receiving of m1 *and* cuts between the

sending and receipt of the message m2.

- A consistent cut cannot violate temporal causality by implying that a result occurred before its cause, as in message m1 being received before the cut and being sent after the cut.



Global state predicates

- A Global State Predicate is a function that maps from the set of global process states to True or False.
- Detecting a condition like deadlock or termination requires evaluating a Global State Predicate.
- A Global State Predicate is stable: once a system enters a state where it is true, such as deadlock or termination, it remains true in all future states reachable from that state.
- However, when we monitor or debug an application, we are interested in non stable predicates.

The Snapshot Algorithm

- Chandy and Lamport defined a snapshot algorithm to determine global states of distributed systems
- The goal of a snapshot is to record a set of process and channel states (a snapshot) for a set of processes so that, even if the combination of recorded states may not have occurred at the same time, the recorded global state is consistent
- The algorithm records states locally; it does not gather global states at one site.
- The snapshot algorithm has some assumptions
- Neither channels nor processes fail
- Reliable communications ensure every message sent is received exactly once
- Channels are unidirectional
- Messages are received in FIFO order
- There is a path between any two processes
- Any process may initiate a global snapshot at anytime
- Processes may continue to function normally during a snapshot

Snapshot Algorithm

- For each process, incoming channels are those which other processes can use to send it

messages. Outgoing channels are those it uses to send messages. Each process records its state and for each incoming channel a set of messages sent to it. The process records foreach channel, any messages sent after it recorded its state and before the sender recorded its own state. This approach can differentiate between states in terms of messages transmitted but not yet received

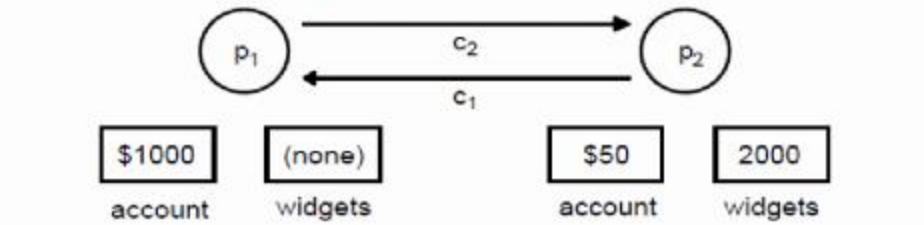
- The algorithm uses special marker messages, separate from other messages, which prompt the receiver to save its own state if it has not done so and which can be used to determine which messages to include in the channel state.
- The algorithm is determined by two rules

Example

•Figure 11.11 shows an initial state for two processes.

- •Figure 11.12 shows four successive states reached and identified after state transitions by the two processes.
- •Termination: it is assumed that all processes will have recorded their states and channel states a finite time after some process initially records its state.

Figure 11.11 Two processes and their initial states



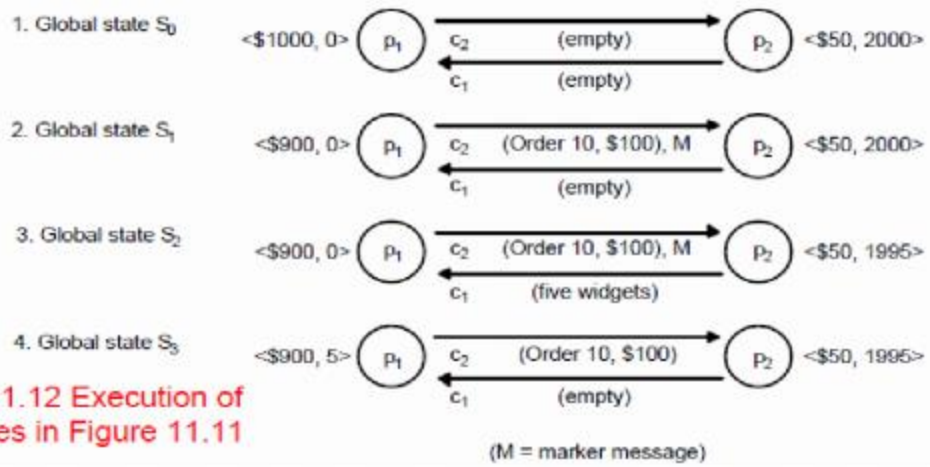


Figure 11.12 Execution of processes in Figure 11.11

(M = marker message)

Characterizing a state

- A snapshot selects a consistent cut from the history of the execution. Therefore the state recorded is consistent. This can be used in an ordering to include or exclude states that have or have not recorded their state before the cut. This allows us to distinguish events as pre-snap or post-snap events.
- The reachability of a state (figure 11.13) can be used to determine stable predicates.

