LECTURE NOTES

ON

DISTRIBUTED SYSTEMS

R18CSE4261

B.TECH. - IV YEAR – I SEMESTER

SRI INDU COLLEGE OF ENGINEERING &TECHNOLOGY
(An Autonomous Institution under UGC, New Delhi)

SRI INDU COLLEGE OF ENGINEERING &TECHNOLOGY

(An Autonomous Institution under UGC, New Delhi)

B.Tech. - IV Year – I Semester **L T P C 3 0 0 3**

(R18CSE4261) Distributed Systems

UNIT 1:
Characterization of Distributed Systems: Introduction, Examples of distributed Systems, Resource Sharing and the web, Challenges.
Systems Models: Introduction, Architectural models and Fundamental models.

UNIT 2: **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.

UNIT 3: **Inter process Communication:** Introduction, The API for the Internet protocols, External data Representation and marshaling, Client-Server Communication, Group Communication, Case study: IPC in UNIX.
Distributed objects and Remote Invocation: Introduction, Communication between distributed objects, Remote Procedure call, Events and notifications, Case study: JAVA RMI.

UNIT 4: **Distributed File Systems**: Introduction, File Service architecture, case Study1: SUN network file Systems, Case Study 2: The Andrew File System.
Name Services: Introduction, Name Services and the Domain Name System, Directory Services, Case study of the Global name Service.
Distributed Shared Memory: Introduction, Design and Implementation issues, Sequential consistency and IVY case study, Release consistency and munin case study, Other consistency models.

UNIT 5: **Transaction and Concurrency control:** Introduction, Transactions, nested Transactions, Locks, Optimistic concurrency control, Timestamp ordering, Comparison of methods for concurrency control.
Distributed Transactions: Introduction, Flat and Nested Distributed Transactions, Atomic commit protocols, Concurrency control in distributed trasactions, Distributed transactions, Distributed deadlocks, Transaction recovery.

Text Book:

TEXT BOOKS: 1. Distributed Systems, Concepts and Design, G Coulouris, J Dollimore and T Kindberg, Pearson Education, 4TH Edition, 2009.

REFERENCES:

1. Distributed Systems: Principles and Paradigms, S. Tanenbaum and Maarten Van Steen, 2nd Edition, PHI.
2. Distributed Systems, An Algorithm Approach, Sukumar Ghosh, Chapman & Hali/CRC, Taylor & Fransis Group, 2007.

Outcomes

- Understand foundations of Distributed Systems.
- Introduce the idea of peer to peer services and file system.
- Understand in detail the system level and support required for distributed system.
- Understand the issues involved in studying process and resource management.

INDEX

UNIT NO	TOPIC	PAGE NO
I	Characterization of Distributed Systems	01 - 22
	System Models	23 - 36
II	Time and Global States	37 - 50
	Coordination and Agreement	51 - 66
III	Inter Process Communication	67 - 83
	Distributed Objects and Remote Invocation	84 - 128
IV	Distributed File Systems	129 - 144
	Name Services	145 - 159
	Distributed Shared Memory	160 - 169
V	Transactions and Concurrency Control	170 - 180
	Distributed Transactions	181 - 194

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

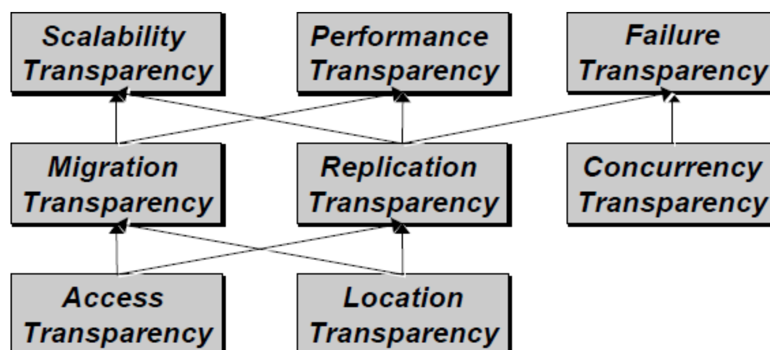
Hardware, software and networks fail!

- 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

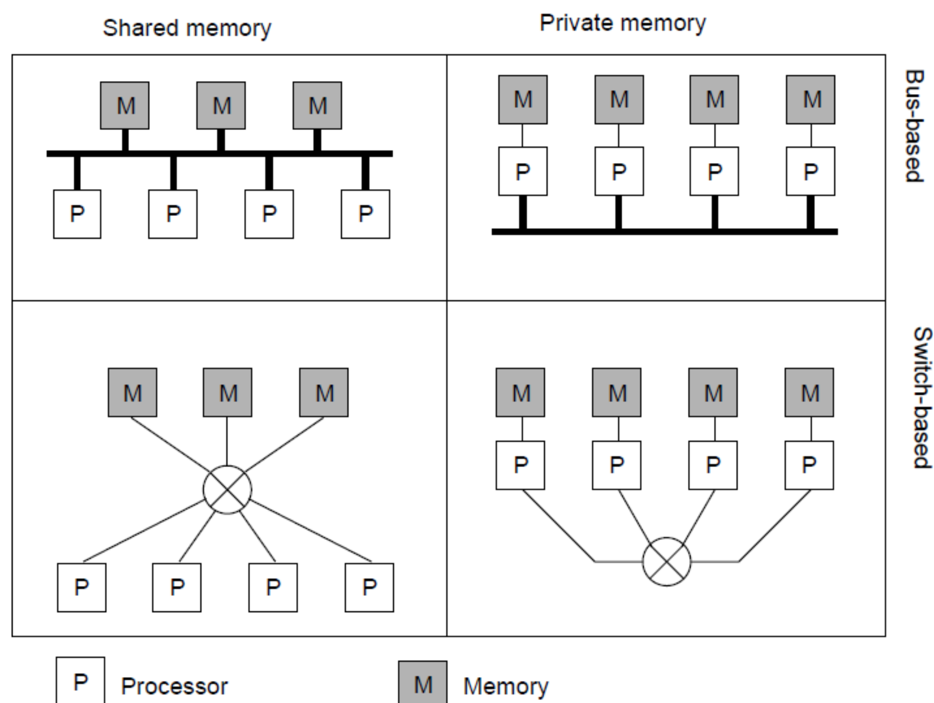
Distributed Systems: Hardware Concepts

- 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

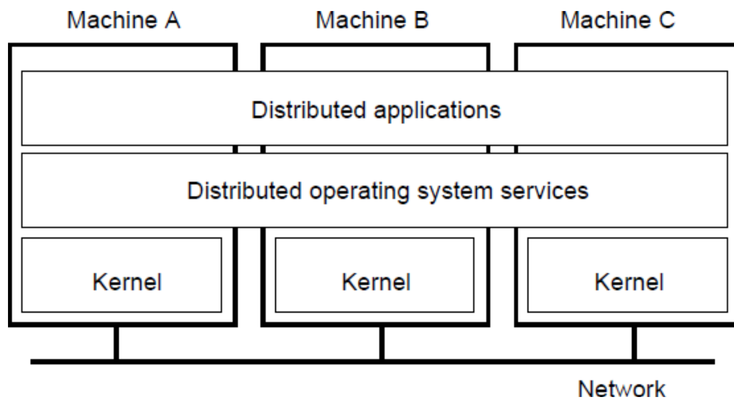
Distributed operating system
_ Network operating system
_ Middleware

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

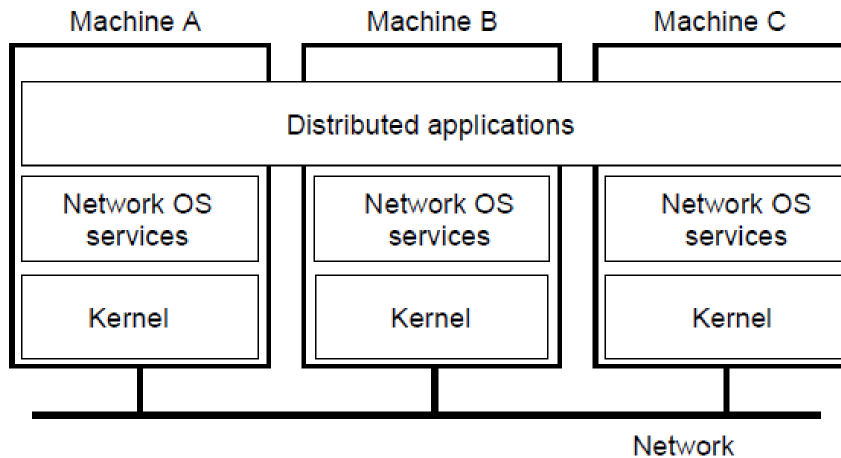
Distributed Operating System

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

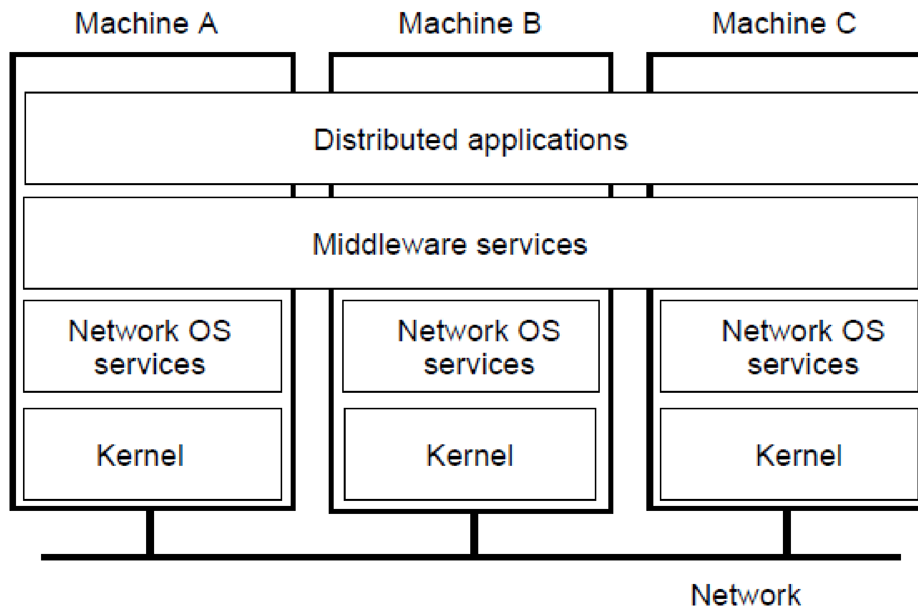


Network Operating System
Some characteristics:
_ Each computer has its own operating system with networking facilities
_ Computers work independently (i.e., they may even have different operating systems)
_ Services are tied to individual nodes (ftp, telnet, WWW)
_ Highly file oriented (basically, processors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

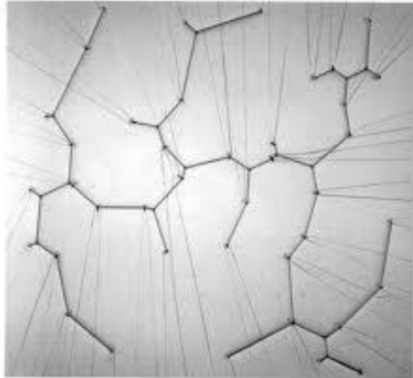
Communication services: Abandon primitive socket based message passing in favor of:
_ Procedure calls across networks
_ Remote-object method invocation
_ Message-queuing systems
_ Advanced communication streams
_ Event notification service
Information system services: Services that help manage data in 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

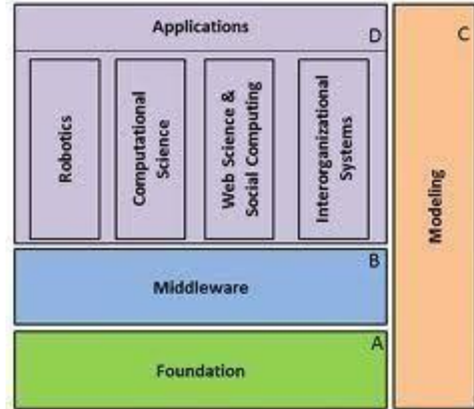
Control services: Services giving applications control over when, where, and how they access data:

_ 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

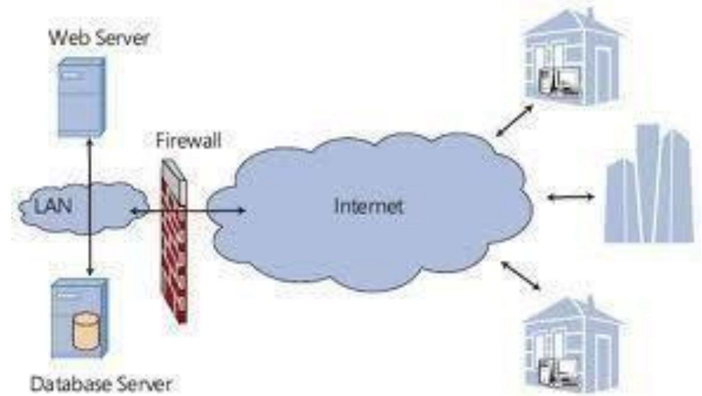
Item	Distributed OS		Network OS	Middle-ware DS
	multiproc.	multicomp.		
1	Very High	High	Low	High
2	Yes	Yes	No	No
3	1	N	N	N
4	Shared memory	Messages	Files	Model specific
5	Global, central	Global, distributed	Per node	Per node
6	No	Moderately	Yes	Varies
7	Closed	Closed	Open	Open



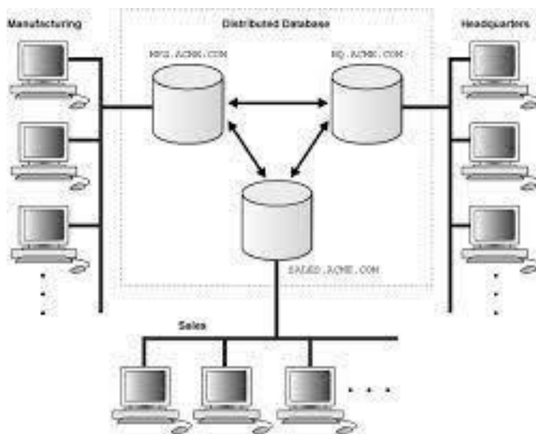


Applications
D
Robotics
Computational Science
Web Science & Social Computing
Interorganizational Systems
C
Modeling
B
Middleware
A
Foundation

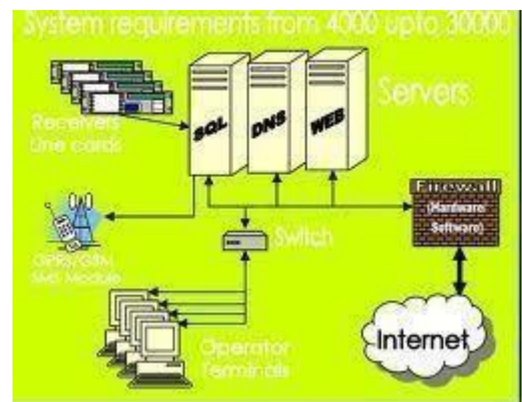
Grid Computing



Web Server
Firewall
LAN
Internet
Database Server

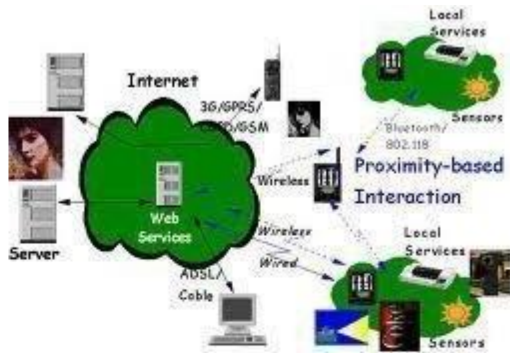


Manufacturing
Distributed Database
Headquarters
MFG.ACME.COM
HQ.ACME.COM
SALES.ACME.COM
Sales



System requirements from 4000 upto 30000
Servers
SQL
DNS
WEB
Firewall
Switch
Operator Terminals
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

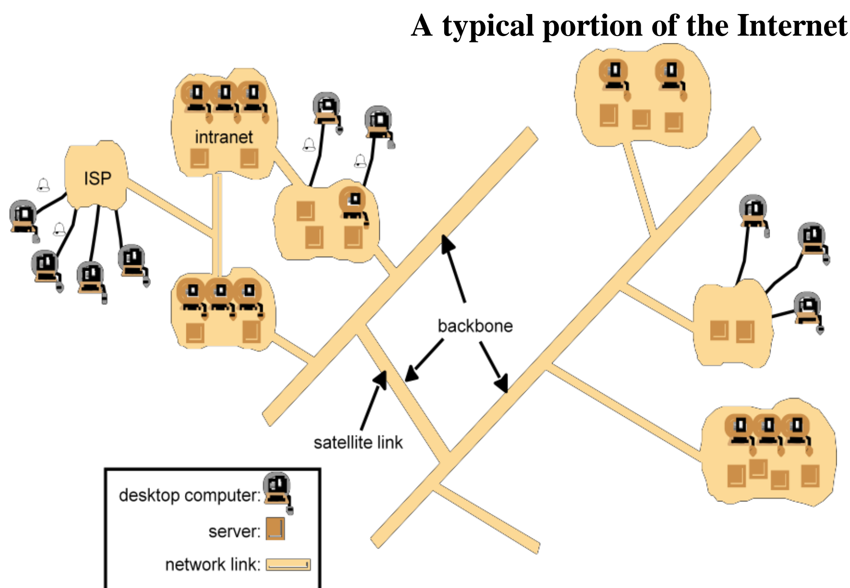
a number of influential trends:

- 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

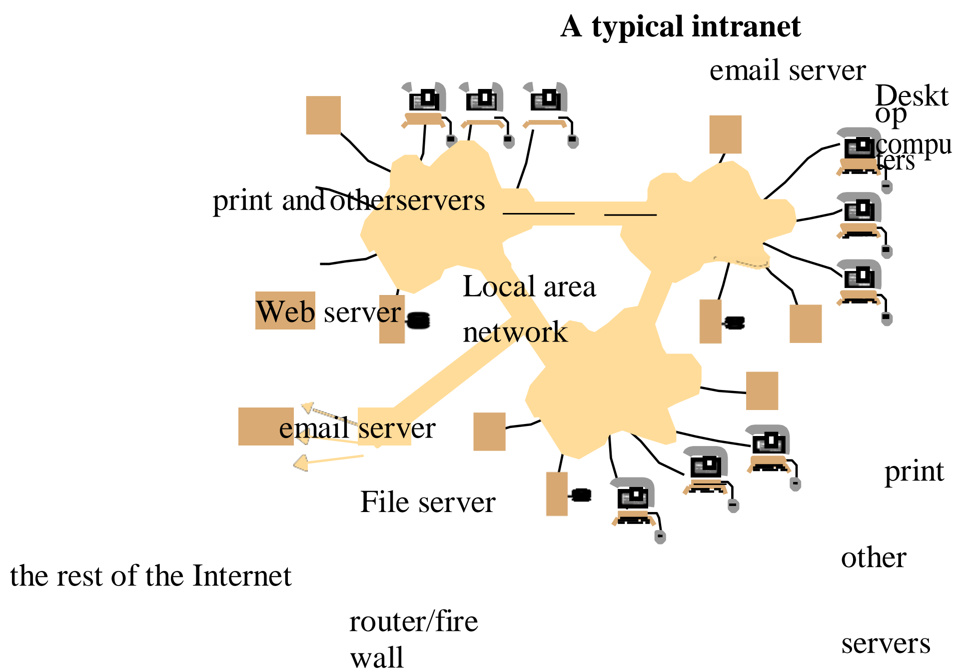
Computers vs. Web servers in the Internet

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

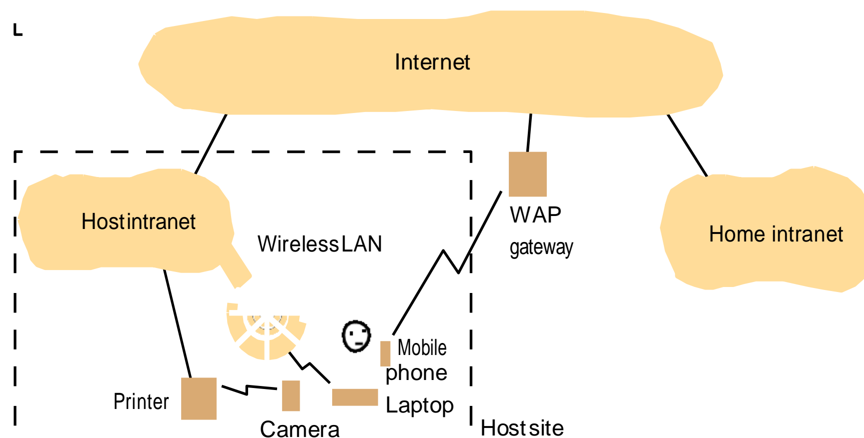
Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

- 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.

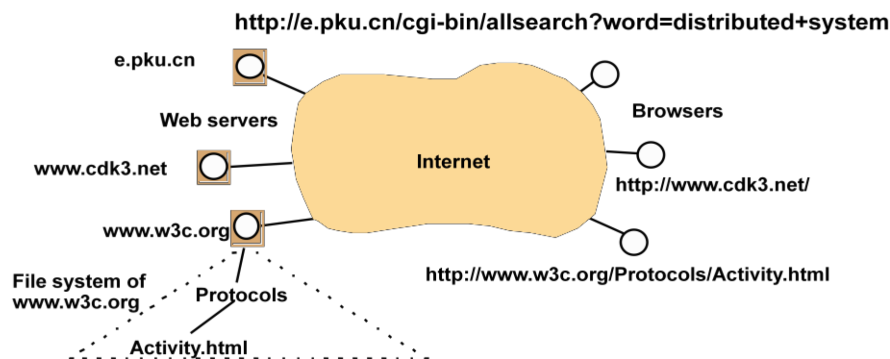
-The browser interprets the HTML text

-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:

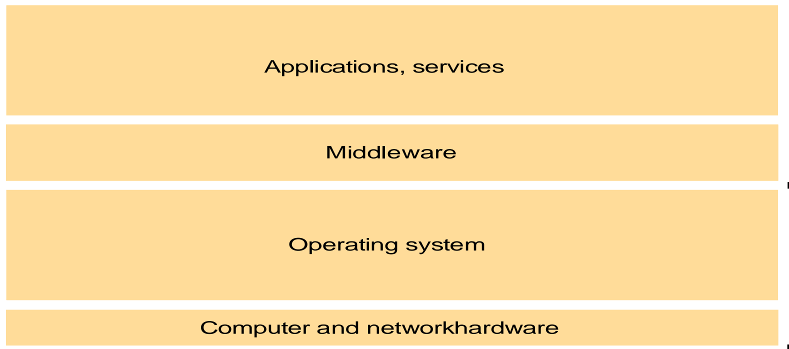
A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates 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

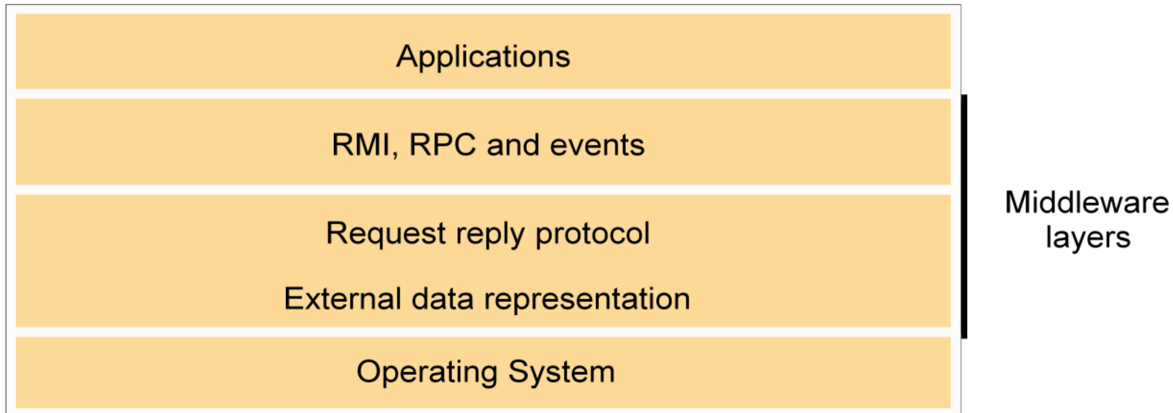
Middleware

- 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

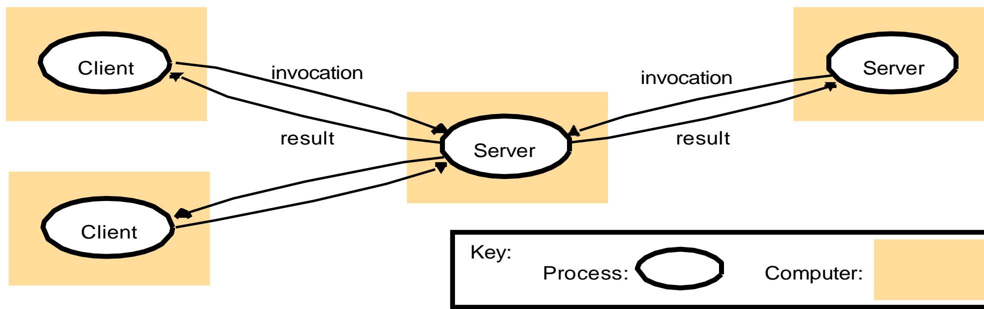
– An interface defines the signatures of a set of methods, including arguments, argument types, return values and exceptions. Implementation details are not included in an interface. A class may implement an interface by specifying behavior for each method in the interface. Interfaces do not have constructors.

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.

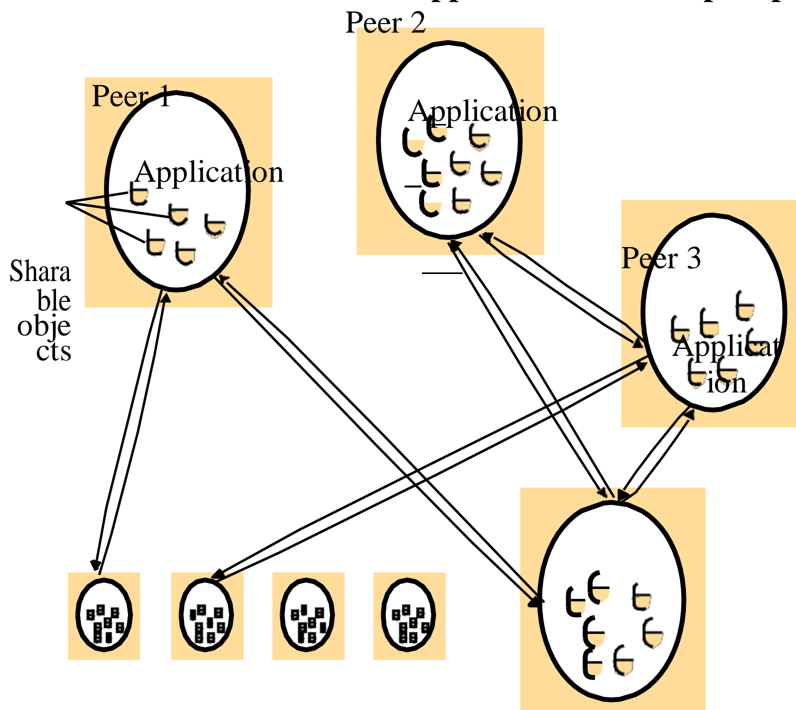
Clients invoke individual servers



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:

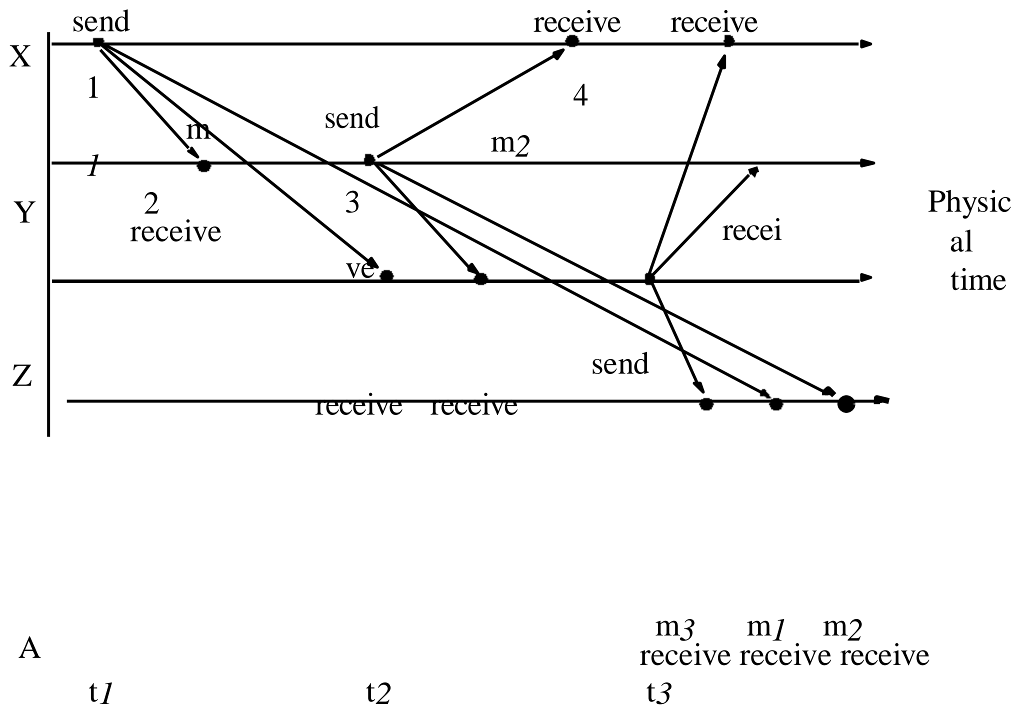
Synchronous distributed systems: Hadzilacos and Toueg define a synchronous distributed system to be one in which the following bounds are defined:
• The time to execute each step of a process has known lower and upper bounds.
• Each message transmitted over a channel is received within a known bounded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



Failure model

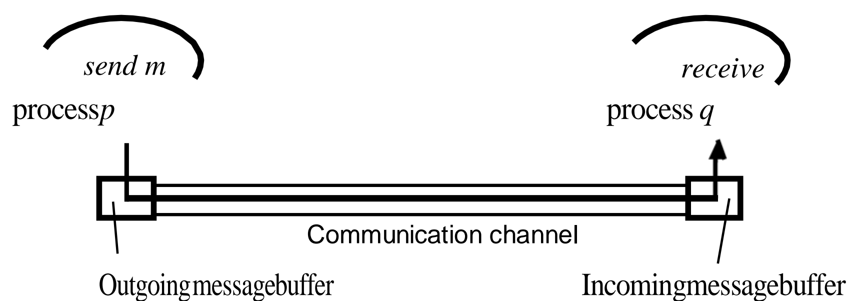
In a distributed system both processes and communication channels may fail – that is, they may depart from what is considered to be correct or desirable behaviour. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. Hadzilacos 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.

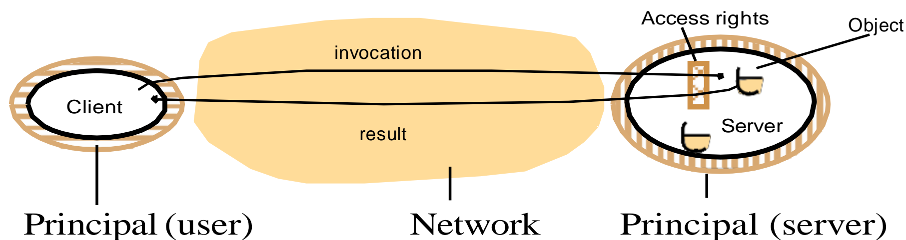
Security model

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

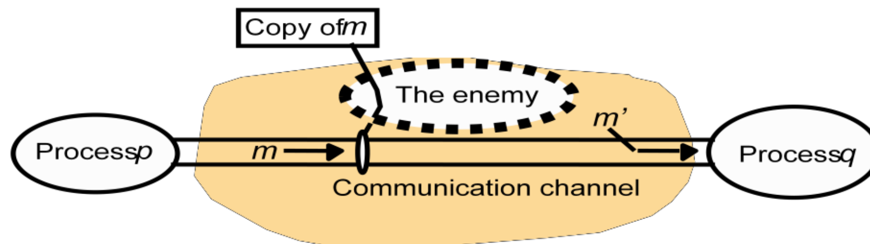
Protecting objects :

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.

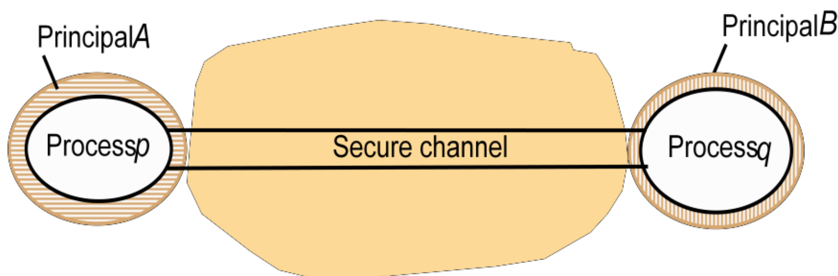
Authentication: The use of shared secrets and encryption provides the basis for the *authentication* of messages – proving the identities supplied by their senders. The basic authentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to 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.

