GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)

Seshadri Rao Knowledge Village, Gudlavalleru – 521 356.

Department of Computer Science and Engineering

HANDOUT

on

Distributed Systems

Vision

To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.

Mission

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behavior & respect for others.
- To foster industry-academia relationship for mutual benefit and growth

Program Educational Objectives

PEO1: Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.

PEO2: Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations

PEO3: Demonstrate commitment and progress in lifelong learning, professional development, leadership and Communicate effectively with professional clients and the public.

HANDOUT ON DISTRIBUTED SYSTEMS

Class & Sem.:IV B.Tech – I Semester Academic Year:2018-19

Branch : CSE Credits :3

==========

1. Brief History and Scope of the Subject

A distributed system is a collection of independent computers that appear to the users of the system as a single computer

- Two aspects:
  - Hardware: autonomous machines
  - Software : the users think of the system as a single computer.

More general definition: A distributed system consists of multiple autonomous computers communicate through that a computer network.

From 1945 until mid-1980s, computers were large and expensive.

- A mainframe costs millions
- A minicomputer costs tens of thousands

Start from mid-1980

- Microprocessors
- Computer networks, LAN, and WAN

Results: Distributed systems

2. Pre-Requisites

OS, Computer Networks Concepts

3. Course Objectives:

To familiarize with the concepts of distributed computing systems.

4. Course Outcomes:

Student will be able to

CO1: Understand the concepts of distributed systems

CO2: Implement different types of architectures in system models

CO3: Design an API by using TCP and UDP

CO4: Design issues of RMI

CO5: Implement Thread and its synchronization

CO6: Analyze the working of various algorithms used to achieve synchronization.

5. Program Outcomes:

Graduates of the Computer Science and Engineering Program will have ability to Engineering graduate will be able to

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and engineering specialization to the solution of *complex engineering problems.*

2. **Problem analysis**: Identify, formulate, research literature, and analyze engineering problems to arrive at substantiated conclusions using first principles of mathematics, natural, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components, processes to meet the specifications with consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively with the engineering community and with society at large. Be able to comprehend and write effective reports documentation. Make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of engineering and management principles and apply these to one's own work, as a member and leader in a team. Manage projects in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

6. Mapping of Course Outcomes with Program Outcomes:

	1	2	3	4	5	6	7	8	9	10	11	12
CO1	3											
CO2		1	2	3								
CO3			1	3								
CO4				1								
CO5	2											
CO6		3		1								

3- High Level Mapping **2-** Medium Level Mapping **1-**Low Level Mapping

7. Prescribed Text Books

1. Andrew S. Tanenbaum, Distributed Operating Systems.
2. George Coulouris, Jean Dollimore, Tim Kindberg, Distributed Systems Concepts and Design – 2nd edition.

8. Reference Text Books

1. Andrew S. Tanenbaum, Maarten Van Steen - Distributed Systems principles and paradigms.

9. URLs and Other E-Learning Resources

a. https://www.cs.helsinki.fi/u/jakangas/Teaching/DistSys/DistSys-08f-1.pdf

b. https://www.vidyarthiplus.com/vp/attachment.php?aid=43022

10. Digital Learning Materials:

- http://192.168.0.49/videos/videosListing/270#

11. Lecture Schedule / Lesson Plan

Topic	No. of Periods	
	Theory	Tutorial
UNIT –1: Characterization of Distributed Systems		
Introduction	1	1
Examples of Distributed Systems	2	
Resource Sharing and the Web	3	1
Challenges	2	
	8	**2**
UNIT – 2: System Models		
Introduction	1	1
Architectural models		
-Software Layers	1	
-System Architecture	1	1
-Variations	1	
-Interface and Objects	1	
-Design requirements for Distributed Architectures	2	
Fundamental models		1
-Interaction model	1	
-Failure model	1	
-Security model	1	
	10	**3**
UNIT – 3: Inter process Communication		
Introduction	1	1
The API for Internet protocols		
- Characteristics of Inter process Communication	1	1
-Sockets, UDP Datagram Communication, TCP Stream Communication	2	
External Data Representation & Marshalling	2	
client/server communication	1	**1**
group communication – IP multicast	1	
-An Implementation of group communication	1	
-Reliability and ordering of multicast	1	
	10	**3**

UNIT – 4: Distributed Objects and Remote Invocation		
Introduction	1	1
Communication between distributed objects		
- Object model, distributed object model	2	
- Design and implementation of RMI	2	
- Distributed Garbage Collection	1	**1**
Remote Procedure Call	1	
Events and notifications	2	
Case Study : JAVA RMI	1	
	10	**2**
UNIT – 5: Operating System Support		
Introduction	1	1
The Operating System layer	1	
Protection	1	
Processes and threads	1	1
- Address space	1	
- Creation of new process	1	
- Threads	2	
	8	**2**
UNIT – 6: Coordination and Agreement -Introduction		
Distributed mutual exclusion	1	1
Elections	1	
Multicast communication	1	
Transactions and replications	1	
System model and group communication	1	
Concurrency control in distributed transactions	1	**1**
Distributed deadlocks	1	
Transaction recovery	1	
Replication		
-Passive Replication	1	
-Active Replication	1	
	10	2
Total No.of Periods:	**56**	**14**

UNIT – I

Objective:

To familiarize with the concepts of distributed computing systems.

Syllabus:

Unit-I: **Characterization of Distributed Systems**

Introduction, Examples of Distributed systems, Resource Sharing and the Web Challenges.

Learning Outcomes:

At the end of the unit, students will be able to:

1. Understand the fundamental concepts of distributed systems
2. Identify the challenges in distributed systems

Learning Material

Introduction

CHARACTERIZATION OF DISTRIBUTED SYSTEMS

1.1 Introduction

Definition: a distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.

Ex: Internet, Intranet, Mobile & Ubiquitous computing

Motivation : The sharing of resources is a main motivation for constructing distributed systems.

The term 'resource' is extends from hardware components such as disks and printers to software-defined entities such as files, databases and data objects of all kinds. It includes the stream of video frames audio connection.

- Internet is a combination of many networks.

  Mobile phone networks, corporate networks, factory networks, campus networks, home networks etc.

All of these separately or in combination share the essential characteristics comes under distributed systems.

- Computers are separated by any distance. They may be on separate continents, in the same building or in the same room.

Consequences of distributed systems :

1. Concurrency: In a network of computers, concurrent program execution is the norm. i.e sharing of 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 to the network.
- Concurrently executing programs must have coordination between them.

2. No global clock: Programs coordinate their actions by exchanging messages. Close coordination depends on time.

- But there are limits to the accuracy of computers in a network to synchronize their clocks – there is no single global notion of the correct time. So, communication is done by sending messages through a network.

3. Independent failures: Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers but that doesn't mean that they stop running.

- 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.

1.2 Examples of distributed systems: Examples of Distributed systems are

1. Internet
2. Intranet
3. Mobile and Ubiquitous Computing

1. Internet:

- The Internet is a interconnected collection of different types of computer networks. Programs on the computers Communicate by passing messages.

- The design and construction of the Internet communication mechanisms (the Internet protocols) is enabling a program running anywhere to address messages to programs anywhere else.

- 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.

- The set of services is open-ended – it can be extended by the addition of server computers and new types of service.

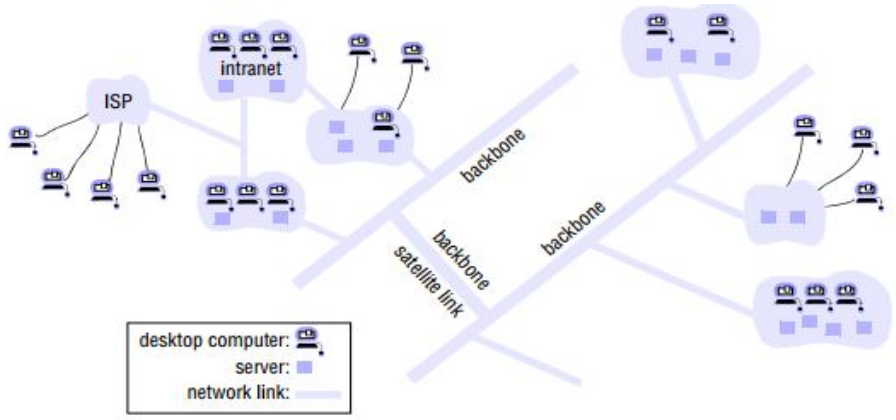


Fig:A typical portion of the internet

- Internet Service Providers (**ISP**s) are companies that provide modem links and other type 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.

- Multimedia services are available in the Internet enabling users to access audio and video data including music, radio, TV channels, phone, and video conferencing.

2. Intranet :

An intranet is a portion of the Internet that is separately administered and has a boundary that can be configured to enforce local security policies.

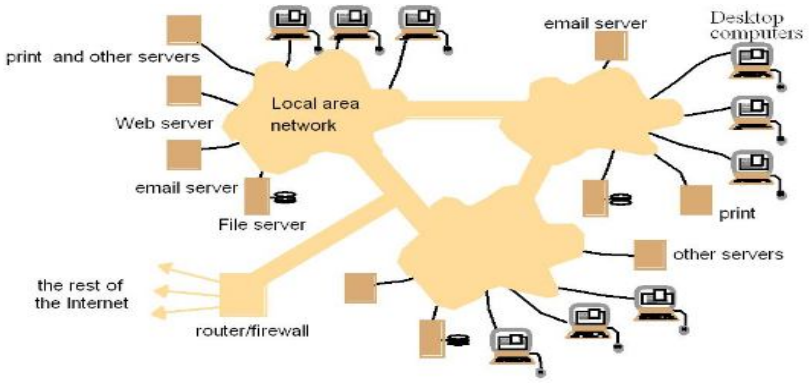


Fig: A typical intranet

- It is composed of several Local Area Networks(LANs) linked by backbone connections.
- An intranet is connected to the internet via a router, which allows the users inside the intranet to make use of services elsewhere such as the Web or email.

- It allows to access the services it provides. but, most of the organizations need to protect their own services from unauthorized use by using 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.
- Some organizations may not wish to connect their internal networks to the Internet at all.

Ex: police, military, hospitals and other security and law enforcement agencies are likely to have at least some internal networks that are isolated from the outside world

Main issues in the design of intranet are:

- File services are needed to enable users to share data.
- Firewall usage between internal and external users.
- The cost of software installation and support is an important issue.

3. Mobile and ubiquitous computing:

- Distributed systems make use of small and portable computing devices in wireless networking

  These devices include:

  • Laptop computers.

  • Handheld devices(mobile phones, smart phones, PDAs, video cameras etc.

  • Wearable devices(smart watches with functionality similar to a PDA)

  • Devices embedded in appliances such as washing machines, hi-fi systems,
  cars and refrigerators.

- **Mobile computing** is to perform 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 are still provided with access to resources via the devices they carry with them.

- The other name for mc is location-aware or context-aware computing.

- **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.

- Ubiquitous and mobile computing overlap, since the mobile user can in principle benefit from computers that are everywhere. But they are distinct,

Ubiquitous computing could benefit users while they remain in a single environment such as the home or a hospital.

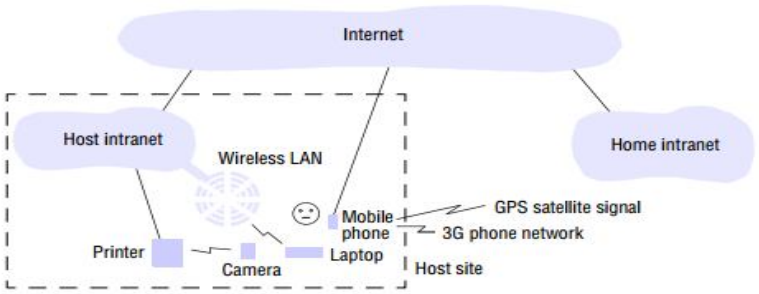


Fig : Portable and handheld devices in a distributed system

- Both intranets are connected to the rest of the Internet. The user has access to three forms of wireless connection. Their laptop has a means of connecting to the host's wireless LAN.
- This network provides coverage of a few hundred metres. It connects to the rest of the host intranet via a gateway or access point.
- The user also has a mobile (cellular) telephone, which is connected to the Internet using the Wireless Application Protocol(WAP) via gateway.
- The phone gives access to pages of simple, textual information, which can communicate over an infra-red link when pointed at a corresponding device such as a printer.

1.3 Resource sharing and the web:

Resource sharing

- We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.
- We share hardware equipments like as printers and disks to reduce costs.
- Patterns of resource sharing may vary.

  At one extreme, a search engine on the Web provides a facility to users throughout the world, users who need never come into contact with one another directly.

  At the other extreme, in computer-supported cooperative working (CSCW), a group of users who cooperate directly share resources such as documents in a small, closed group.
- The term service manages a collection of related resources and presents their functionality to users and applications.

Ex: access shared files through a file service; send documents to printers through a printing service; buy goods through an electronic payment service etc.

- The only access we have to the service is via the set of operations that it exports.

  Ex: a file service provides read, write and delete operations on files.
- Resources in a distributed system are accessed from other computers by means of communication.

- Requests are sent in messages from clients to a server and replies are sent in messages from the server to the clients.

- A complete interaction between a client and a server, from the point when the client sends its request to when it receives the server's response, is called a **remote invocation**.

- Most of the times distributed systems can be constructed in the form of interacting clients and servers. The World Wide Web, email and networked printers all fit this model.

The World Wide Web:

- The **World Wide Web** is an evolving system for publishing and accessing resources and services across the Internet.
- Through commonly available web browsers, users retrieve and view documents of many types, to listen to audio streams and view video streams, and to interact with an unlimited set of services.
- The Web began life at the European centre for nuclear research (CERN), Switzerland, in 1989 as a vehicle for exchanging documents.
- Web provides a hypertext structure among the documents that it stores.

The Web is an open system: It can be extended and implemented in new ways without disturbing its existing functionality.

First, its operation is based on **communication** standards and document or content standards that are freely published and widely implemented.

Ex: many types of browsers, implemented on several platforms.

Second, the Web is open with respect to the types of **resource** that can be published and shared on it.

Ex:a resource on the Web is a web page can be presented to the user, such as media files and PDFs(Portable Document Format).

The Web is based on three main standard technological components:

1)**HyperText Markup Language** (HTML), a language for specifying the contents and layout of pages as they are displayed by web browsers.

2) **Uniform Resource Locators** (URLs), also known as Uniform Resource Identifiers (URIs), which identify documents and other resources stored as part of the Web.

3)**client-server system architecture**, with standard rules for interaction (the HyperText Transfer Protocol – HTTP) by which browsers and other clients fetch documents and other resources from web servers.

1) **HTML** : The Hyper Text Markup Language is used to specify the text and images that make up the contents of a web page, and to specify how they are laid out and formatted for presentation to the user.