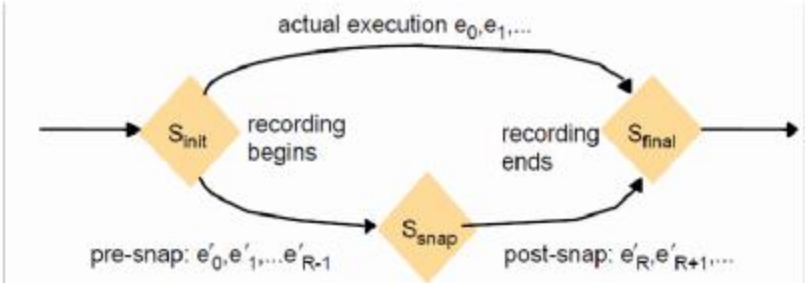


Figure 11.13 Reachability between states in the snapshot algorithm

Coordination And Agreement

Introduction

- Fundamental issue: for a set of processes, how to coordinate their actions or to agree on one or more values?
- even no fixed master-slave relationship between the components
- Further issue: how to consider and deal with failures when designing algorithms
- Topics covered
- mutual exclusion
- how to elect one of a collection of processes to perform a special role
- multicast communication
- agreement problem: consensus and byzantine agreement

Failure Assumptions and Failure Detectors

- Failure assumptions of this chapter
- Reliable communication channels
- Processes only fail by crashing unless state otherwise
- Failure detector: object/code in a process that detects failures of other processes
- unreliable failure detector
- One of two values: unsuspected or suspected
- Evidence of possible failures
- Example: most practical systems
- Each process sends -alive/I'm here‖ message to everyone else
- If not receiving -alive‖ message after timeout, it's suspected
- maybe function correctly, but network partitioned
- reliable failure detector
- One of two accurate values: unsuspected or failure – few practical systems

12.2 Distributed Mutual Exclusion

- Process coordination in a multitasking OS
- **Race condition**: several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place
- **critical section**: when one process is executing in a critical section, no other process is to be allowed to execute in its critical section
- **Mutual exclusion**: If a process is executing in its critical section, then no other processes can be executing in their critical sections
- Distributed mutual exclusion
- Provide critical region in a distributed environment
- message passing
- for example, locking files, locked daemon in UNIX (NFS is stateless, no file-locking at the NFS level)

Algorithms for mutual exclusion

- Problem: an asynchronous system of *N* processes
- processes don't fail
- message delivery is reliable; not share variables
- only one critical region
- application-level protocol: enter(), resourceAccesses(), exit()
- Requirements for mutual exclusion
- Essential
- [ME1] safety: only one process at a time
- [ME2] liveness: eventually enter or exit
- Additional
- [ME3] happened-before ordering: ordering of enter() is the same as HB ordering
- Performance evaluation
- overhead and bandwidth consumption: # of messages sent
- client delay incurred by a process at entry and exit
- throughput measured by synchronization delay: delay between one's exit and next's entry

A central server algorithm

- server keeps track of a token---permission to enter critical region
- a process requests the server for the token
- the server grants the token if it has the token
- a process can enter if it gets the token, otherwise waits
- when done, a process sends release and exits

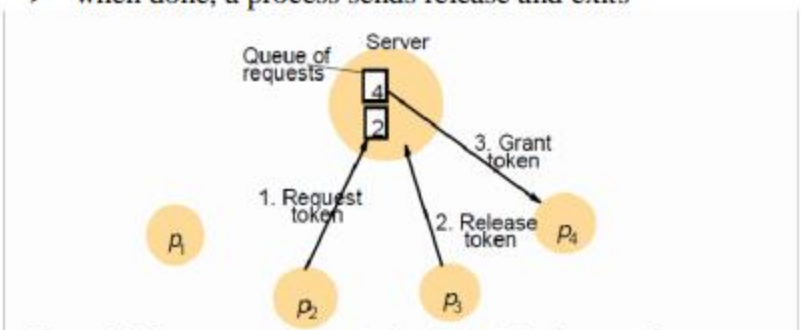


Figure 12.2 Server managing a mutual exclusion token for a set of processes

A central server algorithm: discussion

- Properties
- safety, why?
- liveness, why?
- HB ordering not guaranteed, why?
- Performance
- enter overhead: two messages (request and grant)
- enter delay: time between request and grant
- exit overhead: one message (release)
- exit delay: none
- synchronization delay: between release and grant
- centralized server is the bottleneck