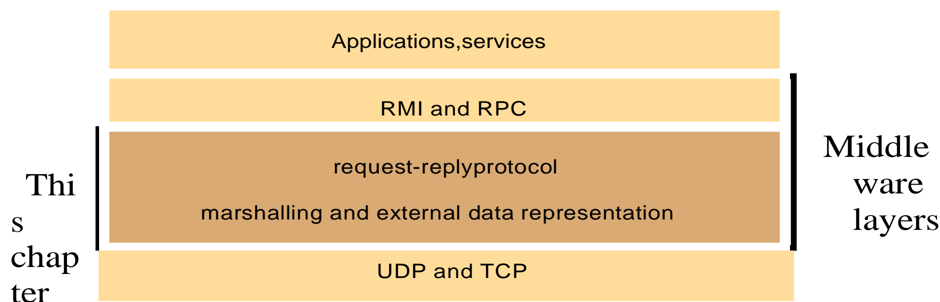
Secure channels: Encryption and authentication are used to build secure channels as a service layer on top of existing communication services. A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal, as shown in the following figure. A secure 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 process*pi* : *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* := *maxVij 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 =<e*i*0, *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

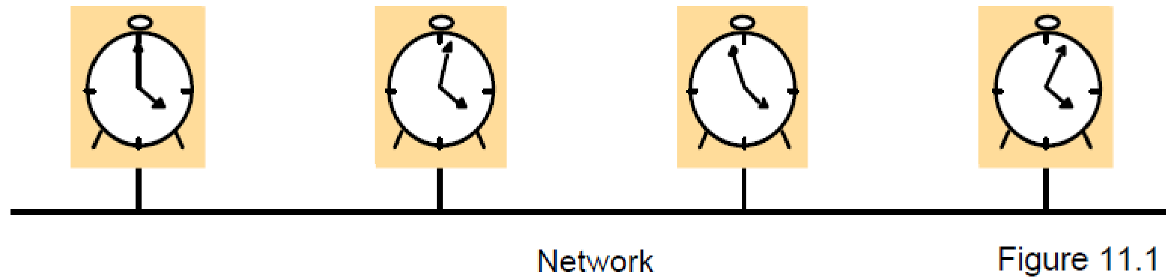


Figure 11.1

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 $2D$ *(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 $p1$ sends its local time t to process $p2$ in a message m
- $p2$ could set its clock to $t + T$trans where Ttrans is the time to transmit m
- Ttrans is unknown but $min \leq T$trans $\leq max$
- uncertainty $u = max\text{-}min$. Set clock to $t + (max - min)/2$ then skew $\leq u/2$

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 ± (Tround/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 $[t+min, t + T\text{round} - min]$
- the width of the range is Tround + 2min

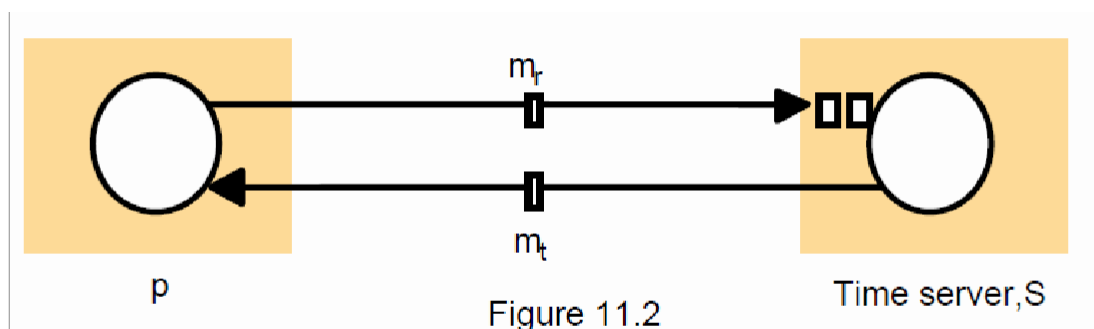


Figure 11.2

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

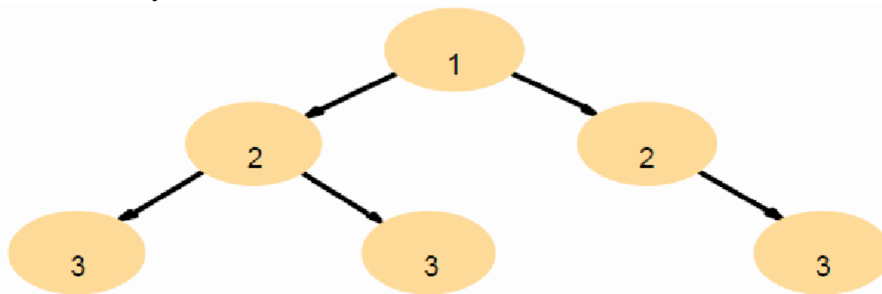


Figure 11.3 An example synchronization subnet in an NTP implementation

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)

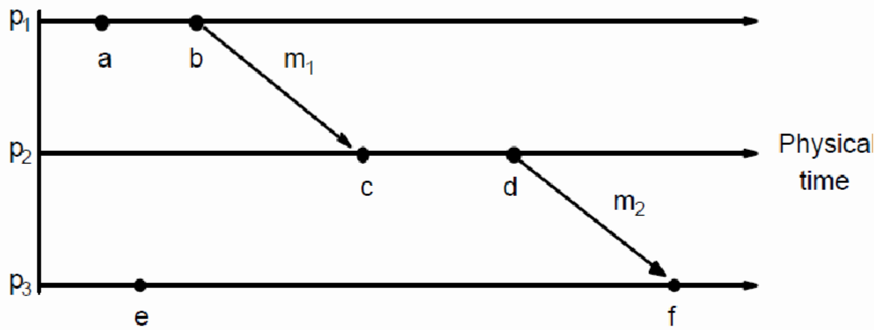
- Higher accuracy. Useful if no hardware multicast.

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*')
- $Ti\text{-}2 = Ti\text{-}3 + t + o$ and $Ti = Ti\text{-}1 + t' - o$
- This gives us (by adding the equations) : $di = t + t' = Ti\text{-}2 - Ti\text{-}3 + Ti - Ti\text{-}1$
- Also (by subtracting the equations)
  - $= oi + (t' - t)/2$ where $oi = (Ti\text{-}2 - Ti\text{-}3 + Ti\text{-}1 - Ti)/2$
- Using the fact that *t*, *t*'>0 it can be shown that
- $oi - di/2 \leq o \leq oi + di/2$.
- 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)

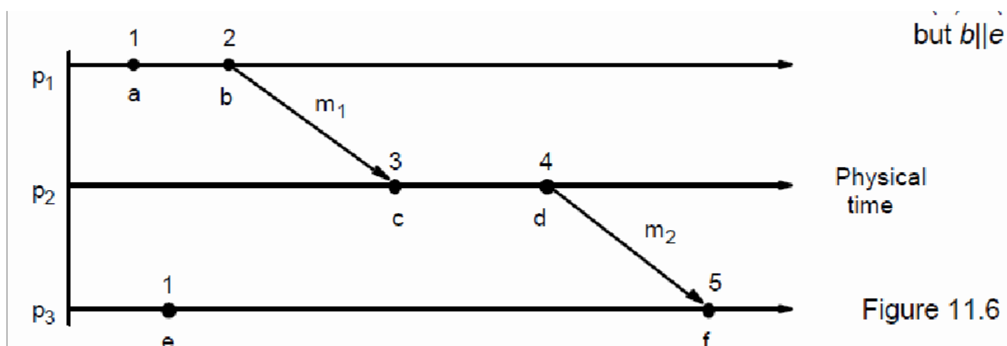
Logical time and logical clocks

- 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 $e \rightarrow_i e'$ in process *p*i, then $e \rightarrow e'$
- HB2: for any message *m*, send(*m*) → receive(*m*)
- HB3: if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$



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 *L*j := *max*(*L*j, *t*) and applies LC1 before timestamping the event *receive* (*m*)
- *e* →*e*' implies *L*(*e*)<*L*(*e*'), but *L*(*e*)<*L*(*e'*) does not imply *e*→*e*'



Totally ordered logical clocks

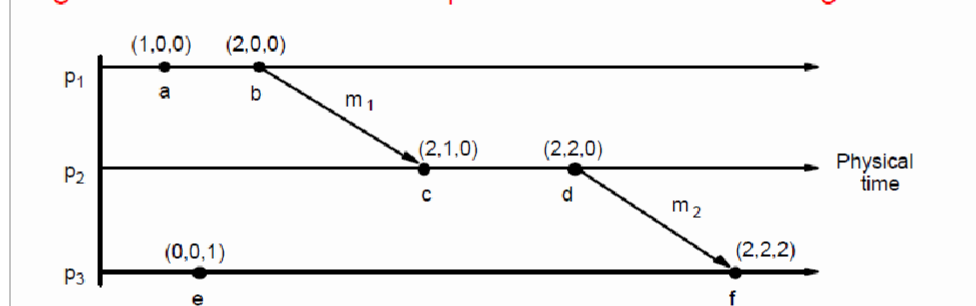
- Some pairs of distinct events, generated by different processes, may have numerically identical Lamport timestamps
- Different processes may have same Lamport time
- Totally ordered logical clocks
- If e is an event occurring at pi with local timestamp Ti, and if e' is an event occurring at pj with 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* → *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
- *Viji*] (*j*≠ *i*) is the number of events at *pj* that *pi* has been affected by VC1: Initially, *Vi*[*j*] := 0 for *pi*, *j*=1.. *N* (*N* processes)
- 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)

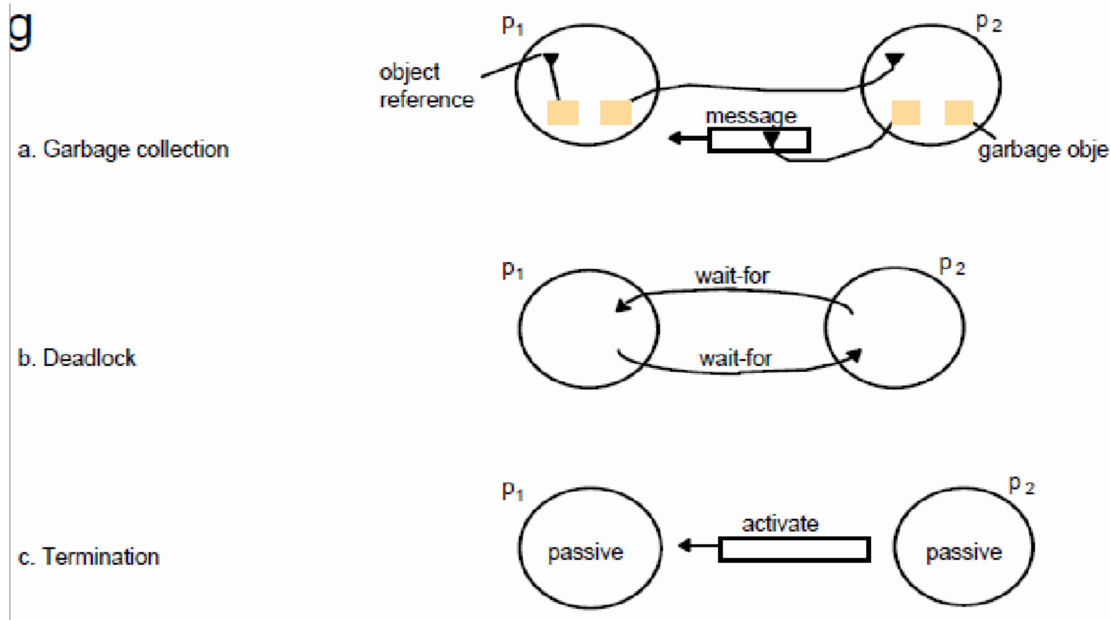
Figure 11.7 Vector timestamps for events shown in Figure 11.5



- 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*→*f* since V(a) < V(f)
- c || e since neither V(c) <= V(e) nor V(e) <= V(c)

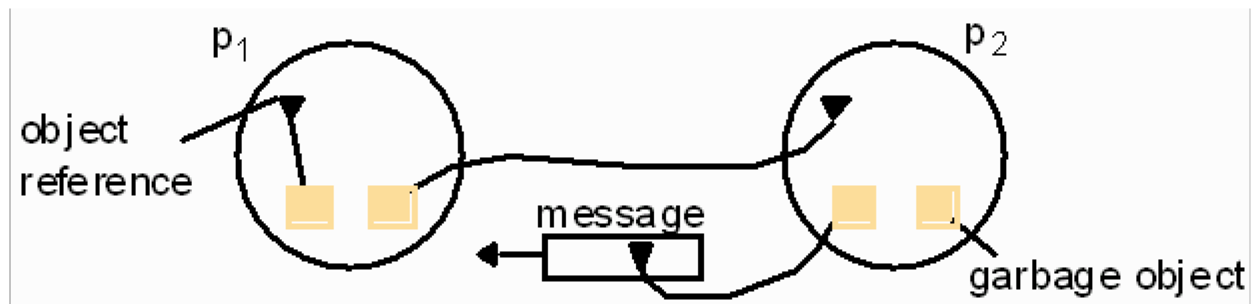
Global states

- 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



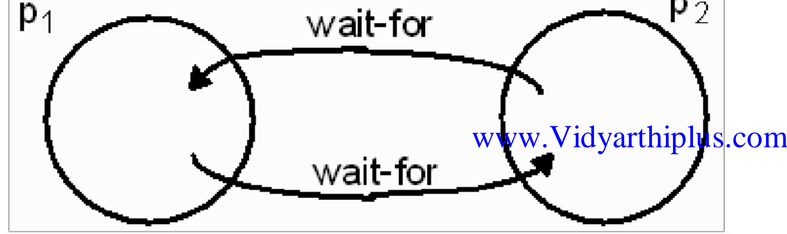
Distributed Garbage Collection

- 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

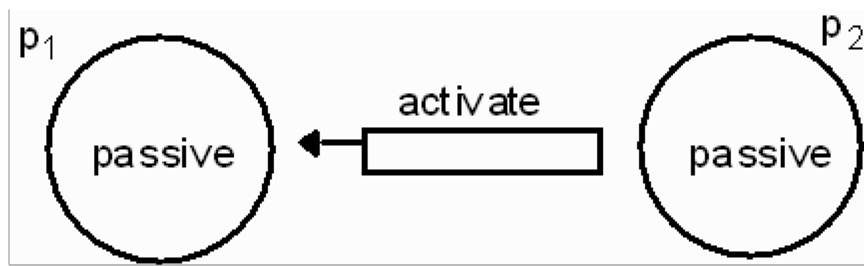
- A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and there is a cycle in the graph of the ***waits-for*** relationship
- In figure 11.8b, both p1 and p2 wait for a message from the other, so both are blocked and the system cannot continue



www.Vidyarthiplus.com

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*



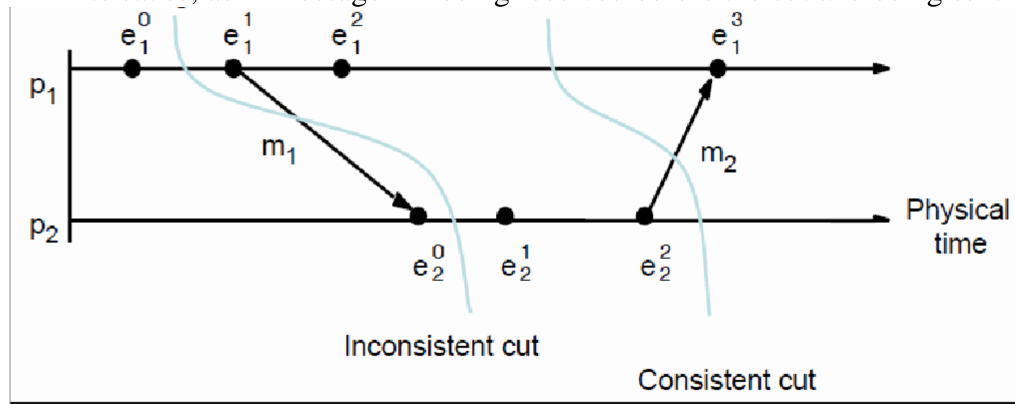
Distributed Debugging

- 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 our capabilities
- 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 any time
- 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

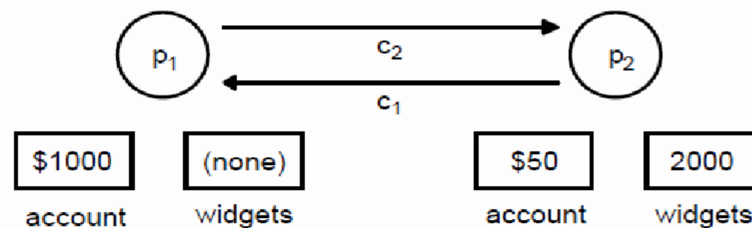
- The algorithm uses special marker messages, separate from other messages, which prompt the receiver to save its own state if it has not done so and which can be used to determine which messages to include in the channel state.
- The algorithm is determined by two rules

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



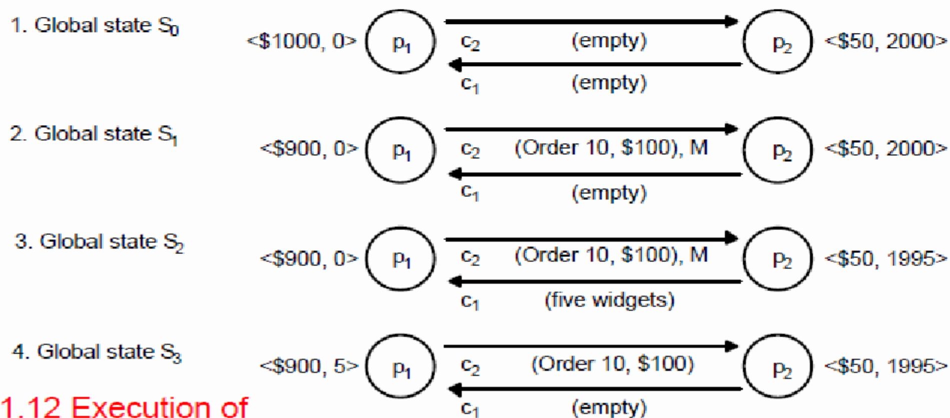


Figure 11.12 Execution of processes in Figure 11.11

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.

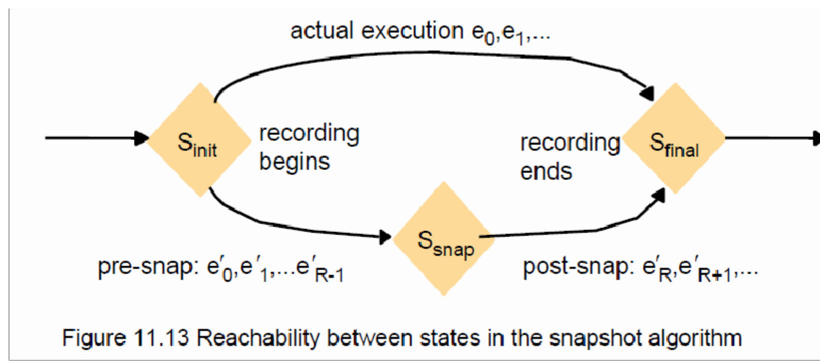


Figure 11.13 Reachability between states in the snapshot algorithm

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

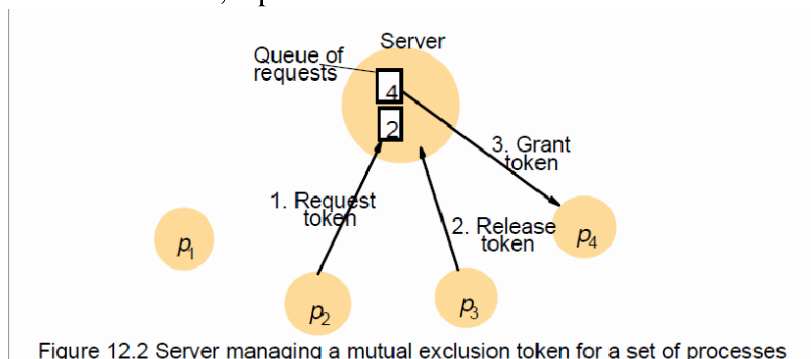


Figure 12.2 Server managing a mutual exclusion token for a set of processes

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) mod *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

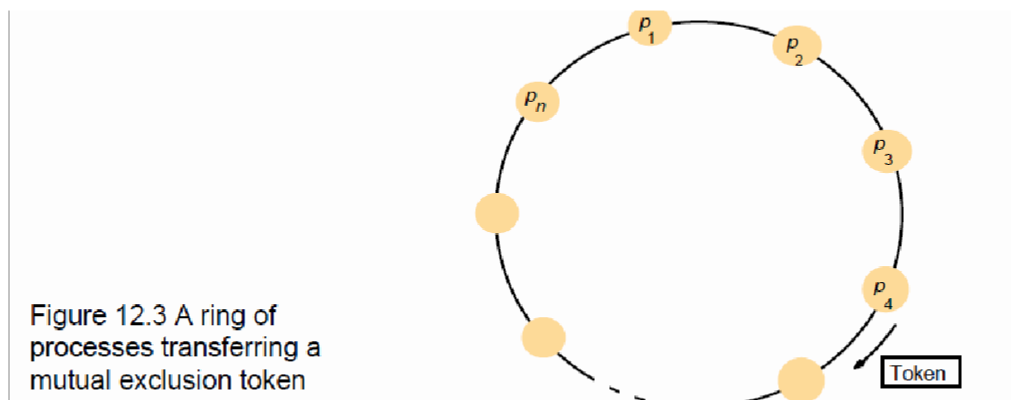


Figure 12.3 A ring of processes transferring a mutual exclusion token

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

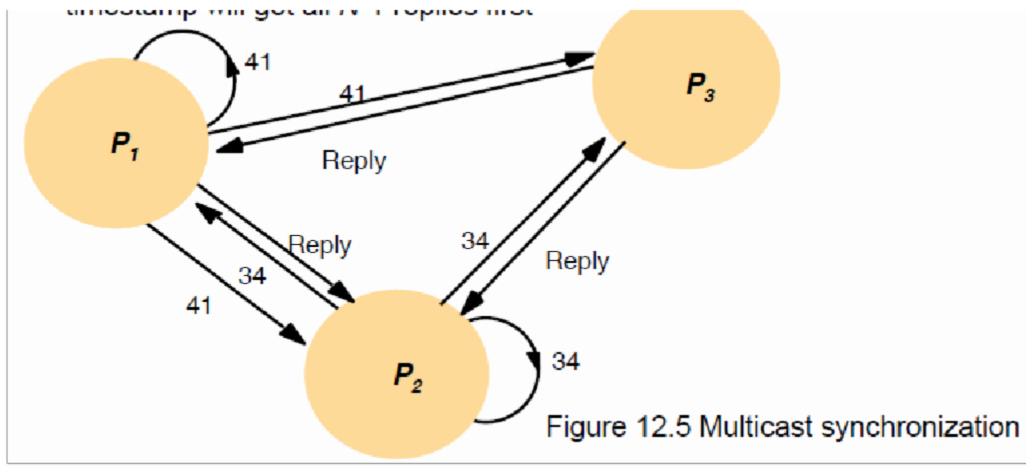


Figure 12.5 Multicast synchronization

Figure 12.4 Ricart and Agrawala's algorithm

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

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

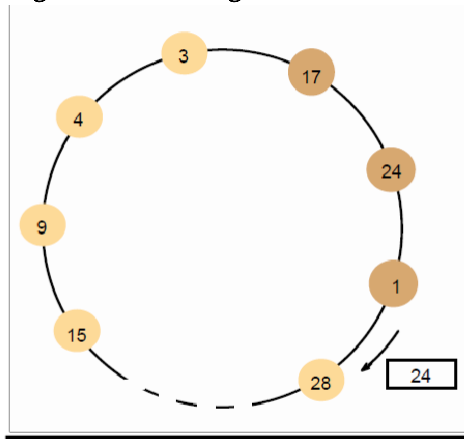
Election: choosing a unique process for a particular role

- 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 –participant‖ 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

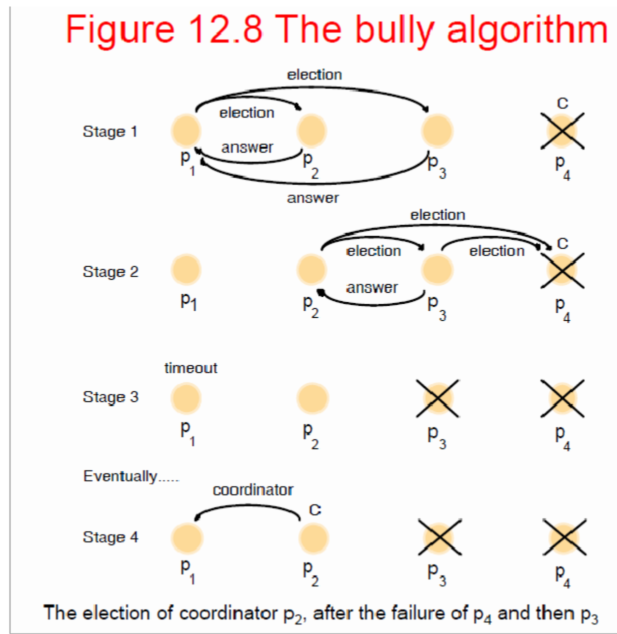
• Receiving an election message

– 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 $elected_i$ to the new coordinator



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*);   // $p \in g$ 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

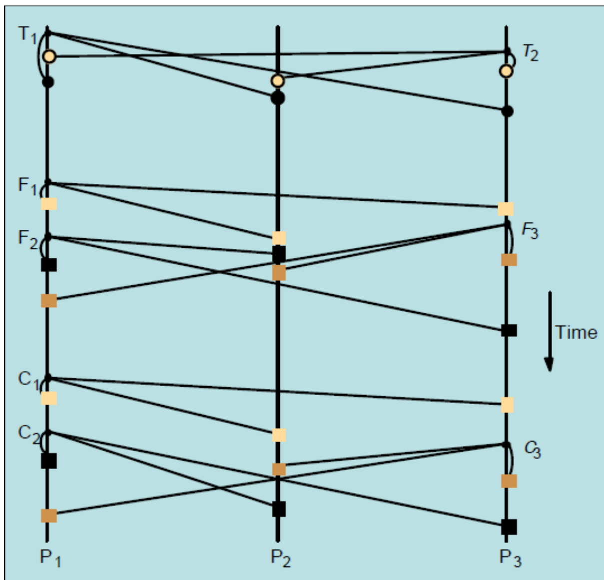
– FIFO ordering and casual ordering are only partial orders

– 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;

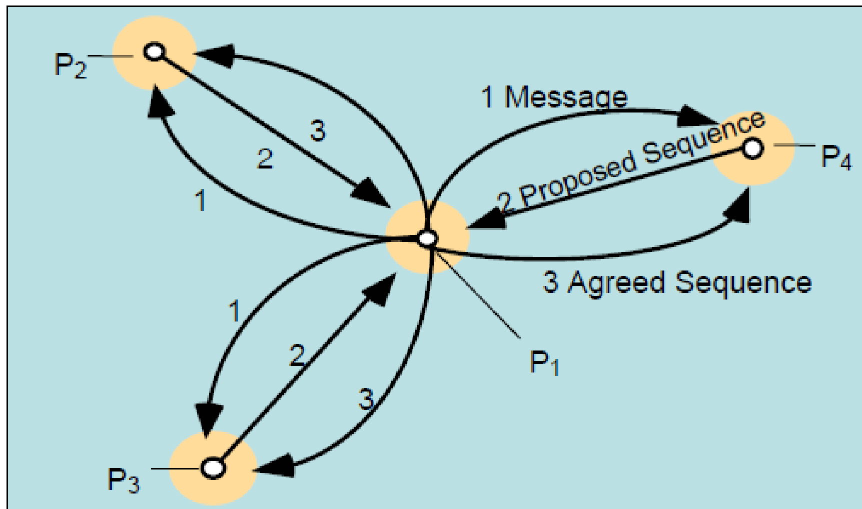
On B-deliver$(m_{order} = <\text{"order"}, i, S>)$ *with* $g = group(m_{order})$
  wait until $<m, i>$ in hold-back queue and $S = r_g$;
  TO-deliver m;   // (after deleting it from the hold-back queue)
  $r_g = S + 1$;

2. Algorithm for sequencer of *g*

On initialization: $s_g := 0$;

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

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 $(i = 1, 2..., N)$

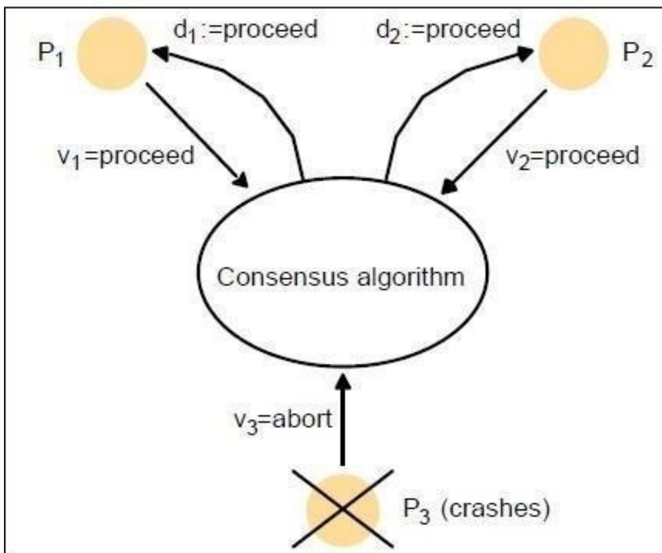
On initialization
$V_i^g[j] := 0$ $(j = 1, 2..., N)$;

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 $V_j^g[k] \le V_i^g[k]$ $(k \ne j)$;
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
  $Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

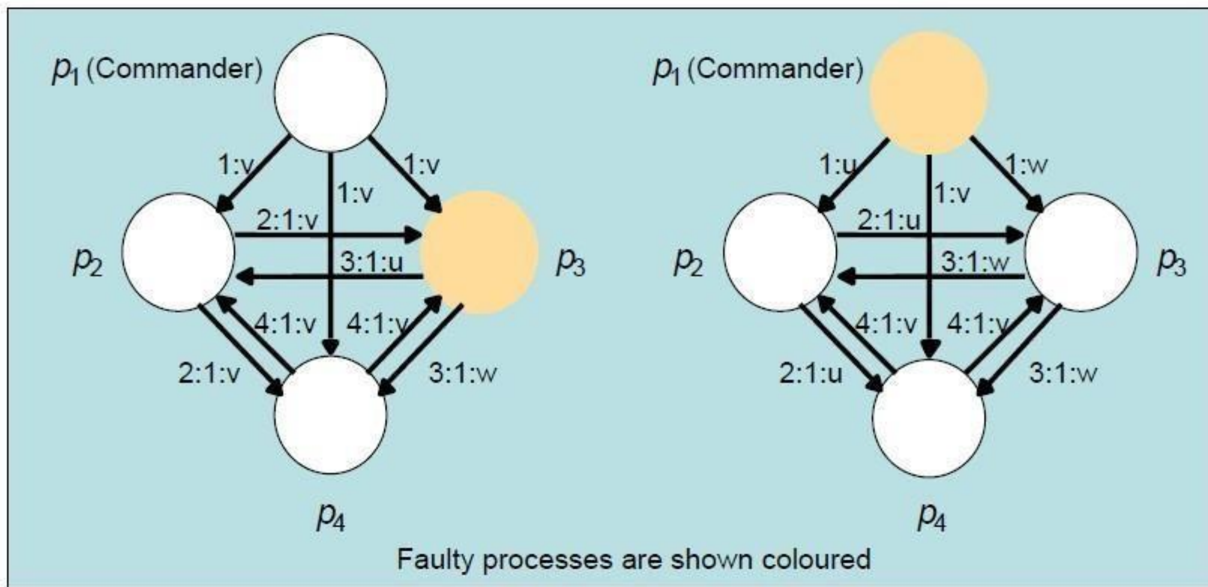
In round r $(1 \le r \le f+1)$
  B-multicast(g, $Values_i^r - Values_i^{r-1}$); // Send only values that have not been sent
  $Values_i^{r+1} := Values_i^r$;
  while (in round r)
  {
      On B-deliver(V_j) *from some* p_j
        $Values_i^{r+1} := Values_i^{r+1} \cup V_j$;
  }

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

UNIT III

Inter Process Communication: Introduction, The API for the internet protocols, External Data Representation and Marshalling, Client-Server Communication, Group Communication, Case Study: IPC in UNIX.
Distributed Objects and Remote Invocation: Introduction, Communication between Distributed Objects, Remote Procedure Call, Events and Notifications, Case study-Java RMI.

Application program interface

The application program interface to UDP provides a *message passing* abstraction– the simplest form of interprocess communication. This enables a sending process to transmit a single message to a receiving process. The independent packets containing these messages are called *datagrams*. In the Java and UNIX APIs, the sender specifies the destination using a socket – an indirect reference to a particular port used by the destination process at a destination computer.

The application program interface to TCP provides the abstraction of a two-way *stream* between pairs of processes. The information communicated consists of a stream of data items with no message boundaries. Streams provide a building block for producer-consumer communication. A producer and a consumer form a pair of processes in which the role of the first is to produce data items and the role of the second is to consume them. The data items sent by the producer to the consumer are queued on arrival at the receiving host until the consumer is ready to receive them. The consumer must wait when no data items are available. The producer must wait if the storage used
to hold the queued data items is exhausted.

The API for the Internet protocols

The general characteristics of interprocess communication and then discuss the Internet protocols as an example, explaining how programmers can use them, either by means of UDP messages or through TCP streams.

The characteristics of interprocess communication

Message passing between a pair of processes can be supported by two message communication operations, *send* and *receive*, defined in terms of destinations and messages. To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes.

Synchronous and asynchronous communication • A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving processes may be either synchronous or asynchronous. In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case,

both *send* and *receive* are *blocking* operations. Whenever a *send* is issued the sending process (or thread) is blocked until the corresponding *receive* is issued. Whenever a *receive* is issued by a process (or thread), it blocks until a message arrives.

In the *asynchronous* form of communication, the use of the *send* operation is *nonblocking* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The *receive* operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

In a system environment such as Java, which supports multiple threads in a single process, the blocking *receive* has no disadvantages, for it can be issued by one thread while other threads in the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage. Non-blocking communication appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control. For these reasons, today's systems do not generally provide the nonblocking form of *receive*.

Message destinations • Chapter 3 explains that in the Internet protocols, messages are sent to (*Internet address*, *local port*) pairs. A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but can have many senders. Processes may use multiple ports to receive messages. Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients.

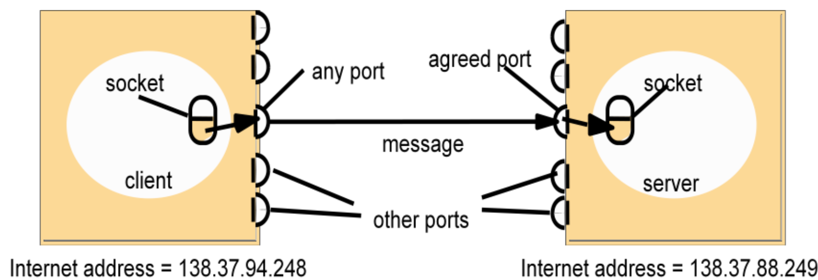
Reliability • As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a 'reasonable' number of packets being dropped or lost. In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

Ordering • Some applications require that messages be delivered in *sender order* – that is, the order in which they were transmitted by the sender. The delivery of messages out of sender order is regarded as a failure by such applications.