- A web page contains such structured items as headings, paragraphs, tables and images. HTML is also used to specify links and which resources are associated with them.
- A typical piece of HTML text follows:

```
<IMG SRC = "http://www.cdk5.net/WebExample/Images/earth.jpg">          1
<P>                                                                    2
Welcome to Earth! Visitors may also be interested in taking a look at the   3
<A HREF = "http://www.cdk5.net/WebExample/moon.html">Moon</A>.         4
</P>                                                                   5
```

- The file name earth.html is stored in web server. A browser retrieves the contents of this file from a web server.
- HTML directives, known as tags, are enclosed by angle brackets, such as <P>.

2) **URLs:** The purpose of a **Uniform Resource Locator** is to identify a resource in such a way as to enable the browser to locate that resource.

- Browsers examine URLs in order to fetch the corresponding resources from web servers. Sometimes the user types a URL into the browser.

Every URL have the following components:

scheme : scheme-specific-location.

scheme', declares which type of URL this is. URLs are required to identify the locations of a variety of resources

Ex: **'**mailto:joe@anISP.net' identifies a user's email address; ftp://ftp.downloadIt.com/software/aProg.exe identifies a file that is to be retrieved using the File Transfer Protocol (FTP) rather than the more commonly used protocol HTTP.

One more example of scheme is 'telnet' (used to log into a computer).

- HTTP URLs are the most widely used, for accessing resources using the standard HTTP protocol.
- An HTTP URL has two main jobs: to identify which web server maintains the resource, and to identify which of the resources at that server is required.

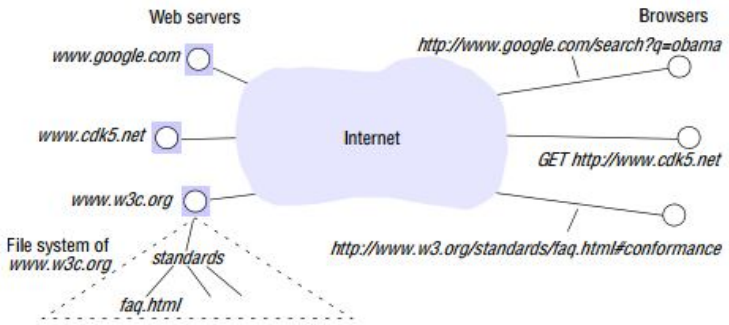


Fig:Web servers and Web browsers

Figure shows three browsers issuing requests for resources managed by three web servers. The topmost browser is issuing a query to a search engine. The middle browser requires the default page of another web site.

The bottommost browser requires a web page that is specified in full, including a path name relative to the server. The files for a given web server are maintained in one or more subtrees (directories) of the server's file system, and each resource is identified by a path name relative to the server.

HTTP URLs are of the following form:

http:// servername [:port] [/pathName] [?query] [#fragment]

- Items in square brackets are optional.
- A full HTTP URL always begins with the string 'http://' followed by a server name, expressed as a Domain Name System (DNS) name.

- It is followed optional 'port' on which the server listens for requests, which is 80 by default. Then comes an optional path name of the server's resource.
- If this is absent then the server's default web page is required. Finally, the URL optionally ends in a query component – for example, when a user submits the entries in a form such as a search engine's query page.

Publishing a resource: to publish a resource on the Web, a user first place the corresponding file in a directory that the web server can access.

3) **Client-Server system architecture (standard rules of interaction-HTTP):** The **Hyper Text Transfer Protocol** defines the ways in which browsers and other types of client interact with web servers.

a) **Request-reply interactions:** HTTP is a 'request-reply' protocol. The client sends a request message to the server containing the URL of the required resource.

- The server looks up the path name and, if it exists, sends back the resource's content in a reply message to the client. Otherwise, it sends back an error response.

b) **Content types:** Browsers are not necessarily capable of handling every type of content.

- When a browser makes a request, it includes a list of the types of content it prefers – for example, in principle it may be able to display images in 'GIF' format but not 'JPEG' format.
- The server may be able to take this into account when it returns content to the browser. The server includes the content type in the reply message so that the browser will know how to process it.

c) **One resource per request:** Clients specify one resource per HTTP request. If a web page contains nine images, say, then the browser will issue a total of ten separate requests to obtain the entire contents of the page.

- Browsers typically make several requests concurrently, to reduce the overall delay to the user.

d) **Simple access control:** By default, any user with network connectivity to a web server can access any of its published resources.

If users wish to restrict access to a resource, then they can configure the server to issue a 'challenge', the corresponding user then has to

prove that they have the right to access the resource, for example, by typing in a password.

1.4 Challenges

The main challenges of distributed systems are

1) Heterogeneity
2) Openness
3) Security
4) Scalability
5) Failure handling
6) Concurrency
7) Transparency

1) **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:

   a. Networks
   b. Computer hardware
   c. Operating systems
   d. Programming languages
   e. Implementations by different developers

   a) Networks:
   The Internet consists of many different sorts of network, their differences are masked. The Internet protocols are implemented over a variety of different networks, all of the computers attached to these networks use the Internet protocols to communicate with one another.

   Ex: 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.

   b) Computer hardware:
   Data types may be represented in different ways on different sorts of hardware

   Ex: To represent Integer data type, there are two alternatives for the byte ordering one is big-endian byte-order the other is little-endian byte-order.

These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware.

c) **Operating systems:**

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.

Ex: The calls for exchanging messages in UNIX are different from the calls in Windows.

d) **Programming languages:**

- 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.

e) **Implementations by different developers**

Programs written by different developers cannot communicate with one another unless they use common standards.

Ex: 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.

Ex: CORBA , JAVA RMI

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 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.

virtual machine: This approach provides a way of making code executable on a variety of host computers:

2) Openness

- The openness of a computer system is the characteristic that determines whether the system can be extended and re implemented in various ways.

- The openness of 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 can be achieved by the specification and documentation of the key software interfaces of the components of a systems by the software developers i.e., the key interfaces are published

  Ex: World Wide Web Consortium (W3C) develops and publishes standards related to the working of the Web [www.w3.org]. Systems that are designed to support resource sharing in this way are termed open.

- distributed systems to emphasize the fact that they are extensible.

3) Security

- The information resources 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)

- The challenge is to send sensitive information in a message over a network in a secure manner. Even though a firewall can be used to form a barrier around an intranet, not deal with ensuring the appropriate use of resources by users, or with the appropriate use of resources in the Internet, that are not protected by firewalls.

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network.

Ex:

1. A doctor might request access to hospital patient data or send additions to that data
2. In e- 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. Both of these challenges can be met by the use of encryption techniques developed for this purpose.

The two security challenges have not yet been fully met are Denial of service attacks, Security of mobile code.

4) 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.

The design of scalable distributed systems presents the following challenges

- **Controlling the cost of physical resources** : The demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it.
- **Controlling the performance loss**: The management of a set of data whose size is proportional to the number of users or resources in the system.
- **Preventing software resources running out**: Lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet), in the late 1970s use 32 bits for this purpose, supply of available Internet addresses is running out. For this reason a new version of the protocol with 128-bit Internet addresses is adopted, it require modifications to many software components.
- **Avoiding performance bottlenecks**: Algorithms should be decentralized to avoid having performance bottlenecks

- The system and application software should not need to change when the scale of the system increases, but this is difficult to achieve.

5) 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.

The following techniques for dealing with failures

- **Detecting failures:** Some failures can be detected. Ex: 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. Ex: remote crashed server in the Internet 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. 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**: It would not be practical to attempt to detect and hide all of the failures that might occur in large network with so many components. Their clients can be designed to tolerate failures.

- **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, it is possible through redundancy.

- **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.

  The design of effective techniques for keeping replicas of rapidly changing data upto-date without excessive loss of performance is a challenge.

➢ Distributed systems provide a high degree of availability in the face of hardware faults. The availability of a system is a measure of the proportion of time that it is available for use.

6) Concurrency:

➢ Both services and applications provide resources that can be shared by clients in a distributed system. There is a possibility that several clients will attempt to access a shared resource at the same time.

➢ Any object that represents a shared resource in a distributed system must be responsible for ensuring that it operates correctly in a concurrent environment.

➢ For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

7) 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.

- **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.

➢ The two most important transparencies are access and location transparency; their presence or absence most strongly affects the utilization of distributed resources. They are sometimes referred to together as network transparency.

UNIT-I

Assignment-Cum-Tutorial Questions

Section A

I.Objective Questions

1. The main motivation for the distributed system is [ ]
A. Discovery B. Sharing C. Searching D. Transparency

2. Which of the following is the example of distributed system [ ]
A. internet B. Intranet C. mobile computing D. All of the above

3. ISP stands for _____ [ ]
A. Internet Service Provider B. Internet Service Publisher
C. Intranet Service Provider D. Intranet Service Publisher

4. Which of the following is a Request-Response protocol _____ [ ]
A.SMTP B. FTP C.HTTP. D.UDP

5. CSCW stands for _____ [ ]
A. computer-supported cooperative working
B. computer-supported coordination working
C. computer-supported couple working
D. None of the above

6. Security component in information is _____ [ ]
A. Integrity B. Availability C. Confidentiality D. All of the above

7. Which software layer provides programming abstraction and masking the heterogeneity of networks [ ]
A. Middleware B. Middle core C. Mobile code D. Hard core

8. What characteristic of a computer system determines whether the system can be extended and re-implemented in various ways. [ ]
A. Openness B. Closeness C. Scalability D. Redundancy

9. Openness can be achieved by_____made available to software developers. [ ]
A. Key Design B. Key Software Interface
C. Key Platform D. None of the above

10. Which of the following approach provides a way of making code executable on any hardware. [ ]
A. Host Machine B. Guest Machine C. Virtual Machine D. All of the above

11. Which security challenges have not been fully met [ ]
A. Password-Based Attack. B. Denial-of-Service Attack.
C. Man-in-the-Middle Attack. D. Compromised-Key Attack.

12. Challenges faced in the design of scalable distributed system [ ]
A. Controlling the cost of physical resources
B. Controlling the performance loss
C. Preventing software resources running out D. All of the above
13. Failures in a distributed system are _____ [ ]
A. Complete B. Partial C. No failures D. All of the above

14. Which internal networks isolated from the outside world. [ ]
A. Police B.Hospitals C. Military
D. Law enforcement agencies E. All of the above

15. In HTTP version 1.0, Clients specify ___ resources per HTTP request

A. 2 B. 1 C. 3 D. 4 [ ]

Section B

II. Descriptive Questions

1. Define Distributed System ? What is the motivation of distributed system? Give some examples of distributed systems.
2. List the consequences of distributed systems with necessary explanation.
3. Summarize about Internet with suitable examples.
4. Explain Intranet with the help of a neat diagram and Explain the role of firewall.
5. Outline concurrency and transparency challenges faced in the development of distributed systems.
6. What is meant by Mobile and Ubiquitous Computing? Explain.
7. List the standard technological components in the web.
8. List the components of WorldWideWeb.
9. Analyze the challenges heterogeneity, openness and security with reference to Distributed system construction.
10. Examine the reasons for designing a system as a Distributed System?

UNIT – II

Objective:

To familiarize with the concepts of different descriptive models and design issues for distributed systems.

Syllabus:

Unit-II: **System Models**

Introduction, Architectural Models- Software Layers, System Architecture, Variations, Interface and Objects, Design Requirements for Distributed Architectures, Fundamental Models-Interaction Model, Failure Model, Security Model.

Learning Outcomes:

At the end of the unit, students will be able to:

1. Implement different types of architectures in System Models.
2. Know the difficulties and threats to distributed systems

Learning Material

Unit-2

SYSTEM MODELS

2.1 Introduction

The properties and design issues of distributed systems can be captured and discussed through the use of descriptive models.

Each type of model is intended to provide an abstract, simplified but consistent description of a relevant aspect of distributed system design.

There are two different models

1) **Architectural models** defines the way in which the components of systems interact with one another and the way in which they are mapped onto an underlying network of computers. i.e the location and interactions of the components.

   Ex: client-server model, peer-peer model.

2) **Fundamental models** take an abstract perspective in order to examine individual aspects of a distributed system.

There are 3 fundamental models. Each fundamental model represents a set of issues that must be addressed in the design of distributed systems.

1) **Interaction Model** - which consider the structure and sequencing of the communication between the elements of the system.

2) **Failure Model** - which consider the ways in which a system may fail to operate correctly.

3) **Security Model** - which consider how the system is protected against attempts to interfere with its correct operation or to steal its data

Their purpose is to specify the design issues, difficulties and threats resolved to fulfill tasks correctly, reliably, securely.

Difficulties and threats for distributed systems

The problems faced by the designers of distributed systems are:

1) Varying modes of use:

- The component parts of systems are subject to wide variations in workload.
  Ex: Some web pages are accessed several million times a day.

- Some parts of a system may be disconnected, or poorly connected some of the time.
  Ex: when mobile computers are included in a system.

- Some applications have special requirements for high communication bandwidth and low latency – Ex : multimedia applications.

2) Wide range of system environments:

- A distributed system must accommodate heterogeneous hardware, operating systems and networks.

- The networks may differ widely in performance – wireless networks operate at a fraction of the speed of local networks.

- Systems of widely differing scales, ranging from tens of computers to millions of computers, must be supported.

3) Internal problems:

- Non-synchronized clocks, conflicting data updates and many modes of hardware and software failure involving the individual system components.

4) External threats:

- Attacks on data integrity and secrecy, denial of service attacks.

2.2 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.
- Architecture model of a distributed system first simplifies and abstracts the functions of the individual components of as distributed system, it considers
    - The placement of components across a network of computers
    - The interrelationships between the components
- Initial simplification is achieved by classifying process as
    1. Sever process
    2. Client process
    3. Peer process
- This classification identifies the responsibilities of each and help us to assess their workloads and to determine the impact of failures.
- Later simplification is processes that cooperate and communicate in a symmetrical manner to perform a task.

2.2.1 Software layers

Layering : 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.

The term software architecture referred to the structuring of software as layers in a single computer in terms of services offered and requested between processes located in the same or different computers.

The process-and-service-oriented view can be expressed in terms of service layers. The organization of these layers is vertical in distributed systems.

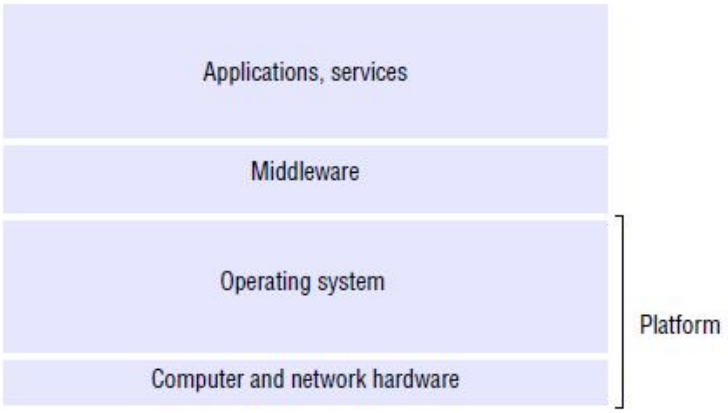


Fig : Software and hardware service layers in distributed systems

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.