A ring-based algorithm

- Arrange processes in a logical ring to rotate a token
- Wait for the token if it requires to enter the critical section
- The ring could be unrelated to the physical configuration
- *pi* sends messages to $p(i+1) \bmod N$
- when a process requires to enter the critical section, waits for the token
- when a process holds the token
- If it requires to enter the critical section, it can enter
- when a process releases a token (exit), it sends to its neighbor
- If it doesn‘t, just immediately forwards the token to its neighbor

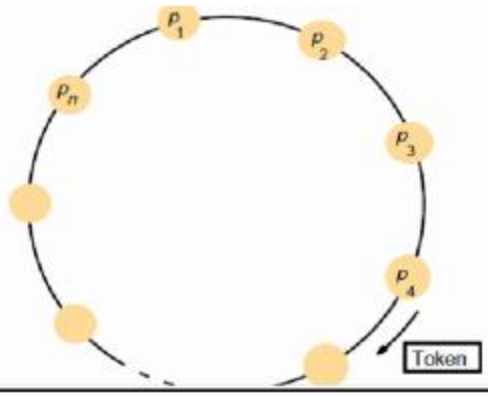


Figure 12.3 A ring of processes transferring a mutual exclusion token

An algorithm using multicast and logical clocks

- Multicast a request message for the token (Ricart and Agrawala [1981])
- enter only if all the other processes reply
- totally-ordered timestamps: $<T, pi>$
- Each process keeps a *state: RELEASED, HELD, WANTED*
- if all have *state* = *RELEASED*, all reply, a process can hold the token and enter
- if a process has *state* = *HELD*, doesn't reply until it exits
- if more than one process has *state* = *WANTED*, process with the lowest timestamp will get all
- *N*-1 replies first

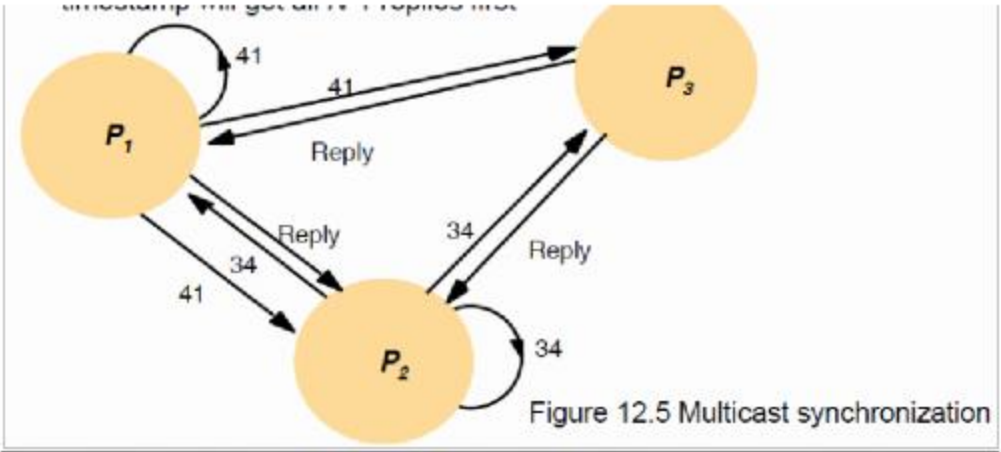


Figure 12.5 Multicast synchronization

Figure 12.4 Ricart and Agrawala's algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = $(N-1)$);
    state := HELD;

request processing deferred here

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

An algorithm using multicast: discussion

- •Properties
- safety, why?
- liveness, why?
- HB ordering, why?
- Performance
- bandwidth consumption: no token keeps circulating
- entry overhead: 2(N-1), why? [with multicast support: 1 + (N -1) = N]
- entry delay: delay between request and getting all replies
- exit overhead: 0 to N-1 messages
- exit delay: none
- synchronization delay: delay for 1 message (one last reply from the previous holder)