Sockets

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for ommunication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh

OS. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process, is shown in the following figure.



For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by
a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number (216) of possible port numbers for use by local processes for receiving messages. Any process
may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

Java API for Internet addresses • As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, *InetAddress*, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames. For example, instances of *InetAddress* that contain Internet addresses can be created by calling a static method of *InetAddress*, giving a DNS hostname as the argument. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host whose DNS name is *bruno.dcs.qmul.ac.uk*, use:

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");

This method can throw an *UnknownHostException*. Note that the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet addresses – 4 bytes in IPv4 and 16 bytes in IPv6.

UDP datagram communication

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive. A datagram is transmitted between processes when one process *sends* it and another *receives* it. To send or receive messages a process must first create a socket bound to an
Internet address of the local host and a local port. A server will bind its socket to a *server port* – one that it makes known to clients so that they can send messages to it. A client binds its socket to any free local port. The *receive* method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

The following are some issues relating to datagram communication:

Message size: The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival. The underlying IP protocol allows packet lengths of up to 216 bytes, which includes the headers as well as the message. However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment them into chunks of that size.

Generally, an application, for example DNS, will decide on a size that is not excessively large but is adequate for its intended use.

Blocking: Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication (a non-blocking *receive* is an option in some implementations). The *send* operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination. On arrival, the message is placed in a queue for the socket that is bound to the destination port. The message can be collected from the queue by an outstanding or future invocation of *receive* on that socket. Messages are discarded at the destination if no process already has a socket bound to the destination port.

Timeouts: The *receive* that blocks forever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

Receive from any: The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin. The *receive* method returns the Internet address and local port of the sender, allowing the recipient to check where the message came from. It is possible to connect a datagram socket to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

Failure model for UDP datagrams • A failure model for communication channels and defines reliable communication in terms of two properties: integrity and validity. The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP datagrams suffer from the following failures:

Omission failures: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination. To simplify the discussion, we regard send-omission and receive-omission failures as omission failures in the communication channel.

-

Ordering: Messages can sometimes be delivered out of sender order. Applications using UDP

datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements.

Use of UDP • For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP. UDP datagrams are sometimes an attractive choice because they do not
suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

• the need to store state information at the source and destination;
• the transmission of extra messages;
• latency for the sender.

Java API for UDP datagrams • The Java API provides datagram communication by means of two classes: *DatagramPacket* and *DatagramSocket*. *DatagramPacket*:

This class provides a constructor that makes an instance out of an array of bytes comprising a message, the length of the message and the Internet address and local port number of the destination socket, as follows:

Datagram packet
array of bytes containing message length of message Internet address port number

An instance of *DatagramPacket* may be transmitted between processes when one process *sends* it and another *receives* it.

UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
        public static void main(String args[]){
        DatagramSocket aSocket = null;
          try{
                aSocket = new DatagramSocket(6789);
                byte[] buffer = new byte[1000];
                while(true){
                  DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                  aSocket.receive(request);
                  DatagramPacket reply = new DatagramPacket(request.getData(),
                        request.getLength(), request.getAddress(), request.getPort());
                  aSocket.send(reply);
                }
          }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
          }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
  }
}
```

DatagramSocket: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can throw a *SocketException* if the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.

UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
        public static void main(String args[]){
        DatagramSocket aSocket = null;
          try{
                aSocket = new DatagramSocket(6789);
                byte[] buffer = new byte[1000];
                while(true){
                  DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                  aSocket.receive(request);
                  DatagramPacket reply = new DatagramPacket(request.getData(),
                        request.getLength(), request.getAddress(), request.getPort());
                  aSocket.send(reply);
                }
          }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
          }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```

TCP stream communication

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:

Message sizes: The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data. The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handed to the application as requested. Applications can, if necessary, force data to be sent immediately.

Lost messages: The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required.

Flow control: The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.

Message duplication and ordering: Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.

-

Message destinations: A pair of communicating processes establish a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. Establishing a connection involves a *connect* request from client to server followed by an *accept* request from server to client before any communication can take place. This could be a considerable overhead for a single client-server request and reply.

Java API for TCP streams • The Java interface to TCP streams is provided in the classes *ServerSocket* and *Socket*:

ServerSocket: This class is intended for use by a server to create a socket at a server port for listening for *connect* requests from clients. Its *accept* method gets a *connect* request from the queue or, if the queue is empty, blocks until one arrives. The result of executing *accept* is an instance of *Socket* – a socket to use for communicating with the client.

Socket: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and port number. It can throw an *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error occurs.

TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
          try{
                int serverPort = 7896;
                s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =
                        new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);          // UTF is a string encoding see Sn 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
          }catch (UnknownHostException e){
                        System.out.println("Sock:"+e.getMessage());
          }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
          }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException
e){System.out.println("close:"+e.getMessage());}}
        }
}
```

TCP server makes a connection for each client and then echoes the client’s request

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

// this figure continues on the next slide

 class Connection extends Thread {
            DataInputStream in;
            DataOutputStream out;
            Socket clientSocket;
            public Connection (Socket aClientSocket) {
               try {
                        clientSocket = aClientSocket;
                        in = new DataInputStream( clientSocket.getInputStream());
                        out =new DataOutputStream( clientSocket.getOutputStream());
                        this.start();
                } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
            }
            public void run(){
               try {                                    // an echo server
                        String data = in.readUTF();
                        out.writeUTF(data);
               } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
               } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
               } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}
            }
 }
```

External data representation and marshalling

The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival. The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures. There are two variants for the ordering of integers: the so-called *big-endian* order, in which the most significant byte comes first; and *little-endian* order, in which it comes last. Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use ASCII character coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character.

One of the following methods can be used to enable any two computers to exchange binary data values:

• The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.

• The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary. Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. An agreed standard for the representation of data structures and primitive values is called an *external data representation*.

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. *Unmarshalling* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Thus marshalling consists of the translation of structured data items and
primitive values into an external data representation. Similarly, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

Three alternative approaches to external data representation and marshalling are discussed:

• CORBA's common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programminglanguages.

-

• Java's object serialization, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

• XML (Extensible Markup Language), which defines a textual fomat for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer. Even in the case of XML, which is textual and therefore more accessible to hand-encoding, software for marshalling and unmarshalling is available for all commonly used platforms and programming environments. Because marshalling requires the consideration of all the finest details of the representation of the primitive components of composite objects, the process is likely to be error-prone if carried out by hand. Compactness is another issue that can be addressed in the design of automatically generated marshalling procedures.

In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation. The HTTP protocol, which is described in Chapter 5, is another example of the textual approach.

Another issue with regard to the design of marshalling methods is whether the marshalled data should include information concerning the type of its contents. For example, CORBA's representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information, but in different ways. Java puts all of the required type information into the serialized form, but XML documents may refer to externally defined sets of names (with types) called *namespaces*.

Although we are interested in the use of an external data representation for the arguments and results of RMIs and RPCs, it does have a more general use for representing data structures, objects or structured documents in a form suitable for transmission in messages or storing in files.

CORBA CDR for constructed types

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

CORBA's Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types, which are described in Figure 4.7. Each argument or result in a remote invocation is
represented by a sequence of bytes in the invocation or result message.

index in sequence of bytes	← 4 bytes →	notes on representation
0–3	5	length of string
4–7	"Smit"	'Smith'
8–11	"h___"	
12–15	6	length of string
16–19	"Lond"	'London'
20-23	"on__"	
24–27	1934	unsigned long

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Marshalling in CORBA • Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL (see Section 8.3.1), which provides a notation for describing the types of the arguments and results of RMI methods.

Java object serialization

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations. An object is an instance of a Java class. For example, the Java class equivalent to the *Person struct* defined in CORBA IDL might be:

```
public class Person implements Serializable {
private String name;
private String place;
private int year;
public Person(String aName, String aPlace, int aYear) {
```

```
name = aName;
place = aPlace;
year = aYear;
}
// followed by methods for accessing the instance variables
}
```

Extensible Markup Language (XML)

XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web. In general, the term *markup language* refers to a textual encoding that represents both a text and details as to its structure or its appearance. Both XML and HTML were derived from SGML (Standardized Generalized Markup Language) [ISO 8879], a very complex markup language. HTML was designed for defining the appearance of web pages. XML was designed for writing structured documents for the Web.

XML data items are tagged with 'markup' strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. That is, in XML, the tags relate to the structure of the text that they enclose, in contrast to HTML, in which the tags specify how a browser could display the text. For a specification of XML, see the pages on XML provided by W3C [www.w3.org VI].

XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services. However, XML is also used in many other ways, including in archiving and retrieval systems – although an XML archive may be larger than a binary one, it has the advantage of being readable on any computer.
Other examples of uses of XML include for the specification of user interfaces and the encoding of configuration files in operating systems.

XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them. For example, clients usually use SOAP messages to communicate with web
services. SOAP is an XML format whose tags are published for use by web services and their clients.

Some external data representations (such as CORBA CDR) do not need to be self describing, because it is assumed that the client and server exchanging a message have prior knowledge of the order and the types of the information it contains. However, XML was intended to be used by multiple applications for different purposes. The provision of tags, together with the use of namespaces to define the meaning of the tags, has made this possible. In addition, the use of tags enables applications to select just those parts of a document it needs to process: it will not be affected by the addition of information relevant to other applications.

-

XML definition of the Person structure

```
<person id="123456789">
            <name>Smith</name>
            <place>London</place>
            <year>1934</year>
            <!-- a comment -->
</person >
```

Remote object references

Java and CORBA that support the distributed object model. It is not relevant to XML. When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A *remote object reference* is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked. Chapter 5 explains that remote object references are also passed as arguments and returned as results of remote method invocations, that each remote object has a single remote object reference and that remote object references can be compared to see whether they refer to the same remote object. Here, we discuss the external representation of remote object references.

Client-server communication

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments) sends a request message to the remote object and returns the reply.

The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest (); acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort); sends the reply message reply to the client at its Internet address and port.

Field	Type
messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

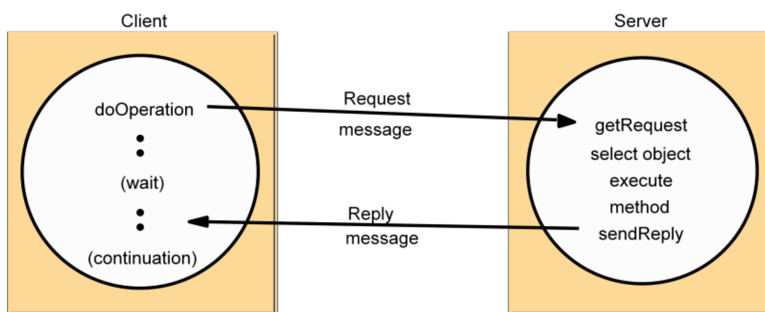
RPC exchange protocols

HTTP request message

HTTP reply message

HTTP version	status code	reason	headers	message body
HTTP/1.1	200	OK		resource data

Request-reply communication



Group communication

A *multicast operation* is more appropriate – this is an operation that sends a single message from one process to
each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired behaviour of a multicast. The simplest multicast rotocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:
1. *Fault tolerance based on replicated services*: A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.

2. *Discovering services in spontaneous networking*: Section 1.3.2 defines service discovery in the context of spontaneous networking. Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.

3. *Better performance through replicated data*: Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value is multicast to the processes managing the replicas.

4. *Propagation of event notifications*: Multicast to a group may be used to notify processes when

something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publishsubscribe protocols may make use of group multicast to disseminate events to subscribers (see Chapter 6).

IP multicast – An implementation of multicast communication

IP multicast • *IP multicast* is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers – ports belong to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. A *multicast group* is specified by a Class D Internet address – that is, an address whose first 4 bits are 1110 in IPv4.

At the application programming level, IP multicast is available only via UDP. An application program performs multicasts by sending UDP datagrams with multicast addresses and ordinary port numbers. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group. At the IP level, a computer
belongs to a multicast group when one or more of its processes has sockets that belong to that group. When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:
Multicast routers: IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single datagrams to routers on other networks, where they are again multicast to local members. To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass – called the *time to live*, or TTL for short. To understand how routers know which other routers have members of a multicast group.

Multicast address allocation: As discussed in Chapter 3, Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA). The management of this address space is reviewed annually, with current practice documented in RPC 3171. This document defines a partitioning of this address space into a number of blocks, including:

• Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.
• Internet Control Block (224.0.1.0 to 224.0.1.225).
• Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.
• Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic (to constrain propagation).

Failure model for multicast datagrams • Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. That is, some but not all of the members of the group may receive it. This can be called *unreliable* multicast, because it does not guarantee that a message will be delivered to any member of a group.

Java API to IP multicast • The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*, which is a subclass of *DatagramSocket* with the additional capability of being able to join multicast groups. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a or any free local port. A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port. A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket.

Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);


         // this figure continued on the next slide
```

Reliability and ordering of multicast

The effect of the failure semantics of IP multicast on the four examples of the use of replication

1. *Fault tolerance based on replicated services*: Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others. In most cases, this service would require that all members receive request messages in the same order as one another.

2. *Discovering services in spontaneous networking*: One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services.

3. *Better performance through replicated data*: Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.

4. *Propagation of event notifications*: The particular application determines the qualities required of multicast. For example, the Jini lookup services use IP multicast to announce their existence

Communication between Distributed Objects

The Object Model

Five Parts of the Object Model

– An object-oriented program consists of a collection of interacting objects

• Objects consist of a set of data and a set of methods

• In DS, object's data should be accessible only via methods

Object References

– Objects are accessed by object references

– Object references can be assigned to variables, passed as arguments, and returned as the result of a method

– Can also specify a method to be invoked on that object

Interfaces

– Provide a definition of the signatures of a set of methods without specifying their implementation

– Define types that can be used to declare the type of variables or of the parameters and return values of methods

Actions

– Objects invoke methods in other objects

– An invocation can include additional information as arguments to perform the behavior specified by the method

– Effects of invoking a method

1. The state of the receiving object may be changed
2. A new object may be instantiated
3. Further invocations on methods in other objects may occur
4. An exception may be generated if there is a problem encountered

Exceptions

– Provide a clean way to deal with unexpected events or errors

– A block of code can be defined to throw an exception when errors or unexpected conditions occur. Then control passes to code that catches the exception

Garbage Collection

– Provide a means of freeing the space that is no longer needed

– Java (automatic), C++ (user supplied)

Distributed Objects

• Physical distribution of objects into different processes or computers in a distributed system

– Object state consists of the values of its instance variables

– Object methods invoked by remote method invocation (RMI)

– Object encapsulation: object state accessed only by the object methods

Usually adopt the client-server architecture

– Basic model

 • Objects are managed by servers and

• Their clients invoke their methods using RMI

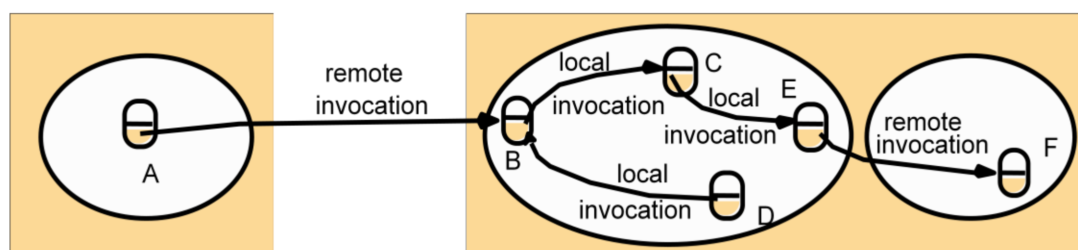
– Steps
1. The client sends the RMI request in a message to the server
2. The server executes the invoked method of the object
3. The server returns the result to the client in another message

– Other models
• Chains of related invocations: objects in servers may become clients of objects in other servers
• Object replication: objects can be replicated for fault tolerance and performance
• Object migration: objects can be migrated to enhancing performance and availability

The Distributed Object Model

Two fundamental concepts: Remote Object Reference and Remote Interface
– Each process contains objects, some of which can receive remote invocations are called remote objects (B, F), others only local invocations
– Objects need to know the remote object reference of an object in another process in order to invoke its methods, called remote method invocations
– Every remote object has a remote interface that specifies which of its methods can be invoked remotely

Remote and local method invocations

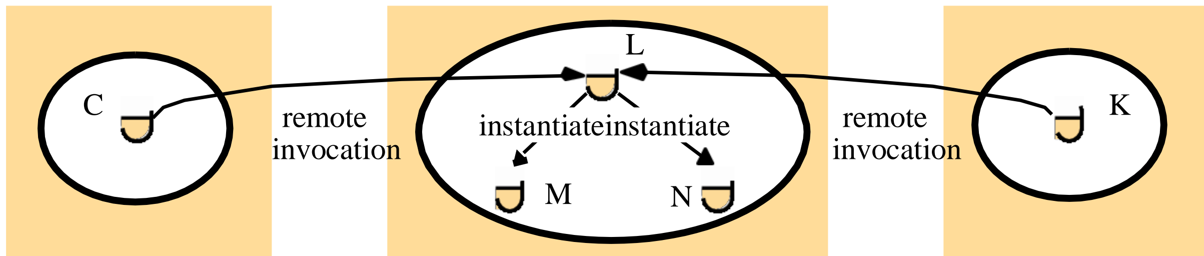


Five Parts of Distributed Object Model

• Remote Object References
– accessing the remote object
– identifier throughout a distributed system
– can be passed as arguments
• Remote Interfaces
– specifying which methods can be invoked remotely
– name, arguments, return type
– Interface Definition Language (IDL) used for defining remote interface

Remote Object and Its remote Interface

• Actions
– An action initiated by a method invocation may result in further invocations on methods in other objects located indifference processes or computers
– Remote invocations could lead to the instantiation of new objects, ie. objects M and N of following figure.



• Exceptions
– More kinds of exceptions: i.e. timeout exception
- RMI should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked
• Garbage Collection
- Distributed garbage collections is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting

Design Issues for RMI
• Two design issues that arise in extension of local method invocation for RMI
  – The choice of invocation semantics
• Although local invocations are executed exactly once, this cannot always be the case for RMI due to transmission error
  – Either request or reply message may be lost
  – Either server or client may be crashed
  – The level of transparency
• Make remote invocation as much like local invocation as possible

RMI Design Issues: Invocation Semantics
• Error handling for delivery guarantees
  – Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed
  – Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server
  – Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations
• Choices of invocation semantics
  – Maybe: the method executed once or not at all (no retry nor retransmit)
  – At-least-once: the method executed at least once
  – At-most-once: the method executed exactly once

Invocation semantics: choices of interest

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

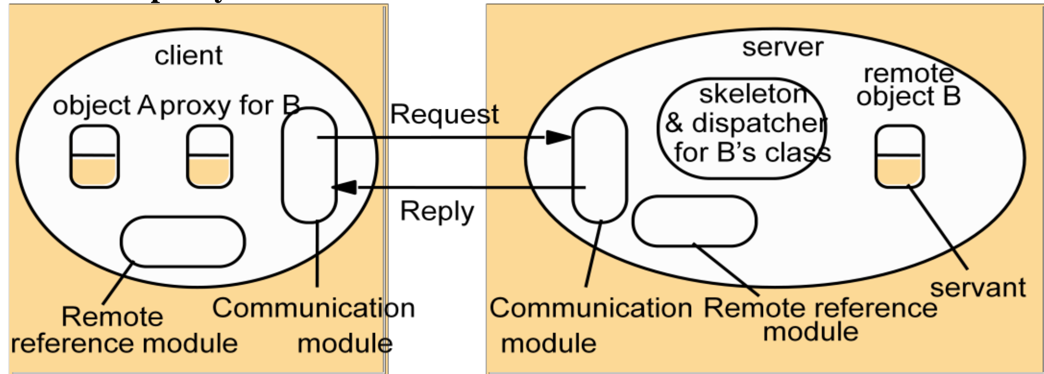
RMI Design Issues: Transparency

• Transparent remote invocation: like a local call
  – marshalling/unmarshalling
  – locating remote objects
  – accessing/syntax
• Differences between local and remote invocations
  – latency: a remote invocation is usually several order of magnitude greater than that of a local one
  – availability: remote invocation is more likely to fail
  – errors/exceptions: failure of the network? server? hard to tell
• syntax might need to be different to handle different local vs remote errors/exceptions (e.g. Argus)
  – consistency on the remote machine:
• Argus: incomplete transactions, abort, restore states [as if the call was never made]

Implementation of RMI

•Communication module
  – Two cooperating communication modules carry out the request-reply protocols: message type, request ID, remote object reference
• Transmit request and reply messages between client and server
• Implement specific invocation semantics
  – The communication module in the server
• selects the dispatcher for the class of the object to be invoked,
• passes on local reference from remote reference module,
• returns request

The role of proxy and skeleton in remote method invocation



• **Remote reference module**

– Responsible for translating between local and remote object references and for creating remote object references

– remote object table: records the correspondence between local and remote object references

  • remote objects held by the process (B on server)

  • local proxy (B on client)

– When a remote object is to be passed for the first time, the module is asked to create a remote object reference, which it adds to its table

• **Servant**

– An instance of a class which provides the body of a remote object

– handles the remote requests

•**RMI software**

– Proxy: behaves like a local object, but represents the remote object

– Dispatcher: look at the methodID and call the corresponding method in the skeleton

– Skeleton: implements the method

  Generated automatically by an interface compiler

Implementation Alternatives of RMI

• **Dynamic invocation**

  – Proxies are static—interface complied into client code

  – Dynamic—interface available during run time

    • Generic invocation; more info in ―Interface Repository‖ (COBRA)

    • Dynamic loading of classes (Java RMI)

•**Binder**

– A separate service to locate service/object by name through table mapping for names and remote object references

• **Activation of remote objects**
  – Motivation: many server objects not necessarily in use all of the time
    • Servers can be started whenever they are needed by clients, similar to inetd
  – Object status: active or passive
    • active: available for invocation in a running process
    • passive: not running, state is stored and methods are pending
  – Activation of objects:
    • creating an active object from the corresponding passive object by creating a new instance of its class
    • initializing its instance variables from the stored state
  – Responsibilities of activator
    • Register passive objects that are available for activation
    • Start named server processes and activate remote objects in them
    • Keep track of the locations of the servers for remote objects that it has already activated

• **Persistent object stores**
  – An object that is guaranteed to live between activations of processes is called a persistent object
  – Persistent object store: managing the persistent objects
    • stored in marshaled from on disk for retrieval
    • saved those that were modified
  – Deciding whether an object is persistent or not:
    • persistent root: any descendent objects are persistent (persistent Java, PerDiS)
    • some classes are declared persistent (Arjuna system)
    • Object location
  – specifying a location: ip address, port #, ...
  – location service for migratable objects
    • Map remote object references to their probable current locations

    • Cache/broadcast scheme (similar to ARP)
– Cache locations
– If not in cache, broadcast to find it
  • Improvement: forwarding (similar to mobile IP)

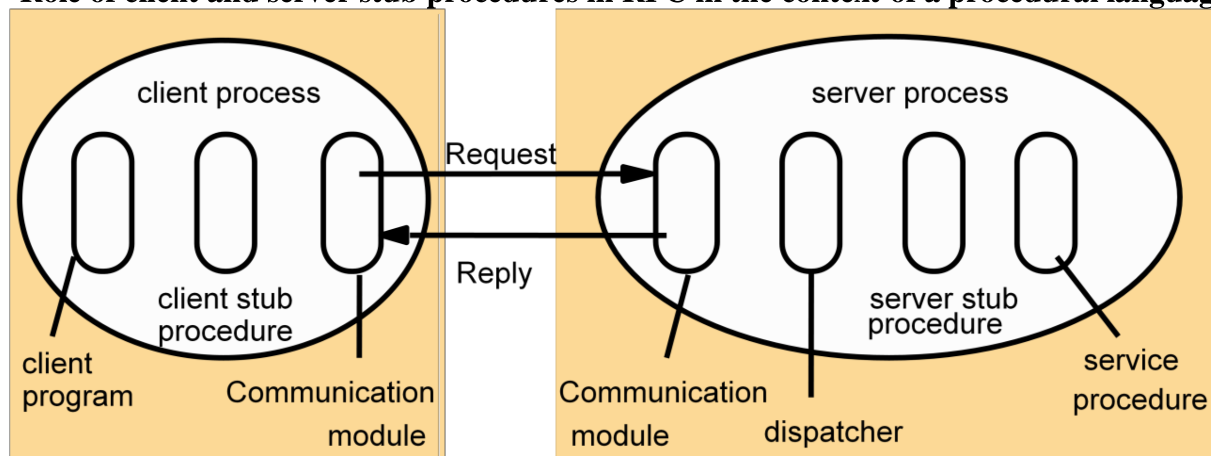
Distributed Garbage Collection
• Aim: ensure that an object
  – continues to exist if a local or remote reference to it is still held anywhere
  – be collected as soon as no object any longer holds a reference to it
• General approach: reference count
• Java's approach
  – the server of an object (B) keeps track of proxies
  – when a proxy is created for a remote object
• addRef(B) tells the server to add an entry
  – when the local host's garbage collector removes the proxy
• removeRef(B) tells the server to remove the entry

  – when no entries for object B, the object on server is deallocated

Remote Procedure Call

• client: "stub" instead of "proxy" (same function, different names)
– local call, marshal arguments, communicate the request
•server:
– dispatcher
– "stub": unmarshal arguments, communicate the results back

Role of client and server stub procedures in RPC in the context of a procedural language



Case Study: Sun RPC

•Sun RPC: client-server in the SUN NFS (network file system)
  – UDP or TCP; in other unix OS as well
  – Also called ONC (Open Network Computing) RPC
•Interface Definition Language (IDL)
  – initially XDR is for data representation, extended to be IDL
  – less modern than CORBA IDL and Java

• program numbers instead of interface names
• procedure numbers instead of procedure names
• single input parameter (structs)
  – rpcgen: compiler for XDR
  • client stub; server main procedure, dispatcher, and server stub
• XDR marshalling, unmarshaling

•Binding (registry) via a local binder - portmapper
  – Server registers its program/version/port numbers with portmapper
  – Client contacts the portmapper at a fixed port with program/version numbers to get the server port
– Different instances of the same service can be run on different computers at different ports

•Authentication
  – request and reply have additional fields
  – unix style (uid, gid), shared key for signing, Kerberos

Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};

struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
  version VERSION {
    void WRITE(writeargs)=1;      1
    Data READ(readargs)=2;        2
  }=2;
} = 9999;
```

Events and Notifications

•Idea behind the use of events

– One object can react to a change occurring in another object

•Events

– Notifications of events: objects that represent events

• asynchronous and determined by receivers what events are interested

– event types

• each type has attributes (information in it)

• subscription filtering: focus on certain values in the attributes (e.g. "buy" events, but only "buy car" events)

•Publish-subscribe paradigm

– publish events to send

– subscribe events to receive

•Main characteristics in distributed event-based systems

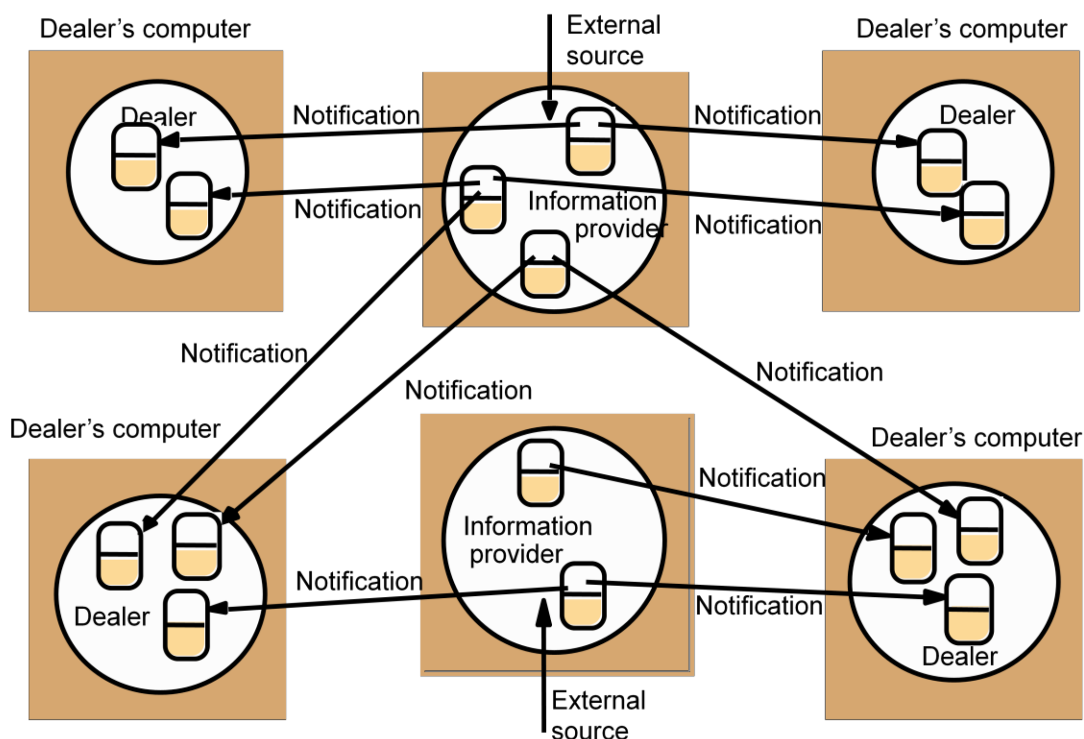
– Heterogeneous: a way to standardize communication in heterogeneous systems

• not designed to communicate directly

– Asynchronous: notifications are sent asynchronously

• no need for a publisher to wait for each subscriber--subscribers come and go

Dealing room system: allow dealers using computers to see the latest information about the market prices of the stocks they deal in

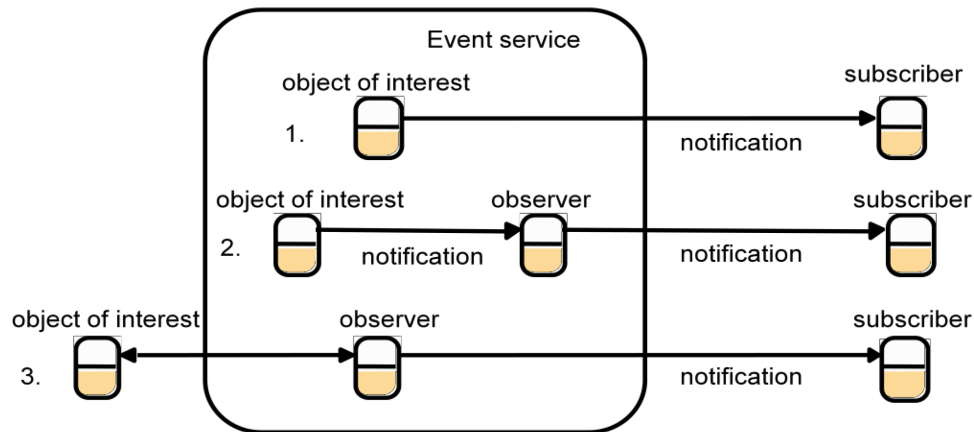


Distributed Event Notification

• Distributed event notification
  – decouple publishers from subscribers via an event service (manager)
• Architecture: roles of participating objects
  – object of interest (usually changes in states are interesting)
  – event
  – notification
  – subscriber
  – observer object (proxy) [reduce work on the object of interest]
•forwarding
• filtering of events types and content/attributes
• patterns of events (occurrence of multiple events, not just one)
• mailboxes (notifications in batch es, subscriber might not be ready)
  – publisher (object of interest or observer object)

• generates event notifications

Example: Distributed Event Notification



•Three cases
– Inside object without an observer: send notifications directly to the subscribers
– Inside object with an observer: send notification via the observer to the subscribers
– Outside object (with an observer)

1. An observer queries the object of interest in order to discover when events occur
2. The observer sends notifications to the subscribers

•Jini

Case Study: Jini Distributed Event Specification

–Allow a potential subscriber in one Java Virtual Machine (JVM) to subscribe to and receive notifications of events in an objectof interest in another JVM
– Main objects

• event generators (publishers)
• remote event listeners (subscribers)
• remote events (events)
• third-party agents (observers)

– An object subscribes to events by informing the event generator about the type of event and specifying a remote event listener as the target for notification

Case Study: Java RMI

Java Remote interfaces *Shape* and *ShapeList and* Java class ShapeListServant implements interface ShapeList

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
             ShapeList aShapeList = new ShapeListServant();          1
           Naming.rebind("Shape List", aShapeList );                 2
             System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
  public static void main(String args[]){
  System.setSecurityManager(new RMISecurityManager());
  ShapeList aShapeList = null;
  try{
     aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");         1
     Vector sList = aShapeList.allShapes();                                2
  } catch(RemoteException e) {System.out.println(e.getMessage());
  }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
     }
}
```

Java class ShapeListServer with main and Java client of ShapreList

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
            ShapeList aShapeList = new ShapeListServant();          1
          Naming.rebind("Shape List", aShapeList );                 2
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
           System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
  public static void main(String args[]){
  System.setSecurityManager(new RMISecurityManager());
  ShapeList aShapeList = null;
  try{
     aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");       1
     Vector sList = aShapeList.allShapes();                               2
  } catch(RemoteException e) {System.out.println(e.getMessage());
  }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
     }
}
```

Naming class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [] list()

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

Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();                    1
                Naming.rebind("Shape List", aShapeList );                     2
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList;          // contains the list of Shapes       1
    private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {    2
        version++;
            Shape s = new ShapeServant( g, version);                     3
            theList.addElement(s);
            return s;
    }
    public  Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

-

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
      private Vector theList;                    // contains the list of Shapes          1
      private int version;
     public ShapeListServant()throws RemoteException{...}
     public Shape newShape(GraphicalObject g) throws RemoteException {          2
         version++;
             Shape s = new ShapeServant( g, version);                                3
             theList.addElement(s);
             return s;
     }
     public  Vector allShapes()throws RemoteException{...}
     public int getVersion() throws RemoteException { ... }
}
```

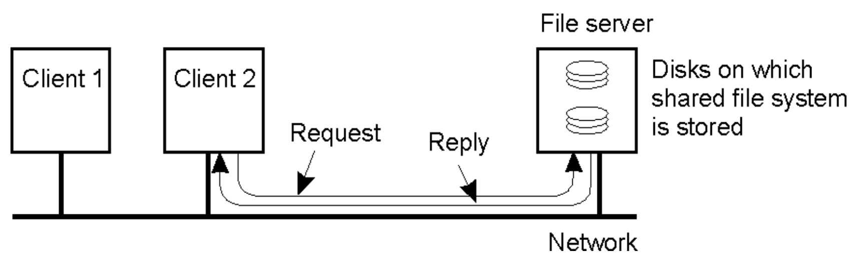
Java RMI Callbacks

•Callbacks
- – server notifying the clients of events
- – why?
  - • polling from clients increases overhead on server
  - • not up-to-date for clients to inform users
- – how
  - • remote object (callback object) on client for server to call
  - • client tells the server about the callback object, server put the client on a list
  - • server call methods on the callback object when events occur
    - – client might forget to remove itself from the list
      - • lease--client expire

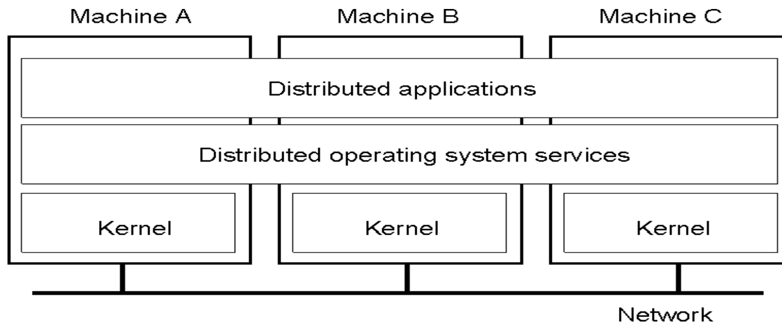
The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources – the processors, memory, networks, and storage media. An operating system such as UNIX (and its variants, such as Linux and Mac OS X) or Windows (and its variants, such as XP, Vista and Windows 7) provides the programmer with, for example, files rather than disk blocks, and with sockets rather than raw network access. It takes over the physical resources on a single node and manages them to present these resource abstractions through the system-call interface.