Ex : 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.

Platform

The lowest-level hardware and software layers are referred to as a platform for distributed systems and applications.

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.

Ex: Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM/Symbian are major examples.

Middleware

It is software layer whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers.

Middleware is represented by processes or objects in a set of computers that interact with each other to implement communication and resource-sharing support for distributed applications.

It is concerned with providing useful building blocks for the construction of software components that can work with one another in a distributed system.

It raises the level of the communication activities of application programs through the support of abstractions are

- remote method invocation
- communication between a group of processes
- notification of events
- the partitioning, placement and retrieval of shared data objects amongst cooperating computers
- the replication of shared data objects
- the transmission of multimedia data in real time

Widely used object oriented middleware products are

- CORBA
- Java RMI
- Web services
- Microsoft's Distributed Component Object Model(DCOM)
- The ISO/ITU-T's Reference Model for Open Distributed Processing (RM-ODP)

Middle ware also provide services for use by application programs. Ex: CORBA offers services like naming, security, transactions, persistent storage and event notification.

Limitations of middleware

- Many distributed applications rely entirely on the services provided by the available middle ware to support their needs for communication and data sharing.

  Ex:An application that is suited to the client-server model such as a database of name and addresses can rely on middleware that provides only remote method invocation.

- Some aspects of the dependability of systems require support at the application level.

  Ex: transfer of large file over unreliable network. TCP provides some error detection and correction, bit it cannot recover from major network interruptions. The mail transfer service adds another level of fault tolerance, maintaining a record of progress and resuming transmission using a new TCP connection if the original one breaks.

2.2.2 System Architectures

The most evident aspect of distributed system design is the division of responsibilities between system components (applications, servers, and other processes) and the placement of the components on computers in the network.

- It has major implication for:
  - Performance
  - Reliability
  - Security

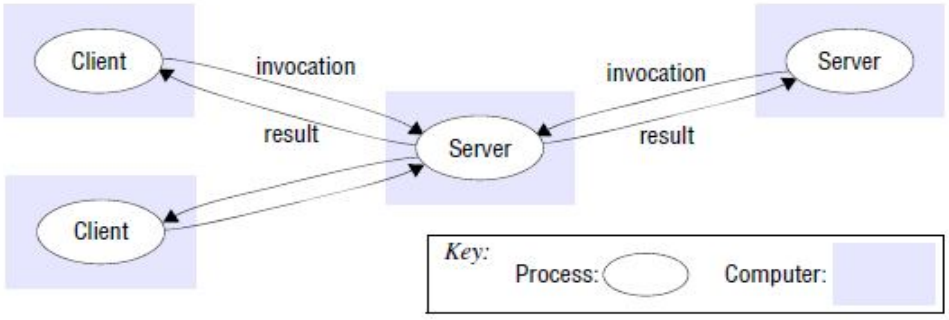
1. Client-Server model:

- Most often architecture for distributed systems.
- Client process interact with individual server processes in a separate host computers in order to access the shared resources
- Servers may in turn be clients of other servers.

E.g. a web server is often a client of a local file server that manages the files in which the web pages are stored.

E.g. a search engine can be 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

Fig :Clients invoke individual servers



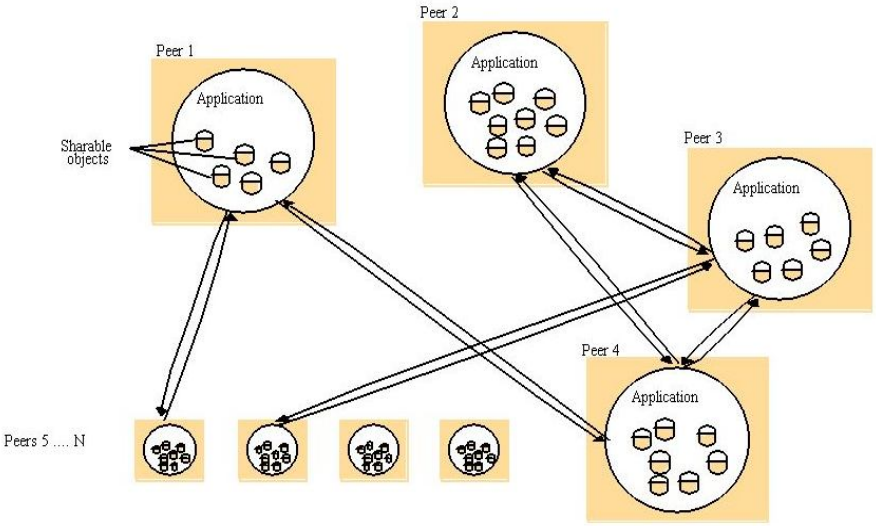
2. **Peer-to-Peer model**

- All of the processes play similar roles, interacting cooperatively as peers to perform a distributed activities or computations without any distinction between clients and servers or the computers that they run on.

- Peer-to-peer applications and systems have been successfully constructed that enable tens or hundreds of thousands of computers to provide access to data and other resources that they collectively store and manage.

E.g., music sharing system Napster

Fig : Peer-to-peer architecture



- Applications are composed of large numbers of peer processes running on separate computers.
- A large number of data objects are shared, an individual computer holds only a small part of the application database, and the storage, processing and communication loads for access to objects are distributed across many computers and network links.
- Each object is replicated in several computers to further distribute the load.

2.2.3 Variations

The problem of client-server model is placing a service in a server at a single address that does not scale well beyond the capacity of computer host and bandwidth of network connections.

To address this problem, several variations of client-server model derived from the consideration of the following factors:

- The use of multiple servers and caches to increase performance and resilience
- The use of mobile code and mobile agents
- Users' need for low-cot computer with limited hardware resources that are simple to manage.
- The requirement to add and remove mobile devices in a convenient manner.

1) Services provided by multiple servers

- Services may be implemented as several server processes in separate host computers interacting as necessary to provide a service to client processes.

- The servers may partition the set of objects on which the service is based and distribute them between themselves, or may maintain replicated copies of them on several hosts.
- The web provides a common example of partitioned data in which each web server manages its own set of resources. A user can employ a browser to access a resource at any one of the servers.

E.g. service based on replication data in the Sun NIS (Network Information Service), used by the computers on a LAN when users log in. Each NIS server has its own replica of the password file containing a list of users' login names and encrypted passwords.

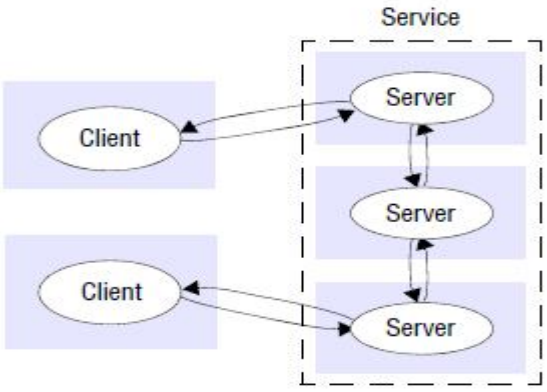


Fig : A service provided by multiple servers

2) Proxy servers and caches

- A cache is a store of recently used data objects.
- When a new object is received at a computer it is added to the cache store, replacing some existing objects if necessary.
- When an object is needed by a client process the caching service first checks the cache and supplies the object from there if an up-to-date copy is available.
- If not, an up-to-data copy is fetched.
- Caches may be collected with each client or they may be located in a proxy server that can be shared by several clients.

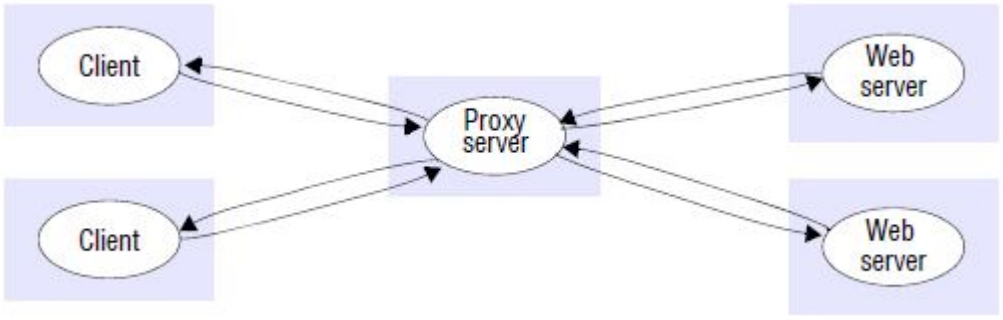


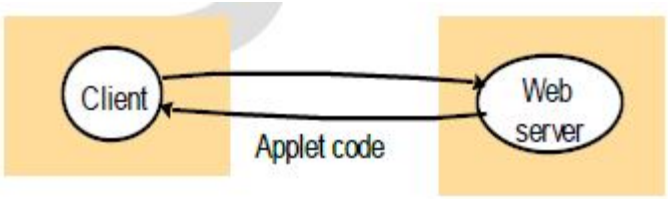
Fig : Web proxy server

3) Mobile code

- Applets are a well-known and widely used example of mobile code.
- Applets downloaded to clients give good interactive response.
- Mobile codes such as Applets are a potential security threat to the local resources in the destination computer.
- Browsers give applets limited access to local resources. For example by providing no access to local user file system.

  a stockbroker might provide a customized service to notify customers of changes in the prices of shares; to use the service, each customer would have to download a special applet that receives updates from the broker's server, display them to the user and perhaps performs automatic to buy and sell operations triggered by conditions set up by the customer and stored locally in the customer's computer.

a) client request results in the downloading of applet code



b) client interacts with the applet

Fig: Web applets

4) Mobile agents

- A running program (code and data) that travels from one computer to another in a network carrying out of a task, usually on behalf of some other process.

  Examples of the tasks that can be done by mobile agents are:
  - To collecting information.
  - To install and maintain software maintain on the computers within an organization.
  - To compare the prices of products from a number of vendors.

- Mobile agents are a potential security threat to the resources in computers that they visit.

- The environment receiving a mobile agent should decide on which of the local resources to be allowed to use.

- Mobile agents themselves can be vulnerable
  - ➢ They may not be able to complete their task if they are refused access to the information they need.

- Mobile agents are a potential security threat to the resources in computers that they visit.

- The environment receiving a mobile agent should decide on which of the local resources to be allowed to use.

- Mobile agents themselves can be vulnerable
  - ➢ They may not be able to complete their task if they are refused access to the information they need

5) Network computers

- It downloads its operating system and any application software needed by the user from a remote file server.

- Applications are run locally but the files are managed by a remote file server.

- Network applications such as a Web browser can also be run.

6) Thin clients

- It is a software layer that supports a window-based user interface on a computer that is local to the user while executing application programs on a remote computer.

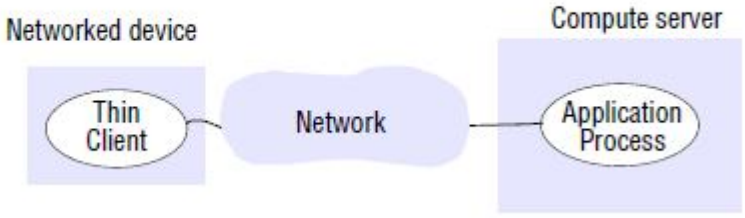
- This architecture has the same low management and hardware costs as the network computer scheme.

- Instead of downloading the code of applications into the user's computer, it runs them on a computer server.

- Compute server is a powerful computer that has the capacity to run large numbers of application simultaneously.

- The compute server will be a multiprocessor or cluster computer running a multiprocessor version of an operation system such as UNIX or Windows.

Thin clients and computer servers



2.2.4 Interfaces and Objects

- **The set of functions available for invocation in a process is specified by one or more interface definitions.**
- **In client-server architecture, each server process is seen as a single entity with a fixed interface defining the functions that can be invoked in it.**
- **Distributed processes can be constructed in object-oriented manner.**
- **Many objects can be encapsulated in server processes, and references to them are passed to other processes so that their methods can be accessed by remote invocation.**
- **This approach is adapted by JAVA RMI and** CORBA.
- **In a static client-server architecture or dynamic object-oriented model the distribution of responsibilities between processes and between computers remains an important aspect of the design.**
- **In traditional model responsibilities are statically allocated. Ex : fileserver responsible only for files not for web pages.**
- **In Object-oriented model new services, new types of object can be intatiated and immediately made available for invocation.**

2.2.5 Design requirements for distributed architectures

➢ **Performance issues**

Performance issues arising from the limited processing and communication capacities of computers and networks are considered under the following subheading:

- Responsiveness – response to interaction
  E.g. a web browser can access the cached pages faster than the non-cached pages.

- Throughput -- The rate at which computational work is done.

- Load balancing - -
  E.g. using applets on clients, remove the load on the server.

➢ **Quality of service**

- The ability of systems to meet deadlines.
- It depends on availability of the necessary computing and network resources at the appropriate time.

- 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.
  E.g. the task of displaying a frame of video.

- The main properties that affect the quality of the service are:
  - Reliability
  - Security
  - Performance
  - Adaptability

- Reliability and security issues are critical in the design of computer system. These two aspects related to the failure and security models.

- The performance aspect of quality of service was defined in terms of responsiveness and computational throughput, performance aspect of QOS is strongly related to the interaction model.

- Adaptability aspect of quality of service is, it able to adapt the changing system configurations.

- QOS applies to operating systems as well as network.

- Each critical resource must be reserved by the applications that require QOS and there must be resource managers that provide guarantees.

➢ **Use of caching and replication**

- Distributed systems overcome the performance issues by the use of data replication and caching.

- A variety of different cache consistency protocols are used to suit different applications.
  Ex: Web-caching protocol

➢ **Dependability issues**

Dependability issues of a computer system defined as:

- Correctness

  Ensuring the correctness of distributed and concurrent programs. Develop the techniques for checking the correctness of distributed system and concurrent programs execution.

- Security

  Security is locating sensitive data and other resources only in computers that can be secured effectively against attack.
  E.g. a hospital database

- Fault tolerance
  - ✓ Dependable applications should continue to function in the presence of faults in hardware, software, and networks.
    - ✓ Reliability is achieved through redundancy – the provision of multiple resources so that the system and application software can reconfigure and continue to perform task in the presence of faults.

2.3 Fundamental models

- Fundamental Models are concerned with a more formal description of the properties that are common in all of the architectural models.

- All architectural models are composed of processes that communicate with each other by sending messages over a computer networks.

- Aspects of distributed systems that are discussed in fundamental models are:

1) Interaction model
   - Computation occurs within processes.
   - The processes interact by passing messages, resulting in:
     - Communication (information flow)
     - Coordination (synchronization and ordering of activities) between processes
   - Interaction model reflects the facts that communication takes place with delays.

2) Failure model
   - Failure model defines and classifies the faults.