Maekawa's voting algorithm

- •Observation: not all peers to grant it access
- Only obtain permission from subsets, overlapped by any two processes
- •Maekawa's approach
- subsets Vi,Vj for process Pi, Pj
- Pi ∈ Vi, Pj ∈ Vj
- Vi ∩ Vj ≠ ∅ , there is at least one common member
- subset |Vi|=K, to be fair, each process should have the same size
- Pi cannot enter the critical section until it has received all K reply messages
- Choose a subset
- Simple way (2√N): place processes in a √N by √N matrix and let Vi be the union of the row and column containing Pi
- If P1, P2 and P3 concurrently request entry to the critical section, then its possible that each process has received one (itself) out of two replies, and none can proceed
- adapted and solved by [Saunders 1987]

Figure 12.6 Maekawa's algorithm

On initialization
  state := RELEASED;
  voted := FALSE;

For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;

On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;

On receipt of a release from p_i at p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

Elections

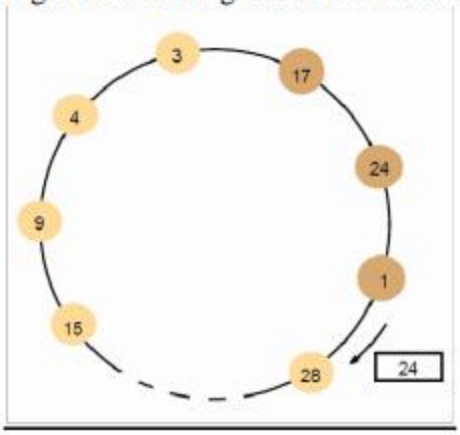
Election: choosing a unique process for a particular role

- All the processes agree on the ***unique*** choice
- For example, server in dist. Mutex assumptions
- Each process can call only one election at a time multiple concurrent elections can be called by different processes
- Participant: engages in an election each process *pi* has variable *electedi* = ? (don't know) initially process with the *largest* identifier wins.
- The (unique) identifier could be any useful value Properties
- [E1] *electedi* of a -participantl process must be P (elected process=largestid) or ⊥ (undefined)
- [E2] liveness: all processes participate and eventually set *electedi* != ⊥ (or crash) Performance
- overhead (bandwidth consumption): # of messages
- turnaround time: # of messages to complete an election

A ring-based election algorithm

- Arrange processes in a logical ring
  - *pi* sends messages to $p(i+1)$ mod N
  - It could be unrelated to the physical configuration
  - Elect the coordinator with the largest id
  - Assume no failures
- Initially, every process is a non-participant. Any process can call an election
  - Marks itself as participant
  - Places its id in an *election* message
  - Sends the message to its neighbor
  - Receiving an election message
- if *id* > *myid*, forward the msg, mark participant
- if *id* < *myid*
  - non-participant: replace *id* with *myid*: forward the msg, mark participant
  - participant: stop forwarding (why? Later, multiple elections)
- if *id* = *myid*, coordinator found, mark non-participant, *electedi* := *id*, send *elected*
  - message with *myid*
  - Receiving an elected message
- *id* != *myid*, mark non-participant, *electedi* := *id* forward the msg
- if *id* = *myid*, stop forwarding

Figure 12.7 A ring-based election in progress



- Receiving an election message:
- if *id* > *myid*, forward the msg, mark participant
- if *id* < *myid*
- non-participant: replace *id* with *myid*: forward the msg, mark participant
- participant: stop forwarding (why? Later, multiple elections)
- if *id* = *myid*, coordinator found, mark non-participant, *electedi* := *id*, send *elected* message with
- *myid*
- Receiving an elected message: – *id* != *myid,* mark non-participant,
- *electedi* := *id* forward the msg
- if *id* = *myid*, stop forwarding

A ring-based election algorithm: discussion

- •Properties
- safety: only the process with the largest id can send an *elected* message
- liveness: every process in the ring eventually participates in the election; extra elections are stopped
- Performance
- one election, best case, when?
- *N election* messages
- *N elected* messages
- turnaround: 2*N* messages
- one election, worst case, when?
- 2*N* - 1 election messages
- *N elected* messages
- turnaround: 3*N* - 1 messages
- can't tolerate failures, not very practical

The bully election algorithm

•Assumption

– Each process knows which processes have higher identifiers, and that it can communicate with all such processes

•Compare with ring-based election

– Processes can crash and be detected by timeouts

• synchronous

• timeout $T = 2T_{transmitting}$ (max transmission delay) + $T_{processing}$ (max processing delay)