The operating system's middleware support role, it is useful to gain some historical perspective by examining two operating system concepts that have come about during the development of distributed systems: network operating systems and distributed operating systems.

Both UNIX and Windows are examples of *network operating systems*. They have a networking capability built into them and so can be used to access remote resources. Access is network-transparent for some – not all – types of resource. For example, through a distributed file system such as NFS, users have network-transparent access to files. That is, many of the files that users access are stored remotely, on a server, and this is largely transparent to their applications.



An operating system that produces a single system image like this for all the resources in a distributed system is called a *distributed operating system*



Middleware and the Operating System

What is a distributed OS?

- Presents users (and applications) with an integrated computing platform that hides the individual computers.
- Has control over all of the nodes (computers) in the network and allocates their resources to tasks without user involvement.
    - In a distributed OS, the user doesn't know (or care) where his programs are running.
- Examples:
    - Cluster computer systems
    - V system, Sprite

- In fact, there are no distributed operating systems in general use, only network operating systems such as UNIX, Mac OS and Windows.
- to remain the case, for two main reasons.

The first is that users have much invested in their application software, which often meets their current problem-solving needs; they will not adopt a new operating system that will not run their applications, whatever efficiency advantages it offers.

The second reason against the adoption of distributed operating systems is that users tend to prefer to have a degree of autonomy for their machines, even in a closely knit organization.

Combination of middleware and network OS

- No distributed OS in general use
    - Users have much invested in their application software
    - Users tend to prefer to have a degree of autonomy for their machines
- Network OS provides autonomy
- Middleware provides network-transparent access resource

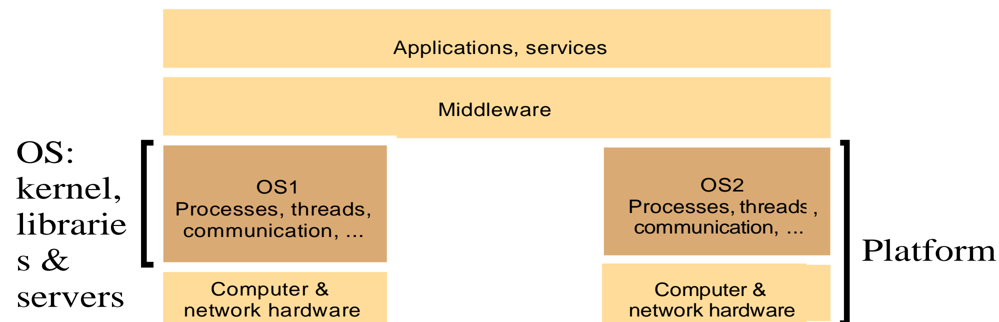
The relationship between OS and Middleware

- Operating System
    - Tasks: processing, storage and communication
    - Components: kernel, library, user-level services
- Middleware
    - runs on a variety of OS-hardware combinations

– remote invocations

Functions that OS should provide for middleware

The following figure shows how the operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.



Node 1 Node 2

Encapsulation: They should provide a useful service interface to their resources – that is, a set of operations that meet their clients' needs. Details such as management of memory and devices used to implement resources should be hidden from clients.

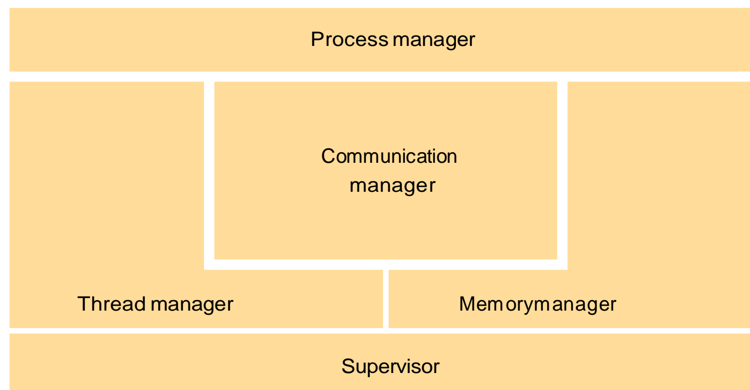
Protection: Resources require protection from illegitimate accesses – for example, files are protected from being read by users without read permissions, and device registers are protected from application processes.

Concurrent processing: Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency.

Communication: Operation parameters and results have to be passed to and from resource managers, over a network or within a computer.

Scheduling: When an operation is invoked, its processing must be scheduled within the kernel or server.

The core OS components

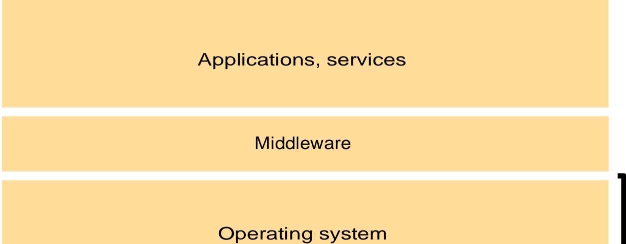


- **Process manager**
  - Handles the creation of and operations upon processes.

- **Thread manager**
  - Thread creation, synchronization and scheduling
- **Communication manager**
  - Communication between threads attached to different processes on the same computer
- **Memory manager**
  - Management of physical and virtual memory
- **Supervisor**
  - Dispatching of interrupts, system call traps and other exceptions
  - control of memory management unit and hardware caches

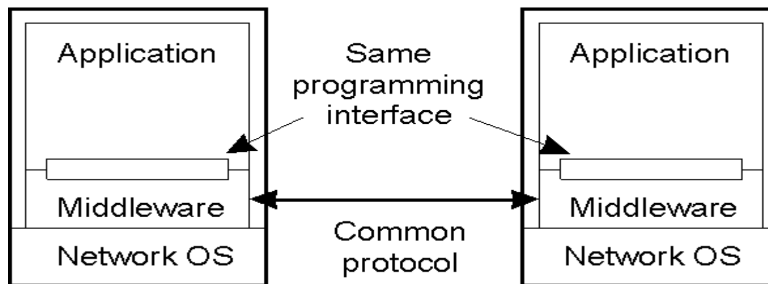
processor and floating point unit register manipulations

Software and hardware service layers in distributed systems



Middleware and Openness

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



Typical Middleware Services

- Communication
- Naming
- Persistence
- Distributed transactions
- Security

Middleware Models

- Distributed files
  - Examples?
- Remote procedure call
  - Examples?

- Distributed objects
  - Examples?
- Distributed documents
  - Examples?
- Others?
  - Message-oriented middleware (MOM)
  - Service oriented architecture (SOA)
  - Document-oriented

Middleware and the Operating System

- Middleware implements abstractions that support network-wide programming. Examples:
  - RPC and RMI (Sun RPC, Corba, Java RMI)
  - event distribution and filtering (Corba Event Notification, Elvin)
  - resource discovery for mobile and ubiquitous computing
  - support for multimedia streaming
- Traditional OS's (e.g. early Unix, Windows 3.0)
  - simplify, protect and optimize the use of local resources
- Network OS's (e.g. Mach, modern UNIX, Windows NT)

– do the same but they also support a wide range of communication standards and enable remote processes to access (some) local resources (e.g. files).

DOS vs. NOS vs. Middleware Discussion

- What is good/bad about DOS?
  - Transparency
  - Other issues have reduced success.
  - Problems are often socio-technological.
- What is good/bad about NOS?
  - Simple.
  - Decoupled, easy to add/remove.
  - Lack of transparency.
- What is good/bad about middleware?
  - Easy to make multiplatform.
  - Easy to start something new.
    - But this can also be bad.

Types of Distributed Oss

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

Illegitimate access

- **Maliciously contrived code**
- **Benign code**
  - contains a bug
  - have unanticipated behavior
- **Example: *read* and *write* in File System**
  - Illegal user vs. access right control
  - Access the file pointer variable directly (*setFilePointerRandomly*) vs. type-safe language
    - Type–safe language, e.g. Java or Modula-3
    - Non-type-safe language, e.g. C or C++

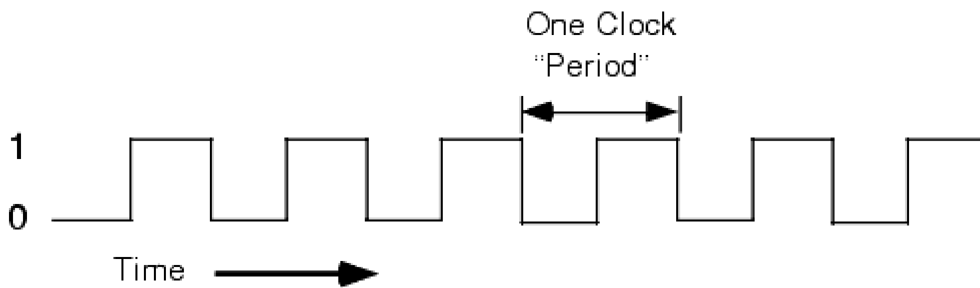
Kernel and Protection

- **Kernel**
  - always runs

– complete access privileges for the physical resources

- **Different execution mode**
  - *An address space:* a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, e.g.: read only or read-write
  - *supervisor* mode (*kernel process*) / *use*r mode (*user process*)
  - Interface between kernel and user processes: system call trap
- **The price for protection**
  - switching between different processes take many processor cycles
  - a system call trap is a more expensive operation than a simple method call

The System Clock



system clock frequency

clock period(One full period is also called a clock cycle)

"Hertz" (Hz) meaning one cycle per second

10 MHz : 100 nanoseconds

1GHz: 1 nanoseconds

Process and thread

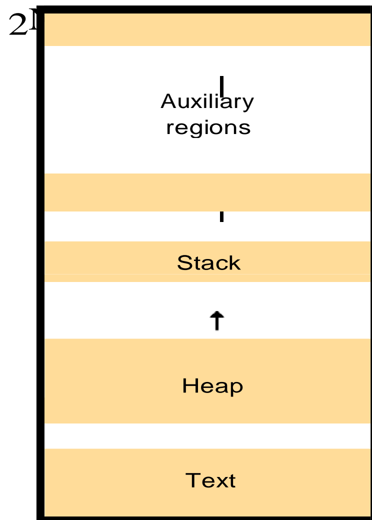
- **Process**
  - A program in execution
  - Problem: sharing between related activities are awkward and expensive
  - Nowadays, a process consists of an *execution environment* together with one or more *threads*
  - an analogy at page 215
- **Thread**
  - Abstraction of a single activity
  - Benefits
    - Responsiveness
    - Resource sharing
    - Economy
    - Utilization of MP architectures

Execution environment

- the unit of resource management
- Consist of
  - An address space
  - Thread synchronization and communication resources such as semaphores and communication interfaces (e.g. sockets)
  - Higher-level resources such as open files and windows
- Shared by threads within a process

Address space

- **Address space**
  - a unit of management of *a process*'s virtual memory
  - Up to 2^{32} bytes and sometimes up to 2^{64} bytes
  - consists of one or more regions
- **Region**
  - an area of continuous virtual memory that is accessible by the threads of the owning process
- **The number of regions is indefinite**
  - Support a separate stack for each thread
  - access *mapped file*
  - Share memory between processes
- **Region can be shared**
  - Libraries
  - Kernel
  - Shared data and communication
  - Copy-on-write



Creation of new process in distributed system

- **Creating process by the operation system**
  - *Fork, exec* in UNIX
- **Process creation in distributed system**
  - The choice of a target host
  - The creation of an execution environment, an initial thread

Choice of process host

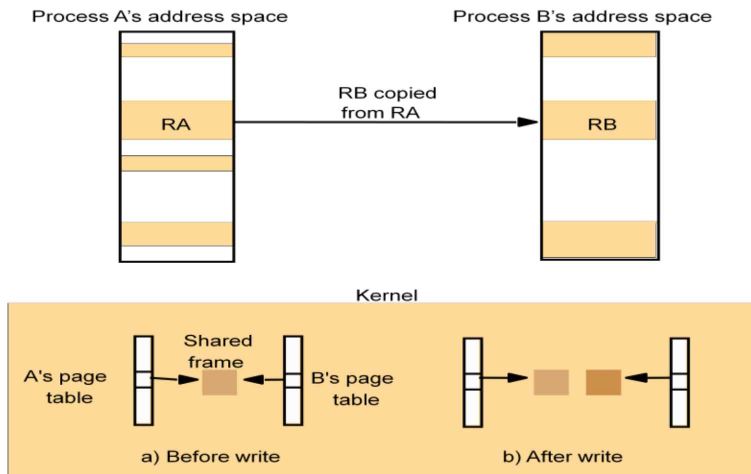
- **Choice of process host**
  - running new processes at their originator's computer
  - sharing processing load between a set of computers
- **Load sharing policy**
  - Transfer policy: situate a new process locally or remotely?

- Location policy: which node should host the new process?
    - Static policy without regard to the current state of the system
    - Adaptive policy applies heuristics to make their allocation decision
- Migration policy: when&where should migrate the running process?

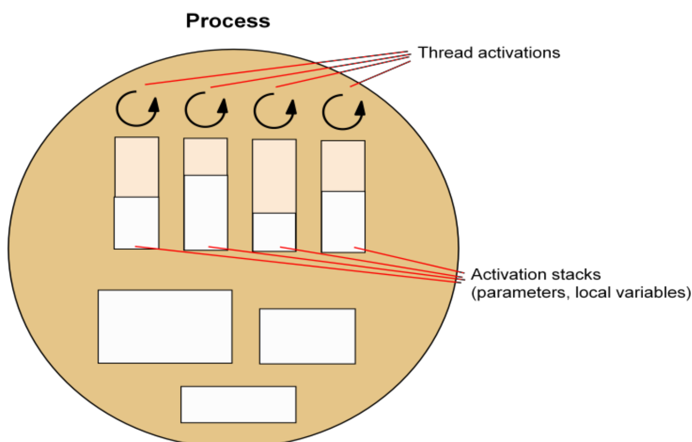
- **Load sharing system**
    - Centralized
    - Hierarchical
    - Decentralized

Creation of a new execution environment

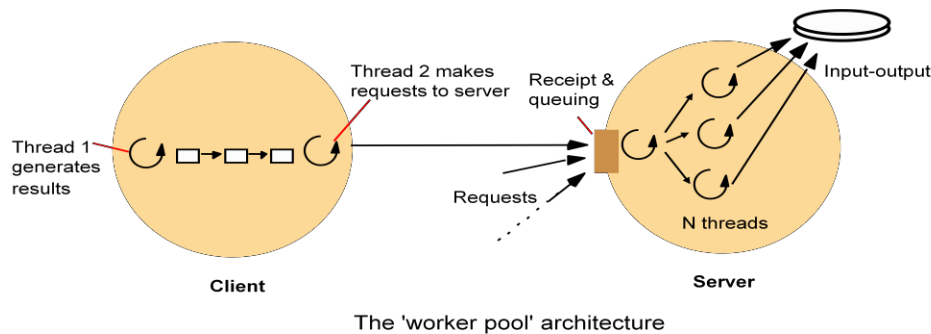
- Initializing the address space
    - Statically defined format
    - With respect to an existing execution environment, e.g. *fork*
- *Copy-on-write* scheme



Threads concept and implementation



Client and server with threads



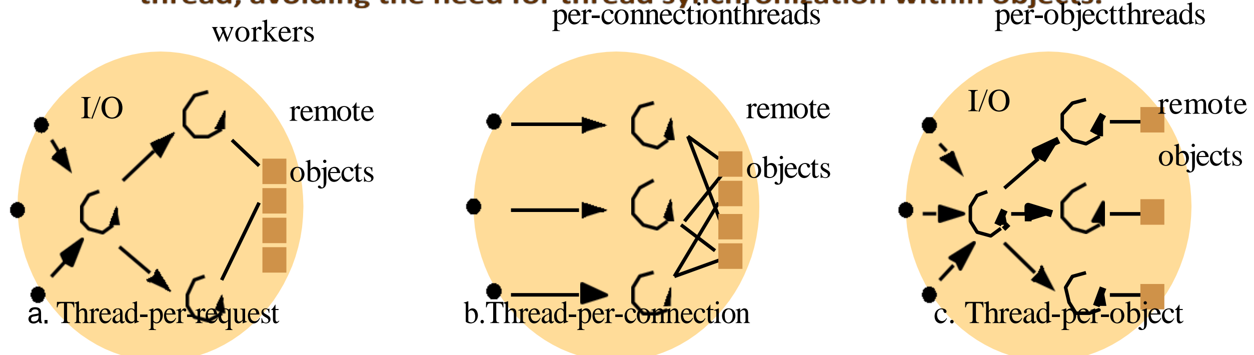
The 'worker pool' architecture

Alternative server threading architectures

(a) would be useful for UDP-based service, e.g. NTP

(b) is the most commonly used - matches the TCP connection model

(c) is used where the service is encapsulated as an object. E.g. could have multiple shared whiteboards with one thread each. Each object has only one thread, avoiding the need for thread synchronization within objects.



Threads versus multiple processes

- Creating a thread is (much) cheaper than a process (~10-20 times)
- Switching to a different thread in same process is (much) cheaper (5-50 times)
- Threads within same process can share data and other resources more conveniently and efficiently (without copying or messages)
- Threads within a process are not protected from each other

State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

Threads implementation

Threads can be implemented:

- in the OS kernel (Win NT, Solaris, Mach)
- at user level (e.g. by a thread library: C threads, pthreads), or in the language (Ada, Java).

+ lightweight - no system calls

+ modifiable scheduler

+ low cost enables more threads to be employed

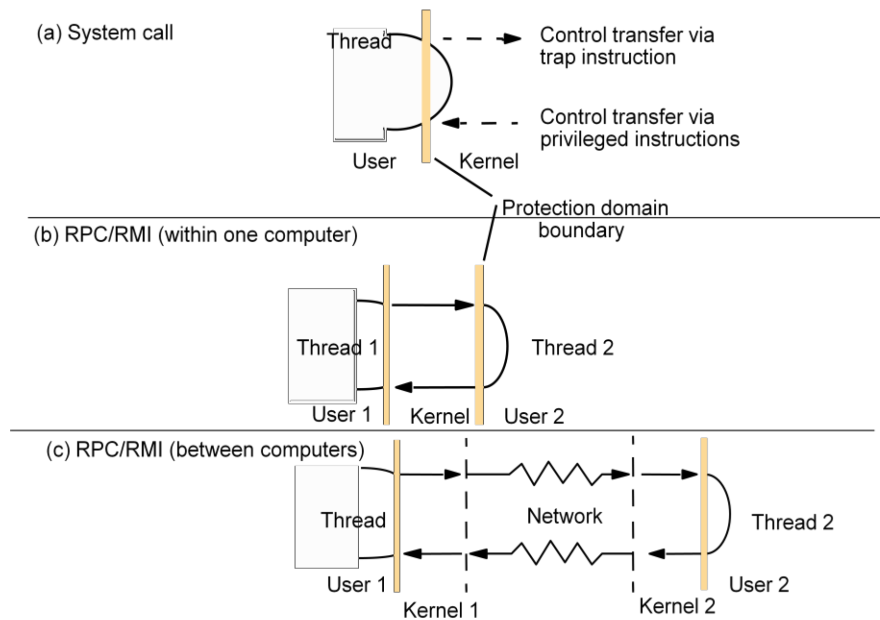
  - not pre-emptive
  - can exploit multiple processors
  - - page fault blocks all threads

- hybrid approaches can gain some advantages of both
  - user-level hints to kernel scheduler
  - hierarchic threads (Solaris 2)
  - event-based (SPIN, FastThreads)

Implementation of invocation mechanisms

- **Communication primitives**
  - TCP(UDP) Socket in Unix and Windows
  - *DoOperation*, *getRequest*, *sendReply* in Amoeba
  - Group communication primitives in V system
- **Protocols and openness**
  - provide standard protocols that enable internetworking between middleware
  - integrate novel low-level protocols without upgrading their application
  - Static stack
    - new layer to be integrated permanently as a –driver‖
  - Dynamic stack
    - protocol stack be composed on the fly
    - E.g. web browser utilize wide-area wireless link on the road and faster Ethernet connection in the office
- **Invocation costs**
  - Different invocations
  - The factors that matter
    - synchronous/asynchronous, *domain transition*, communication across a network, thread scheduling and switching
- **Invocation over the network**
  - Delay: the total RPC call time experienced by a client
  - Latency: the fixed overhead of an RPC, measured by null RPC
  - Throughput: the rate of data transfer between computers in a single RPC
  - An example
    - Threshold: one extra packet to be sent, might be an extra acknowledge packet is needed

Invocations between address spaces



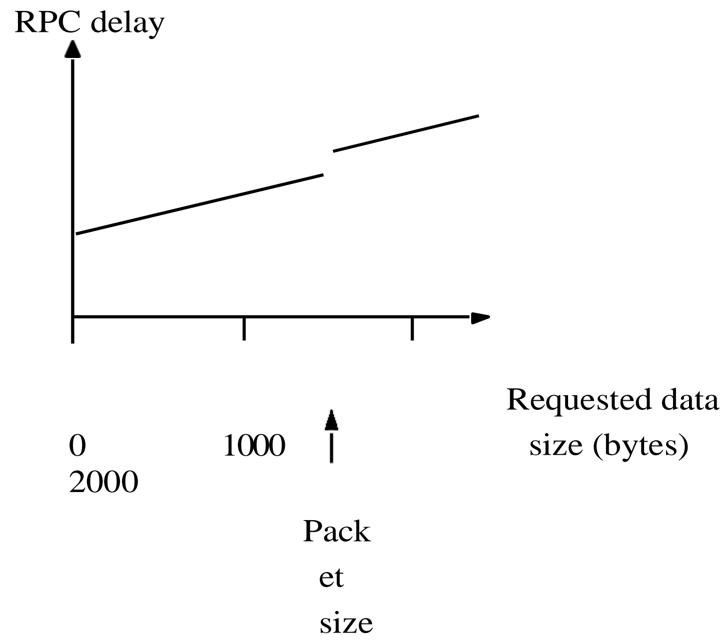
Support for communication and invocation

- The performance of RPC and RMI mechanisms is critical for effective distributed systems.
  - Typical times for 'null procedure call':
  - Local procedure call < 1 microseconds
  - Remote procedure call ~ 10 milliseconds
  - 'network time' (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only .01 millisecond; the remaining delays must be in OS and middleware - latency, not communication time.
- Factors affecting RPC/RMI performance
  - marshalling/unmarshalling + operation despatch at the server
  - data copying:- application -> kernel space -> communication buffers
  - thread scheduling and context switching:- including kernel entry
  - protocol processing:- for each protocol layer
  - network access delays:- connection setup, network latency

Improve the performance of RPC

- Memory sharing
  - rapid communication between processes in the same computer
- Choice of protocol
  - TCP/UDP
    - E.g. Persistent connections: several invocations during one
  - OS's buffer collect several small messages and send them together
- Invocation within a computer
  - Most cross-address-space invocation take place within a computer
  - LRPC (lightweight RPC)

RPC delay against parameter size


RPC delay
Requested data
size (bytes)
0
1000
2000
Pack
et
size

- A client stub marshals the call arguments into a message, sends the request message and receives and unmarshals the reply.
- At the server, a worker thread receives the incoming request, or an I/O threadreceives the request and passes it to a worker thread; in either case, the worker calls the appropriate server stub.
- The server stub unmarshals the request message, calls the designated procedure, and marshals and sends the reply.
- The following are the main components accounting for remote invocation delay, besides network transmission times:

Marshalling: Marshalling and unmarshalling, which involve copying and converting data, create a significant overhead as the amount of data grows.

Data copying: Potentially, even after marshalling, message data is copied several times in the course of an RPC:

1. across the user–kernel boundary, between the client or server address space and kernel buffers;
2. across each protocol layer (for example, RPC/UDP/IP/Ethernet);
3. between the network interface and kernel buffers.

Transfers between the network interface and main memory are usually handled by direct memory access (DMA). The processor handles the other copies.

Packet initialization: This involves initializing protocol headers and trailers, including checksums. The cost is therefore proportional, in part, to the amount of data sent.

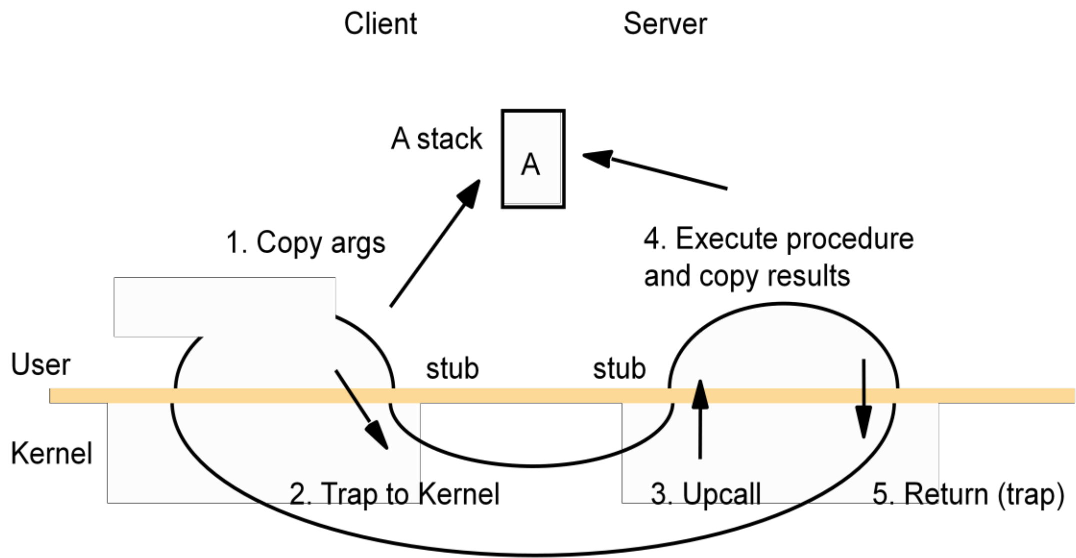
Thread scheduling and context switching: These may occur as follows:

1. Several system calls (that is, context switches) are made during an RPC, as stubs invoke the kernel's communication operations.
2. One or more server threads is scheduled.
3. If the operating system employs a separate network manager process, then each

Send involves a context switch to one of its threads.

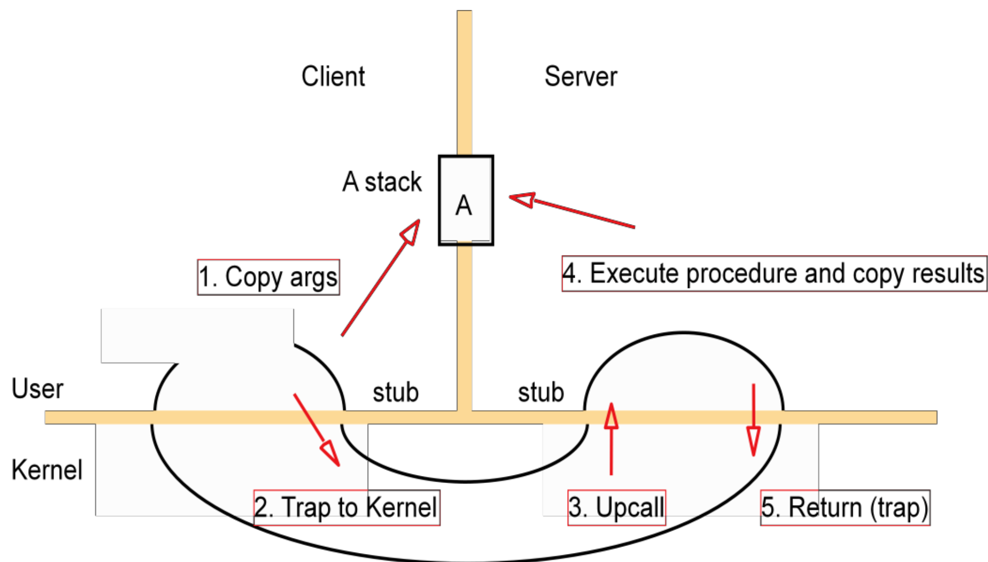
Waiting for acknowledgements: The choice of RPC protocol may influence delay, particularly when large amounts of data are sent.

A lightweight remote procedure call



Bershad's LRPC

- Uses shared memory for interprocess communication
  - while maintaining protection of the two processes
  - arguments copied only once (versus four times for conventional RPC)
- Client threads can execute server code
  - via protected entry points only (uses capabilities)
- Up to 3 x faster for local invocations

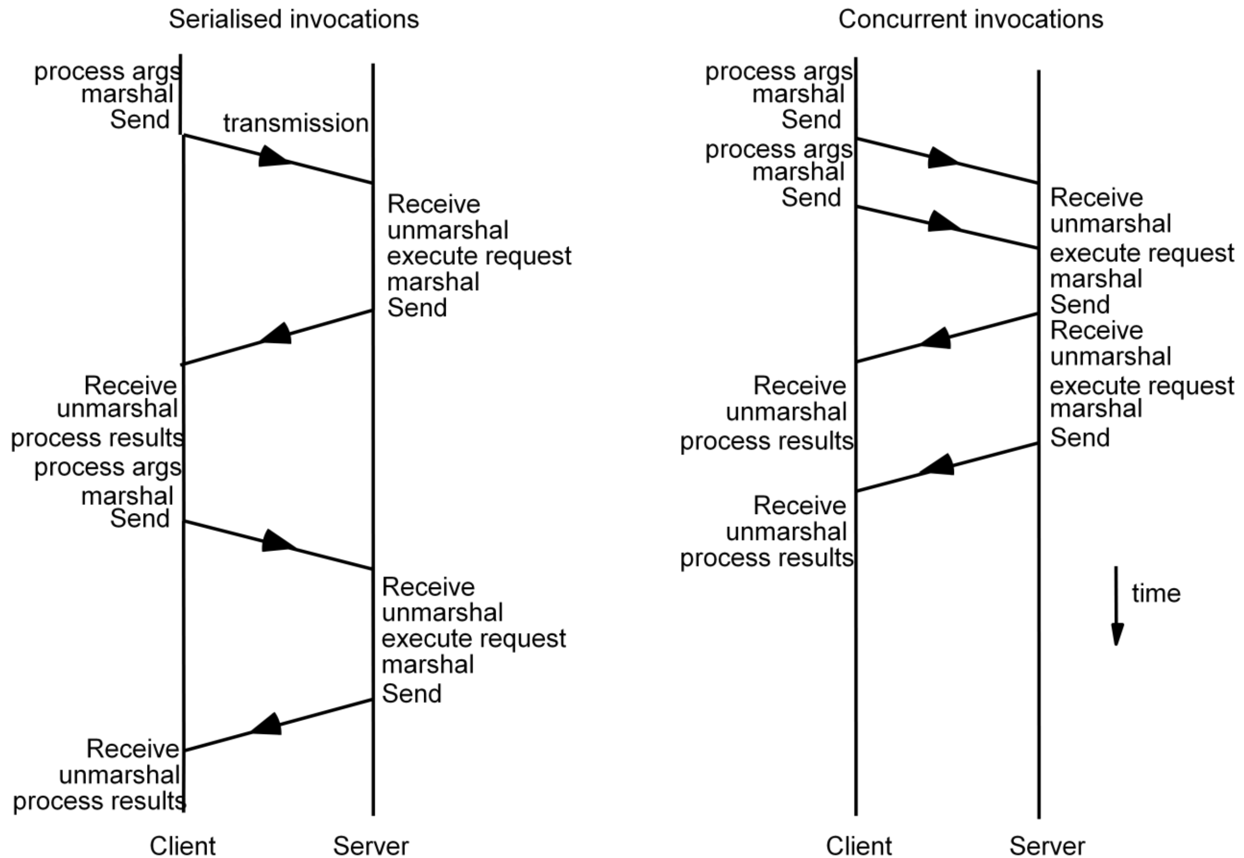


Asynchronous operation

- Performance characteristics of the Internet
  - High latencies, low bandwidths and high server loads
  - Network disconnection and reconnection.
  - outweigh any benefits that the OS can provide
- Asynchronous operation
  - Concurrent invocations
    - E.g., the browser fetches multiple images in a home page by concurrent *GET* requests
  - Asynchronous invocation: non-blocking call
    - E.g., CORBA *oneway* invocation: maybe semantics, or collect result by a separate call
- Persistent asynchronous invocations
  - Designed for *disconnected operation*
  - Try indefinitely to perform the invocation, until it is known to have succeeded or failed, or until the application cancels the invocation
  - QRPC (Queued RPC)
    - Client queues outgoing invocation requests in a stable log
    - Server queues invocation results
- The issues to programmers
  - How user can continue while the results of invocations are still not known?

The following figure shows the potential benefits of interleaving invocations (such as HTTP requests) between a client and a single server on a single-processor machine. In the serialized case, the client marshals the arguments, calls the *Send* operation and then waits until the reply from the server arrives – whereupon it *Receives*, unmarshals and then processes the results. After this it can make the second invocation.