3) Security model
   - Security model defines and classifies the forms of attacks.
   - It provides a basis for analysis of threats to a system
   - It is used to design of systems that are able to resist threats

2.3.1 Interaction model

Distributed systems are composed of many processes, interacting in the following ways:

- Multiple server processes may cooperate with one another to provide a service
  E.g. Domain Name Service

- A set of peer processes may cooperate with one another to achieve a common goal
  E.g. voice conferencing

- Two significant factors affecting interacting processes in a distributed system are:

  ★ Communication performance is often a limiting characteristic.

  ★ It is impossible to maintain a single global notion of time

Interaction Model-Communication Channels

2.3.1.1 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
  - By simple message passing over a computer network
- Communication over a computer network has the performance characteristics such as:
  - Latency

    ★ The delay between the start of a message's transmission from one process to the beginning of its receipt by another.

  - Bandwidth
    ★ The total amount of information that can be transmitted over a computer network in a given time.

    ★ Communication channels using the same network, have to share the available bandwidth.
  - Jitter
    ★ The variation in the time taken to deliver a series of messages.
    ★ It is relevant to multimedia data.
    For example, if consecutive samples of audio data are played with differing time intervals then the sound will be badly distorted.

Interaction Model-Computer Clock

2.3.1.2 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.

- Two processes running on different computers can associate timestamp with their events.

- Even if two processes read their clock at the same time, their local clocks may supply different time.

- This is because computer clock drift from perfect time and their drift rates differ from one another.
- Clock drift rate refers to the relative amount that a computer clock differs 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 would eventually vary quite significantly unless corrections are applied.
- There are several techniques to correcting time on computer clocks.

  For example, computers may use radio signal receivers to get readings from GPS (Global Positioning System) with an accuracy about 1 microsecond.

Interaction Model-Variations

2.3.1.3 Two variants of the interaction model

In a distributed system it is hard to set time limits on the time taken for process execution, message delivery or clock drift.

Two models of time assumption in distributed systems are:

1. **Synchronous distributed systems**
   - It has a strong assumption of time
   - 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.

2. **Asynchronous distributed system**
   - It has no assumption about time.
   - There is no bound on process execution speeds.
     - Each step may take an arbitrary long time.
   - There is no bound on message transmission delays.
     - A message may be received after an arbitrary long time.
   - There is no bound on clock drift rates.
     - The drift rate of a clock is arbitrary.

2.3.2 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 behavior.

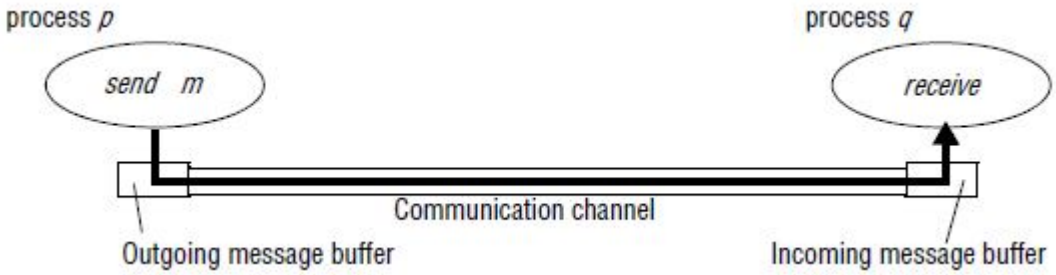
Types of failures:

- Omission Failures
- Arbitrary Failures

Omission failure

- Omission failures refer to cases when a process or communication channel fails to perform actions that it is supposed to do.
- The chief omission failure of a process is to crash. In case of the crash, the process has halted and will not execute any further steps of its program.
- Another type of omission failure is related to the communication which is called communication omission failure shown in Figure.

Fig :Processes and channels



- The communication channel produces an omission failure if it does not transport a message from "p"s outgoing message buffer to "q"'s incoming message buffer.

- This is known as "dropping messages" and is generally caused by lack of buffer space at the receiver or at an gateway or by a network transmission error, detected by a checksum carried with the message data.

Arbitrary failure

- Arbitrary failure is used to describe the worst possible failure semantics, in which any type of error may occur.

  E.g. a process may set wrong values in its data items, or it may return a wrong value in response to an invocation.

- Communication channel can suffer from arbitrary failures.
  E.g. message contents may be corrupted or non-existent messages may be delivered or real messages may be delivered more than once.

- The omission failures are classified together with arbitrary failures shown in Figure

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* operation 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 or commit omissions; a process may stop or take an incorrect step.

Figure : Omission and arbitrary failures

2.3.3 Security model

- The security of a distributed system can be achieved by securing the processes and the channels used in their interactions.
- Also, by protecting the objects that they encapsulate against unauthorized access.

Protecting objects

- Access rights
  - Access rights specify who is allowed to perform the operations on a object who is allowed to read or write its state.

- Principal
  - Principal is the authority associated with each invocation and each result.
  - A principal may be a user or a process.
  - The invocation comes from a user and the result from a server.
- The sever is responsible for

  - Verifying the identity of the principal (user) behind each invocation.
    - Checking that they have sufficient access rights to perform the requested operation on the particular object invoked. Rejecting those that do not.

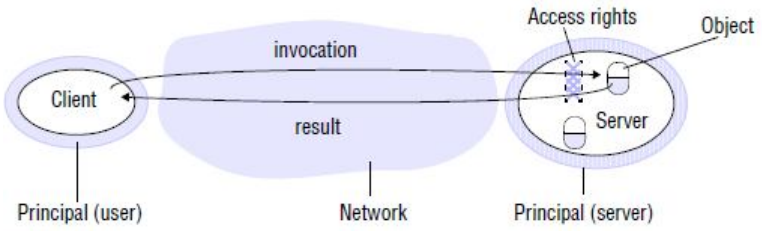


Figure : Objects and principals

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 is open.

The enemy

- To model security threats, we assume an enemy that is capable of sending any message to any process and reading or copying any message between a pair of processes.

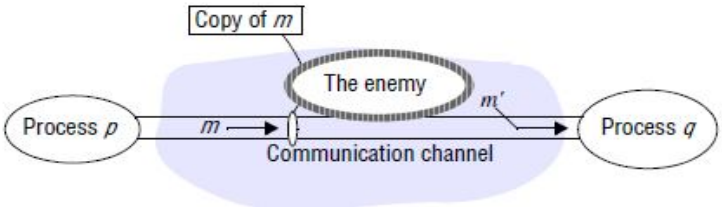


Figure :The enemy

- Threats from a potential enemy are classified as:
  - Threats to processes
  - Threats to communication channels
  - Denial of service

UNIT-II
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions ions

1. Which of the following applications require high communication bandwidth and low latency? [ ]

   A. Multi Media Applications B. Client-Server Applications
   C.Mobile Applications D.None of the above

2. Object-based architectures are [ ]
   A)Natural units of decomposition B)Accessed via interface
   C)Connected via RMI D)All the above

3. Identify the examples of Platform [ ]
   A.IntelX86/windows B.IntelX86/linux
   C.IntelX86/solaris D.All the above

4. The type of user interface supported by thin client is [ ]
   A.windowbased B.clientbased
   C.serverbased D.peerbased

5. Structurally, a network includes a setof nodes interconnected by a set of transmission lines, and each connection is called as [ ]
   A.Server B.Client C.Link D.Host

6. Which of the following comes under internal problems [ ]
   i)Non-synchronized clocks ii)Conflicting data updates
   iii)Many modes of hardware and software iv)Denial of service
   A. i,ii,iii B. i&ii C. i&iii D. i,ii,iii & iv

7. The problem with Client-Server System is [ ]
   A)Setting up the server is a complex technical task as well as maintaining and sorting out technical problems.
   B)Servers are expensive
   C)Processes requests too fast
   D)Both A and B

8. Which combination of failures will be caused across the server due to lack of reply or response from the server across the Distributed Systems
   i)omission failures ii)Timing failures
   iii)Arbitary failures iv)None of the above

A. i, ii B. ii, iii C. i, iii D. iv [ ]

9. The communication channel produces an omission failure if it does not transport a message from **p's** outgoing message buffer to **q's** incoming message buffer this is known as
   a)holding message b)dropping message [ ]
   c)buffered message d)None of the above.

10. To build asynchronous distribute system, what is required for the processes to perform tasks [ ]
    a)Sufficient process cycles
    b)Network capacity
    c)Supply clocks with bounded drift rates
    d)All of the above.

Section B:

Descriptive Questions

1) Identify difficulties and threats to distributed systems.
2) Summarize the Software Layers of distributed system architectural model.
3) Explain about the protection of objects against unauthorized access.
4) Illustrate the Architectural design of distributed system.
5) List different descriptive models of Distributed System, Explain in brief.
6) Explain about design requirements for distributed architectures.
7) Illustrate the distributed application based on peer-to-peer architecture.
8) Identify different types of failure models in distributed systems and their recovery.
9) "It is hard to set time limits on the time taken for process execution and message delivery in the interaction model", how to resolve this problem?
10) Analyze the important aspects of Quality of service and identify non functional properties that affect the quality of service.

UNIT-III

Interprocess Communication

Objective :

To familiarize the characteristics of protocols for communication between processes in a distributed system.

Syllabus:

Interprocess Communication: Introduction, The API for the Internet Protocols- The Characteristics of Interprocess communication, Sockets, UDP Datagram Communication, TCP Stream Communication; External Data Representation and Marshalling; Client Server Communication; Group Communication- IP Multicast- an implementation of group communication, Reliability and Ordering of Multicast.

Learning Outcomes

At the end of the unit student will be able to

1) Design an API by using TCP and UDP
2) Implement group communication.

3.1 INTRODUCTION

- The java API for interprocess communication in the internet provides both datagram and stream communication.

- The two communication patterns that are most commonly used in distributed programs:
  - Client-Server communication
    - The request and reply messages provide the basis for remote method invocation (RMI) or remote procedure call (RPC).

- Group communication
  - ❖ The same message is sent to several processes.

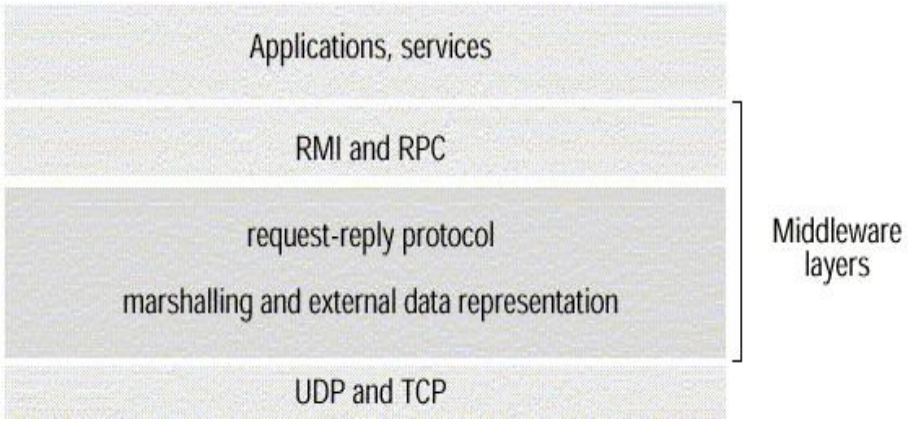


Fig: **Middleware layers**

- ➢ Remote Method Invocation (RMI)
  - It allows an object to invoke a method in an object in a remote process.
    - ❖ E.g. CORBA and Java RMI

- ➢ Remote Procedure Call (RPC)
  - It allows a client to call a procedure in a remote server.

- ➢ The application program interface (API) to UDP provides a message passing abstraction.
  - Message passing is the simplest form of interprocess communication.
  - API enables a sending process to transmit a single message to a receiving process.
  - The independent packets containing theses messages are called datagrams.
  - In the Java and UNIX APIs, the sender specifies the destination using a socket.

- Socket is an indirect reference to a particular port used by the destination process at a destination computer.

- The application program interface (API) 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.
- Request-reply protocols are designed to support client-server communication in the form of either RMI or RPC.
- Group multicast protocols are designed to support group communication.
- Group multicast is a form of interprocess communication in which one process in a group of processes transmits the same message to all members of the group.

3.2 The API for the Internet Protocols

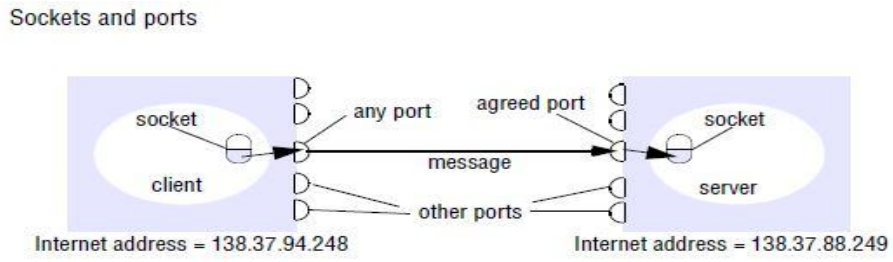
3.2.1 : The Characteristics of Interprocess Communication

- Synchronous and asynchronous communication
  - In the synchronous form, both send and receive are blocking operations.
  - In the asynchronous form, the use of the send operation is non-blocking and the receive operation can have blocking and non-blocking variants.
- Message destinations
  - Messages are send to Internet address and local port pair.
  - A local port is a message destination within a computer, specified as an integer.
  - A port has an exactly one receiver but can have many senders.
- Reliability
  - A reliable communication is defined in terms of validity and integrity.
  - A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost.
  - For integrity, messages must arrive uncorrupted and without duplication.
- Ordering
  - Some applications require that messages be delivered in sender order.

3.2.2 : Sockets

- Internet IPC mechanism of Unix and other operating systems (BSD Unix, Solaris, Linux, Windows NT, Macintosh OS)
- Processes in the above OS can send and receive messages via a socket.
- Sockets need to be bound to a port number and an internet address in order to send and receive messages.
- Each socket has a transport protocol (TCP or UDP).
- Messages sent to some internet address and port number can only be received by a process using a socket that is bound to this address and port number.
- Processes cannot share ports (exception: TCP multicast).
- Both forms of communication, UDP and TCP, use the socket abstraction, which provides and endpoint for communication between processes.
- Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process.

Figure . Sockets and ports



3.2.3: UDP Datagram Communication

- UDP datagram properties
  - No guarantee of order preservation
  - Message loss and duplications are possible

- Necessary steps

- Creating a socket
- Binding a socket to a port and local Internet address
  - ❖ A client binds to any free local port
  - ❖ A server binds to a server port

➢ Receive method
- It returns Internet address and port of sender, plus message.