•Three types of messages

– Election: announce an election

– Answer: in response to Election

– Coordinator: announce the identity of the elected process

The bully election algorithm: howto

• Start an election when detect the coordinator has failed or begin to replace the coordinator, which has lower identifier

– Send an election message to all processes with higher id's and waits for answers (except the failed coordinator/process)

• If no answers in time *T*

– Considers it is the coordinator

– sends coordinator message (with its id) to all processes with lower id's

• else

– waits for a coordinator message and starts an election if T' timeout

– To be a coordinator, it has to start an election

• A higher id process can replace the current coordinator (hence "bully")

– The highest one directly sends a coordinator message to all process with lower identifiers

• Receiving an election message

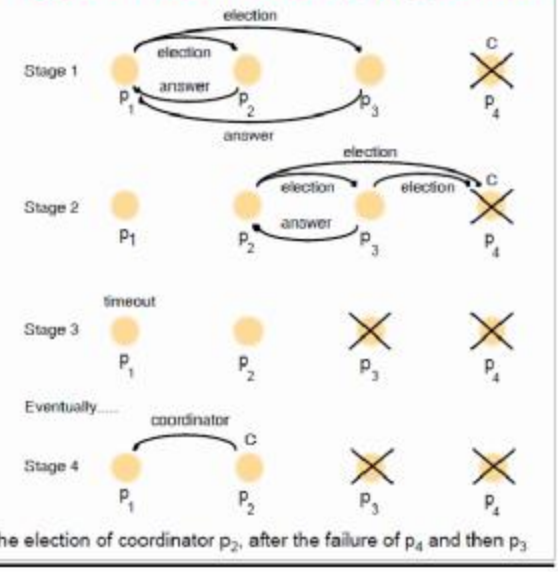
– sends an answer message back

– starts an election if it hasn't started one—send election messages to all higher-id processes (including the "failed" coordinator—the coordinator might be up by now)

• Receiving a coordinator message

– set *electedi* to the new coordinator

Figure 12.8 The bully algorithm



The election of coordinator p_2, after the failure of p_4 and then p_3

–

The bully election algorithm: discussion

- Properties
- safety:
- a lower-id process always yields to a higher-id process
- However, it's guaranteed
- if processes that have crashed are replaced by processes with the same identifier since message delivery order might not be guaranteed and
- failure detection might be unreliable
- liveness: all processes participate and know the coordinator at the end
- Performance
- best case: when?
- overhead: *N*-2 *coordinator* messages
- turnaround delay: no *election/answer* messages

Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
- The set of messages that every process of the group should receive
- On the delivery ordering across the group members
- Challenges
- Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
- Delivery guarantees ensure that operations are completed
- Types of group
- Static or dynamic: whether joining or leaving is considered Closed or open

- A group is said to be closed if only members of the group can multicast to it. Reliable

Multicast

- Simple basic multicasting (B-multicast) is sending a message to every process that is a member of a defined group
- B-multicast (g, m) for each process p ∈ group g, send (p, message m)
- On receive (m) at p: B-deliver (m) at p
- Reliable multicasting (R-multicast) requires these properties
- Integrity: a correct process sends a message to only a member of the group
- Validity: if a correct process sends a message, it will eventually bedelivered
- Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

Figure 12.10 Reliable multicast algorithm

```
On initialization
   Received := {};

For process p to R-multicast message m to group g
   B-multicast(g, m);        // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)
   if (m ∉ Received)
   then
            Received := Received ∪ {m};
            if (q ≠ p) then B-multicast(g, m); end if
            R-deliver m;
   end if
```

Types of message ordering

Three types of message ordering

– ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message before another, every correct process will deliver the first message before the other

– ***Casual ordering***: any correct process that delivers the second message will deliver the previous message first

– ***Total ordering***: if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first