Times for serialized and concurrent invocations



In the concurrent case, the first client thread marshals the arguments and calls the *Send* operation. The second thread then immediately makes the second invocation. Each thread waits to receive its results. The total time taken is liable to be lower than in the serialized case, as the figure shows. Similar benefits apply if the client threads make concurrent requests to several servers, and if the client executes on a multiprocessor even greater throughput is potentially possible, since the two threads' processing can also be overlapped.

Operating System Architecture

- A key principle of distributed systems is openness.
- The major kernel architectures:
    - Monolithic kernels
    - Micro-kernels

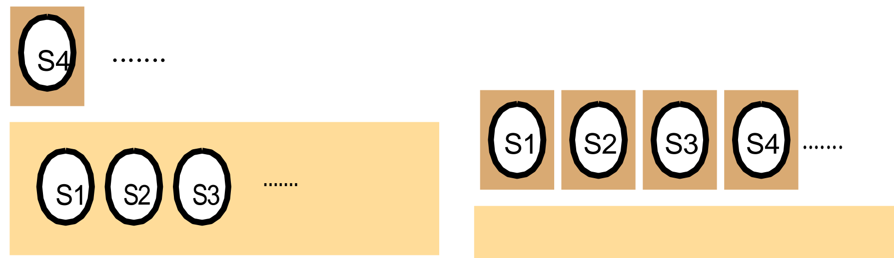
- An open distributed system should make it possible to:
    - Run only that system software at each computer that is necessary for its particular role in the system architecture. For example, system software needs for PDA and dedicated server are different. Loading redundant modules wastes memory resources.
    - Allow the software (and the computer) implementing any particular service to be changed independent of other facilities.
    - Allow for alternatives of the same service to be provided, when this is required to suit different users or applications.

- Introduce new services without harming the integrity of existing ones.

- A guiding principle of operating system design:
  - The separation of fixed resource management –mechanisms– from resource management –policies‖, which vary from application to application and service to service.
  - For example, an ideal scheduling system would provide mechanisms that enable a multimedia application such as videoconferencing to meet its real-time demands The kernel would provide only the most basic mechanisms upon which the general resource management tasks at a node are carried out.
  - Server modules would be dynamically loaded as required, to implement the required resourced management policies for the currently running applications.
  - while coexisting with a non-real-time application such as web browsing.

- Monolithic Kernels
  - A monolithic kernel can contain some server processes that execute within its address space, including file servers and some networking.
  - The code that these processes execute is part or the standard kernel configuration.

Monolithic kernel and microkernel



MonolithicKernel Microkernel

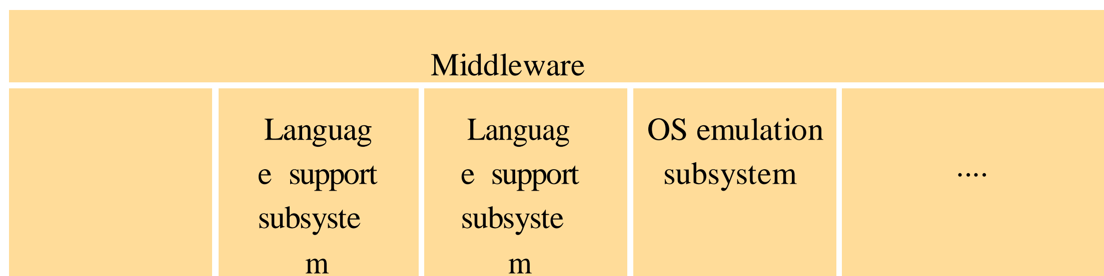
Key:

Server: Kernelcodeanddata: Dynamically loaded serverplugin

- Microkernel
  - The microkernel appears as a layer between hardware layer and a layer consisting of major systems.

    If performance is the goal, rather than portability, then middleware may use the facilities of the microkernel directly.

The role of the microkernel



Microkernel

Hardware

The microkernel supports middleware via subsystems

- Monolithic and Microkernel comparison
  - The advantages of a microkernel
    - Its extensibility
    - Its ability to enforce modularity behind memory protection boundaries.
    - Its small kernel has less complexity.
  - The advantages of a monolithic
    - The relative efficiency with which operations can be invoked because even invocation to a separate user-level address space on the same node is more costly.

- Hybrid Approaches
  - Pure microkernel operating system such as Chorus & Mach have changed over a time to allow servers to be loaded dynamically into the kernel address space or into a user-level address space.
    In some operating system such as SPIN, the kernel and all dynamically loaded modules grafted onto the kernel execute within a single address space

Case Study of a Distributed Operating System

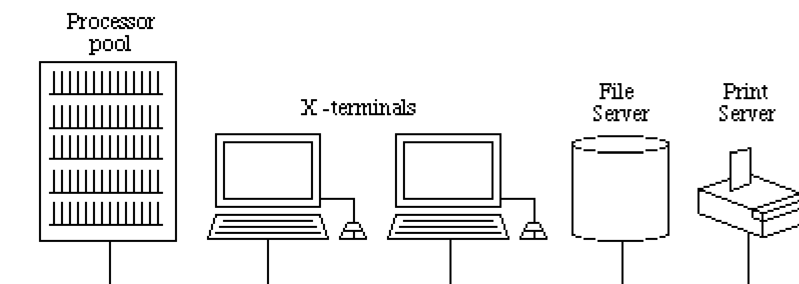
Introduction to Amoeba

- Originated at a university in Holland, 1981
- Currently used in various EU countries
- Built from the ground up. UNIX emulation added later
- Goal was to build a transparent distributed operating system
- Resources, regardless of their location, are managed by the system, and the user is unaware of where processes are actually run

---

The Amoeba System Architecture

- Assumes that a large number of CPUsare available and that each CPU ha 10s of Mb of memory
- CPUs are organised into processor pools



- CPUs do not need to be of the same architecture (can mix SPARC, Motorola PowerPC, 680x0, Intel, Pentium, etc.)
- When a user types a command, system determines which CPU(s) to execute it on. CPUs can be timeshared.
- Terminals are X-terminals or PCs running X emulators
- The processor pool doesn't have to be composed of CPU boards enclosed in a cabinet, they can be on PCs, etc., in different rooms, countries,...
- Some servers (e.g., file servers) run on dedicated processors, because they need to be available all the time

The Amoeba Microkernel

- The Amoeba microkernel is used on all terminals (with an on-board processor), processors, and servers
- The microkernel

  manages processes and threads

  provides low-level memory management support

  supports interprocess communication (point-to-point and group)

  handles low-level I/O for the devices attached to the machine

---

The Amoeba Servers: Introduction

- OS functionality not provided by the microkernel is performed by Amoeba servers
- To use a server, the client calls a stub procedure which marshalls parameters, sends the message, and blocks until the result comes back

Server Basics

- Amoeba uses capabilities
- Every OS data structure is an object, managed by a server
- To perform an operation on an object, a client performs an RPC with the appropriate server, specifying the object, the operation to be performed and any parameters needed.
- The operation is transparent (client does not know where server is, nor how the operation is performed)
- Capabilites

  To create an object the client performs an RPC with the server

  Server creates the object and returns a capability

  To use the object in the future, the client must present the correct capability

Bits	48	24	8	48
	Server Port	Object	Rights	Check

The check field is used to protect the capability against forgery

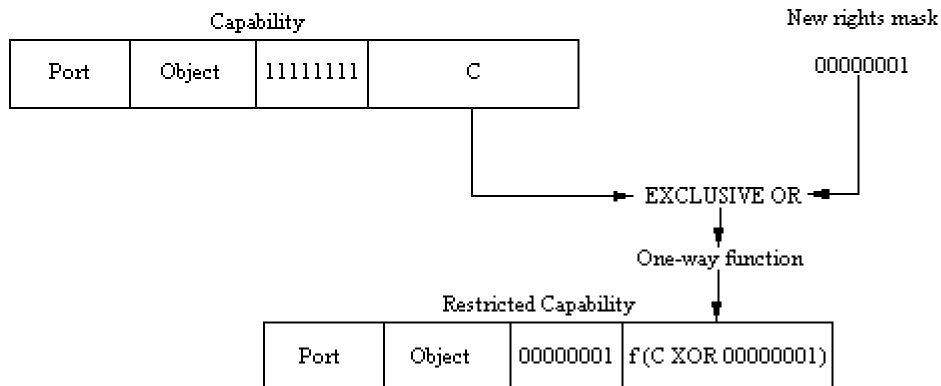
- Object protection

  When an object is created, server generates random check field, which it stores both in the capability and in its own tables

  The rights bits in the capability are set to on

  The server sends the owner capability back to the client

  Creating a capability with restricted rights



  Client can send this new capability to another process

Process Management

- All processes are objects protected by capabilities
- Processes are managed at 3 levels

  by process servers, part of the microkernel

  by library procedures which act as interfaces

  by the run server, which decides where to run the processes

- Process management uses process descriptors

  Contains:

    platform description

    process' owner's capability

    etc

Memory Management

- Designed with performance, simplicity and economics in mind
- Process occupies contiguous segments in memory
- All of a process is constantly in memory
- Process is never swapped out or paged

Communication

- Point-to-point (RPC) and Group

The Amoeba Servers

The File System

- Consists of the Bullet (File) Server, the Directory Server, and the Replication Server

The Bullet Server

- Designed to run on machines with large amounts of RAM and huge local disks
- Used for file storage
- Client process creates a file using the *create* call
- Bullet server returns a capability that can be used to *read* the file with
- Files are immutable, and file size is known at file creation time. Contiguous allocation policies used

The Directory Server

- Used for file naming
- Maps from ASCII names to capabilities
- Directories also protected by capabilities
- Directory server can be used to name ANY object, not just files and directories

The Replication Server

- Used for fault tolerence and performance
- Replication server creates copies of files, when it has time

Other Amoeba Servers

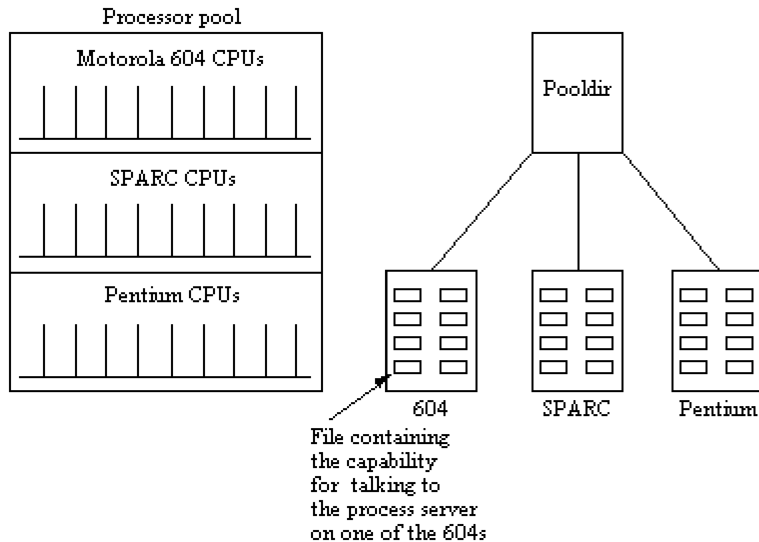
The Run Server

- When user types a command, two decisions have to be made

On which architecture should the process be run?

Which processor should be chosen?

- Run server manages the processor pools



- Uses processes process descriptor to identify appropriate target architecture
- Checks which of the available processors have sufficient memory to run the process
- Estimates which of the remaining processor has the most available compute power

The Boot Server

- Provides a degree of fault tolerance
- Ensures that servers are up and running
- If it discovers that a server has crashed, it attempts to restart it, otherwise selects another processor to provide the service
- Boot server can be replicated to guard against its own failure

PEER-TO-PEER SYSTEMS

Peer-to-peer (**P2P**) computing or networking is a distributed application architecture that partitions tasks or work loads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes.

Peers make a portion of their resources, such as processing power, disk storage or network and width, directly available to other network participants, without the need for central coordination by

servers or stable hosts.[1] Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model in which the consumption and supply of resources is divided. Emerging collaborative P2P systems are going beyond the era of peers doing similar things while sharing resources, and are looking for diverse peers that can bring in unique resources and capabilities to a virtual community thereby empowering it to engage in greater tasks beyond those that can be accomplished by individual peers, yet that are beneficial to all the peers.[2]

While P2P systems had previously been used in many application domains,[3] the architecture was popularized by the file sharing system Napster, originally released in 1999. The concept has inspired new structures and philosophies in many areas of human interaction. In such social contexts, peer-to-peer as a meme refers to theegalitarian social networking that has emerged throughout society, enabled by Internettechnologies in general.

The demand for services in the Internet can be expected to grow to a scale that is limited only by the size of the world's population. The goal of peer-to-peer systems is to enable the sharing of data and resources on a very large scale by eliminating any requirement for separately managed servers and their associated infrastructure. The scope for expanding popular services by adding to the number of the computers hosting them is limited when all the hosts must be owned and managed by the service provider. Administration and fault recovery costs tend to dominate. The network bandwidth that can be provided to a single server site over available physical links is also a major constraint. System-level services such as Sun NFS (Section 12.3), the Andrew File System (Section 12.4) or video servers (Section 20.6.1) and application-level services such as Google, Amazon or eBay all exhibit this problem to varying degrees.

Peer-to-peer systems aim to support useful distributed services and applications using data and computing resources available in the personal computers and workstations that are present in the Internet and other networks in ever-increasing numbers. This is increasingly attractive as the performance difference between desktop and server machines narrows and broadband network connections proliferate. But there is another, broader aim: has defined peer-topeer applications as 'applications that exploit resources available at the edges of the Internet – storage, cycles, content, human presence'. Each type of resource sharing mentioned in that definition is already represented by distributed applications available for most types of personal computer. The purpose of this chapter is to describe some general techniques that simplify the construction of peer-to-peer applications and enhance their scalability, reliability and security.

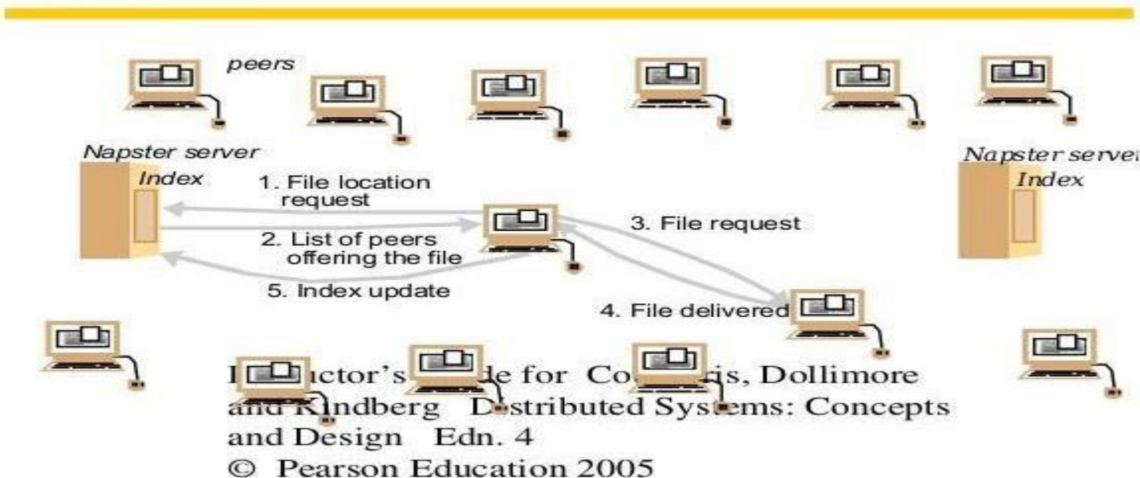
Traditional client-server systems manage and provide access to resources such as files, web pages or other information objects located on a single server computer or a small cluster of tightly coupled

servers. With such centralized designs, few decisions are required about the placement of the resources or the management of server hardware resources, but the scale of the service is limited by the server hardware capacity and network connectivity. Peer-to-peer systems provide access to information resources located on computers throughout a network (whether it be the Internet or a corporate network). Algorithms for the placement and subsequent retrieval of information objects are a key aspect of the system design. The aim is to deliver a service that is fully decentralized and self-organizing, dynamically balancing the storage and processing loads between all the participating computers as computers join and leave the service. Peer-to-peer systems share these characteristics:

- Their design ensures that each user contributes resources to the system.
- Although they may differ in the resources that they contribute, all the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.
- Their correct operation does not depend on the existence of any centrally administered systems.
- They can be designed to offer a limited degree of anonymity to the providers and users of resources.
- A key issue for their efficient operation is the choice of an algorithm for the placement of data across many hosts and subsequent access to it in a manner that balances the workload and ensures availability without adding undue overheads.

Napster and its legacy

The first application in which a demand for a globally scalable information storage and retrieval service emerged was the downloading of digital music files. Both the need for and the feasibility of a peer-to-peer solution were first demonstrated by the Napster filesharing system [OpenNap 2001] which provided a means for users to share files. Napster became very popular for music exchange soon after its launch in 1999. At its peak, several million users were registered and thousands were swapping music files simultaneously. Napster's architecture included centralized indexes, but users supplied the files, which were stored and accessed on their personal computers. Napster's method of operation is illustrated by the sequence of steps shown in Figure 10.2.



Note that in step 5clients are expected to add their own music files to the pool of shared resources by transmitting a link to the Napster indexing service for each available file. Thus the motivation for Napster and the key to its success was the making available of a large, widely distributed set of files to users throughout the Internet, fulfilling Shirky's dictum by providing access to 'shared resources at the edges of the Internet'. Napster was shut down as a result of legal proceedings instituted against the operators of the Napster service by the owners of the copyright in some of the material

(i.e., digitally encoded music) that was made available on it (see the box below). Anonymity for the receivers and the providers of shared data and other resources is a concern for the designers of peer-to-peer systems. In systems with many nodes, the routing of requests and results can be made sufficiently tortuous to conceal their source and the contents of files can be distributed across multiple nodes, spreading the responsibility for making them available. Mechanisms for anonymous communication that are resistant to most forms of traffic analysis are available If files are also encrypted before they are placed on servers, the owners of the servers can plausibly deny any knowledge of the contents. But these anonymity techniques add to the cost of resource sharing, and recent work has shown that the anonymity available is weak against some attacks The Freenet projects are focused on providing Internet-wide file services that offer anonymity for the providers and users of the shared files. Ross Anderson has proposed the Eternity Service , a storage service that provides long-term guarantees of data

Peer-to-peer systems and copyright ownership issues

The developers of Napster argued that they were not liable for the infringement of the copyright owners' rights because they were not participating in the copying process, which was performed entirely between users' machines. Their argument failed because the index servers were deemed an essential part of the process. Since the index servers were located at well-known addresses, their operators were unable to remain anonymous and so could be targeted in lawsuits.

A more fully distributed file-sharing service might have achieved a better separation of legal responsibilities, spreading the responsibility across all of the users and thus making the pursuit of legal remedies very difficult, if not impossible.

Whatever view one takes about the legitimacy of file copying for the purpose of sharing copyright-protected material, there are legitimate social and political justifications for the anonymity of clients and servers in some application contexts. The most persuasive justification arises when anonymity is used to overcome censorship and maintain freedom of expression for individuals in oppressive societies or organizations. It is known that email and web sites have played a significant role in achieving public awareness at times of political crisis in such societies; their role could be strengthened if the authors could be protected by anonymity.

Peer-to-peer middleware systems are designed specifically to meet the need for the automatic placement and subsequent location of the distributed objects managed by peer-to-peer systems and applications.

Functional requirements • The function of peer-to-peer middleware is to simplify the construction of services that are implemented across many hosts in a widely distributed network. To achieve this it must enable clients to locate and communicate with any individual resource made available to a service, even though the resources are widely distributed amongst the hosts. Other important requirements include the ability to add new resources and to remove them at will and to add hosts to the service and remove them. Like other middleware, peer-to-peer middleware should offer a simple programming interface to application programmers that is independent of the types of distributed resource that the application manipulates.

Non-functional requirements • To perform effectively, peer-to-peer middleware must also address the following non-functional requirements

Global scalability: One of the aims of peer-to-peer applications is to exploit the hardware resources of very large numbers of hosts connected to the Internet. Peer-topeer middleware must therefore be designed to support applications that access millions of objects on tens of thousands or hundreds of thousands of hosts.

Load balancing: The performance of any system designed to exploit a large number of computers depends upon the balanced distribution of workload across them. For the systems we are

considering, this will be achieved by a random placement of resources together with the use of replicas of heavily used resources.
Optimization for local interactions between neighbouring peers: The ‘network distance’ between nodes that interact has a substantial impact on the latency of individual interactions, such as client requests for access to resources. Network traffic loadings are also impacted by it. The middleware should aim to place resources close to the nodes that access them the most.

Accommodating to highly dynamic host availability: Most peer-to-peer systems are constructed from host computers that are free to join or leave the system at any time. The hosts and network segments used in peer-to-peer systems are not owned or managed by any single authority; neither their reliability nor their continuous participation in the provision of a service is guaranteed. A major challenge for peerto- peer systems is to provide a dependable service despite these facts. As hosts join the system, they must be integrated into the system and the load must be redistributed to exploit their resources. When they leave the system whether voluntarily or involuntarily, the system must detect their departure and redistribute their load and resources.

Routing overlays

In peer-to-peer systems a distributed algorithm known as a *routing overlay* takes responsibility for locating nodes and objects. The name denotes the fact that the middleware takes the form of a layer that is responsible for routing requests from any client to a host that holds the object to which the request is addressed. The objects of interest may be placed at and subsequently relocated to any node in the network without client involvement. It is termed an overlay since it implements a routing mechanism in the application layer that is quite separate from any other routing mechanisms deployed at the network level such as IP routing. The routing overlay ensures that any node can access any object by routing each request through a sequence of nodes, exploiting knowledge at each of them to locate the destination object. Peer-to-peer systems usually store multiple replicas of objects to ensure availability. In that case, the routing overlay maintains knowledge of the location of all the available replicas and delivers requests to the nearest ‘live’ node (i.e. one that has not failed) that has a copy of the relevant object. The GUIDs used to identify nodes and objects are an example of the ‘pure’ names. These are also known as opaque identifiers, since they reveal nothing about the locations of the objects to which they refer. The main task of a routing overlay is the following:
Routing of requests to objects: A client wishing to invoke an operation on an object submits a request including the object’s GUID to the routing overlay, which routes the request to a node at which a replica of the object resides.
But the routing overlay must also perform some other tasks:
Insertion of objects: A node wishing to make a new object available to a peer-to-peer service computes a GUID for the object and announces it to the routing overlay,which then ensures that the object is reachable by all other clients.
Deletion of objects: When clients request the removal of objects from the service the routing overlay must make them unavailable.
Node addition and removal: Nodes (i.e., computers) may join and leave the service. When a node joins the service, the routing overlay arranges for it to assume some of the responsibilities of other nodes. When a node leaves (either voluntarily or as a result of a system or network fault), its responsibilities are distributed amongst the other nodes.

Overlay case studies: Pastry, Tapestry

The prefix routing approach is adopted by both Pastry and Tapestry. Pastry is the message routing infrastructure deployed in several applications including PAST [Druschel and Rowstron

2001], an archival (immutable) file storage system implemented as a distributed hash table with the

API and Squirrel, a peerto- peer web caching service described. Pastry has a straightforward but effective design that makes it a good first example for us to study in detail. Tapestry is the basis for the OceanStore storage system, which we describe in . It has a more complex architecture than Pastry because it aims to support a wider range of locality approaches.

Pastry

All the nodes and objects that can be accessed through Pastry are assigned 128-bit GUIDs. For nodes, these are computed by applying a secure hash function to the public key with which each node is provided. For objects such as files, the GUID is computed by applying a secure hash function to the object's name or to some part of the object's stored state. The resulting GUIDs have the usual properties of secure hash values – that is, they are randomly distributed in the range 0 to 2128–1. They provide no clues as to the value from which they were computed, and clashes between GUIDs for different nodes or objects are extremely unlikely. (If a clash occurs, Pastry detects it and takes remedial action.) In a network with *N* participating nodes, the Pastry routing algorithm will correctly route a message addressed to any GUID in *O(log N)* steps. If the GUID identifies a node that is currently active, the message is delivered to that node; otherwise, the message is delivered to the active node whose GUID is numerically closest to it. Active nodes take responsibility for processing requests addressed to all objects in their numerical neighbourhood. Routing steps involve the use of an underlying transport protocol (normally UDP) to transfer the message to a Pastry node that is 'closer' to its destination. But note that the closeness referred to here is in an entirely artificial space – the space of GUIDs. The real transport of a message across the Internet between two Pastry nodes may require a substantial number of IP hops.

UNIT IV

Distributed File Systems: Introduction, File service Architecture, Case Study1: Sun Network File System, Case Study 2: The Andrew File System.
Name Services: Introduction, Name Services and the Domain Name System, Directory Services, Case study of the Global Name Service.
Distributed Shared Memory: Introduction Design and Implementation issues, Sequential consistency and Ivy case study, Release consistency and Munin case study, other consistency models.

DISTRIBUTED FILE SYSTEMS

A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files. File systems provide directory services, which convert a file name (possibly a hierarchical one) into an internal identifier (e.g. inode, FAT index). They contain a representation of the file data itself and methods for accessing it (read/write). The file system is responsible for controlling access to the data and for performing low-level operations such as buffering frequently used data and issuing disk I/O requests.

A distributed file system is to present certain degrees of transparency to the user and the system:
Access transparency: Clients are unaware that files are distributed and can access them in the same way as local files are accessed.
Location transparency: A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.
Concurrency transparency: All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.
Failure transparency: The client and client programs should operate correctly after a server failure.
Heterogeneity: File service should be provided across different hardware and operating system platforms.
Scalability: The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).
Replication transparency: To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.
Migration transparency: Files should be able to move around without the client's knowledge.
Support fine-grained distribution of data: To optimize performance, we may wish to locate

individual objects near the processes that use them.
Tolerance for network partitioning: The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

File service types

To provide a remote system with file service, we will have to select one of two models of operation. One of these is the upload/download model. In this model, there are two fundamental operations: *read file* transfers an entire file from the server to the requesting client, and *write file* copies the file back to the server. It is a simple model and efficient in that it provides local access to the file when it is being used. Three problems are evident. It can be wasteful if the client needs access to only a small amount of the file data. It can be problematic if the client doesn't have enough space to cache the entire file. Finally, what happens if others need to modify the same file?

The second model is a remote access model. The file service provides remote operations such as *open*, *close*, *read bytes*, *write bytes*, *get attributes*, etc. The file system itself runs on servers. The drawback in this approach is the servers are accessed for the duration of file access rather than once to download the file and again to upload it.

Another important distinction in providing file service is that of understanding the difference between *directory service* and *file service*. A directory service, in the context of file systems, maps human-friendly textual names for files to their internal locations, which can be used by the file service. The file service itself provides the file interface (this is mentioned above). Another component of file distributed file systems is the client module. This is the client-side interface for file and directory service. It provides a local file system interface to client software (for example, the vnode file system layer of a UNIX kernel).

Introduction

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.
- Distributed file systems support the sharing of information in the form of files and hardware resources.
- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Figure 1 provides an overview of types of storage system.

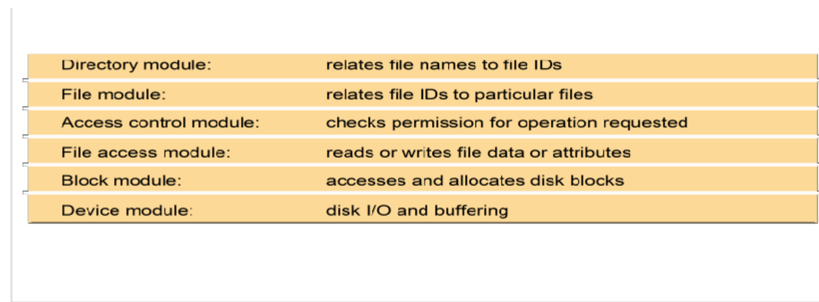
Figure 1. Storage systems and their properties

	Sharing	*Persistence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:
1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

- Figure 2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system.

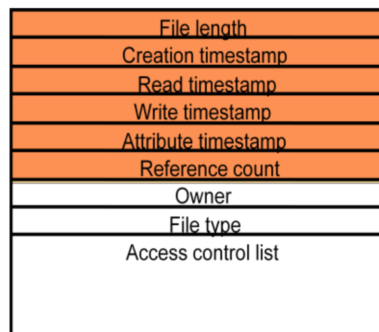
Figure 2. File system modules



Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- Files contain both data and attributes.
- A typical attribute record structure is illustrated in Figure 3.

Figure 3. File attribute record structure



- Figure 4 summarizes the main operations on files that are available to applications in UNIX systems.

Figure 4. UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from buffer. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to offset (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Gets the file attributes for file *name* into *buffer*.

- Distributed File system requirements
  - Related requirements in distributed file systems are:
    - Transparency
    - Concurrency
    - Replication
    - Heterogeneity
    - Fault tolerance
    - Consistency
    - Security
    - Efficiency

Case studies

File service architecture • This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a *stateless* implementation of the server module.

SUN NFS • Sun Microsystems's *Network File System* (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

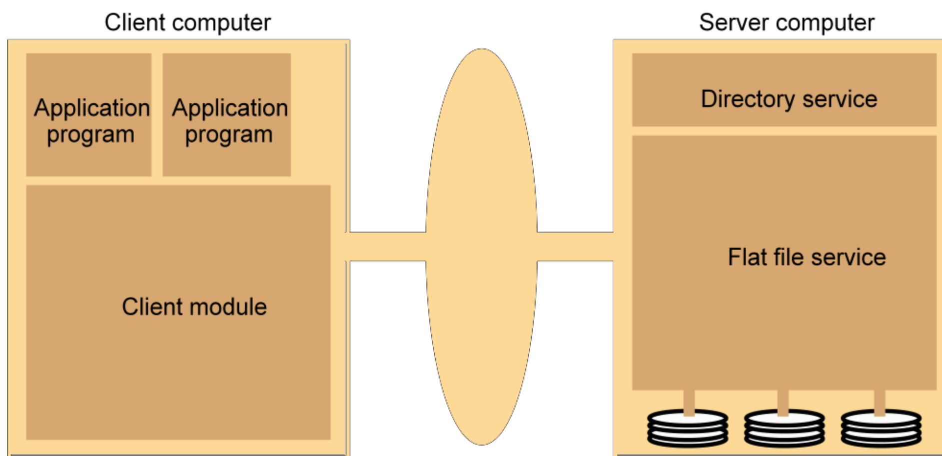
Andrew File System • Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system. The design of the Andrew File System (henceforth abbreviated AFS) reflects

an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version.

File Service Architecture

- An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components:
  - A flat file service
  - A directory service
  - A client module.
- The relevant modules and their relationship is shown in Figure 5.

Figure 5. File service architecture



- The Client module implements exported interfaces by flat file and directory services on server side.
- Responsibilities of various modules can be defined as follows:
  - Flat file service:
    - Concerned with the implementation of operations on the contents of file. Unique File Identifiers (UFIDs) are used to refer to files in all requests for

      flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.
  - Directory service:
    - Provides mapping between text names for the files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.
  - Client module:
    - It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.

- It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.

- Flat file service interface:
  - Figure 6 contains a definition of the interface to a flat file service.

Figure 6. Flat file service operations

Read(FileId, i, n) -> Data -throws BadPosition	if $1 \le i \le Length(File)$: Reads a sequence of up to n items from a file starting at item *i* and returns it in *Data.*
Write(FileId, i, Data) -throws BadPosition	if $1 \le i \le Length(File)+1$: Write a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() -> FileId	Creates a new file of length0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 3.)

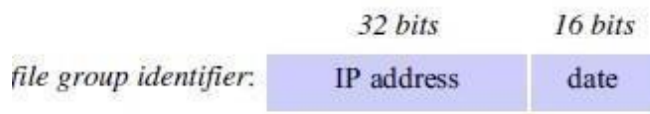
- Access control
  - In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.
- Directory service interface
  - Figure 7 contains a definition of the RPC interface to a directory service.

Figure 7. Directory service operations

Lookup(Dir, Name) -> FileId -throws NotFound	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, File) -throws NameDuplicate	If *Name* is not in the directory, adds(*Name,File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name)	If *Name* is in the directory, the entry containing *Name* is removed from the directory. If *Name* is not in the directory: throws an exception.
GetNames(Dir, Pattern) -> NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*.

- Hierarchic file system
    - A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.
- File Group
    - A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.
        - A similar construct is used in a UNIX file system.
        - It helps with distributing the load of file serving between several servers.
        - File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).

To construct globally unique ID we use some unique attribute of the machine on which it is created. E.g: IP number, even though the file group may move subsequently.

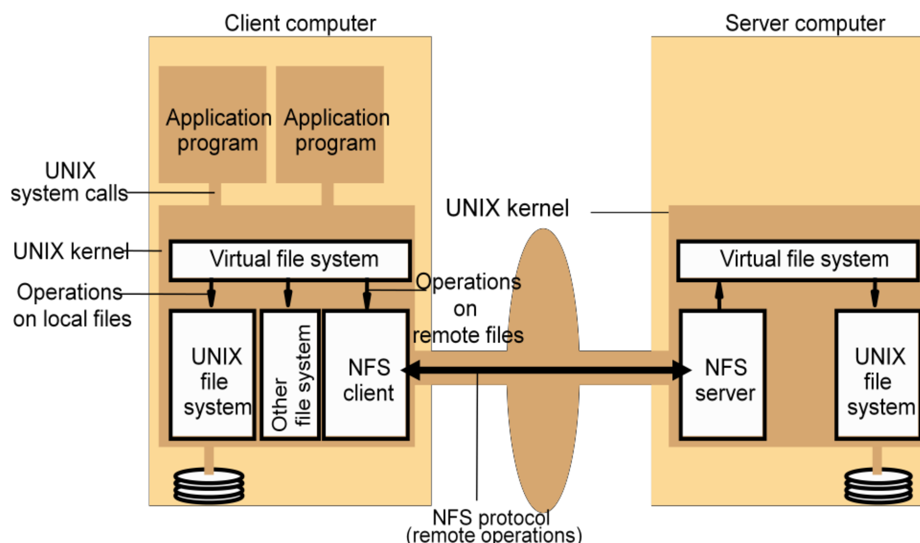


DFS: Case Studies

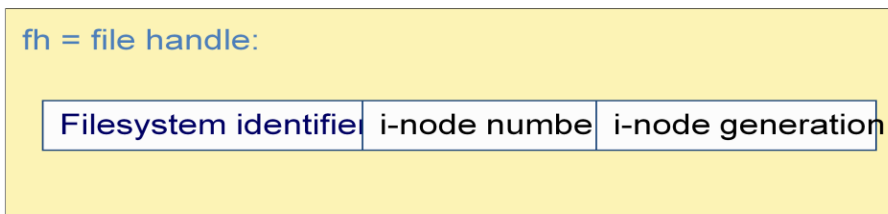
- NFS (Network File System)
    - Developed by Sun Microsystems (in 1985)
    - Most popular, open, and widely used.
    - NFS protocol standardized through IETF (RFC 1813)
- AFS (Andrew File System)
    - Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
    - A research project to create campus wide file system.
    - Public domain implementation is available on Linux (LinuxAFS)
    - It was adopted as a basis for the DCE/DFS file system in the Open Software Foundation (OSF, www.opengroup.org) DEC (Distributed Computing Environment

NFS architecture

Figure 8 shows the architecture of Sun NFS



- The file identifiers used in NFS are called file handles.