➢ Issues related to datagram communications are:
- Message size
  - ❖ IP allows for messages of up to 216 bytes.
  - ❖ Most implementations restrict this to around 8 kbytes.
  - ❖ Any application requiring messages larger than the maximum must fragment.
  - ❖ If arriving message is too big for array allocated to receive message content, truncation occurs.
- Blocking
  - ❖ Send: non-blocking
    - ▪ upon arrival, message is placed in a queue for the socket that is bound to the destination port.
  - ❖ Receive: blocking
    - ▪ Pre-emption by timeout possible
    - ▪ If process wishes to continue while waiting for packet, use separate thread
- Timeout
- Receive from any

➢ UDP datagrams suffer from following failures:
- Omission failure
- Messages may be dropped occasionally,
- Ordering

Java API for UDP Datagrams

- The Java API provides datagram communication by two classes:
  - DatagramPacket
    - It provides a constructor to make an array of bytes comprising:
      - Message content
      - Length of message
      - Internet address
      - Local port number
    - It provides another similar constructor for receiving a message.

  - DatagramSocket
    - This class supports sockets for sending and receiving UDP datagram.
    - It provides a constructor with port number as argument.
    - No-argument constructor is used to choose a free local port.
    - DatagramSocket methods are:
      - send and receive
      - setSoTimeout
      - connect
- Example
- The process creates a socket, sends a message to a server at port 6789 and waits to receive a reply.

import java.net.*;

```
import java.io.*;

public class UDPClient{

public static void main(String args[]){

// args give message contents and destination hostname try {
DatagramSocket aSocket = new DatagramSocket(); // create socket byte [] m = args[0].getBytes();
InetAddress aHost = InetAddress.getByName(args[1]); // DNS lookup int serverPort = 6789;
DatagramPacket request =
new DatagramPacket(m, args[0].length(), aHost, serverPort);
aSocket.send(request); //send nessage byte[] buffer = new byte[1000];
DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
aSocket.receive(reply); //wait for reply System.out.println("Reply: " + new String(reply.getData())); aSocket.close();
}catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());}
} finally{if (aSocket !=null)aSocket.close()}
}
}
```

Figure. UDP client sends a message to the server and gets a reply (Above Java Code)

- Example

- The process creates a socket, bound to its server port 6789 and waits to receive a request message from a 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()}
}
}
```

Figure. UDP server repeatedly receives a request and sends it back to the client (Above Java Code)

3.2.4: TCP Stream Communication

- The API to the TCP protocol provides the abstraction of a stream of bytes to be written to or read from.
    - Characteristics of the stream abstraction:
        - Message sizes
        - Lost messages
        - Flow control
        - Message destinations

- Use of TCP
  - Many services that run over TCP connections, with reserved port number are:
    - HTTP (Hypertext Transfer Protocol)
    - FTP (File Transfer Protocol)
    - Telnet
    - SMTP (Simple Mail Transfer Protocol)

Issues related to stream communication:

- Matching of data items
- Blocking
- Threads

- Java API for TCP streams
  - The Java interface to TCP streams is provided in the classes:
    - ServerSocket
      - It is used by a server to create a socket at server port to listen for connect requests from clients.
    - Socket
      - It is used by a pair of processes with a connection.
      - The client uses a constructor to create a socket and connect it to the remote host and port of a server.
      - It provides methods for accessing input and output streams associated with a socket.

- Example
- The client process creates a socket, bound to the hostname and server port 6789.

Example : The server process opens a server socket to its server port 6789 and listens for connect requests.

```
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 socket:"+e.getMessage());}
}
}
```

Figure:. TCP server makes a connection for each client and then echoes the client's request

```
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("readline:"+e.getMessage());}
} finally {try{clientSocket.close();}catch(IOException e){/*close failed*/}}
}
}
```

Figure. TCP server makes a connection for each client and then echoes the client's request

3.3: External Data Representation

- The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes.
- Irrespective of the form of communication used, the data structure must be converted to a sequence of bytes before transmission and rebuilt on arrival.
- External Data Representation is an agreed standard for the representation of data structures and primitive values.
- Data representation problems are:

  - Using agreed external representation, two conversions necessary
  - Using sender's or receiver's format and convert at the other end

- Marshalling

  - Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.

- Unmarshalling

  - Unmarshalling is the process of disassembling a collection of data on arrival to produce an equivalent collection of data items at the destination.

- Three approaches to external data representation and marshalling are:
  - CORBA
  - Java's object serialization
  - XML
- Marshalling and unmarshalling activities is usually performed by middleware layer
- Marshalling is likely error-prone if carried out by hand

3. 3.1: CORBA Common Data Representation (CDR)

- CORBA Common Data Representation (CDR)
  - CORBA CDR is the external data representation defined with CORBA 2.0.
    - It consists 15 primitive types:
      - Short (16 bit)
      - Long (32 bit)
      - Unsigned short
      - Unsigned long
      - Float(32 bit)
      - Double(64 bit)
      - Char
      - Boolean(TRUE,FALSE)
      - Octet(8 bit)
      - Any(can represent any basic or constructed type)
  - Composite type are shown in Figure

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

Figure. CORBA CDR for constructed types

- Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in above table / Figure.
- Below Figure/ table shows a message in CORBA CDR that contains the three fields of a struct whose respective types are string, string, and unsigned long.

➢ example: struct with value {'Smith', 'London', 1934}

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*

Figure. CORBA CDR message

3.3.2 :Java object serialization

➢ In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation.

➢ An object is an instance of a Java class.

- Example, the Java class equivalent to the Person struct

```
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

}
```

Serialized values				Explanation
Person	8-byte version number		h0	class name, version number
3	int year	java.lang.String name	java.lang.String place	number, type and name of instance variables
1934	5 Smith	6 London	h1	values of instance variables

Figure '. Indication of Java serialization form

3.3.4 Remote Object References

- Remote object references are needed when a client invokes an object that is located on a remote server.
- A remote object reference is passed in the invocation message to specify which object is to be invoked.
- Remote object references must be unique over space and time.
- In general, may be many processes hosting remote objects, so remote object referencing must be unique among all of the processes in the various computers in a distributed system.
- generic format for remote object references is shown in below Figure.

32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface of remote object

Figure : Representation of a remote object references

- internet address/port number: process which created object
- time: creation time

- object number: local counter, incremented each time an object is created in the creating process
- interface: how to access the remote object (if object reference is passed from one client to another)

3. 4: Client-Server Communication

- The client-server communication is designed to support the roles and message exchanges in typical client-server interactions.
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.
- Often built over UDP datagrams
- Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary
- Avoidance of connection establishment overhead
- No need for flow control due to small amounts of data are transferred
- The request-reply protocol was based on a trio of communication primitives:
- doOperation, getRequest, and sendReply shown in Figure .

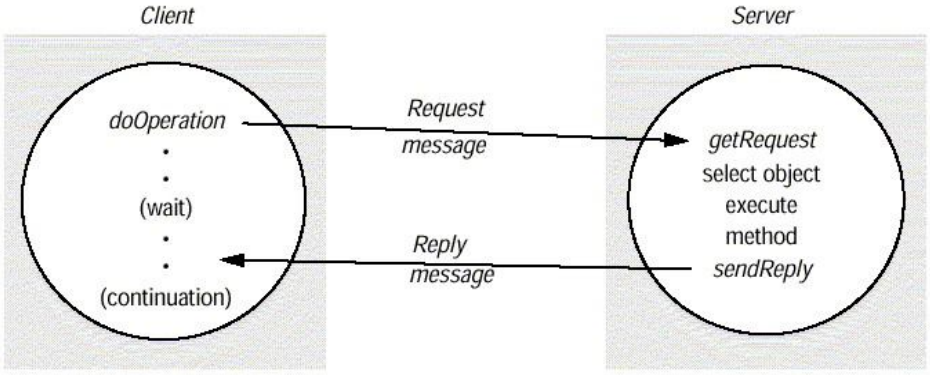


Figure:. Request-reply communication

- The designed request-reply protocol matches requests to replies.

- If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.
- Figure outlines the three communication primitives.

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.

Figure :. Operations of the request-reply protocol

- The information to be transmitted in a request message or a reply message is shown in Figure .

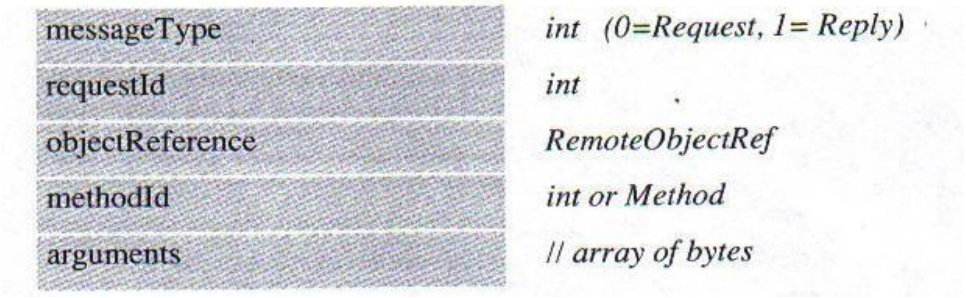


Figure : Request-reply message structure

- In a protocol message
  - The first field indicates whether the message is a request or a reply message.
  - The second field request id contains a message identifier.
  - The third field is a remote object reference .
  - The forth field is an identifier for the method to be invoked.
- Message identifier
  - A message identifier consists of two parts:

- ❖ A requestId, which is taken from an increasing sequence of integers by the sending process
- ❖ An identifier for the sender process, for example its port and Internet address.

➢ Failure model of the request-reply protocol

- If the three primitive doOperation, getRequest, and sendReply are implemented over UDP datagram, they have the same communication failures.
  - ❖ Omission failure
  - ❖ Messages are not guaranteed to be delivered in sender order.

➢ RPC exchange protocols

- Three protocols are used for implementing various types of RPC.
  - ❖ The request (R) protocol.
  - ❖ The request-reply (RR) protocol.

The request-reply-acknowledge (RRA) protocol.

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Figure. RPC exchange protocols

➢ In the R protocol, a single request message is sent by the client to the server.

➢ The R protocol may be used when there is no value to be returned from the remote method.

➢ The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol.

➢ RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply.

➢ HTTP: an example of a request-reply protocol

- HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.

- HTTP protocol steps are:
  - ❖ Connection establishment between client and server at the default server port or at a port specified in the URL
  - ❖ client sends a request
  - ❖ server sends a reply
  - ❖ connection closure
- HTTP 1.1 uses persistent connections.
  - ❖ Persistent connections are connections that remains open over a series of request-reply exchanges between client and server.
- Resources can have MIME-like structures in arguments and results.
- A Mime type specifies a type and a subtype, for example:
  - ❖ text/plain
  - ❖ text/html
  - ❖ image/gif
  - ❖ image/jpeg
- HTTP methods
  - ❖ GET
    - Requests the resource, identified by URL as argument.
    - If the URL refers to data, then the web server replies by returning the data
    - If the URL refers to a program, then the web server runs the program and returns the output to the client.

method	*URL*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure :. HTTP request message

- HEAD
  - This method is similar to GET, but only meta data on resource is returned (like date of last modification, type, and size)
- POST
  - Specifies the URL of a resource (for instance, a server program) that can deal with the data supplied with the request.
  - This method is designed to deal with:
    - Providing a block of data to a data-handling process
    - Posting a message to a bulletin board, mailing list or news group.
    - Extending a dataset with an append operation
- PUT
  - Supplied data to be stored in the given URL as its identifier.
- DELETE
  - The server deletes an identified resource by the given URL on the server.
- OPTIONS
  - A server supplies the client with a list of methods.
  - It allows to be applied to the given URL
- TRACE
  - The server sends back the request message
- A reply message specifies
  - The protocol version
  - A status code
  - Reason
  - Some headers
  - An optional message body

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

3.5: **Group Communication**

- The pairwise exchange of messages is not the best model for communication from one process to a group of other processes.
- A multicast operation is more appropriate.
- Multicast operation is an operation that sends a single message from one process to each of the members of a group of processes.
- The simplest way of multicasting, provides no guarantees about message delivery or ordering.
- Multicasting has the following characteristics:
  - 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.
  - Finding the discovery servers in 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.
  - Better performance through replicated data
    - Data are replicated to increase the performance of a service.
  - Propagation of event notifications
    - Multicast to a group may be used to notify processes when something happens.

3.5.1: IP multicast

- IP multicast is built on top of the Internet protocol, IP.
- IP multicast allows the sender to transmit a single IP packet to a multicast group.
- A multicast group is specified by class D IP address for which first 4 bits are 1110 in IPv4.
- The membership of a multicast group is dynamic.
- A computer belongs to a multicast group if one or more processes have sockets that belong to the multicast group.
- The following details are specific to IPv4:

- Multicast IP routers
  - ❖ IP packets can be multicast both on local network and on the wider Internet.
  - ❖ Local multicast uses local network such as Ethernet.
  - ❖ 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.
- Multicast address allocation
  - ❖ Multicast addressing may be permanent or temporary.
  - ❖ Permanent groups exist even when there are no members.
  - ❖ Multicast addressing by temporary groups must be created before use and cease to exit when all members have left.
  - ❖ The session directory (sd) program can be used to start or join a multicast session.
  - ❖ session directory provides a tool with an interactive interface that allows users to browse advertised multicast sessions and to advertise their own session, specifying the time and duration.
- Java API to IP multicast
  - ❖ The Java API provides a datagram interface to IP multicast through the class MulticastSocket, which is a subset of DatagramSocket with the additional capability of being able to join multicast groups.
  - ❖ The class MulticastSocket provides two alternative constructors , allowing socket to be creative to use either a specified local port, or any free local port.

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
   public static void main(String args[]){
    // args give message contents and 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);
        byte[] buffer = new byte[1000];
        for(int i=0; i< 3;i++) {                    // get messages from others in group
            DatagramPacket messageIn = new DatagramPacket(buffer, buffer.length);
            s.receive(messageIn);
            System.out.println("Received:" +new String(messageIn.getData()));
            }
        s.leaveGroup(group);
    }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    }catch (IOException e){System.out.println("IO: " + e.getMessage());
    }finally {if(s != null) s.close();}
 }

}
```

Figure :. Multicast peer joins a group and sends and receives datagrams

- A process can join a multicast group with a given multicast address by invoking the joinGroup method of its multicast socket.
- A process can leave a specified group by invoking the leaveGroup method of its multicast socket.
- The Java API allows the TTL to be set for a multicast socket by means of the setTimeToLive method. The default is 1, allowing the multicast to propagate only on the local network.

3.5.2 Reliability and Ordering of Multicast

Some examples of the effects of reliability and ordering .We now consider 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.
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.
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.