•Note that

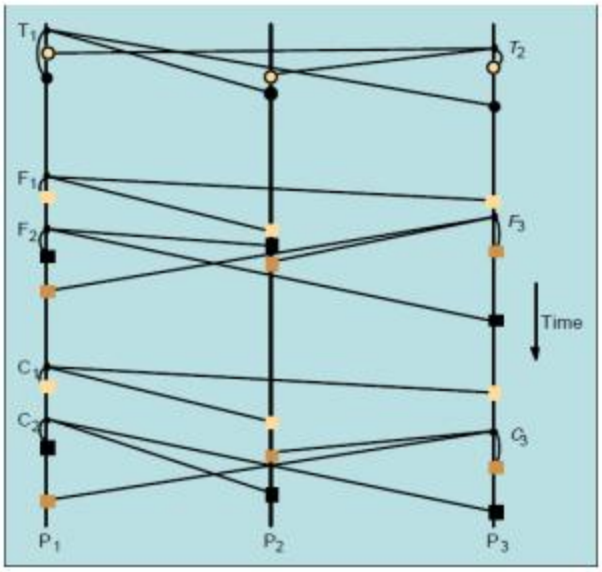
– FIFO ordering and casual ordering are only partial orders

– Not all messages are sent by the same sending process

– Some multicasts are concurrent, not able to be ordered by happened before

– Total order demands consistency, but not a particular order

Figure 12.12 Total, FIFO and causal ordering of multicast messages



Notice

- the consistent ordering of totally ordered messages *T1* and *T2*,
- the FIFO-related messages *F1* and *F2* and
- the causally related messages *C1* and *C3* and
- the otherwise arbitrary delivery ordering of messages

Note that *T1* and *T2* are delivered in opposite order to the physical time of message creation

Bulletin board example (FIFO ordering)

• A bulletin board such as Web Board at NJIT illustrates the desirability of consistency and FIFO ordering. A user can best refer to preceding messages if they are delivered in order. Message 25 in Figure 12.13 refers to message 24, and message 27 refers to message 23.

• Note the further advantage that Web Board allows by permitting messages to begin threads by replying to a particular message. Thus messages do not have to be displayed in the same order they are delivered

Bulletin board: *os.interesting*

Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 12.13 Display from bulletin board program

Implementing total ordering

• The normal approach to total ordering is to assign totally ordered identifiers to multicast messages, using the identifiers to make ordering decisions.
• One possible implementation is to use a sequencer process to assign identifiers. See Figure 12.14. A drawback of this is that the sequencer can become a bottleneck.
• An alternative is to have the processes collectively agree on identifiers. A simple algorithm is shown in Figure 12.15.

Figure 12.14 Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
  B-multicast($g \cup \{sequencer(g)\}$, <*m*, *i*>);

On B-deliver(<*m*, *i*>) *with* $g = group(m)$
  Place <*m*, *i*> in hold-back queue;

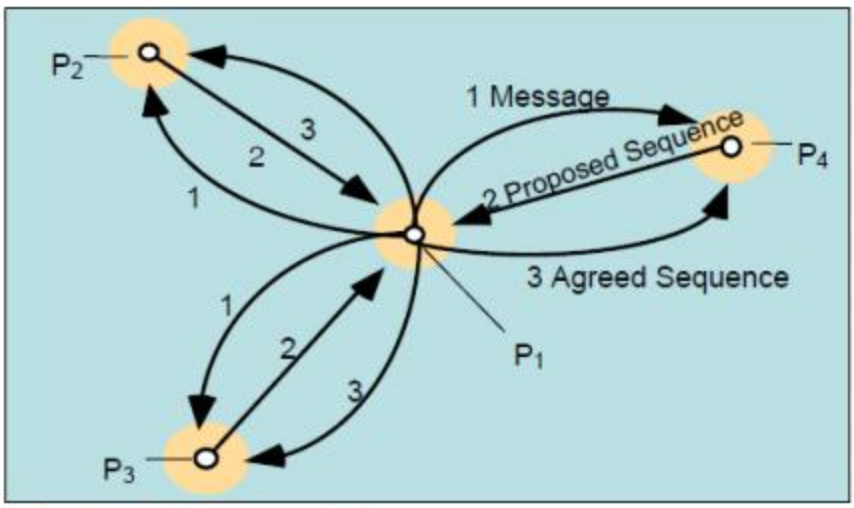
On B-deliver(m_{order} = <"order", *i*, *S*>) *with* $g = group(m_{order})$
  wait until <*m*, *i*> in hold-back queue and $S = r_g$;
  TO-deliver m; // (after deleting it from the hold-back queue)
  $r_g = S + 1$;

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver(<*m*, *i*>) *with* $g = group(m)$
  B-multicast(*g*, <"order", *i*, s_g>);
  $s_g := s_g + 1$;