- A simplified representation of the RPC interface provided by NFS version 3 servers is shown in Figure 9.

Figure 9. NFS server operations (NFS Version 3 protocol, simplified)

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> status*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

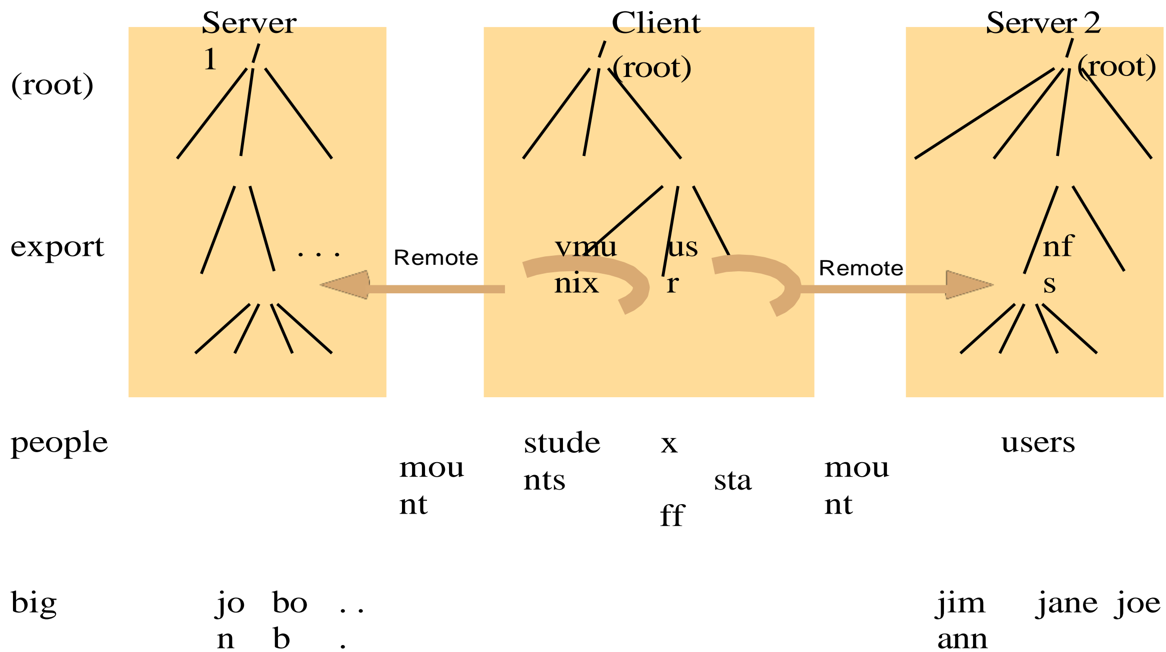
- NFS access control and authentication
  - The NFS server is stateless server, so the user's identity and access rights must be checked by the server on each request.
    - In the local file system they are checked only on the file's access permission attribute.
  - Every client request is accompanied by the userID and groupID
    - It is not shown in the Figure 8.9 because they are inserted by the RPC system.
  - Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution.

- Mount service
  - Mount operation:

mount(remotehost, remotedirectory, localdirectory)

- Server maintains a table of clients who have mounted filesystems at that server.
- Each client maintains a table of mounted file systems holding:
  < IP address, port number, file handle>
- Remote file systems may be hard-mounted or soft-mounted in a client computer.
- Figure 10 illustrates a Client with two remotely mounted file stores.

Figure 10. Local and remote file systems accessible on an NFS client



- Automounter
  - The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever an ‘empty’ mount point is referenced by a client.
    - Automounter has a table of mount points with a reference to one or more NFS servers listed against each.
    - it sends a probe message to each candidate server and then uses the mount service to mount the file system at the first server to respond.
  - Automounter keeps the mount table small.
  - Automounter Provides a simple form of replication for read-only file systems.
    - E.g. if there are several servers with identical copies of /usr/lib then each server will have a chance of being mounted at some clients.
- Server caching
  - Similar to UNIX file caching for local files:
    - pages (blocks) from disk are held in a main memory buffer cache until the space is required for newer pages. Read-ahead and delayed-write optimizations.
    - For local files, writes are deferred to next sync event (30 second intervals).
    - Works well in local context, where files are always accessed through the local cache, but in the remote case it doesn't offer necessary synchronization guarantees to clients.

  - NFS v3 servers offers two strategies for updating the disk:
    - Write-through - altered pages are written to disk as soon as they are received at the server. When a write() RPC returns, the NFS client knows that the page is on the disk.
    - Delayed commit - pages are held only in the cache until a commit() call is received for the relevant file. This is the default mode used by NFS v3 clients. A commit() is issued by the client whenever a file is closed.
- Client caching
  - Server caching does nothing to reduce RPC traffic between client and server
    - further optimization is essential to reduce server load in large networks.
    - NFS client module caches the results of read, write, getattr, lookup and readdir operations
    - synchronization of file contents (one-copy semantics) is not guaranteed when two or more clients are sharing the same file.
  - Timestamp-based validity check
    - It reduces inconsistency, but doesn't eliminate it.
    - It is used for validity condition for cache entries at the client:

$$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$$

t	freshness guarantee
Tc	time when cache entry was last validated
Tm	time when block was last updated at server
T	current time

- it is configurable (per file) but is typically set to 3 seconds for files and 30 secs. for directories.
- it remains difficult to write distributed applications that share files with NFS.

- Other NFS optimizations
  - Sun RPC runs over UDP by default (can use TCP if required).
  - Uses UNIX BSD Fast File System with 8-kbyte blocks.
  - reads() and writes() can be of any size (negotiated between client and server).
  - The guaranteed freshness interval t is set adaptively for individual files to reduce getattr() calls needed to update Tm.
  - File attribute information (including Tm) is piggybacked in replies to all file requests.
- NFS performance
  - Early measurements (1987) established that:
    - Write() operations are responsible for only 5% of server calls in typical UNIX environments.
      - hence write-through at server is acceptable.
    - Lookup() accounts for 50% of operations -due to step-by-step pathname resolution necessitated by the naming and mounting semantics.
  - More recent measurements (1993) show high performance.
    - see www.spec.org for more recent measurements.
- NFS summary
  - NFS is an excellent example of a simple, robust, high-performance distributed service.
  - Achievement of transparencies are other goals of NFS:
    - Access transparency:
      - The API is the UNIX system call interface for both local and remote files.
    - Location transparency:
      - Naming of filesystems is controlled by client mount operations, but transparency can be ensured by an appropriate system configuration.
    - Mobility transparency:
      - Hardly achieved; relocation of files is not possible, relocation of filesystems is possible, but requires updates to client configurations.
    - Scalability transparency:
      - File systems (file groups) may be subdivided and allocated to

separate servers.

- Replication transparency:
  - – Limited to read-only file systems; for writable files, the SUN Network Information Service (NIS) runs over NFS and is used to replicate essential system files.
- Hardware and software operating system heterogeneity:
  - – NFS has been implemented for almost every known operating system and hardware platform and is supported by a variety of filling systems.
- Fault tolerance:
  - – Limited but effective; service is suspended if a server fails. Recovery from failures is aided by the simple stateless design.
- Consistency:
  - – It provides a close approximation to one-copy semantics and meets the needs of the vast majority of applications.
  - – But the use of file sharing via NFS for communication or close coordination between processes on different computers cannot be recommended.
- Security:
  - – Recent developments include the option to use a secure RPC implementation for authentication and the privacy and security of the data transmitted with read and write operations.
  - – Efficiency:
    - NFS protocols can be implemented for use in situations that generate very heavy loads.

Case Study: The Andrew File System (AFS)

AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

Whole-file serving: The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).

Whole file caching: Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' *open* requests in preference to remote copies whenever possible.

- Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations.
- AFS is implemented as two software components that exist at UNIX processes called Vice and Venus**.**

Scenario • Here is a simple scenario illustrating the operation of AFS:

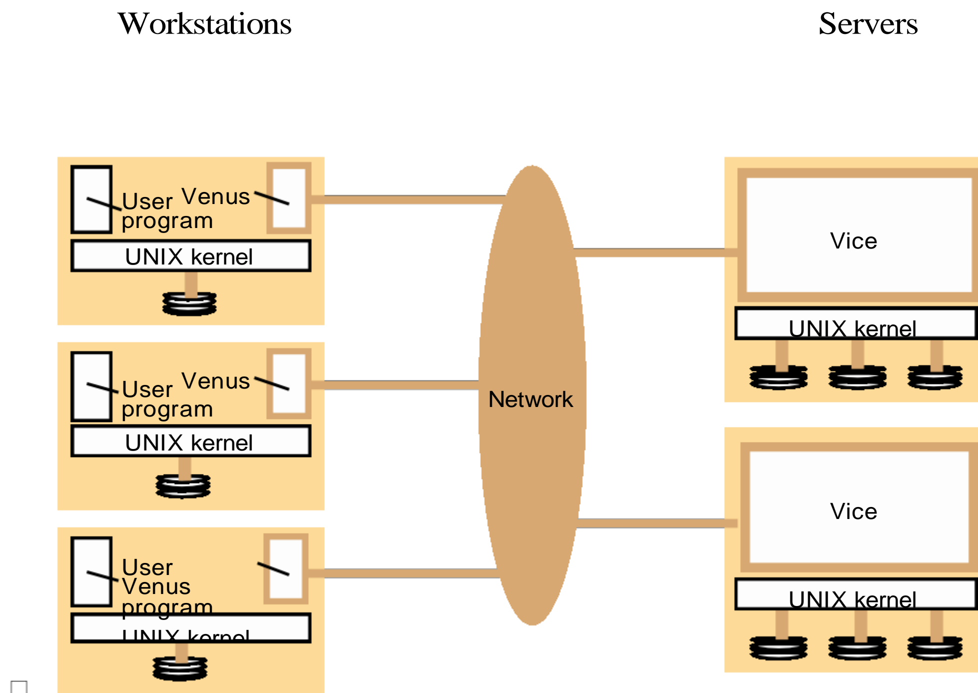
1. When a user process in a client computer issues an *open* system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.

2. The copy is stored in the local UNIX file system in the client computer. The copy is then *open*ed and the resulting UNIX file descriptor is returned to the client.

3. Subsequent *read*, *write* and other operations on the file by processes in the client computer are applied to the local copy.

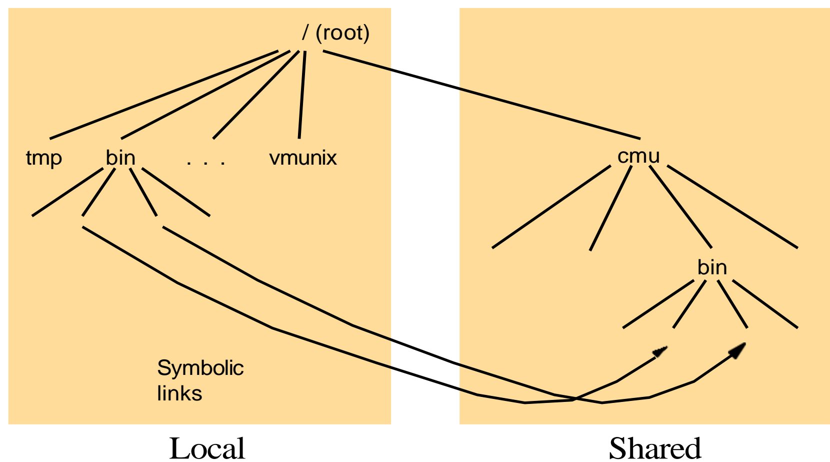
4. When the process in the client issues a *close* system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client's local disk is retained in case it is needed again by a user-level process on the same workstation.

Figure 11. Distribution of processes in the Andrew File System



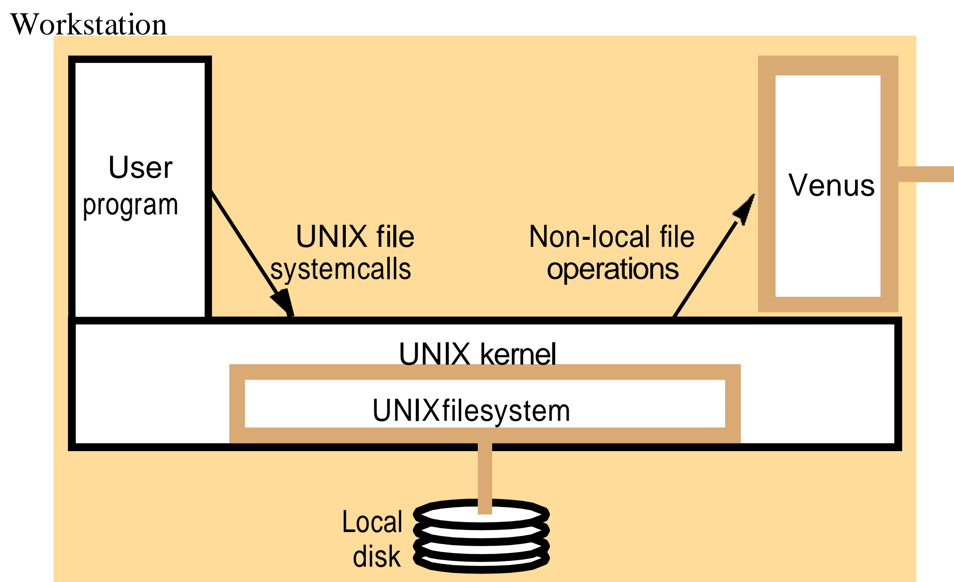
- The files available to user processes running on workstations are either local or shared.
- Local files are handled as normal UNIX files.
- They are stored on the workstation's disk and are available only to local user processes.
- Shared files are stored on servers, and copies of them are cached on the local disks of workstations.
- The name space seen by user processes is illustrated in Figure 12.

Figure 12. File name space seen by clients of AFS



- The UNIX kernel in each workstation and server is a modified version of BSD UNIX.
- The modifications are designed to intercept open, close and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer. (Figure 13)

Figure 13. System call interception in AFS



- Figure 14 describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues system calls.

Figure 14. implementation of file system calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→ ←	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.		
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

- Figure 15 shows the RPC calls provided by AFS servers for operations on files.

Figure 15. The main components of the Vice service interface

Fetch(fid) -> attr, data	Returns the attributes (status) and, optionally, the contents of file identified by the *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() -> fid	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

Other aspects

AFS introduces several other interesting design developments and refinements that we outline here, together with a summary of performance evaluation results:

1. UNIX kernel modifications
2. Location database
3. Threads
4. Read-only replicas
5. Bulk transfers
6. Partial file caching
7. Performance
8. Wide area support

Naming Services

Which one is easy for humans and machines? and why?

- 74.125.237.83 or **google.com**
- 128.250.1.22 or distributed systems website
- 128.250.1.25 or Prof. Buyya
- Disk 4, Sector 2, block 5 OR /usr/raj/hello.c

Introduction

- In a distributed system, names are used to refer to a wide variety of resources such as:
  - Computers, services, remote objects, and files, as well as users.
- Naming is fundamental issue in DS design as it facilitates communication and resource sharing.
  - A name in the form of URL is needed to access a specific web page.
  - Processes cannot share particular resources managed by a computer system unless they can name them consistently
  - Users cannot communicate within one another via a DS unless they can name one another, with email address.
- Names are not the only useful means of identification: descriptive attributes are another.

What are Naming Services?

- How do Naming Services facilitate communication and resource sharing?
  - An URL facilitates the localization of a resource exposed on the Web.
    - *e.g., abc.net.au means it is likely to be an Australian entity?*
  - A consistent and uniform naming helps processes in a distributed system to interoperate and manage resources.
    - *e.g., commercials use .com; non-profit organizations use .org*
  - Users refers to each other by means of their names (i.e. email) rather than their system ids
  - Naming Services are not only useful to locate resources but also to gather additional information about them such as attributes

What are Naming Services?

In a Distributed System, a Naming Service is a specific service whose aim is to provide a consistent and uniform naming of resources, thus allowing other programs or services to localize them and obtain the required metadata for interacting with them.

Key benefits

- Resource localization
- Uniform naming
- Device independent address (e.g., you can move domain name/web site from one server to another server seamlessly).

The role of names and name services

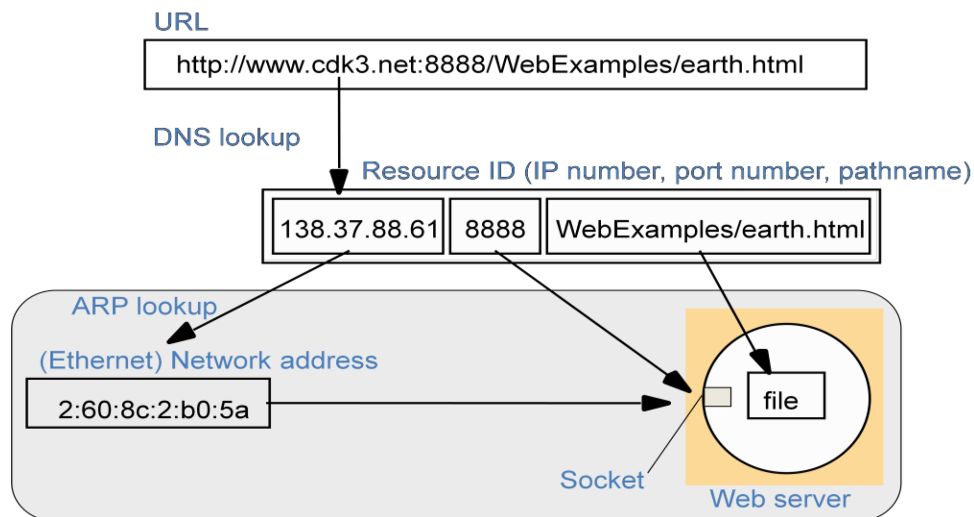
- Resources are accessed using *identifier* or *reference*

- An identifier can be stored in variables and retrieved from tables quickly
  - Identifier includes or can be transformed to an address for an object
    - *E.g. NFS file handle, Corba remote object reference*
  - A *name* is human-readable value (usually a string) that can be *resolved* to an identifier or address
    - *Internet domain name, file pathname, process number*
    - *E.g ./etc/passwd, http://www.cdk3.net/*
- For many purposes, names are preferable to identifiers
  - because the binding of the named resource to a physical location is deferred and can be changed
  - because they are more meaningful to users
- Resource names are *resolved* by name services
  - to give identifiers and other useful attributes

Requirements for name spaces

- Allow simple but meaningful names to be used
- Potentially infinite number of names
- Structured
  - to allow similar subnames without clashes
  - to group related names
- Allow re-structuring of name trees
  - for some types of change, old programs should continue to work
- Management of trust

Composed naming domains used to access a resource from a URL



A key attribute of an entity that is usually relevant in a distributed system is its address. For example:

• The DNS maps domain names to the attributes of a host computer: its IP address, the type of entry (for example, a reference to a mail server or another host) and, for example, the length of time the host's entry will remain valid.
• The X500 directory service can be used to map a person's name onto attributes including their email address and telephone number.
• The CORBA Naming Service maps the name of a remote object onto its remote object reference, whereas the Trading Service maps the name of a remote object onto its remote object reference, together with an arbitrary number of attributes describing the object in terms understandable by human users.

Name Services and the Domain Name System

- A name service stores a collection of one or more naming contexts, sets of bindings between textual names and attributes for objects such as computers, services, and users.
- The major operation that a name service supports is to resolve names.

Uniform Resource Identifiers

Uniform Resource Identifiers (URIs) came about from the need to identify resources on the Web, and other Internet resources such as electronic mailboxes. An important goal was to identify resources in a coherent way, so that they could all be processed by common software such as browsers. URIs are 'uniform' in that their syntax incorporates that of indefinitely many individual types of resource identifiers (that is, URI schemes), and there are procedures for managing the global namespace of schemes. The advantage of uniformity is that it eases the process of introducing new types of identifier, as well as using existing types of identifier in new contexts, without disrupting existing usage.

Uniform Resource Locators: Some URIs contain information that can be used to locate and access a resource; others are pure resource names. The familiar term Uniform Resource Locator (URL) is often used for URIs that provide location information and specify the method for accessing the resource.

Uniform Resource Names: Uniform Resource Names (URNs) are URIs that are used as pure resource names rather than locators. For example, the URI:

mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com

Navigation

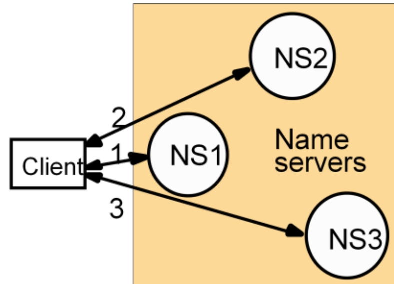
Navigation is the act of chaining multiple Naming Services in order to resolve a single name to the corresponding resource.

- Namespaces allows for structure in names.
- URLs provide a default structure that decompose the location of a resource in
  - protocol used for retrieval
  - internet end point of the service exposing the resource
  - service specific path
- This decomposition facilitates the resolution of the name into the corresponding resource
- Moreover, structured namespaces allows for iterative navigation…

Iterative navigation

Reason for NFS iterative name resolution

This is because the file service may encounter a symbolic link (i.e. an *alias*) when resolving a name. A symbolic link must be interpreted in the client's file system name space because it may point to a file in a directory stored at another server. The client computer must determine which server this is, because only the client knows its mount points



A client iteratively contacts name servers NS1–NS3 in order to resolve a name

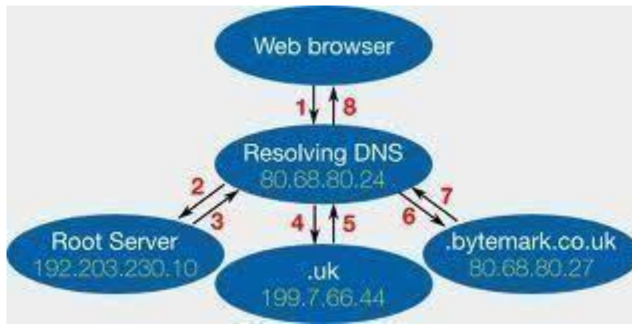
Server controlled navigation

- In an alternative model, name server coordinates naming resolution and returns the results to the client. It can be:
  - Recursive:
    - *it is performed by the naming server*
    - *the server becomes like a client for the next server*
    - *this is necessary in case of client connectivity constraints*
  - Non recursive:
    - *it is performed by the client or the first server*
    - *the server bounces back the next hop to its client*

Non-recursive and recursive server-controlled navigation

DNS offers recursive navigation as an option, but iterative is the standard technique. Recursive navigation must be used in domains that limit client access to their DNS information for security reasons.

A name server NS1 communicates with other name servers on behalf of a client



7. **Lecture Notes: (To be attached)**

8. **Textbook :**

   1. George Coulouris, Jean Dollimore, Tim Kindberg, , "Distributed Systems: Concepts and Design", 4th Edition, Pearson Education, 2005. **PP. 350-356.**

9. **Application**

The Domain Name System is a name service design whose main naming database is used across the Internet.

This original scheme was soon seen to suffer from three major shortcomings:
• It did not scale to large numbers of computers.
• Local organizations wished to administer their own naming systems.
• A general name service was needed – not one that serves only for looking up computer addresses.

Domain names • The DNS is designed for use in multiple implementations, each of which may have its own name space. In practice, however, only one is in widespread use, and that is the one used for naming across the Internet. The Internet DNS name space is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called *generic domains*) in use across the Internet were:

com – Commercial organizations
edu – Universities and other educational institutions
gov – US governmental agencies
mil – US military organizations

net – Major network support centres
org – Organizations not mentioned above
int – International organizations

New top-level domains such as *biz* and *mobi* have been added since the early 2000s. A full list of current generic domain names is available from the Internet Assigned Numbers Authority [www.iana.org I]. In addition, every country has its own domains:

us – United States
uk – United Kingdom
fr – France
... – ...

DNS - The Internet Domain Name System

- A distributed naming database (specified in RFC 1034/1305)
- Name structure reflects administrative structure of the Internet
- Rapidly resolves domain names to IP addresses
  - exploits caching heavily
  - typical query time ~100 milliseconds
- Scales to millions of computers
  - partitioned database
  - caching
- Resilient to failure of a server
  - Replication

Basic DNS algorithm for name resolution (domain name -> IP number)

- Look for the name in the local cache
- Try a superior DNS server, which responds with:

– another recommended DNS server

– the IP address (which may not be entirely up to date)

DNS name servers: Hierarchical organisation

Note: Name server names are in italics, and the corresponding domains are in parentheses.
Arrows denote name server entries

DNS in typical operation

DNS server functions and configuration

- Main function is to resolve domain names for computers, i.e. to get their IP addresses
  - caches the results of previous searches until they pass their 'time to live'
- Other functions:
  - get *mail host* for a domain
  - reverse resolution - get domain name from IP address
  - Host information - type of hardware and OS
  - Well-known services - a list of well-known services offered by a host
  - Other attributes can be included (optional)

DNS resource records

The DNS architecture allows for recursive navigation as well as iterative navigation. The resolver specifies which type of navigation is required when contacting a name server. However,

name servers are not bound to implement recursive navigation. As was pointed out above, recursive navigation may tie up server threads, meaning that other requests might be delayed.

Record type	*Meaning*	*Main contents*
A	A computer address	IP number
NS	An authoritative name server	Domain name for server
CNAME	The canonical name for an alias	Domain name for alias
SOA	Marks the start of data for a zone	Parameters governing the zone
WKS	A well-known service description	List of service names and protocols
PTR	Domain name pointer (reverse lookups)	Domain name
HINFO	Host information	Machine architecture and operating system
MX	Mail exchange	List of <*preference, host*> pairs
TXT	Text string	Arbitrary text

The data for a zone starts with an *SOA*-type record, which contains the zone parameters that specify, for example, the version number and how often secondaries should refresh their copies. This is followed by a list of records of type *NS* specifying the name servers for the domain and a list of records of type *MX* giving the domain names of mail hosts, each prefixed by a number expressing its preference. For example, part of the database for the domain *dcs.qmul.ac.uk* at one point is shown in the following figure where the time to live *1D* means 1 day.

DNS zone data records

domain name	*time to live*	*class*	*type*	*value*
dcs.qmul.ac.uk	*1D*	*IN*	*NS*	*dns0*
dcs.qmul.ac.uk	*1D*	*IN*	*NS*	*dns1*
dcs.qmul.ac.uk	*1D*	*IN*	*MX*	*1 mail1.qmul.ac.uk*
dcs.qmul.ac.uk	*1D*	*IN*	*MX*	*2 mail2.qmul.ac.uk*

The majority of the remainder of the records in a lower-level zone like *dcs.qmul.ac.uk* will be of type *A* and map the domain name of a computer onto its IP address. They may contain some aliases for the well-known services, for example:

domain name	*time to live*	*class*	*type*	*value*
www	*1D*	*IN*	*CNAME*	traffic
traffic	*1D*	*IN*	*A*	138.37.95.150

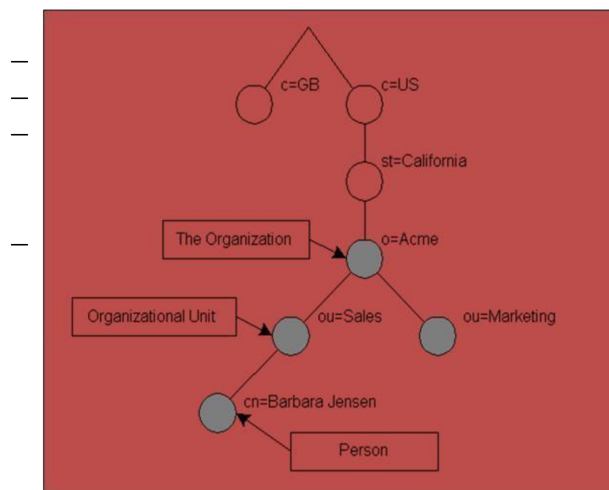
If the domain has any subdomains, there will be further records of type *NS* specifying their name servers, which will also have individual *A* entries. For example, at one point the database for *qmul.ac.uk* contained the following records for the name servers in its subdomain

dcs.qmul.ac.uk:

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

DNS issues

- Name tables change infrequently, but when they do, caching can result in the delivery of stale data.
  - Clients are responsible for detecting this and recovering
- Its design makes changes to the structure of the name space difficult. For example:
  - merging previously separate domain trees under a new root
  - moving subtrees to a different part of the structure (e.g. if Scotland became a separate country, its domains should all be moved to a new country-level domain.)
- Directory service: 'yellow pages' for the resources in a network
  Retrieves the set of names that satisfy a given description
  - e.g. X.500, LDAP, MS Active Directory Services
    - *(DNS holds some descriptive data, but:*
      - the data is very incomplete
      - DNS isn't organised to search it)
- Discovery service:- a directory service that also:
  is automatically updated as the network configuration changes
  meets the needs of clients in spontaneous networks (Section 2.2.3)
  discovers services required by a client (who may be mobile) within the current *scope*, for example, to find the most suitable printing service for image files after arriving at a hotel.
  Examples of discovery services: Jini discovery service, the 'service location protocol', the 'simple service discovery protocol' (part of UPnP), the 'secure discovery service'.



The name services store collections of <*name, attribute*> pairs, and how the attributes are looked up from a name. It is natural to consider the dual of this arrangement, in which *attributes* are used as values to be looked up. In these services, textual names can be considered to be just another attribute. Sometimes users wish to find a particular person or resource, but they do not know its name, only some of its other attributes.

For example, a user may ask: 'What is the name of the user with telephone number 020-555 9980?' Likewise, sometimes users require a service, but they are not concerned with what system entity supplies that service, as long as the service is conveniently accessible.

For example, a user might ask, 'Which computers in this building are Macintoshes running the Mac OS X operating system?' or 'Where can I print a high-resolution colour image?'

A service that stores collections of bindings between names and attributes and that looks up entries that match attribute-based specifications is called a *directory service*.

Examples are Microsoft's Active Directory Services, X.500 and its cousin LDAP, Univers and Profile.

Directory services are sometimes called *yellow pages services*, and conventional name services are correspondingly called *white pages services*, in an analogy with the traditional types of telephone directory. Directory services are also sometimes known as *attribute-based name services*.

A directory service returns the sets of attributes of any objects found to match some specified attributes. So, for example, the request 'TelephoneNumber = 020 5559980' might return {'Name = John Smith', 'TelephoneNumber = 020 555 9980', 'emailAddress = john@dcs.gormenghast.ac.uk', ...}.

The client may specify that only a subset of the attributes is of interest – for example, just the email addresses of matching objects. X.500 and some other directory services also allow objects to be looked up by conventional hierarchic textual names. The Universal Directory and Discovery Service (UDDI), which was presented in Section 9.4, provides both white pages and yellow pages services to provide information about organizations and the web services they offer.

UDDI aside, the term *discovery service* normally denotes the special case of a directory service for services provided by devices in a spontaneous networking environment. As Section 1.3.2 described, devices in spontaneous networks are liable to connect and disconnect unpredictably. One core difference between a discovery service and other directory services is that the address of a directory service is normally well known and preconfigured in clients, whereas a device entering a spontaneous networking environment has to resort to multicast navigation, at least the first time it accesses the local discovery service.

Attributes are clearly more powerful than names as designators of objects: programs can be written to select objects according to precise attribute specifications where names might not be known. Another advantage of attributes is that they do not expose the structure of organizations

to the outside world, as do organizationally partitioned names. However, the relative simplicity of use of textual names makes them unlikely to be replaced by attribute-based naming in many applications.

Discovery service

- A database of services with lookup based on service description or type, location and other criteria, E.g.
  1. Find a printing service in this hotel compatible with a Nikon camera
  2. Send the video from my camera to the digital TV in my room.
- Automatic registration of new services
- Automatic connection of guest's clients to the discovery service

Global Name Service (GNS)

- Designed and implemented by Lampson and colleagues at the DEC Systems Research Center (1986)
- Provide facilities for resource location, email addressing and authentication
- When the naming database grows from small to large scale, the structure of name space may change
  - the service should accommodate it
- Cache consistency ?

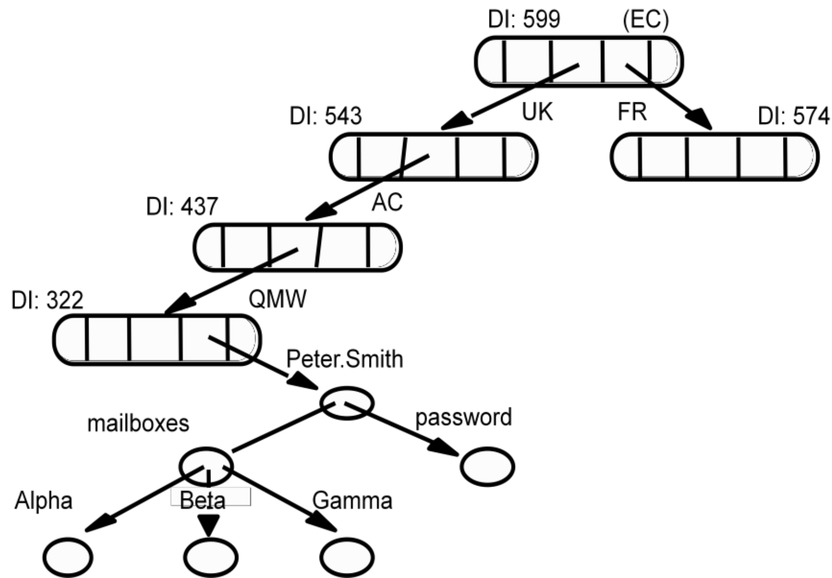
The GNS manages a naming database that is composed of a tree of directories holding names and values. Directories are named by multi-part pathnames referred to a root, or relative to a working directory, much like file names in a UNIX file system. Each directory is also assigned an integer, which serves as a unique *directory identifier* (DI). A directory contains a list of names and references. The values stored at the leaves of the directory tree are organized into *value trees*, so that the attributes associated with names can be structured values.