UNIT-III
Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. The granting of the use of a resource for a period of time is called as [ ]

a)lease b)rent c)revoke d)grant

2. A remote procedure call is initiated by [ ]

a) server b) client c) both a and b d) none of them

3. Socket style API for windows is known as [ ]

a) wsock b) winsock c) wins d) all of the above

4. Whic of the following is a TCP name for a transport service access point [ ]

a) port b) pipe c) node d) link

5. Which of the following is the type of socket [ ]

a)Datagram b)Stream c)Raw d)all of the above

6. Which of the following allows a client to call a procedure in a remote server. [ ]

a) Remote procedure call(RPC) c) Remote process call
b) Remote access call d) none of the above

7. JavaAPI provides a datagram interface to IP multicast through which of the following clas

[ ]

a)multicastsocket b)multipinsocket c)clientsocket d)serversocket

8. Which operation is performed repeatedly with the same effect as if it had been performed Exactly once . [ ]

a) identical operation b)idempotent operatio c)unique operation d)single operation

9. In which format we have to store the records correspondence between local object references In that process and remote object references [ ]

a)remote object table b)remote method call

c)remote procedure call d)port table

10. Recognize the different objects involved in the Jini distributed event specification

i)eventgenerators ii)remoteeventlisteners

iii)remoteevents iv)thirdpartyagents [ ]

a)i&ii b)ii&iv c)i&ii&iii&iv d)iii&iv

11. Identify type of module that is responsible for translating between local and remote object references and for creating remote object references. [ ]

a)remote reference b)object referenc c)local reference d)module reference

SECTION-B

SUBJECTIVE QUESTIONS

1) List different characteristics of IPC.
2) Identify the issues related to datagram communication.
3) Does java supports object serialization? Support your argument with necessary explanation.
4) Summarize the design issues for RMI.
5) Differentiate TCP stream communication and client Server communication.
6) Write CORBACDR message format for **structwithvalue{‘smith’,’london’,1934}** and List different primitive types used in CORBA Common Data Representation(CDR).
7) Sketch the format of the client-server request-reply communication protocol and outline the different communication primitives syntax.
8) Illustrate the concept of socket and ports with internet addresses ranging from 138.37.94.248 to 138.37.94.249.
9) Identify the problems in external data representation and suggest approaches for external data representation and marshalling.

How is RMI implemented in java?

Unit-IV
Distributed Objects and Remote Invocation

Syllabus: Distributed Objects and Remote Invocation: Introduction, Communication between Distributed Objects- Object Model, Distributed Object Modal, Design Issues for RMI, Implementation of RMI, Distributed Garbage Collection; Remote Procedure Call, Events and Notifications, Case Study: JAVA RMI

Topic 01: INTRODUCTION

- Objects that can receive remote method invocations are called remote objects and they implement a remote interface.
- Programming models for distributed applications are:
  - Remote Procedure Call (RPC)
    - Client calls a procedure implemented and executing on a remote computer
    - Call as if it was a local procedure
  - Remote Method Invocation (RMI)
    - Local object invokes methods of an object residing on a remote computer
    - Invocation as if it was a local method call
  - Event-based Distributed Programming
    - Objects receive asynchronous notifications of events happening on remote computers/processes
- Middleware
  - Software that provides a programming model above the basic building blocks of processes and message passing is called middleware.
  - The middleware layer uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events.

(Figure 1)

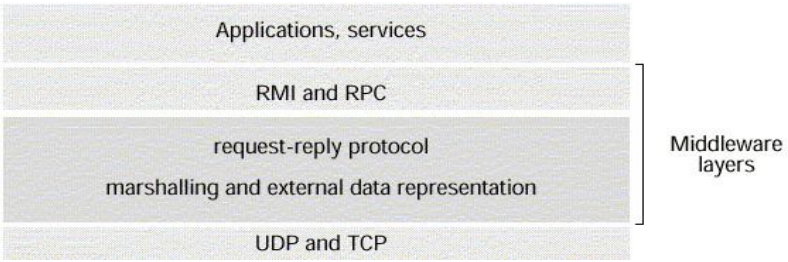


Figure 1. Middleware layers

- **Transparency Features of Middleware**
  - Location transparency:
    - In RMI and RPCs, the client calls a procedeure/method without knowledge of the location of invoked method/procedure.
  - Transport protocol transparency:
    - E.g., request/reply protocol used to implement RPC can use either UDP or TCP.
  - Transparency of computer hardware
    - They hide differences due to hardware architectures, such as byte ordering.
  - Transparency of operating system
    - It provides independency of the underlying operating system.
  - Transparency of programming language used
    - E.g., by use of programming language independent Interface Definition Languages (IDL), such as CORBA IDL.

```
// In file Person.idl
struct Person {
string name;
string place;
long year;
} ;
interface PersonList {
readonly attribute string listname;
void addPerson(in Person p) ;
void getPerson(in string name, out Person p);
long number();
};
```

Figure 2. CORBA IDL example

- Interfaces for RMI and RPC
  - An explicit interface is defined for each module.

- An Interface hides all implementation details.
- Accesses the variables in a module can only occur through methods specified in interface.
- Interface in distributed system
  - No direct access to remote variables is possible
    - Using message passing mechanism to transmit data objects and variables
      - » Request-reply protocols
      - » Local parameter passing mechanisms (by value, by reference) is not applicable to remote invocations
    - Specify input, output as attribute to parameters
      - » Input: transmitted with request message
      - » Output: transmitted with reply message
  - Pointers are not valid in remote address spaces
    - Cannot be passed as argument along interface
    - RPC and RMI interfaces are often seen as a client/server system
  - Service interface (in client server model)
    - Specification of procedures and methods offered by a server
  - Remote interface (in RMI model)
    - Specification of methods of an object that can be invoked by objects in other processes
- Interface Definition Languages (IDL)
  - Impossible to specify direct access to variables in remote classes
  - Hence, access only through specified interface
  - Desirable to have language-independent IDL that compiles into access methods in application programming language

❖ Example: CORBA IDL

(Figure 2)

Topic No 2: **Remote Procedure Call (RPC)**

- A remote procedure call (RPC) is similar to a remote method invocation (RMI).
- A client program calls a procedure in another program running in a server process.
- RPC, like RMI, may be implemented to have one of the choices of invocation semantics - at-least-once, at-most-once are generally chosen.
- RPC is generally implemented over a request-reply protocol.
- The software that support RPC is shown in Figure 3.

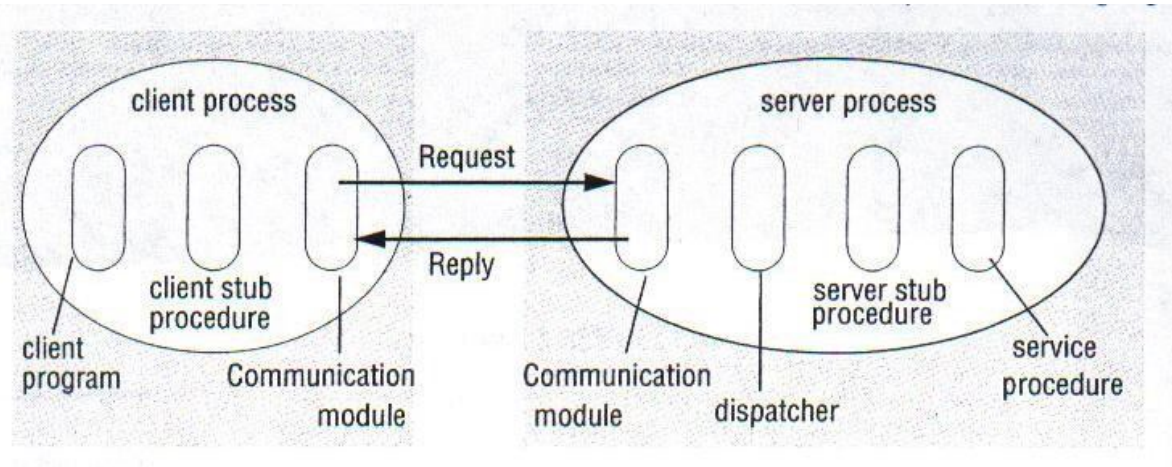


Figure 3. Role of client and server stub procedures in RPC in the context of a procedural language

- RPC only addresses procedure calls.
- RPC is not concerned with objects and object references.
- A client that accesses a server includes one stub procedure for each procedure in the service interface.
- A client stub procedure is similar to a proxy method of RMI (discussed later).
- A server stub procedure is similar to a skeleton method of RMI (discussed later).

RPC Example 1: Local Program

* A first program (hello.c) to test rpc. Use of this program is for

* testing the logic of the rpc programs.

```
#include <stdio.h>
```

```
int
main (void) {
    static char * result;
    static char msg[256];
    printf("getting ready to return value\n");
    strcpy(msg, "Hello world");
    result= msg;
    printf("Returning %s\n", result);
    return (0);
} /
```

Protocol Definition Program

- The name of this program is hello.x
- The number at the end is version number and should be updated each time the service is updated to make sure the active old copies is not responding to the client program.

```
program HELLO {
  version ONE{
  string PRINT_HELLO() = 1;
  } = 1 ;
} = 0x2000059;
```

Client Program

- Now we are ready to use rpcgen (command for generating the required programs).
- Note that so far we have only hello.c and hello.x
- After running "rpcgen -a -C hello.x" the directory contain following files:

```
-rw-rw-r-- 1 aabhari aabhari  131 Oct  5 12:15 hello.c
-rw-rw-r-- 1 aabhari aabhari  688 Oct  5 12:19 hello.h
-rw-rw-r-- 1 aabhari aabhari   90 Oct  5 12:18 hello.x
-rw-rw-r-- 1 aabhari aabhari  776 Oct  5 12:19 hello_client.c
-rw-rw-r-- 1 aabhari aabhari  548 Oct  5 12:19 hello_clnt.c
```

-rw-rw-r-- 1 aabhari aabhari 316 Oct 5 12:19 hello_server.c

-rw-rw-r-- 1 aabhari aabhari 2076 Oct 5 12:19 hello_svc.c

- The two templates that we should modify for this example are hello_client.c and hello_server.c.

Template of hello_client Program

* This is sample code generated by rpcgen.

* These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"

void hello_1(char *host)

{

    CLIENT *clnt;

    char * *result_1;

    char *print_hello_1_arg;

#ifndef DEBUG

    clnt = clnt_create (host, HELLO, ONE, "udp");

    if (clnt == NULL) {

        clnt_pcreateerror (host);

        exit (1);

    }

#endif  /* DEBUG */

    result_1 = print_hello_1((void*)&print_hello_1_arg, clnt);

    if (result_1 == (char **) NULL) {

        clnt_perror (clnt, "call failed");

    }

#ifndef DEBUG

 clnt_destroy (clnt);

#endif   /* DEBUG */

}
```

```
Int main (int argc, char *argv[])
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    hello_1 (host);
exit (0);
}
```

hello_client Program

- We have to modified hello_client template program.
- The modifications for our first example are very simple.
- Next show the modified program of hello_client that needs only few lines.
- * This is sample code generated by rpcgen.
- * These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"
#include <stdlib.h>
#include <stdio.h>
void
hello_1(char *host)
{
    CLIENT *clnt;
    char * *result_1;
    char *print_hello_1_arg;
```

```
#ifndef DEBUG
clnt = clnt_create (host, HELLO, ONE, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif  /* DEBUG */
    result_1 = print_hello_1((void*)&print_hello_1_arg, clnt);
    if (result_1 == (char **) NULL)
        clnt_perror (clnt, "call failed");
    else printf(" from server: %s\n",*result_1);
#ifndef DEBUG
    clnt_destroy (clnt);
#endif   /* DEBUG */
}
int
main (int argc, char *argv[])
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    hello_1 (host);
exit (0);
} //end clinet_server.c
```

Template of hello-server Program

* This is sample code generated by rpcgen.

* These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"

char **

print_hello_1_svc(void *argp, struct svc_req *rqstp)

{

    static char * result;

    /* insert server code here */

    return &result;

}
```

hello-server Program

* This is sample code generated by rpcgen.

* These are only templates and you can use them as a guideline for developing your own functions.

```
#include "hello.h"

char **

print_hello_1_svc(void *argp, struct svc_req *rqstp)

{

    static char * result;

    static char msg[256];

    printf("getting ready to return value\n");

    strcpy(msg, "Hello world");

    result= msg;

    printf("Returning\n");

    return &result;
```

```
}
```

Making Client and Server Program

- To compile the client

leda% gcc hello_client.c hello_clnt.c -o client -lnsl

- To compile the server

leda% gcc hello_server.c hello_svc.c -o server -lnsl

- To run the server use

leda% ./server

- To run the client use

elara% ./client leda

RPC

- rpcgen facilitates the generation of client and server stubs from the IDL program.
- It even generates client and server template programs.
- The option -a is passed to rpcgen and also all the generation of all support files including the make files.
- The -a option causes the rpcgen to halt with warnings if template files with default names exist.
- Consider turning a factorial program into a client-server program using RPC.

RPC Example 2

```
/* A program to calculate factorial numbers. */

#include <stdio.h>

void main(void){

   long int f_numb, calc_fact(int);

   int     number;

   printf("Factorial Calculator\n") ;

   printf("Enter a positive integer value");

   scanf("%d", &number);

   if (number < 0)

    printf("Positive value only!\n");
```

```
    else if ((f_numb = calc_fact(number)) > 0)
        printf("%d! = %d\n", number, f_numb);
    else
        printf("Sorry %d! is out of my range!\n", number);
 }
/* Calculate the factorial number and return the result or return 0 if
   the value is out of range. */
   long int calc_fact(int n) {
    long int  total = 1, last = 0;
    int      idx;
    for (idx = n; id<= 1; --idx)     {
        total *=idx;
        if (total < last)              /* Have we gone out of range? */
          return (0);
        last = total;
     }
  return (total);
}
/* The protocol definition file for the factorial program.
   the file name is fact.x */

 program FACTORIAL {
   version ONE{
   long int CALC_FAC(int) = 1;
   } = 1 ;
} = 0x2000049;
```

- We may now use rpcgen with the option flags -a -C to generate header file, the client and the server stub files and in addition, the client and server template programs.

- The content of the fact_client.c program is as shown below.

/* This is sample code generated by rpcgen.

These are only templates and you can use them as aguideline for developing your own functions. */

```
#include "fact.h"

void factorial_1(char *host)

{

  CLIENT *clnt;

   long *result_1

   int calc_fac_1_arg;

    #ifndef DEBUG

    clnt = clnt_create(host, FACTORIAL, ONE, "netpath");

    if (clnt == (CLIENT *) NULL) {

    clnt_pcreateerror(host);

     exit(1);

}

#endif  /* DEBUG */

     result_1 = calc_fac_1(&calc_fac_1_arg, clnt);

     if (result_1 == (long *) NULL) {

     clnt_perror(clnt, "call failed");

      }

      #ifndef DEBUG

      clnt_destroy(clnt);

       #endif  /* DEBUG */

}

main(int argc, char *argv[])

{

   char *host;
```

```
   if (arg < 2) {
   printf("usage: %s server_host\n", argv[0]);
   exit(1);
}
   host = argv[1];
    factorial_1(host);
}
```

- The template code for client needs to be modified to conform our original program.