Figure 12.15 The ISIS algorithm for total ordering



Each process q in group g keeps
• Aq g: the largest agreed sequence number it has observed so far for the group g
• Pq g: its own largest proposed sequence number

Algorithm for process p to multicast a message m to group g
1. B-multicasts <m, i> to g, where i is a unique identifier for m

2. Each process q replies to the sender p with a proposal for the message's agreed sequence number of Pq g :=Max(Aq g, Pq g)+1
3. Collects all the proposed sequence numbers and selects the largest one a as the next agreed sequence number. It then B-multicasts <i, a> to g.
4. Each process q in g sets Aq g := Max(Aq g, a) and attaches a to the message identified by i

Implementing casual ordering
• Causal ordering using vector timestamps (Figure 12.16)
– Only orders multicasts, and ignores one-to-one messages between processes
– Each process updates its vector timestamp before delivering a message to maintain the count of precedent messages

Algorithm for group member p_i $(i = 1, 2..., N)$

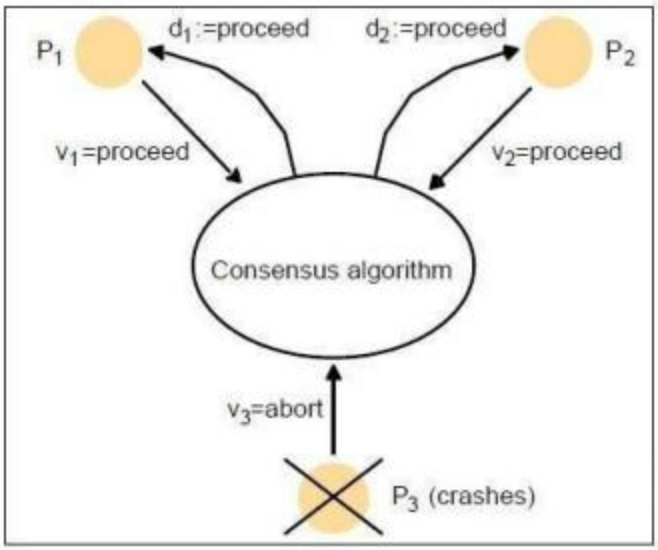
On initialization
$V_i^g[j] := 0$ $(j = 1, 2..., N)$;

To CO-multicast message m to group g
$V_i^g[i] := V_i^g[i] + 1$;
B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$
place $<V_j^g, m>$ in hold-back queue;
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;
CO-deliver m; // after removing it from the hold-back queue
$V_i^g[j] := V_i^g[j] + 1$;

Consensus and related problems
• Problems of agreement
– For processes to agree on a value (consensus) after one or more of the processes has proposed what that value should be
– Covered topics: byzantine generals, interactive consistency, totally ordered multicast
• The byzantine generals problem: a decision whether multiple armies should attack or retreat, assuming that united action will be more successful than some attacking and some retreating
• Another example might be space ship controllers deciding whether to proceed or abort. Failure handling during consensus is a key concern
• Assumptions
– communication (by message passing) is reliable
– processes may fail
• Sometimes up to f of the N processes are faulty

Consensus Process
1. Each process pi begins in an undecided state and proposes a single value vi, drawn from a set D (i=1…N)
2. Processes communicate with each other, exchanging values
3. Each process then sets the value of a decision variable di and enters the decided state



Two processes propose "proceed." One proposes "abort," but then crashes. The two remaining processes decide proceed.

Figure 12.17 Consensus for three processes

Requirements for Consensus

• Three requirements of a consensus algorithm
– ***Termination***: Eventually every correct process sets its decision variable
– ***Agreement***: The decision value of all correct processes is the same: if pi and pj are correct and have entered the *decided* state, then di=dj
(i,j=1,2, …, N)
– ***Integrity***: If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value

The byzantine generals problem

• Problem description
– Three or more generals must agree to *attack* or to *retreat*
– One general, the *commander*, issues the order
– Other generals, the *lieutenants*, must decide to attack or retreat
– One or more generals may be treacherous
• A *treacherous general* tells one general to attack and another to retreat
• Difference from consensus is that a single process supplies the value to agree on
• Requirements
– *Termination*: eventually each correct process sets its decision variable
– *Agreement*: the decision variable of all correct processes is the same
– *Integrity*: if the commander is correct, then all correct processes agree on the value that the commander has proposed (but the commander need not be correct)