Names in the GNS have two parts: <*directory name*, *value name*>. The first part identifies a directory; the second refers to a value tree, or some portion of a value tree.

GNS Structure

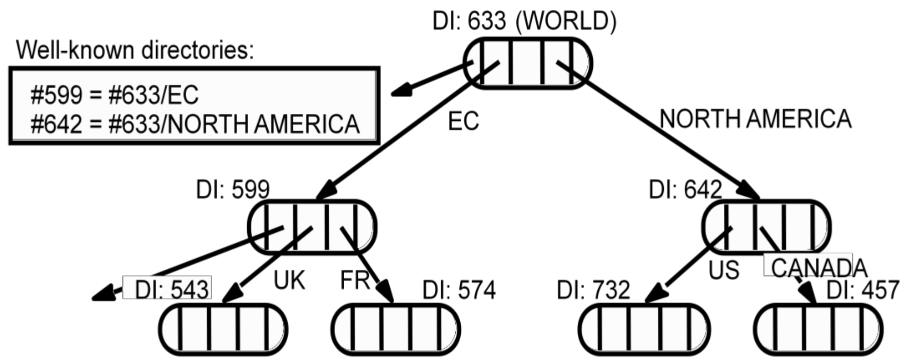
- Tree of directories holding names and values
- Muti-part pathnames refer to the root or relative working directory (like Unix file system)
- Unique Directory Identifier (DI)
- A directory contains list of names and references
- Leaves of tree contain value trees (structured values)

GNS directory tree and value tree



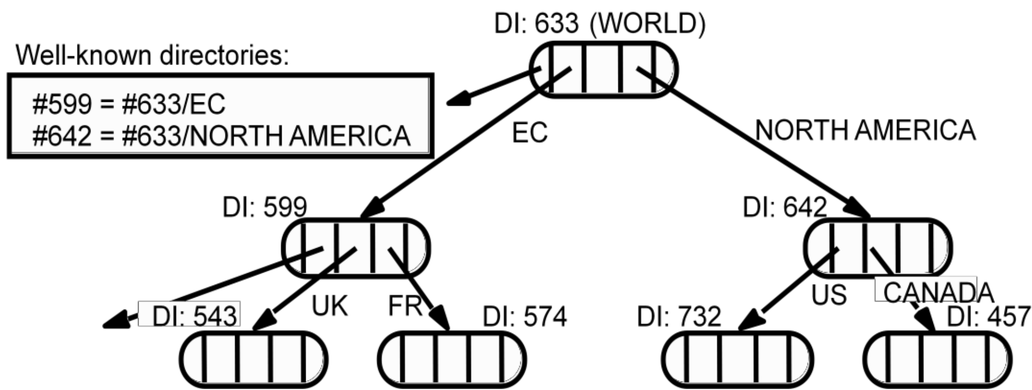
Accommodating changes

- How to integrate naming trees of two previously separate GNS services
- But what is for '/UK/AC/QMV, Peter.Smith' ?



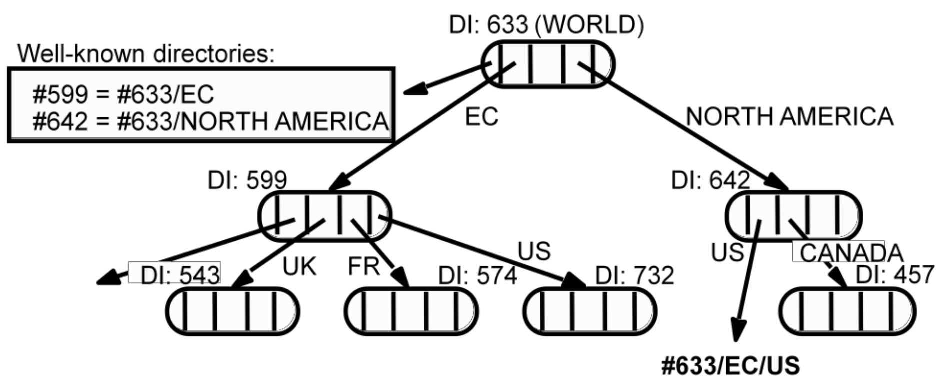
Using DI to solve changes

- Using the name '#599/UK/AC/QMV, Peter.Smith'



Restructuring of database

- Using symbolic links



X500 Directory Service

X.500 is a directory service used in the same way as a conventional name service, but it is primarily used to satisfy descriptive queries and is designed to discover the names and attributes of other users or system resources. Users may have a variety of requirements for searching and browsing in a directory of network users, organizations and system resources to obtain information about the entities that the directory contains. The uses for such a service are likely to be quite diverse. They range from enquiries that are directly analogous to the use of telephone directories, such as a simple 'white pages' access to obtain a user's electronic mail address or a 'yellow pages' query aimed, for example, at obtaining the names and telephone numbers of garages specializing in the repair of a particular make of car, to the use of the directory to access personal details such as job roles, dietary habits or even photographic images of the individuals.

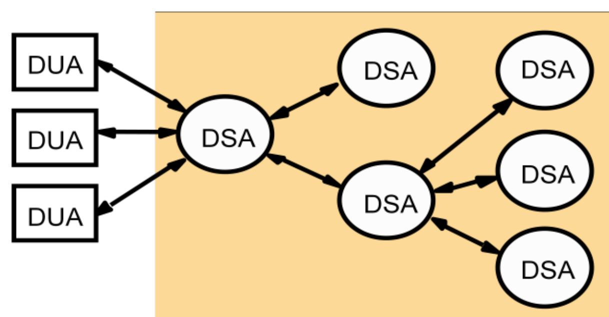
- Standard of ITU and ISO organizations
- Organized in a tree structure with name nodes as in the case of other name servers
- A wide range of attributes are stored in each node

- Directory Information Tree (DIT)
- Directory Information Base (DIB)

X.500 service architecture

The data stored in X.500 servers is organized in a tree structure with named nodes, as in the case of the other name servers discussed in this chapter, but in X.500 a wide range of attributes are stored at each node in the tree, and access is possible not just by name but also by searching for entries with any required combination of attributes. The X.500 name tree is called the *Directory Information Tree* (DIT), and the entire directory structure including the data associated with the nodes, is called the *Directory Information Base* (DIB). There is intended to be a single integrated DIB containing information provided by organizations throughout the world, with portions of the DIB located in individual X.500 servers. Typically, a medium-sized or large organization would provide at least one server. Clients access the directory by establishing a connection to a server and issuing access requests. Clients can contact any server with an enquiry. If the data required are not in the segment of the DIB held by the contacted server, it will either invoke other servers to resolve the query or redirect the client to another server.

- **Directory Server Agent (DSA)**
- **Directory User Agent (DUA)**



In the terminology of the X.500 standard, servers are *Directory Service Agents* (DSAs), and their clients are termed *Directory User Agents* (DUAs). Each entry in the DIB consists of a name and a set of attributes. As in other name servers, the full name of an entry corresponds to a path through the DIT from the root of the tree to the entry. In addition to full or *absolute* names, a DUA can establish a context, which includes a base node, and then use shorter relative names that give the path from the base node to the named entry.

An X.500 DIB Entry

info	
Alice Flintstone, Departmental Staff, Department of Computer Science, University of Gormenghast, GB	
commonName Alice.L.Flintstone Alice.Flintstone Alice Flintstone A. Flintstone	*uid* alf *mail* alf@dcs.gormenghast.ac.uk Alice.Flintstone@dcs.gormenghast.ac.uk
surname Flintstone	*roomNumber* Z42
telephoneNumber +44 986 33 4604	*userClass* Research Fellow

Part of the X.500 Directory Information Tree

The data structure for the entries in the DIB and the DIT is very flexible. A DIB entry consists of a set of attributes, where an attribute has a *type* and one or more *values*. The type of each attribute is denoted by a type name (for example, *countryName*, *organizationName*, *commonName*, *telephoneNumber*, *mailbox*, *objectClass*). New attribute types can be defined if they are required. For each distinct type name there is a corresponding type definition, which includes a type description and a syntax definition in the ASN.1 notation (a standard notation for syntax definitions) defining representations for all permissible values of the type.

DIB entries are classified in a manner similar to the object class structures found in object-oriented programming languages. Each entry includes an *objectClass* attribute, which determines the class (or classes) of the object to which an entry refers. *Organization*, *organizationalPerson* and *document* are all examples of *objectClass* values. Further classes can be defined as they are required. The definition of a class determines which attributes are mandatory and which are optional for entries of the given class. The definitions of classes are organized in an inheritance hierarchy in which all classes except one (called *topClass*) must contain an *objectClass* attribute, and the value of the *objectClass* attribute must be the names of one or more classes. If there are several *objectClass* values, the object inherits the mandatory and optional attributes of each of the classes.

X.500 Service (root)

... France (country) Great Britain (country) Greece (country)...

... BT Plc (organization) University of Gormenghast (organization)

... Computing Service (organizationalUnit)
Department of Computer Science (organizationalUnit)
Engineering Department (organizationalUnit) ...

... Departmental Staff (organizationalUnit)
ely (applicationProcess)
Research Students (organizationalUnit) ...

... Alice Flintstone (person) ... Pat King (person) James Healey (person) Janet Papworth (person) ...

Administration and updating of the DIB • The DSA interface includes operations for adding, deleting and modifying entries. Access control is provided for both queries and updating operations, so access to parts of the DIT may be restricted to certain users or classes of user

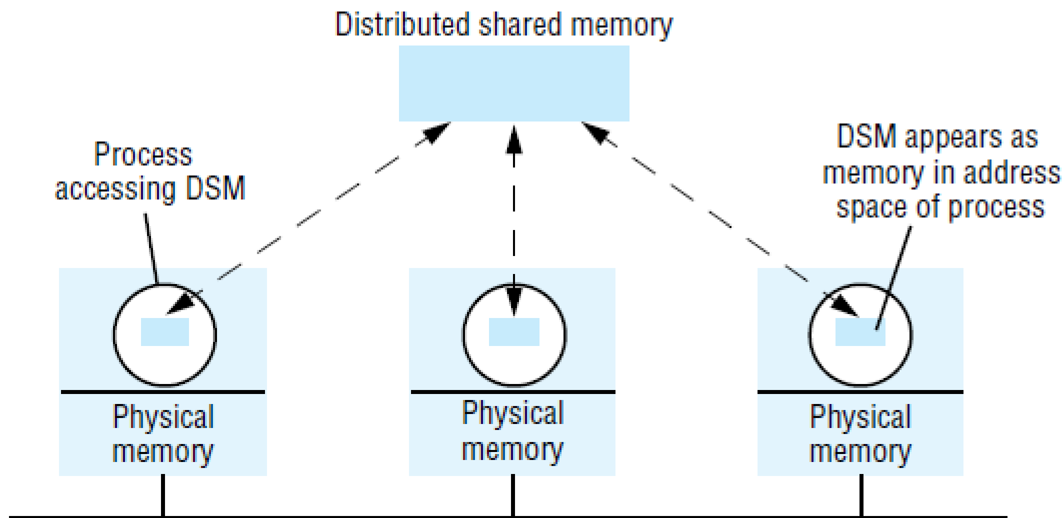
Lightweight Directory Access Protocol • X.500's assumption that organizations would provide information about themselves in public directories within a common system has proved largely unfounded. group at the University of Michigan proposed a more lightweight approach called the *Lightweight Directory Access Protocol* (LDAP), in which a DUA accesses X.500 directory services directly over TCP/IP instead of the upper layers of the ISO protocol stack.

DISTRIBUTED SHARED MEMORY

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another.

The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client-server systems, where clients normally view server-held resources as abstract data and access them by request

(for reasons of modularity and protection).



Message passing cannot be avoided altogether in a distributed system: in the absence of physically shared memory, the DSM runtime support has to send updates in messages between computers. DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access.

In distributed memory multiprocessors and clusters of off-the-shelf computing components (see Section 6.3), the processors do not share memory but are connected by a very high-speed network. These systems, like general-purpose distributed systems, can scale to much greater numbers of processors than a shared-memory multiprocessor's 64 or so. A central question that has been pursued by the DSM and multiprocessor research communities is whether the investment in knowledge of shared memory algorithms and the associated software can be directly transferred to a more scalable distributed memory architecture.

Message passing versus DSM

As a communication mechanism, DSM is comparable with message passing rather than with request-reply-based communication, since its application to parallel processing, in particular, entails the use of asynchronous communication. The DSM and message passing approaches to programming can be contrasted as follows:

Programming model:

Under the message passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory

the processes involved share variables directly, so no marshalling is necessary – even of pointers to shared variables – and thus no separate communication operations are necessary.

Efficiency :

Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message passing platforms on the same hardware – at least in the case of relatively small numbers of computers (ten or so). However, this result cannot be generalized. The performance of a program based on DSM depends upon many factors, as we shall discuss below – particularly the pattern of data sharing.

Implementation approaches to DSM

Distributed shared memory is implemented using one or a combination of specialized hardware, conventional paged virtual memory or middleware:

Hardware:

Shared-memory multiprocessor architectures based on a NUMA architecture rely on specialized hardware to provide the processors with a consistent view of shared memory. They handle memory LOAD and STORE instructions by communicating with remote memory and cache modules as necessary to store and retrieve data.

Paged virtual memory:

Many systems, including Ivy and Mether , implement DSM as a region of virtual memory occupying the same address range in the address space of every participating process.

```
#include "world.h"
struct shared { int a, b; };
Program Writer:
main()
{
struct shared *p;
methersetup(); /* Initialize the Mether runtime */
p = (struct shared *)METHERBASE;
/* overlay structure on METHER segment */
p->a = p->b = 0; /* initialize fields to zero */
while(TRUE){ /* continuously update structure fields */
p –>a = p –>a + 1;
p –>b = p –>b - 1;
}
```

```
}
```

Program Reader:

```
main()
{
struct shared *p;
methersetup();
p = (struct shared *)METHERBASE;
while(TRUE){ /* read the fields once every second */
printf("a = %d, b = %d\n", p ->a, p ->b);
sleep(1);
}
}
```

Middleware:

Some languages such as Orca, support forms of DSM without any hardware or paging support, in a platform-neutral way. In this type of implementation, sharing is implemented by communication between instances of the user-level support layer in clients and servers. Processes make calls to this layer when they access data items in DSM. The instances of this layer at the different computers access local data items and communicate as necessary to maintain consistency.

Design and implementation issues

The synchronization model used to access DSM consistently at the application level; the DSM consistency model, which governs the consistency of data values accessed from different computers; the update options for communicating written values between computers; the granularity of sharing in a DSM implementation; and the problem of thrashing.

Structure

A DSM system is just such a replication system. Each application process is presented with some abstraction of a collection of objects, but in this case the 'collection' looks more or less like memory. That is, the objects can be addressed in some fashion or other. Different approaches to DSM vary in what they consider to be an 'object' and in how objects are addressed. We consider three approaches, which view DSM as being composed respectively of contiguous bytes, language-level objects or immutable data items.

Byte-oriented

This type of DSM is accessed as ordinary virtual memory – a contiguous array of bytes. It is the

$R(x)a$ – a *read* operation that reads the value a from location x.

$W(x)b$ – a *write* operation that stores value b at location x.

view illustrated above by the Mether system. It is also the view of many other DSM systems, including Ivy.It allows applications (and language implementations) to impose whatever data structures they want on the shared memory. The shared objects are directly addressible memory locations (in practice, the shared locations may be multi-byte words rather than individual bytes). The only operations upon those objects are *read* (or LOAD) and *write* (or STORE). If x and y are two memory locations, then we denote instances of these operations as follows:

Object-oriented

The shared memory is structured as a collection of language-level objects with higher-level semantics than simple *read* / *write* variables, such as stacks and dictionaries. The contents of the shared memory are changed only by invocations upon these objects and never by direct access to their member variables. An advantage of viewing memory in this way is that object semantics can be utilized when enforcing consistency.

Immutable data

When reading or taking a tuple from tuple space, a process provides a tuple specification and the tuple space returns any tuple that matches that specification – this is a type of associative addressing. To enable processes to synchronize their activities, the *read* and *take* operations both block until there is a matching tuple in the tuple space.

Synchronization model

Many applications apply constraints concerning the values stored in shared memory. This is as true of applications based on DSM as it is of applications written for sharedmemory multiprocessors (or indeed for any concurrent programs that share data, such as operating system kernels and multi-threaded servers). For example, if a and b are two variables stored in DSM, then a constraint might be that $a=b$ always. If two or moreprocesses execute the following code:

$$a := a + 1;$$

$$b := b + 1;$$

then an inconsistency may arise. Suppose a and b are initially zero and that process 1gets as far as setting a to 1. Before it can increment b, process 2 sets a to 2 and b to 1.

Consistency model

The local replica manager is implemented by a combination of middleware (the DSM runtime layer in each process) and the kernel. It is usual for middleware to perform the majority of DSM processing. Even in a page-based DSM implementation, the kernel usually provides only basic page mapping, page-fault handling and communication mechanisms and middleware is

responsible for implementing the page-sharing policies. If DSM segments are persistent, then one or more storage servers (for example, file servers) will also act as replica managers.

Two processes accessing shared variables

Process 1

```
br := b;
ar := a;
if(ar ≥ br) then
    print ("OK");
```

Process 2

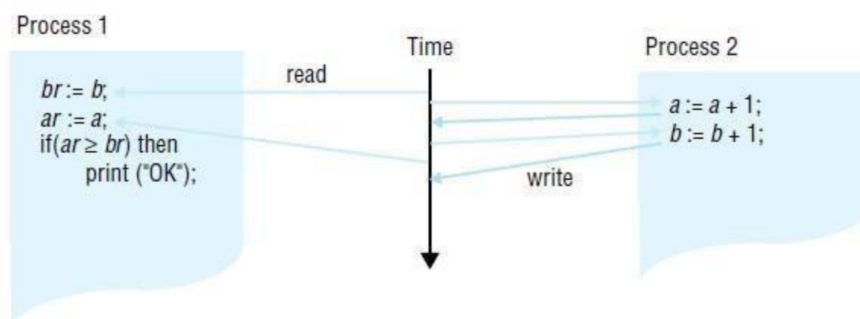
```
a := a + 1;
b := b + 1;
```

Sequential consistency

A DSM system is said to be sequentially consistent if *for any execution* there is some interleaving of the series of operations issued by all the processes that satisfies the following two criteria:

SC1: The interleaved sequence of operations is such that if R(x) a occurs in the sequence, then either the last write operation that occurs before it in the interleaved sequence is W(x) a, or no write operation occurs before it and *a* is the initial value of *x*.

SC2: The order of operations in the interleaving is consistent with the program order in which each individual client executed them.



Coherence

Coherence is an example of a weaker form of consistency. Under coherence, every process agrees on the order of write operations to the same location, but they do not necessarily agree on the ordering of write operations to different locations. We can think of coherence as sequential consistency on a locationby- location basis. Coherent DSM can be implemented by taking a protocol for implementing sequential consistency and applying it separately to each unit of replicated data – for example, each page.

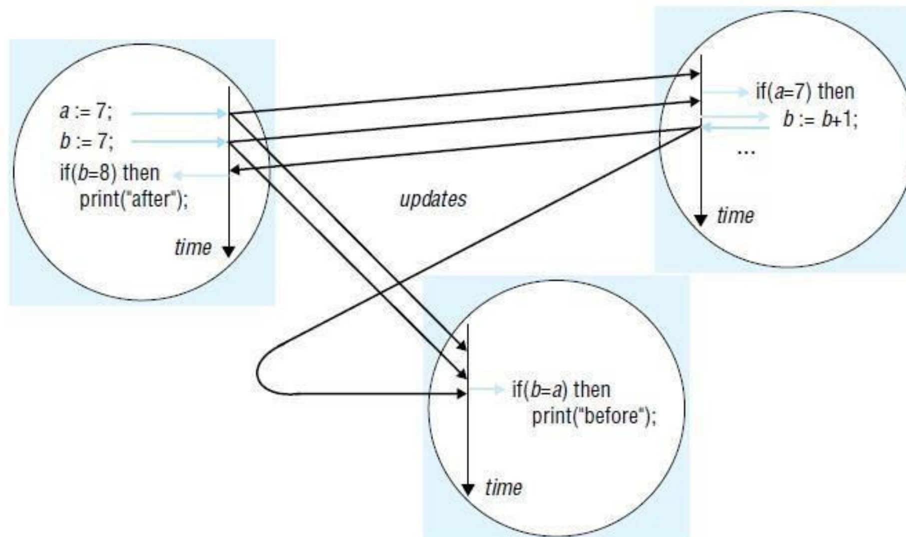
Weak consistency

This model exploits knowledge of synchronization operations in order to relax memory consistency, while appearing to the programmer to implement sequential consistency (at least, under certain conditions that are beyond the scope of this book). For example, if the programmer uses a lock to implement a critical section, then a DSM system can assume that no other process may access the data items accessed under mutual exclusion within it. It is therefore redundant for the DSM system to propagate updates to these items until the process leaves the critical section. While items are left with ‘inconsistent’ values some of the time, they are not accessed at those points; the execution appears to be sequentially consistent.

Update options

Two main implementation choices have been devised for propagating updates made by one process to the others: write-update and write-invalidate. These are applicable to a variety of DSM consistency models, including sequential consistency. In outline, the options are as follows:

Write-update: The updates made by a process are made locally and multicast to all other replica managers possessing a copy of the data item, which immediately modify the data read by local processes. Processes read the local copies of data items, without the need for communication. In addition to allowing multiple readers, several processes may write the same data item at the same time; this is known as multiple-reader/multiple-writer sharing.



Write-invalidate: This is commonly implemented in the form of multiple-reader/ single-writer sharing. At any time, a data item may either be accessed in read-only mode by one or more processes, or it may be read and written by a single process. An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is first sent to all other copies to invalidate them and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data (that is, data that are not up to date). Any processes attempting to access the data item are blocked if a writer exists.

Granularity

An issue that is related to the structure of DSM is the granularity of sharing. Conceptually, all processes share the entire contents of a DSM. As programs sharing DSM execute, however, only certain parts of the data are actually shared and then only for certain times during the execution. It would clearly be very wasteful for the DSM implementation always to transmit the entire contents of DSM as processes access and update it.

Thrashing

A potential problem with write-invalidate protocols is thrashing. Thrashing is said to occur where the DSM runtime spends an inordinate amount of time invalidating and transferring shared data compared with the time spent by application processes doing useful work. It occurs when several processes compete for the same data item, or for falsely shared data items.

RESOURCE MANAGEMENT

Resource Management is the efficient and effective development of an organization's resources when they are needed. Such resources may include financial resources, inventory, human skills,

production resources, or information technology (IT).

In the realm of project management, processes, techniques and philosophies as to the best approach for allocating resources have been developed. These include discussions on functional vs. cross-functional resource allocation as well as processes espoused by organizations like the Project Management Institute (PMI) through their Project Management Body of Knowledge (PMBOK) methodology of project management. Resource management is a key element to activity resource estimating and project human resource management. Both are essential components of a comprehensive project management plan to execute and monitor a project successfully As is the case with the larger discipline of project management, there are resource management softwaretools available that automate and assist the process of resource allocation to projects and portfolio resource transparency including supply and demand of resources. The goal of these tools typically is to ensure that: (i) there are employees within our organization with required specific skill set and desired profile required for a project, (ii) decide the number and skill sets of new employees to hire, and (iii) allocate the workforce to various projects.[3]

Corporate Resource Management Process

Large organizations usually have a defined corporate resource management process which mainly guarantees that resources are never over-allocated across multiple projects Peter Drucker wrote of the need to focus resources, abandoning a less promising initiatives for every new project taken on, as fragmentation inhibits results

Techniques

One resource management technique is resource leveling. It aims at smoothing the stock of resources on hand, reducing both excess inventories and shortages.

The required data are: the demands for various resources, forecast by time period into the future as far as is reasonable, as well as the resources' configurations required in those demands, and the supply of the resources, again forecast by time period into the future as far as is reasonable.

The goal is to achieve 100% utilization but that is very unlikely, when weighted by important metrics and subject to constraints, for example: meeting a minimum service level, but otherwise minimizing cost. A Project Resource Allocation Matrix (PRAM) is maintained to visualize the resource allocations against various projects.

The principle is to invest in resources as stored capabilities, then unleash the capabilities as demanded.

A dimension of resource development is included in resource management by which investment in resources can be retained by a smaller additional investment to develop a new capability that is demanded, at a lower investment than disposing of the current resource and replacing it with another that has the demanded capability.

In conservation, resource management is a set of practices pertaining to maintaining natural systems

integrity. Examples of this form of management are air resource management, soil conservation, forestry, wildlife management and water resourcemanagement. The broad term for this type of resource management is natural resource management (NRM).

Load balancing (computing)

load balancing distributes workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units or disk drives. Load balancing aims to optimize resource use, maximizethroughput, minimize response time, and avoid overload of any single resource. Using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy. Load balancing usually involves dedicated software or hardware, such as a multilayer switch or a Domain Name System server process.

Load balancing differs from channel bonding in that load balancing divides traffic between network interfaces on a network socket (OSI model layer 4) basis, while channel bonding implies a division of traffic between physical interfaces at a lower level, either per packet (OSI model Layer 3) or on a data link (OSI model Layer 2) basis with a protocol like shortest path bridging.

One of the most commonly used applications of load balancing is to provide a single Internet service from multiple servers, sometimes known as a server farm. Commonly load-balanced systems include popular web sites, large Internet Relay Chatnetworks, high-bandwidth File Transfer Protocol sites, Network News Transfer Protocol (NNTP) servers, Domain Name System (DNS) servers, and databases.

Round-robin DNS

An alternate method of load balancing, which does not necessarily require a dedicated software or hardware node, is called*round robin DNS*. In this technique, multiple IP addresses are associated with a single domain name; clients are expected to choose which server to connect to. Unlike the use of a dedicated load balancer, this technique exposes to clients the existence of multiple backend servers. The technique has other advantages and disadvantages, depending on the degree of control over the DNS server and the granularity of load balancing desired.

Another more effective technique for load-balancing using DNS is to delegate www.example.org as a sub-domain whose zone is served by each of the same servers that are serving the web site. This technique works particularly well where individual servers are spread geographically on the Internet. For example,

```
one.example.org A 192.0.2.1
two.example.org A 203.0.113.2
www.example.org NS one.example.org
www.example.org NS two.example.org
```

However, the zone file for www.example.org on each server is different such that each server resolves its own IP Address as the A-record.[1] On server *one* the zone file for www.example.org reports:

```
@ in a 192.0.2.1
```

On server *two* the same zone file contains:

```
@ in a 203.0.113.2
```

This way, when a server is down, its DNS will not respond and the web service does not receive any traffic. If the line to one server is congested, the unreliability of DNS ensures less HTTP traffic reaches that server. Furthermore, the quickest DNS response to the resolver is nearly always the one from the network's closest server, ensuring geo-sensitive load-balancing. A short TTL on the A-record helps to ensure traffic is quickly diverted when a server goes down. Consideration must be given the possibility that this technique may cause individual clients to switch between individual servers in mid-session.

Client-Side Random Load Balancing]

One more approach to load balancing is to deliver list of server IPs to the client, and then to have client randomly select the IP from the list on each connection. This essentially relies on all clients causing similar load, and the Law of Large Numbersto achieve reasonably flat load distribution across servers. It has been claimed that client-side random load balancing tends to provide better load distribution then round-robin DNS; this has been attributed to caching issues with round-robin DNS, which in case of large DNS caching servers, tend to skew the distribution for round-robin DNS, while client-side random selection remains unaffected regardless of DNS caching.

With this approach, the method of delivery of list of IPs to the client can vary, and may be implemented as a DNS list (delivered to all the clients without any round-robin), or via hardcoding it to the list. If "smart client" is used, detecting that randomly selected server is down, and connecting randomly again, it also provides fault tolerance.

Server-side Load Balancers

For Internet services, server-side load balancer is usually a software program that is listening on the port where external clients connect to access services. The load balancer forwards requests to one of the "backend" servers, which usually replies to the load balancer. This allows the load balancer to reply to the client without the client ever knowing about the internal separation of functions. It also prevents clients from contacting back-end servers directly, which may have security benefits by hiding the structure of the internal network and preventing attacks on the kernel's network stack or unrelated services running on other ports.

Some load balancers provide a mechanism for doing something special in the event that all backend servers are unavailable. This might include forwarding to a backup load balancer, or displaying a message regarding the outage.

It is also important that the load balancer itself does not become a single point of failure. Usually load balancers are implemented in high-availability pairs which may also replicate session persistence data if required by the specific application.

Scheduling algorithms

Numerous scheduling algorithms are used by load balancers to determine which back-end server to send a request to. Simple algorithms include random choice or round robin. More sophisticated load balancers may take additional factors into account, such as a server's reported load, least response

times, up/down status (determined by a monitoring poll of some kind), number of active connections, geographic location, capabilities, or how much traffic it has recently been assigned.

Persistence

An important issue when operating a load-balanced service is how to handle information that must be kept across the multiple requests in a user's session. If this information is stored locally on one backend server, then subsequent requests going to different backend servers would not be able to find it. This might be cached information that can be recomputed, in which case load-balancing a request to a different backend server just introduces a performance issue.

Ideally the cluster of servers behind the load balancer should be session-aware, so that if a client connects to any backend server at any time the user experience is unaffected. This is usually achieved with a shared database or an in-memory session database, for example Memcached.

One basic solution to the session data issue is to send all requests in a user session consistently to the same backend server. This is known as *persistence* or *stickiness*. A significant downside to this technique is its lack of automatic failover: if a backend server goes down, its per-session information becomes inaccessible, and any sessions depending on it are lost. The same problem is usually relevant to central database servers; even if web servers are "stateless" and not "sticky", the central database is (see below).

Assignment to a particular server might be based on a username, client IP address, or be random. Because of changes of the client's perceived address resulting from DHCP, network address translation, and web proxies this method may be unreliable. Random assignments must be remembered by the load balancer, which creates a burden on storage. If the load balancer is replaced or fails, this information may be lost, and assignments may need to be deleted after a timeout period or during periods of high load to avoid exceeding the space available for the assignment table. The random assignment method also requires that clients maintain some state, which can be a problem, for example when a web browser has disabled storage of cookies. Sophisticated load balancers use multiple persistence techniques to avoid some of the shortcomings of any one method.

Another solution is to keep the per-session data in a database. Generally this is bad for performance because it increases the load on the database: the database is best used to store information less transient than per-session data. To prevent a database from becoming a single point of failure, and to improve scalability, the database is often replicated across multiple machines, and load balancing is used to spread the query load across those replicas. Microsoft's ASP.net State Server technology is an example of a session database. All servers in a web farm store their session data on State Server and any server in the farm can retrieve the data.

In the very common case where the client is a web browser, a simple but efficient approach is to store the per-session data in the browser itself. One way to achieve this is to use a browser cookie, suitably time-stamped and encrypted. Another isURL rewriting. Storing session data on the client is generally the preferred solution: then the load balancer is free to pick any backend server to handle a request. However, this method of state-data handling is poorly suited to some complex business logic scenarios, where session state payload is big and recomputing it with every request on a server is not feasible. URL rewriting has major security issues, because the end-user can easily alter the submitted URL and thus change session streams.

Yet another solution to storing persistent data is to associate a name with each block of data, and use a distributed hash table to pseudo-randomly assign that name to one of the available servers, and then store that block of data in the assigned server.

Load balancer features

Hardware and software load balancers may have a variety of special features. The fundamental feature of a load balancer is to be able to distribute incoming requests over a number of backend servers in the cluster according to a scheduling algorithm. Most of the following features are vendor specific:

- *Asymmetric load:* A ratio can be manually assigned to cause some backend servers to get a greater share of the workload than others. This is sometimes used as a crude way to account for some servers having more capacity than others and may not always work as desired.
- *Priority activation:* When the number of available servers drops below a certain number, or load gets too high, standby servers can be brought online.
- *SSL Offload and Acceleration:* Depending on the workload, processing the encryption and authentication requirements of an SSL request can become a major part of the demand on the Web Server's CPU; as the demand increases, users will see slower response times, as the SSL overhead is distributed among Web servers. To remove this demand on Web servers, a balancer can terminate SSL connections, passing HTTPS requests as HTTP requests to the Web servers. If the balancer itself is not overloaded, this does not noticeably degrade the performance perceived by end users. The downside of this approach is that all of the SSL processing is concentrated on a single device (the balancer) which can become a new bottleneck. Some load balancer appliances include specialized hardware to process SSL. Instead of upgrading the load balancer, which is quite expensive dedicated hardware, it may be cheaper to forgo SSL offload and add a few Web servers. Also, some server vendors such as Oracle/Sun now incorporate cryptographic acceleration hardware into their CPUs such as the T2000. F5 Networks incorporates a dedicated SSL acceleration hardware card in their local traffic manager (LTM) which is used for encrypting and decrypting SSL traffic. One clear benefit to SSL offloading in the balancer is that it enables it to do balancing or content switching based on data in the HTTPS request.
- *Distributed Denial of Service (DDoS) attack protection:* load balancers can provide features such as SYN cookies and delayed-binding (the back-end servers don't see the client until it finishes its TCP handshake) to mitigate SYN floodattacks and generally offload work from the servers to a more efficient platform.
- *HTTP compression:* reduces amount of data to be transferred for HTTP objects by utilizing gzip compression available in all modern web browsers. The larger the response and the further away the client is, the more this feature can improve response times. The tradeoff is that this feature puts additional CPU demand on the Load Balancer and could be done by Web servers instead.
- *TCP offload:* different vendors use different terms for this, but the idea is that normally each HTTP request from each client is a different TCP connection. This feature utilizes HTTP/1.1 to consolidate multiple HTTP requests from multiple clients into a single TCP socket to the back-end servers.
- *TCP buffering:* the load balancer can buffer responses from the server and spoon-feed the data out to slow clients, allowing the web server to free a thread for other tasks faster than it would if it had to send the entire request to the client directly.
- *Direct Server Return:* an option for asymmetrical load distribution, where request and reply have different network paths.
- *Health checking:* the balancer polls servers for application layer health and removes failed servers from the pool.
- *HTTP caching:* the balancer stores static content so that some requests can be handled without contacting the servers.
- *Content filtering:* some balancers can arbitrarily modify traffic on the way through.