```
/* This is sample code generated by rpcgen.
   These are only templates and you can use them
   as aguideline for developing your own functions. */
#include "fact.h"
#include  <unistd.h>      /* added because we will call exit*/
long int factorial_1(int calc_fac_1_arg, char *host)
{
  CLIENT *clnt;
   long *result_1;
#ifndef  DEBUG
        clnt = clnt_create(host, FACTORIAL, ONE, "netpath");
if (clnt == (CLIENT *) NULL) {
    clnt_pcreateerror(host);
    exit(1);
}
     #endif         /* DEBUG */
     result_1 = calc_fac_1(&calc_fac_1_arg, clnt);
     if (result_1 == (long *) NULL) {
     clnt_perror(clnt, "call failed");
```

```
    }
    #ifndef DEBUG
            clnt_destroy(clnt);
    #endif          /* DEBUG */
        return *result_1;
}
main(int argc, char *argv[])
{
  char *host;
/* Add own declarations here */
   long int  f_numb;
   int       number;

   if (arg < 2) {
   printf("usage: %s server_host\n", argv[0]);
   exit(1);
}
   host = argv[1];
/* This is the code from the previous main in program fact.c */
    printf("Factorial Calculation\n");
    printf("Enter a positive integer value");
    scanf("%d", &number);
    if  (number < 0)
       printf("Positive values only\n");
    else if ((f_numb = factorial_1(number, host)) >0 )
          printf("%d! = %d\n", number, f_numb);
     else
        printf("Sorry %d! is out of my range!\n", number);
```

```
}
```

- Here is the fact_server.c template generated by rpcgen

```
/* This is sample code generated by rpcgen.
   These are only templates and you can use them
   as aguideline for developing your own functions. */
   #include "fact.h"
    long int * calc_fac_1_srv(int *argp, struct svc_req *rqstp)
    {
        static long result;
      /* insert server code here */
            return(&result);
     }
```

- Here is the fact_server.c template with modification code

```
/* This is sample code generated by rpcgen.
   These are only templates and you can use them
   as aguideline for developing your own functions.  */
#include "fact.h"
    long int * calc_fac_1_srv(int *argp, struct svc_req *rqstp)
    {
        static long int result;
      /*  insert server code here */
        long int total = 1, last = 0;
        int idx;

for(idx = *argp; idx – 1; --idx)
            total *= idx;
            if (total <= last)      /* Have we gone out of range? */
```

```
        {
           result = 0;
           return (&result);
         }
        last = total;
      }
        result = total;
        return (&result);
  }
```

- Here is a modified makefile.fact
- # This is a template make file generated by rpcgen

```
#parameters
#added CC = gcc to use gcc
CC = gcc
CLIENT = fact_client
SERVER = fact_server
SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = fact.x
TARGETS_SVC.c = fact_svc.c fact_server.c
TARGETS_CLNT.c = fact_clnt.c fact_client.c
TARGETS = fact.h fact_clnt.c fact_svc.c fact_client.c fact_server.c
OBJECT_CLNT = $(SOURCES_CLNT.c:%.c=%.o) $(TARGETS_CLNT.c:%.c=%.o)
OBJECT_SVC = $(SOURCES_SVC.c:%.c=%.o) $(TARGETS_SVC.c:%.c=%.o)
```

```
# Compiler flags

CFLAGS += -g

LDLIBS += -lnsl

# added –C flag to PRCGENFLAGS or add –lm to LDLIBS

RPCGENFLAGS = -C

#Targets

all : $(CLIENT) $(SERVER)

$(TARGETS) : $(SOURCES.x)

        rpcgen $(RPCGENFLAGS) $(SOURCES.x)

$(OBJECTS_CLNT) : $(SOURCES_CLNT.c) $ (SOURCES_CLNT.h) \

          $(TARGETS_CLNT.c)

$(OBJECTS_SVC) : $(SOURCES_SVC.c) $ (SOURCES_SVC.h) \

          $(TARGETS_SVC.c)

$(CLIENT) : $(OBJECTS_CLNT)

        $ (LINK.c) –o $(CLIENT) $(OBJECTS_CLNT) $(LDLIBS)

$(SERVER) : $(OBJECTS_SVC)

        $ (LINK.c) –o $(SERVER) $(OBJECTS_SVC) $(LDLIBS)

Clean:

      $(RM) core $(TARGETS) $(OBJECTS_CLNT) $(OBJECTS_SVC) \

      $(CLIENT) $(SERVER)
```

Strength and Weaknesses of RPC

- RPC is not well suited for adhoc query processing. Consider the use of RPC to make SQL requests on servers that return arbitrary size of rows and number of tuples.
- It is not suited for transaction processing without special modification.
- A separate special mode of quering is proposed – Remote Data Access (RDA).
- RDA is specially suited for DBMS.
- In a general client_server environment both RPC and RDA are needed.

Topic No 3: **Java RMI**

- Java RMI extends the Java object model to provide support for distributed objects in the Java language.
- It allows object to invoke methods on remote objects using the same syntax as for local invocation.
- It is a single language system – remote interfaces are defined in the Java language.
- An object making a remote invocation needs to be able to handle RemoteExceptions that are thrown in the event of communication subsystem failures
- A remote object must implement Remote interface.
- The next example is a simple Hello world program that is implemented in Java RMI

Java RMI- Example 1

- In this example we do the followings:
  - Define the remote interface
  - Implement the server
  - Implement the client
  - Compile the source files
  - Start the Java RMI registry, server, and client

Hello.java - a remote interface

```
import java.rmi.Remote;

import java.rmi.RemoteException;

public interface Hello extends Remote {

   String sayHello() throws RemoteException;

}
```

```
// Implement the server

import java.rmi.registry.LocateRegistry;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {
```

```
  public Server() {}
  public String sayHello() {
        return "Hello, world!";
  }
public static void main(String args[]) {
        try {
          Server obj = new Server();
          Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
          // Bind the remote object's stub in the registry
          Registry registry = LocateRegistry.getRegistry();
          registry.bind("Hello", stub);
          System.err.println("Server ready");
        } catch (Exception e) {
          System.err.println("Server exception: " + e.toString());
          e.printStackTrace();
        }
  }
}
```

// Implement the client

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
  private Client() {}
public static void main(String args[]) {
        String host = (args.length < 1) ? null : args[0];
    try {
      Registry registry = LocateRegistry.getRegistry(host);
```

```
        Hello stub = (Hello) registry.lookup("Hello");
        String response = stub.sayHello();
        System.out.println("response: " + response);
      } catch (Exception e) {
        System.err.println("Client exception: " + e.toString());
        e.printStackTrace();
      }
    }
}
```

- The source files for this example can be compiled as follows:
  - javac Hello.java Server.java Client.java
  - Start the Java RMI registry :
  - rmiregistry
  - Start the Server: java Server
  - Start the Client: java Client

Java RMI Example 2

- Shared whiteboard
  - » It allows a group of users to share a common view of a drawing surface containing graphical objects.
  - » The server maintains the current state of a drawingClients can poll the server about the latest shape of a drawing.
  - » The server attaches version numbers to new arriving shapes.

- Remote Interface
  - Remote interfaces are defined by extending an interface called Remote provided in the java.rmi package.
  - Figure 4 shows an example of two remote interface called Shape and ShapeList.

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;          1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;   2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Figure 4. Java Remote interfaces Shape and ShapeList

- In this example, GraphicalObject is a class that holds the state of a graphical object.
- GraphicalObject must implement the Serializable interface.
- Ordinary and remote objects can appear as input and output arguments.

- Parameter and result passing
  - Passing remote objects
    - The result is passed by (object) reference.
    - In line2, the return value of the method newShape is defined as shape - a remote interface.
    - When a remote object reference is received, an RMI can be issued on the object refered to by this reference.
  - Passing non-remote objects
    - The result is passed by value.
    - A new object is created locally, with the state differing from the original object.
- Downloading of Classes
  - Classes can be transferred from one Java VM to another.
  - Parameters are passed by value or by reference.

- If the recipient does not yet possess the class of an object passed by value, its code is downloaded automatically.
- If the recipient of a remote object reference does not yet possess the class for a proxy, its code is downloaded automatically.

- RMIregistry
  - The RMIregistry is the binder for Java RMI.
  - The RMIregistry runs on every server that hosts remote objects.
  - It maps local object names of the form *//computerName:port/objName* to object references.
  - This service is not a global service.
  - Clients need to query a particular host to get reference.

(Figure 5)

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name,

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

String [] list()

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

Figure 5. The Naming class of Java RMIregistry

```
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());}
}
}
```

Figure 6. Java class ShapeListServer with main method

- In Figure 6:
  - Security Manager
    - implements various security policies for client accesses
  - Main method
    - 1: create instance of ShapeListServant
    - 2: binds name "ShapeList" to newly created instance in RMIRegistry
- *ShapeListServant* implements *ShapeList*
  - Figure 7 gives an outline of the class ShapeListServant.
    - 1: UnicastRemoteObject - objects that live only as long as creating process
    - 2: factory method - client can request creation of a new object

```
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{...}
}
```

Figure 7. Java class ShapeListServant implements interface ShapeList

- Client program
  - A simplified client for the ShapeList sever is illustrated in Figure 8.
    - polling loop:
      - 1: look up remote reference
      - 2: invoke allShapes() in remote object

```
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());}
 }
}
```

Figure 8. Java client of ShapeList

- Figure 9 shows the inheritance structure of the classes supporting Java RMI servers.

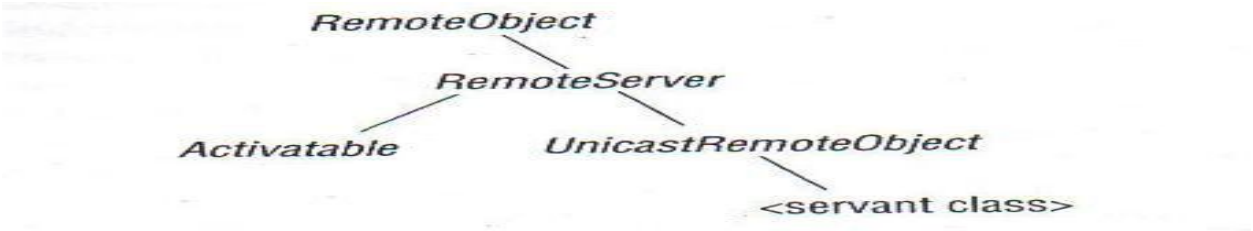


Figure 9. Classes supporting Java RMI

UNIT-IV
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. In RMI Architecture which layer Intercepts method calls made by the client/redirects these calls to a remote RMI service? [ ]

a.Stub&SkeletonLayer b.ApplicationLayer
c.RemoteReferenceLayer d.TransportLayer

2. An RMI Server is responsible for_____ [ ]

a.Creating an instance of the remote object
b.Exporting the remote object
c.Binding the instance of the remote object to the RMI registry
d.All the above

3. In RMI, the objects are passed by_____. [ ]

a.Value b.Reference
c.Value and Reference d.None of the above

4. Which of the following exceptions needs to be handled in an RMI client program? [ ]

a.RemoteException b.NotBoundException
c.MalFormedURLException d. All the above

5. Which allows client programs to call procedures in server programs running in separate process and generally in different computers from the client [ ]

a)RMI b)RPC c)IDL d)TCP

6. Which of the following is an instance of a class which provides the body of a remote object [ ]

a)server b)servent c)client d)none of the above

7. Which method keeps a history of result messages to enable lost results to be retransmitted without re-executing the operations of the server [ ]

a)retransmission of results b) retransmission of request
c)retransmission of process d)retransmission of messages

8. Which of the following retransmits the request message until either a reply is received or the server is assumed to have failed [ ]

a)retry request message b)request message
c)retransmit message d)exit message

9. Which method of the Naming class (found in java.rmi) is used to update the RMI registry on the server machine? [ ]

a)rebind() b)lookup() c)Both A & B d)None of the above

10. In RMI which layer defines and supports the invocation semantics of the RMIconnection and this layer maintains the session during the method call?

a)The Stub & Skeleton Layer b)The Application Layer
c)The Remote Reference Layer d)The Transport Layer [ ]

SECTION-B

SUBJECTIVE QUESTIONS

1. List the design issues for remote method invocation.
2. Summarize the following

   i)javaRMI ii)Events and Notifications

3. Illustrate the implementation of RMI in distributed system.
4. How distributed garbage collector works in cooperation with the local garbage collectors.
5. Describe the process of RPC and Identify its strengths and weaknesses.
6. Explain about the different types of interfaces used in distributed systems.
7. Explain the communication between distributed objects by RMI.
8. Illustrate the invocation semantics of remote method invocation.
9. Illustrate dealing room system with the help of events and notifications concept.
10. How would you incorporate persistent asynchronous communication in to a model of communication based on RMI to remote objects?

Unit-5
Operating System Support

Syllabus: Introduction, the Operating System Layer, Protection, Process and Threads; Address Space, Creation of a New Process, Threads

Topic 01: INTRODUCTION

- Many distributed operating systems have been investigated, but there are none in general/wide use.
- But network operating system are in wide use for various reasons both technical and non-technical.
  - Users have much invested in their application software; they will not adopt a new operating system that will not run their applications.
  - The second reason against the adoption of distributed operating system is that users tend to prefer to have a degree of autonomy for their machines, even in a organization.
- Unix and Windows are two examples of network operating systems.
- Those have a networking capability built into them and so can be used to access remote resources using basic services such as rlogin and telnet.
- The combination of middleware and network operating systems provides an acceptance balance between the requirement of autonomy and network transparency.
- The network operating systems allows users to run their favorite word processor and other standalone applications.

- Middleware enables users to take advantage of services that become available in their distributed system.

Topic 02: Operating System Layer

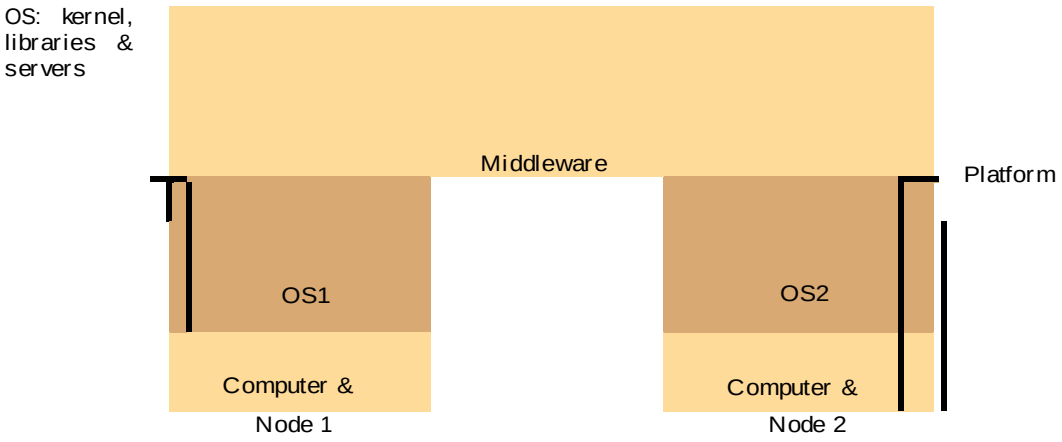


Figure 1. System layers

 Figure 1 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.