The interactive consistency problem

• Interactive consistency: all correct processes agree on a vector of values, one for each process. This is called the decision vector
– Another variant of consensus
• Requirements
– *Termination*: eventually each correct process sets its decision variable
– *Agreement*: the decision vector of all correct processes is the same
– *Integrity*: if any process is correct, then all correct processes decide the correct value for that

process

Relating consensus to other problems

• Consensus (C), Byzantine Generals (BG), and Interactive Consensus (IC) are all problems concerned with making decisions in the context of arbitrary or crash failures

• We can sometimes generate solutions for one problem in terms of another. For example

– We can derive IC from BG by running BG N times, once for each process with that process acting as commander

– We can derive C from IC by running IC to produce a vector of values at each process, then applying a function to the vector's values to derive a single value.

– We can derive BG from C by

• Commander sends proposed value to itself and each remaining process

• All processes run C with received values

• They derive BG from the vector of C values

Consensus in a Synchronous System

• Up to f processes may have crash failures, all failures occurring during f+1 rounds. During each round, each of the correct processes multicasts the values among themselves

• The algorithm guarantees all surviving correct processes are in a position to agree

• Note: any process with f failures will require at least f+1 rounds to agree

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f + 1)$

B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round *r*)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f + 1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

Limits for solutions to Byzantine Generals

• Some cases of the Byzantine Generals problems have no solutions

– Lamport *et al* found that if there are only 3 processes, there is no solution

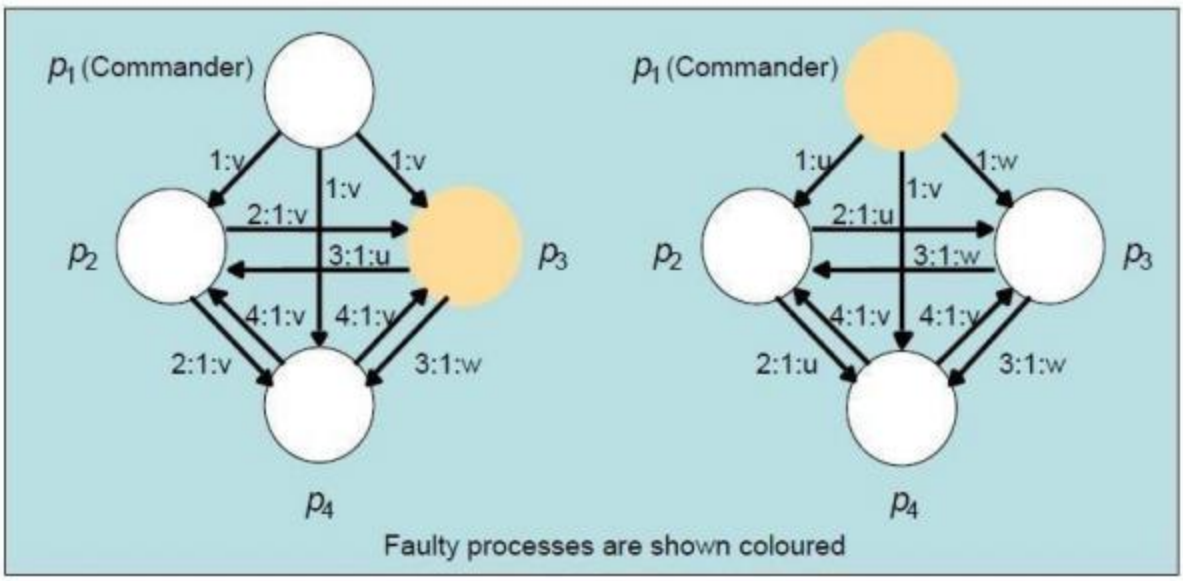
– Pease *et al* found that if the total number of processes is less than three times the number of failures plus one, there is no solution

• Thus there is a solution with 4 processes and 1 failure, if there are two rounds

– In the first, the commander sends the values

– while in the second, each lieutenant sends the values it received

Figure 12.20 Four Byzantine generals



Asynchronous Systems

• All solutions to consistency and Byzantine generals problems are limited to synchronous systems

• Fischer *et al* found that there are no solutions in an asynchronous system with even one failure

• This impossibility is circumvented by *masking faults* or using *failure detection*

• There is also a partial solution, assuming an *adversary* process, based on *introducing random values* in the process to prevent an effective thwarting strategy. This does not always reach consensus