- *HTTP security:* some balancers can hide HTTP error pages, remove server identification headers from HTTP responses, and encrypt cookies so that end users cannot manipulate them.
- *Priority queuing:* also known as rate shaping, the ability to give different priority to different traffic.
- *Content-aware switching:* most load balancers can send requests to different servers based on the URL being requested, assuming the request is not encrypted (HTTP) or if it is encrypted (via HTTPS) that the HTTPS request is terminated (decrypted) at the load balancer.
- *Client authentication:* authenticate users against a variety of authentication sources before allowing them access to a website.
- *Programmatic traffic manipulation:* at least one balancer allows the use of a scripting language to allow custom balancing methods, arbitrary traffic manipulations, and more.
- *Firewall:* direct connections to backend servers are prevented, for network security reasons Firewall is a set of rules that decide whether the traffic may pass through an interface or not.
- *Intrusion prevention system:* offer application layer security in addition to network/transport layer offered by firewall security.

Sharing annotations

Munin implements a variety of consistency protocols, which are applied at the granularity of individual data items. The protocols are parameterized according to the following options:

- whether to use a write-update or write-invalidate protocol;
- whether several replicas of a modifiable data item may exist simultaneously;
- whether or not to delay updates or invalidations (for example, under release consistency);
- whether the item has a fixed owner, to which all updates must be sent;
- whether the same data item may be modified concurrently by several writers;
- whether the data item is shared by a fixed set of processes;
- whether the data item may be modified.

Read-only: No updates may be made after initialization and the item may be freely copied. *Migratory*: Processes typically take turns in making several accesses to the item, at least one of which is an update. For example, the item might be accessed within a critical section. Munin always gives both read and write access together to such an object, even when a process takes a read fault. This saves subsequent write-fault processing.

Write-shared: Several processes update the same data item (for example, an array) concurrently, but this annotation is a declaration from the programmer that the processes do not update the same parts of it. This means that Munin can avoid false sharing but must propagate only those words in the data item that are actually updated at each process. To do this, Munin makes a copy of a page (inside a write-fault handler) just before it is updated locally. Only the differences

between the two versions are sent in an update.

Producer-consumer: The data object is shared by a fixed set of processes, only one of which updates it. As we explained when discussing thrashing above, a writeupdate protocol is most suitable here. Moreover, updates may be delayed under the model of release consistency, assuming that the processes use locks to synchronize their accesses.

Reduction: The data item is always modified by being locked, read, updated and unlocked. An example of this is a global minimum in a parallel computation, which must be fetched and modified atomically if it is greater than the local minimum. These items are stored at a fixed owner. Updates are sent to the owner, which propagates them.

Result: Several processes update different words within the data item; a single process reads the whole item. For example, different 'worker' processes might fill in different elements of an array, which is then processed by a 'master' process. The point here is that the updates need only be propagated to the master and not to the workers (as would occur under the 'write-shared' annotation just described).

Conventional: The data item is managed under an invalidation protocol similar to that described in the previous section. No process may therefore read a stale version of the data item.

OTHER CONSISTENCY MODELS

Models of memory consistency can be divided into *uniform models*, which do not distinguish between types of memory access, and *hybrid models*, which do distinguish between ordinary and synchronization accesses (as well as other types of access).

Other uniform consistency models include:

Causal consistency: Reads and writes may be related by the happened-before relationship . This is defined to hold between memory operations when either (a) they are made by the same process; (b) a process reads a value written by another process; or (c) there exists a sequence of such operations linking the two operations. The model's constraint is that the value returned by a read must be consistent with the happened-before relationship.

Processor consistency: The memory is both coherent and adheres to the pipelined RAM model (see below). The simplest way to think of processor consistency is that the memory is coherent and that all processes agree on the ordering of any two write accesses made by the same process – that is, they agree with its program order.

Pipelined RAM: All processors agree on the order of writes issued by any given processor

In addition to release consistency, hybrid models include:

Entry consistency: Entry consistency was proposed for the Midway DSM system. In this model, every shared variable is bound to a synchronization object such as a lock, which governs access to that variable. Any process that first acquires the lock is guaranteed to read the latest value of the variable. A process wishing to write the variable must first obtain the corresponding lock in 'exclusive' mode – making it the only process able to access the variable.

Several processes may read the variable concurrently by holding the lock in nonexclusive mode. Midway avoids the tendency to false sharing in release consistency, but at the expense of increased programming complexity.

Scope consistency: This memory model [Iftode *et al.* 1996] attempts to simplify the programming model of entry consistency. In scope consistency, variables are associated with synchronization objects largely automatically instead of relying on the programmer to associate locks with variables explicitly. For example, the system can monitor which variables are updated in a critical section.

Weak consistency: Weak consistency [Dubois *et al.* 1988] does not distinguish between *acquire* and *release* synchronization accesses. One of its guarantees is that all previous ordinary accesses complete before *either* type of synchronization access completes.

Common Object Request Broker Architecture (CORBA)

CORBA is a middeware design that allows application programs to communicate with one another irrespective of their programming languages, their hardware and software platforms, the networks they communicate over and their implementors.

Applications are built from CORBA objects, which implement interfaces defined in CORBA's interface definition language, IDL. Clients access the methods in the IDL interfaces of CORBA objects by means of RMI. The middleware component that supports RMI is called the Object Request Broker or ORB.

Introduction

The OMG (Object Management Group) was formed in 1989 with a view to encouraging the adoption of distributed object systems in order to gain the benefits of object-oriented

programming for software development and to make use of distributed systems, which were becoming widespread. To achieve its aims, the OMG advocated the use of open systems based on standard object-oriented interfaces. These systems would be built from heterogeneous hardware, computer networks, operating systems and programming languages.

An important motivation was to allow distributed objects to be implemented in any programming language and to be able to communicate with one another. They therefore designed an interface language that was independent of any specific implementation language.

They introduced a metaphor, the *object request broker*(or ORB), whose role is to help a client to invoke a method on an object. This role involves locating the object, activating the object if necessary and then communicating the client's request to the object, which carries it out and replies.

In 1991, a specification for an object request broker architecture known as CORBA (Common Object Request Broker Architecture) was agreed by a group of companies. This was followed in 1996 by the CORBA 2.0 specification, which defined standards enabling implementations made by different developers to communicate with one another. These standards are called the General Inter-ORB protocol or GIOP. It is intended that GIOP can be implemented over any transport layer with connections. The implementation of GIOP for the Internet uses the TCP protocol and is called the Internet Inter-ORB Protocol or IIOP [OMG 2004a]. CORBA 3 first appeared in late 1999 and a component model has been added recently.

The main components of CORBA's language-independent RMI framework are the following:

- An interface definition language known as IDL,
- The GIOP defines an external data representation, called CDR. It also defines specific formats for the messages in a request-reply protocol. In addition to request and reply messages, it specifies messages for enquiring about the location of an object, for cancelling requests and for reporting errors.
- The IIOP, an implementation of GIOP defines a standard form for remote object references,

CORBA RMI

Programming in a multi-language RMI system such as CORBA RMI requires more of the programmer than programming in a single-language RMI system such as Java RMI.

The following new concepts need to be learned:

- the object model offered by CORBA;
- the interface definition language and its mapping onto the implementation language.

CORBA's object model

The CORBA object model is similar to the one described in , but clients are not necessarily objects – a client can be any program that sends request messages to remote objects and receives replies. The term *CORBA object* is used to refer to remote objects. Thus, a CORBA object implements an IDL interface, has a remote object reference and is able to respond to invocations of methods in its IDL interface. A CORBA object can be implemented by a language that is not objectoriented, for example without the concept of class. Since implementation languages will have different notions of class or even none at all, the class concept does not exist in CORBA. Therefore classes cannot be defined in CORBA IDL, which means that instances of classes cannot be passed as arguments.

CORBA IDL

These are preceded by definitions of two *structs*, which are used as parameter types in defining the methods. Note in particular that *GraphicalObject* is defined as a *struct* , whereas it was a class in the Java RMI example. A component whose type is a *struct* has a set of fields containing values of various types like the instance variables of an object, but it has no methods.

Parameters and results in CORBA IDL:

Each parameter is marked as being for input or output or both, using the keywords *in* , *out* or *inout* illustrates a simple example of the use of those keywords

IDL interfaces *Shape* and *ShapeList*

```
struct Rectangle{                                                          1
      long width;
      long height;
      long x;
      long y;
} ;
struct GraphicalObject {                                                   2
      string type;
      Rectangle enclosing;
      boolean isFilled;
};
interface Shape {                                                          3
      long getVersion() ;
      GraphicalObject getAllState() ;     // returns state of the GraphicalObject
};
typedef sequence <Shape, 100> All;                                         4
interface ShapeList {                                                      5
      exception FullException{ };                                          6
      Shape newShape(in GraphicalObject g) raises (FullException);         7
      All allShapes();               // returns sequence of remote object references  8
      long getVersion() ;
};
```

The semantics of parameter passing are as follows:

Passing CORBA objects:

Any parameter whose type is specified by the name of an IDL interface, such as the return value *Shape* in line 7, is a reference to a CORBA object and the value of a remote object reference is passed.

Passing CORBA primitive and constructed types:

Arguments of primitive and constructed types are copied and passed by value. On arrival, a new value is created in the recipient's process. For example, the *struct GraphicalObject* passed as argument (in line 7) produces a new copy of this *struct* at the server.

Type *Object* :

Object is the name of a type whose values are remote object references. It is effectively a common supertype of all of IDL interface types such as *Shape* and *ShapeList*.

Exceptions in CORBA IDL:

CORBA IDL allows exceptions to be defined in interfaces and thrown by their methods. To illustrate this point, we have defined our list of shapes in the server as a sequence of a fixed length (line 4) and have defined *FullException* (line 6), which is thrown by the method *newShape* (line 7) if the client attempts to add a shape when the sequence is full.

Invocation semantics:

Remote invocation in CORBA has *at-most-once* call semantics as the default. However, IDL may specify that the invocation of a particular method has *maybe* semantics by using the *oneway* keyword. The client does not block on *oneway* requests, which can be used only for methods without results.

The CORBA Naming service

It is a binder that provides operations including *rebind* for servers to register the remote object references of CORBA objects by name and *resolve* for clients to look them up by name. The names are structured in a hierarchic fashion, and each name in a path is inside a structure called a *NameComponent* . This makes access in a simple example seem rather complex.

CORBA pseudo objects

Implementations of CORBA provide interfaces to the functionality of the ORB that programmers need to use. In particular, they include interfaces to two of the components in the *ORB cor*e and the *Object Adaptor*

CORBA client and server example

This is followed by a discussion of callbacks in CORBA. We use Java as the client and server languages, but the approach is similar for other languages. The interface compiler *idlj* can be applied to the CORBA interfaces to generate the following items:

Java interfaces generated by *idlj* from CORBA interface *ShapeList*.

```
public interface ShapeListOperations {
    Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
    Shape[] allShapes();
    int getVersion();
}
```

```
public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
        org.omg.CORBA.portable.IDLEntity { }
```

.

- The equivalent Java interfaces – two per IDL interface. The name of the first Java interface ends in *Operations* – this interface just defines the operations in the IDL interface. The Java second interface has the same name as the IDL interface and implements the operations in the first interface as well as those in an interface suitable for a CORBA object.
- The server skeletons for each *idl* interface. The names of skeleton classes end in *POA* , for example *ShapeListPOA*.
- The proxy classes or client stubs, one for each IDL interface. The names of these classes end in *Stub* , for example *_ShapeListStub*
- A Java class to correspond to each of the *structs* defined with the IDL interfaces. In our example, classes *Rectangle* and *GraphicalObject* are generated. Each of these classes contains a declaration of one instance variable for each field in the corresponding *struct* and a pair of constructors, but no other methods.
- Classes called helpers and holders, one for each of the types defined in the IDL interface. A helper class contains the *narrow* method, which is used to cast down from a given object reference to the class to which it belongs, which is lower down the class hierarchy. For example, the *narrow* method in *ShapeHelper* casts down to class *Shape* . The holder classes deal with *out* and *inout* arguments, which cannot be mapped directly onto Java.

Server program

The server program should contain implementations of one or more IDL interfaces. For a server written in an object-oriented language such as Java or C++, these implementations are implemented as servant classes. CORBA objects are instances of servant classes.

When a server creates an instance of a servant class, it must register it with the POA, which makes the instance into a CORBA object and gives it a remote object reference. Unless this is done, the CORBA object will not be able to receive remote invocations. Readers who studied Chapter 5 carefully may realize that registering the object with the POA causes it to be recorded in the CORBA equivalent of the remote object table.

ShapeListServant class of the Java server program for CORBA interface *ShapeList*

```
import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
      private POA theRootpoa;
      private Shape theList[ ];
      private int version;
       private static int n=0;
      public ShapeListServant(POA rootpoa){
            theRootpoa = rootpoa;
            // initialize the other instance variables
      }
      public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException {1
            version++;
            Shape s = null;
            ShapeServant shapeRef = new ShapeServant( g, version);
            try {
                  org.omg.CORBA.Object ref = theRoopoa.servant_to_reference(shapeRef);  2
                  s = ShapeHelper.narrow(ref);
            } catch (Exception e) {}
             if(n >=100) throw new ShapeListPackage.FullException();
            theList[n++] = s;
            return s;
      }
      public  Shape[ ] allShapes(){ ... }
      public int getVersion() { ... }
}
```

Java class *ShapeListServer*

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class ShapeListServer {
    public static void main(String args[]) {
      try{
        ORB orb = ORB.init(args, null);                                             1
        POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));   2
        rootpoa.the_POAManager().activate();                                         3
        ShapeListServant SLSRef = new ShapeListServant(rootpoa);                     4
        org.omg.CORBA.Object ref = rootpoa.servant_to_reference(SLSRef);             5
        ShapeList SLRef = ShapeListHelper.narrow(ref);
        org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);                    6
        NameComponent nc = new NameComponent("ShapeList", "");                       7
        NameComponent path[] = {nc};                                                 8
        ncRef.rebind(path, SLRef);                                                   9
        orb.run();                                                                  10
      } catch (Exception e) { ... }
    }
}
```

The client program

It creates and initializes an ORB (line 1), then contacts the Naming Service to get a reference to the remote *ShapeList* object by using its *resolve* method (line 2). After that it invokes its method *allShapes* (line 3) to obtain a sequence of remote object references to all the *Shapes* currently held at the server. It then invokes the *getAllState* method (line 4), giving as argument the first remote object reference in the sequence returned; the result is supplied as an instance of the *GraphicalObject* class.

Java client program for CORBA interfaces *Shape* and *ShapeList*

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                                   1
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
            ShapeListHelper.narrow(ncRef.resolve(path));                      2
            Shape[] sList = shapeListRef.allShapes();                         3
            GraphicalObject g = sList[0].getAllState();                       4
        } catch(org.omg.CORBA.SystemException e) {...}
    }
}
```

Callbacks

Callbacks can be implemented in CORBA in a manner similar to the one described for Java RMI For example, the *WhiteboardCallback* interface may be defined as follows:

```
interface WhiteboardCallback {
oneway void callback(in int version);
};
```

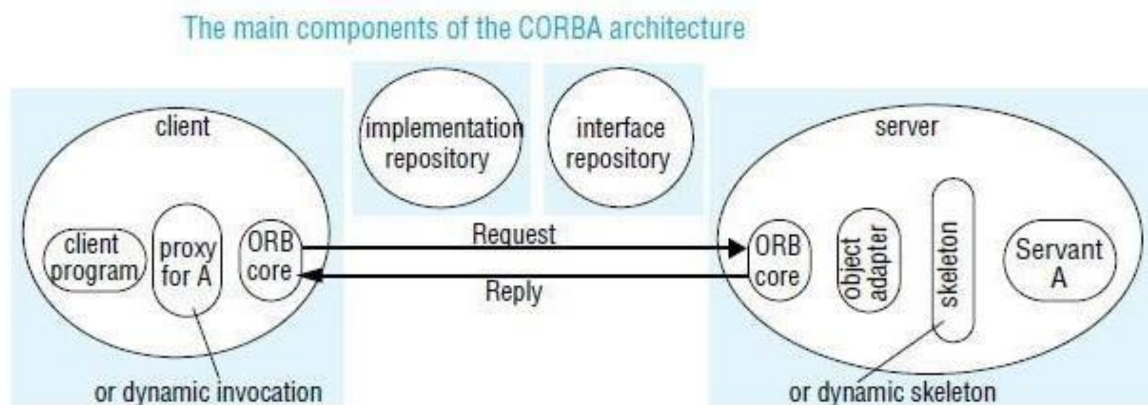
This interface is implemented as a CORBA object by the client, enabling the server to send the client a version number whenever new objects are added. But before the server can do this, the client needs to inform the server of the remote object reference of its object. To make this possible, the *ShapeList* interface requires additional methods such as *register* and *deregister*, as follows:

```
int register(in WhiteboardCallback callback);
void deregister(in int callbackId);
```

After a client has obtained a reference to the *ShapeList* object and created an instance of *WhiteboardCallback*, it uses the *register* method of *ShapeList* to inform the server that it is interested in receiving callbacks. The *ShapeList* object in the server is responsible for keeping a list of interested clients and notifying all of them each time its version number increases when a new object is added.

The architecture of CORBA

The architecture is designed to support the role of an object request broker that enables clients to invoke methods in remote objects, where both clients and servers can be implemented in a variety of programming languages. The main components of the CORBA architecture are illustrated in Figure



CORBA provides for both static and dynamic invocations. Static invocations are used when the remote interface of the CORBA object is known at compile time, enabling client stubs and server skeletons to be used. If the remote interface is not known at compile time, dynamic invocation must be used. Most programmers prefer to use static invocation because it provides a more natural programming model.

ORB core ◊ The role of the ORB core is similar to that of the communication module . In addition, an ORB core provides an interface that includes the following:

- operations enabling it to be started and stopped;
- operations to convert between remote object references and strings;
- operations to provide argument lists for requests using dynamic invocation.

Object adapter

The role of an *object adapter* is to bridge the gap between CORBA objects with IDL interfaces and the programming language interfaces of the corresponding servant classes. This role also includes that of the remote reference and dispatcher modules. An object adapter has the following tasks:

- it creates remote object references for CORBA objects;
- it dispatches each RMI via a skeleton to the appropriate servant;
- it activates and deactivates servants.

An object adapter gives each CORBA object a unique *object name*, which forms part of its remote object reference. The same name is used each time an object is activated. The object name may be specified by the application program or generated by the object adapter. Each CORBA object is registered with its object adapter, which may keep a remote object table that maps the names of CORBA objects to their servants.

Portable object adapter

The CORBA 2.2 standard for object adapters is called the Portable Object Adapter. It is called portable because it allows applications and servants to be run on ORBs produced by different developers [Vinoski 1998]. This is achieved by means of the standardization of the skeleton classes and of the interactions between the POA and the servants. The POA supports CORBA objects with two different sorts of lifetimes:

- those whose lifetimes are restricted to that of the process their servants are instantiated in;
- those whose lifetimes can span the instantiations of servants in multiple processes.

Skeletons

Skeleton classes are generated in the language of the server by an IDL compiler. As before, remote method invocations are dispatched via the appropriate skeleton to a particular servant, and the skeleton unmarshals the arguments in request messages and marshals exceptions and results in reply messages.

Client stubs/proxies

These are in the client language. The class of a proxy (for object oriented languages) or a set of stub procedures (for procedural languages) is generated from an IDL interface by an IDL compiler for the client language. As before, the client stubs/proxies marshal the arguments in invocation requests and unmarshal exceptions and results in replies.

Implementation repository

- An implementation repository is responsible for activating registered servers on demand and for locating servers that are currently running. The object adapter name is used to refer to servers when registering and activating them.
- An implementation repository stores a mapping from the names of object adapters to the pathnames of files containing object implementations.

- Object implementations and object adapter names are generally registered with the implementation repository when server programs are installed.
- When object implementations are activated in servers, the hostname and port number of the server are added to the mapping.

Interface repository

The role of the interface repository is to provide information about registered IDL interfaces to clients and servers that require it. For an interface of a given type it can supply the names of the methods and for each method, the names and types of the arguments and exceptions. Thus, the interface repository adds a facility for reflection to CORBA

Dynamic invocation interface

The dynamic invocation interface allows clients to make dynamic invocations on remote CORBA objects. It is used when it is not practical to employ proxies. The client can obtain from the interface repository the necessary information about the methods available for a given CORBA object. The client may use this information to construct an invocation with suitable arguments and send it to the server.

Dynamic skeletons

If a server uses dynamic skeletons, then it can accept invocations on the interface of a CORBA object for which it has no skeleton. When a dynamic skeleton receives an invocation, it inspects the contents of the request to discover its target object, the method to be invoked and the arguments. It then invokes the target.

Legacy code

The term *legacy code* refers to existing code that was not designed with distributed objects in mind. A piece of legacy code may be made into a CORBA object by defining an IDL interface for it and providing an implementation of an appropriate object adapter and the necessary skeletons.

CORBA Interface Definition Language

The CORBA Interface Definition Language, IDL, provides facilities for defining modules, interfaces, types, attributes and method signatures. IDL has the same lexical rules as C++ but has additional keywords to support distribution, for example *interface*, *any*, *attribute*, *in*, *out*, *inout*, *readonly*, *raises*. It also allows standard C++ preprocessing facilities.

IDL Modules

The module construct allows interfaces and other IDL type definitions to be grouped in logical units. A *module* defines a naming scope, which prevents names defined within a module clashing with names defined outside it.

IDL module *Whiteboard.*

```
module Whiteboard {
    struct Rectangle{
    ...} ;
    struct GraphicalObject {
    ...};
    interface Shape {
    ...};
    typedef sequence <Shape, 100> All;
    interface ShapeList {
    ...};
};
```

IDL interface

An IDL interface describes the methods that are available in CORBA objects that implement that interface. Clients of a CORBA object may be developed just from the knowledge of its IDL interface.

IDL methods

The general form of a method signature is:

[oneway] <return_type> <method_name> (parameter1,..., parameterL)
[raises (except1,..., exceptN)] [context (name1,..., nameM)]

where the expressions in square brackets are optional. For an example of a method signature that contains only the required parts, consider:

void getPerson(in string name, out Person p);

IDL types

IDL supports fifteen primitive types, which include short (16-bit), long (32- bit), unsigned short, unsigned long, float (32-bit), double (64-bit), char, Boolean (TRUE, FALSE), octet (8-bit), and any (which can represent any primitive or constructed type).

Attributes

IDL interfaces can have attributes as well as methods. Attributes are like public class fields in Java. Attributes may be defined as *readonly* where appropriate. The attributes are private to CORBA objects, but for each attribute declared, a pair of accessor methods is generated automatically by the IDL compiler, one to retrieve the value of the attribute and the other to set it. For *readonly* attributes, only the getter method is provided. For example, the *PersonList* interface defined in Figure 5.2 includes the following definition of an attribute: *readonly attribute string listname;*

Inheritance

IDL interfaces may be extended. For example, if interface *B* extends interface *A*, this means that it may add new types, constants, exceptions, methods and attributes to those of *A*. An extended interface can redefine types, constants and exceptions, but is not allowed to redefine methods. A value of an extended type is valid as the value of a parameter or result of the parent type. For example, the type *B* is valid as the value of a parameter or result of the type *A*.

interface A { };

interface B: A{ };

interface C {};

interface Z : B, C {};

IDL constructed types.

Type	*Examples*	*Use*
sequence	*typedef sequence <Shape, 100> All;* *typedef sequence <Shape> All* bounded and unbounded sequences of *Shapes*	Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.
string	*string name;* *typedef string<8> SmallString;* unbounded and bounded sequences of characters	Defines a sequences of characters, terminated by the null character. An upper bound on the length may be specified.
array	*typedef octet uniqueId[12];* *typedef GraphicalObject GO[10][8]*	Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.
record	*struct GraphicalObject {* *string type;* *Rectangle enclosing;* *boolean isFilled;* *};*	Defines a type for a record containing a group of related entities. *Structs* are passed by value in arguments and results.
enumerated	*enum Rand* *(Exp, Number, Name);*	The enumerated type in IDL maps a type name onto a small set of integer values.
union	*union Exp switch (Rand) {* *case Exp: string vote;* *case Number: long n;* *case Name: string s;* *};*	The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an *enum*, which specifies which member is in use.

CORBA SERVICES

CORBA includes specifications for services that may be required by distributed objects. In particular, the Naming Service is an essential addition to any ORB. The CORBA services include the following:

- *Naming Service*:
- *Event Service and Notification Service*:
- *Security service*:
- *Trading service*:

In contrast to the Naming Service which allows CORBA objects to be located by name, the Trading Service [OMG 2000a] allows them to be located by attribute – that is, it is a directory service. Its database contains a mapping from service types and their associated attributes onto remote object references of CORBA objects. The service type is a name, and each attribute is a name-value pair. Clients make queries by specifying the type of service required, together with other arguments specifying constraints on the values of attributes, and preferences for the order in which to receive matching offers. Trading servers can form federations in which they not only use their own databases but also perform queries on behalf of one anothers' clients.

- *Transaction service and concurrency control service*:

The object transaction service [OMG 2003] allows distributed CORBA objects to participate in either flat or nested transactions. The client specifies a transaction as a sequence of RMI calls, which are introduced by *begin* and terminated by *commit* or *rollback* (*abort*). The ORB attaches a transaction identifier to each remote invocation and deals with *begin*, *commit* and *rollback* requests. Clients can also suspend and resume transactions. The transaction service carries out a two-phase commit protocol. The concurrency control service [OMG 2000b] uses locks to apply concurrency control to the access of CORBA objects. It may be used from within transactions or independently.

- *Persistent state service*:

An persistent objects can be implemented by storing them in a passive form in a persistent object store while they are not in use and activating them when they are needed. Although ORBs activate CORBA objects with persistent object references, getting their implementations from the

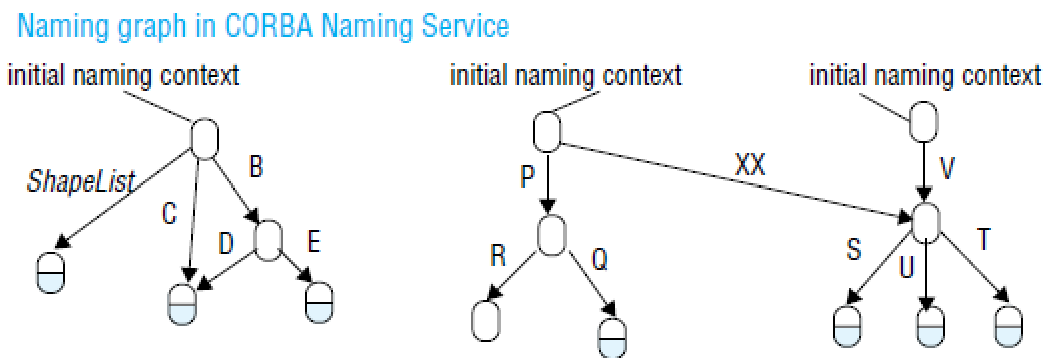
implementation repository, they are not responsible for saving and restoring the state of CORBA objects.

- *Life cycle service*

The life cycle service defines conventions for creating, deleting, copying and moving CORBA objects. It specifies how clients can use factories to create objects in particular locations, allowing persistent storage to be used if required. It defines an interface that allows clients to delete CORBA objects or to move or copy them to a specified location.

CORBA Naming Service

The CORBA Naming Service is a sophisticated example of the binder described in Chapter 5. It allows names to be bound to the remote object references of CORBA objects within naming contexts.



a *naming context* is the scope within which a set of names applies – each of the names within a context must be unique. A name can be associated with either an object reference for a CORBA object in an application or with another context in the naming service.

The names used by the CORBA Naming Service are two-part names, called Name Components, each of which consists of two strings, one for the name and the other for the kind of the object. The kind field provides a single attribute that is intended for use by applications and may contain any useful descriptive information; it is not interpreted by the Naming Service.

Although CORBA objects are given hierarchic names by the Naming Service, these names cannot be expressed as pathnames like those of UNIX files.

Part of the CORBA Naming Service *NamingContext* interface in IDL

struct NameComponent { string id; string kind; };

typedef sequence <NameComponent> Name;

interface NamingContext {

void bind (in Name n, in Object obj);
binds the given name and remote object reference in my context.

void unbind (in Name n);
removes an existing binding with the given name.

void bind_new_context(in Name n);
creates a new naming context and binds it to a given name in my context.

Object resolve (in Name n);
looks up the name in my context and returns its remote object reference.

void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
returns the names in the bindings in my context.

};

CORBA Event Service

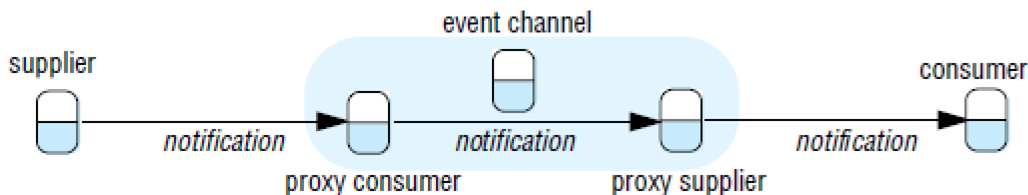
The CORBA Event Service specification defines interfaces allowing objects of interest, called *suppliers*, to communicate notifications to subscribers, called *consumers*. The notifications are communicated as arguments or results of ordinary synchronous CORBA remote method invocations. Notifications may be propagated either by being *pushed* by the supplier to the consumer or *pulled* by the consumer from the supplier. In the first case, the consumers implement the *PushConsumer* interface which includes a method *push* that takes any CORBA data type as argument. Consumers register their remote object references with the suppliers. The supplier invokes the *push* method, passing a notification as argument. In the second case, the supplier implements the *PullSupplier* interface, which includes a method *pull* that receives any CORBA data type as its return value. Suppliers register their remote object references with the consumers. The consumers invoke the *pull* method and receive a notification as result.

The notification itself is transmitted as an argument or result whose type is *any*, which means that the objects exchanging notifications must have an agreement about the contents of notifications. Application programmers, however, may define their own IDL interfaces with notifications of any desired type.

Event channels are CORBA objects that may be used to allow multiple suppliers to communicate with multiple consumers in an asynchronous manner. An event channel acts as a

buffer between suppliers and consumers. It can also multicast the notifications to the consumers. Communication via an event channel may use either the push or pull style. The two styles may be mixed; for example, suppliers may push notifications to the channel and consumers may pull notifications from it.

CORBA event channels



CORBA Notification Service

The CORBA Notification Service extends the CORBA Event Service, retaining all of its features including event channels, event consumers and event suppliers. The event service provides no support for filtering events or for specifying delivery requirements. Without the use of filters, all the consumers attached to a channel have to receive the same notifications as one another. And without the ability to specify delivery requirements, all of the notifications sent via a channel are given the delivery guarantees built into the implementation.

The notification service adds the following new facilities:

- Notifications may be defined as data structures. This is an enhancement of the limited utility provided by notifications in the event service, whose type could only be either *any* or a type specified by the application programmer.
- Event consumers may use filters that specify exactly which events they are interested in. The filters may be attached to the proxies in a channel. The proxies will forward notifications to event consumers according to constraints specified in filters in terms of the contents of each notification.
- Event suppliers are provided with a means of discovering the events the consumers are interested in. This allows them to generate only those events that are required by the consumers.
- Event consumers can discover the event types offered by the suppliers on a channel, which enables them to subscribe to new events as they become available.

- It is possible to configure the properties of a channel, a proxy or a particular event. These properties include the reliability of event delivery, the priority of events, the ordering required (for example, FIFO or by priority) and the policy for discarding stored events.
- An event type repository is an optional extra. It will provide access to the structure of events, making it convenient to define filtering constraints.

A structured event consists of an event header and an event body. The following example illustrates the contents of the header:

domain type	*event type*	*event name*	*requirements*
"home"	"burglar alarm"	"21 Mar at 2pm"	"priority", 1000

The following example illustrates the information in the body of a structured event:

← *filterable part* →			
name, value	*name, value*	*name, value*	*remainder*
"bell", "ringing"	"door", "open"	"cat", "outside"	

Filter objects are used by proxies in making decisions as to whether to forward each notification. A filter is designed as a collection of constraints, each of which is a data structure with two components:

- A list of data structures, each of which indicates an event type in terms of its domain name and event type, for example, *"home"*, *"burglar alarm"*. The list includes all of the event types to which the constraint should apply.
- A string containing a boolean expression involving the values of the event types listed above. For example:

  ("domain type" == "home" && "event type" == "burglar alarm") &&

  ("bell" != "ringing" !! "door" == "open")

CORBA Security Service

The CORBA Security Service [Blakley 1999, Baker 1997, OMG 2002b] includes the following:

- Authentication of principals (users and servers); generating credentials for principals (that is, certificates stating their rights); delegation of credentials is supported

- Access control can be applied to CORBA objects when they receive remote method invocations. Access rights may for example be specified in access control lists (ACLs).
- Security of communication between clients and objects, protecting messages for integrity and confidentiality.
- Auditing by servers of remote method invocations.
- Facilities for non-repudiation. When an object carries out a remote invocation on behalf of a principal, the server creates and stores credentials that prove that the invocation was done by that server on behalf of the requesting principal.

CORBA allows a variety of security policies to be specified according to requirements. A message-protection policy states whether client or server (or both) must be authenticated, and whether messages must be protected against disclosure and/or modification.

Access control takes into account that many applications have large numbers of users and even larger numbers of objects, each with its own set of methods. Users are supplied with a special type of credential called a *privilege* according to their roles.

Objects are grouped into *domains*. Each domain has a single access control policy specifying the access rights for users with particular privileges to objects within that domain. To allow for the unpredictable variety of methods, each method is classified in terms of one of four generic methods (*get*, *set*, *use* and *manage*). *Get* methods just return parts of the object state, *set* methods alter the object state, *use* methods cause the object to do some work, and *manage* methods perform special functions that are not intended to be available for general use. Since CORBA objects have a variety of different interfaces, the access rights must be specified for each new interface in terms of the above generic methods.

In its simplest form, security may be applied in a manner that is transparent to applications. It includes applying the required protection policy to remote method invocations, together with auditing. The security service allows users to acquire their individual credentials and privileges in return for supplying authentication data such as a password.