- Kernels and server processes are the components that manage resources and present clients with an interface to the resources.
- The OS facilitates:
  - Encapsulation
  - Protection
  - Concurrent processing
- Invocation mechanism is a means of accessing an encapsulated resource.
- The core OS components:
  - Process manager
  - Thread manager
  - Memory manager
  - Communication manager
  - Supervisor

Topic 03: Protection

- Protection means protecting resources (files) from illegitimate accesses.

Kernels and protection

- The kernel is a program that is distinguished by the facts that it always runs and its code is executed with complete access privileges for the physical resources on its host computer.
- It can control the memory management unit and set the processor registers so that no other code may access the machine's physical resources except in acceptable ways.
- A kernel process executes with the processor in supervisor (privileged) mode;
- The kernel arranges that other processes execute in user (unprivileged) mode.
- The kernel also sets up *address spaces* to protect itself and other processes from the accesses of an aberrant process
- To provide processes with their required virtual memory layout.
- An *address space* is a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, such as read only or read-write. A process cannot access memory outside its address space.
- The terms *user process* or *user-level process* are normally used to describe one that executes in user mode and has a user-level address space.
- When a process executes application code, it executes in a distinct user-level address space for that application.
- When the same process executes kernel code, it executes in the kernel's address space.

- The process can safely transfer from a user-level address space to the kernel's address space via an exception such as an interrupt or a *system call trap* – the invocation mechanism for resources managed by the kernel.

- A system call trap is implemented by a machine-level *TRAP* instruction, which puts the processor into supervisor mode and switches to the kernel address space.

Topic 04: Process and Threads

- Process
  - A process consists of an execution environment together with one or more threads.
  - A thread is the operating system abstraction of an activity.
  - An execution environment is the unit of resource management: a collection of local kernel managed resources to which its threads have access.
  - An execution environment consists of :
    - An address space
    - Thread synchronization and communication resources (e.g., semaphores, sockets)
    - Higher-level resources (e.g., file systems, windows)
  - Address spaces
    - An address space is a unit of management of a process's virtual memory.
    - It is large (typically up to 2^{32} bytes, and sometimes up to 2^{64} bytes) and consists of one or more **regions**, separated by inaccessible areas of virtual memory.
    - A region is an area of contiguous virtual memory that is accessible by the threads of the owning process.
  - Regions do not overlap.
  - The aim of having multiple threads of execution is :
    - its extent (lowest virtual address and size);
    - read/write/execute permissions for the process's threads;
    - whether it can be grown upwards or downwards

Motivation for Indefinite Number of Regions:

- Need to support a separate stack for each thread.
- To enable files

- *Mapped file* is one that is accessed as an array of bytes in memory.

- The need to share memory between processes or between processes and the kernel
  - A *shared memory* region(shared region) is one that is backed by the same physical memory as one or more regions belonging to other address spaces.
- The uses of shared regions include the following:
  - *Libraries*

- *Kernel*
- *Data sharing and communication*

Topic 05: Creation of a new process

- For a distributed system, the design of the process-creation mechanism has to take into account the utilization of multiple computers; consequently, the process-support infrastructure is divided into separate system services.
- The creation of a new process can be separated into two independent aspects:

  - The **choice of a target host**, for example, the host may be chosen from among the nodes in a cluster of computers acting as a compute server.
  - The creation of an execution environment.

CHOICE OF PROCESS HOST

- The choice of the node at which the new process will reside – the process allocation decision – is a matter of policy.
- In general, *process allocation policies* range from always running new processes at their originator's workstation to sharing the processing load between a set of computers.
- The *transfer policy* determines whether to situate a new process locally or remotely(load).
- The *location policy* determines which node should host a new process selected for transfer.
- *Process location policies* may be static or adaptive.
- The *static policies* operate without regard to the current state of the system, although they are designed according to the system's expected long-term characteristics.
- They are based on a mathematical analysis aimed at optimizing a parameter such as the overall process throughput.
- They may be deterministic or probabilistic,
- Adaptive policies, apply heuristics to make their allocation decisions, based on unpredictable runtime factors such as a measure of the load on each node.
- Load-sharing systems may be **centralized, hierarchical or decentralized**.
- In the first case there is one load manager component
- In the second there are several, organized in a tree structure.
- **Load managers** collect information about the nodes and use it to allocate new processes to nodes.
- In ***hierarchical systems***, managers make process allocation decisions as far down the tree as possible, but managers may transfer processes to one another, via a common ancestor, under certain load conditions.
- In a ***decentralized load-sharing system***, nodes exchange information with one another directly to make allocation decisions.

- The **Spawn system for** example, considers nodes to be 'buyers' and 'sellers' of computational resources and arranges them in a (decentralized) 'market economy'.
- In ***sender-initiated load-sharing algorithms***, the node that requires a new process to be created is responsible for initiating the transfer decision. It typically initiates a transfer when its own load crosses a threshold.
- By contrast, in ***receiver-initiated algorithms***, a node whose load is below a given threshold advertises its existence to other nodes so that relatively loaded nodes can transfer work to it.
- ***Migratory load-sharing systems*** can shift load at any time, not just when a new process is created. They use a mechanism called process migration: the transfer of an executing process from one node to another.

CREATION OF A NEW EXECUTION ENVIRONMENT

- Once the host computer has been selected, a new process requires an execution environment consisting of an address space with initialized contents.
- There are ***two*** approaches to defining and initializing the address space of a newly created process.
- The first approach is used where the address space is of a statically defined format. For example, it could contain just a program text region, heap region and stack region.
- Address space regions are initialized from an executable file or filled with zeros as appropriate
- Alternatively, the address space can be defined with respect to an existing execution environment
- In the case of UNIX *fork* semantics, for example, the newly created child process physically shares the parent's text region and has heap and stack regions that are copies of the parent's in extent (as well as in initial contents).
- This scheme has been generalized so that each region of the parent process may be inherited by (or omitted from) the child process.
- An inherited region may either be shared with or logically copied from the parent's region. When parent and child share a region, the page frames (units of physical memory corresponding to virtual memory pages) belonging to the parent's region are mapped simultaneously into the corresponding child region.

Copy-on-write

- **Copy-on-write** is a general technique – for example, it is also used in copying large messages – so we take some time to explain its operation here.

- Let us follow through an example of regions RA and RB, whose memory is shared copy-on-write between two processes, A and B .
- For the sake of definiteness, let us assume that process A set region RA to be copy-inherited by its child, process B, and that the region RB was thus created in process B.
- We assume, for the sake of simplicity, that the pages belonging to region A are resident in memory.
- Initially, all page frames associated with the regions are shared between the two processes' page tables.
- The pages are initially write-protected at the hardware level, even though they may belong to regions that are logically writable.
- If a thread in either process attempts to modify the data, a hardware exception called a page fault is taken.
- Let us say that process B attempted the write.
- The **page fault handler** allocates a new frame for process B and copies the original frame's data into it byte for byte.
- The old frame number is replaced by the new frame number in one process's page table – it does not matter which – and the old frame number is left in the other page table.
- The two corresponding pages in processes A and B are then each made writable once more at the hardware level. After all of this has taken place, process B's modifying instruction is allowed to proceed.

Threads:

Threads are schedulable activities attached to processes.

The aim of having multiple threads of execution is :
To maximize degree of concurrent execution between operations

- To enable the overlap of computation with input and output
- To enable concurrent processing on multiprocessors.
- Threads can be helpful within servers:
  - Concurrent processing of client's requests can reduce the tendency for servers to become bottleneck.
    - E.g. one thread can process a client's request while a second thread serving another request waits for a disk access to complete.

Processes vs. Threads

- Threads are "lightweight" processes,
- Processes are expensive to create but threads are easier to create and destroy.

Threads programming

- Thread programming is concurrent programming.
- Java provides methods for creating, destroying and synchronizing threads.
- Programs can manage threads in groups.
  - Every thread belongs to one group assigned at thread creation time.
  - Thread groups are useful to shield various applications running in parallel on one Java Virtual Machine (JVM).

- A thread in one group cannot perform management operations on a thread in another group.
- E.g., an application thread may not interrupt a system windowing (AWT) thread.

Thread synchronization

- The main difficult issues in multi-threaded programming are the sharing of objects and the techniques used for thread coordination and cooperation.
- Each thread's local variables in methods are private to it.

Threads have private stack.

Threads do not have private copies of static (class) variables or object instance variables.

Java provides the synchronized keyword for thread coordination.

any object can only be accessed through one invocation of any of its synchronized methods.

an object can have synchronized and non-synchronized methods.

synchronized addTo() and removeFrom() methods to serialize requests in worker pool example.

Threads can be blocked and woken up

The thread awaiting a certain condition calls an object's wait() method.

The other thread calls notify() or notifyAll() to awake one or all blocked threads.

example

- When a worker thread discovers that there are no requests to process, it calls wait() on the instance of Queue.
- When the I/O thread adds a request to the queue, it calls the queue's notify() method to wake up the worker.

UNIT-V
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Which system call returns the process identifier of a terminated cild?
   a)wait b)exit c)fork d)get
2. Which one of the following is not shared by threads? [ ]
a)program counter b)stack c)both(a)and(b) d)none of the above
3. A process can be
a)single threaded b)multi threaded
c)both(a)and(b) d)none of the above
4. Which one of the following is not a valid state of a thread? [ ]
a)running b)parsing
c)ready d)blocked
5. Which of the following cannot be scheduled by the kernel? [ ]
a)kernel level thread b)user level thread c)process d)none of the above
6. A heavy weight process: [ ]
a)has multiple threads of execution
b)has a single thread of execution
c)can have multiple or a single thread for execution
d)None of these
7. Which process can be affected by other processes executing in the system?
a)cooperating process b)child process [ ]
c)parent process d)init process
8. If a process is executing in its critical section, then no other processes can be executing in their critical section. This condition is called [ ]
a)mutual exclusion b)critical exclusion
c)synchronous exclusion d)asynchronous exclusion
9. Which one of the following is a synchronization tool? [ ]
a)thread b)pipe c)semaphore d)socket
10. If a kernel thread performs a blocking system call,_____.
a. the kernel can schedule another thread in the application for execution.
b. the kernel can not schedule another thread in the same application for execution.
c. the kernel must schedule another thread of a different application for execution.
d. the kernel must schedule another thread of the same application on a different processor.
11. In the Many to One model, if a thread makes a blocking system call: [ ]
a. the entire process will be blocked
b. a part of the process will stay blocked, with the rest running
c. the entire process will run
d. None of these
12. Multithreading an interactive program will increase responsiveness to the user by: [ ]

a. continuing to run even if a part of it is blocked
b. waiting for one part to finish before the other begins
c. asking the user to decide the order of multi threading
d. None of these

13. If the kernel is single threaded, then any user level thread performing a blocking system call will: [ ]
a. cause the entire process to run along with the other threads
b. cause the thread to block with the other threads running
c. cause the entire process to block even if the other threads are available to run
d. None of these

14. Thread synchronization is required because [ ]
a. all threads of a process share the same address space
b. all threads of a process share the same global variables
c. all threads of a process can share the same files
d. all the above

15. Which one of the following hides the location where in the network the file is stored?

a)transparent distributed file system
b)hidden distributed file system
c)escaped distribution file system
d)spy distributed file system

16. The address of the next instruction to be executed by the current process is provided by the [ ]
a)CPU registers b)program counter c)process stack d)pipe

SECTION-B

SUBJECTIVE QUESTIONS

1. Identify the need for protection? Explain various protection mechanisms supported by operating systems
2. Summarize the elements of an address space?
3. Illustrate the architecture of multi threaded server
4. Outline the operating system address space of threads in a distributed systems?
5. Summarize operating systems architecture.
6. How is new process created in a distributed systems? How is it different from Unix operating system.
7. Differentiate the process and thread in distributed environment.
8. Identify different constructors and management methods used in threads programming.
9. Compare the worker-pool multithreading architecture with the thread per request architecture.
10. How does copy-on-write of an inherited region done from parent to child process with an example

Unit-6

Coordination and Agreement

Syllabus: Introduction, Distributed Mutual Exclusion, Elections, Multicast, Communication. Transactions & Replications: Introduction, System Model and Group Communication, Concurrency Control in Distributed Transactions, Distributed Deal Locks, Transaction Recovery; Replication-Introduction, Passive (primary) Replication, Active Replication..

Topic01: INTRODUCTION

Failure Assumptions and Failure Detectors

reliable communication channels

process failures: crashes

failure detector: object/code in a process that detects failures of other processes

unreliable failure detector

unsuspected or suspected (evidence of possible failures)

each process sends ``alive" message to everyone else

not receiving ``alive" message after timeout

most practical systems

reliable failure detector

unsuspected or failure

synchronous system

few practical systems

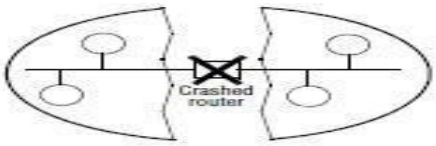


Figure : A network Partition

Topic 02: Distributed Mutual Exclusion

Distributed mutual exclusion for resource sharing

A collection of process **share resources**, mutual exclusion is needed to prevent interference and ensure consistency. (critical section)

No shared variables or facilities are provided by single local kernel to solve it. Require a solution that is based solely on message passing.

Application level protocol for executing a critical section

- enter() // enter critical section-block if necessary
- resrouceAccess() //access shared resoruces
- exit() //leave critical section-other processes may enter

Essential requirements:

ME1: (safety) at most one process may execute in the critical section

ME2: (liveness) Request to enter and exit the critical section eventually succeed.

ME3(ordering) One request to enter the CS happened-before another, then entry to the CS is granted in that order.

ME2 implies freedom from both deadlock and starvation. Starvation involves fairness condition. The order in which processes enter the critical section. It is not possible to use the request time to order them due to lack of global clock. So usually, we use happen-before ordering to order message requests.

Performance Evaluation

Bandwidth consumption, which is proportional to the number of messages sent in each entry and exit operations.

The **client delay** incurred by a process at each entry and exit operation.

throughput of the system. Rate at which the collection of processes as a whole can access the critical section. Measure the effect using the **synchronization delay**

between one process exiting the critical section and the next process entering it; the shorter the delay is, the greater the throughput is.

Sub topic 1.2:Central Server Algorithm

The simplest way to grant permission to enter the critical section is to employ a server.

A process sends a request message to server and awaits a reply from it.

If a reply constitutes a token signifying the permission to enter the critical section.

If no other process has the token at the time of the request, then the server replied immediately with the token.

If token is currently held by other processes, the server does not reply but queues the request.

Client on exiting the critical section, a message is sent to server, giving it back the token.

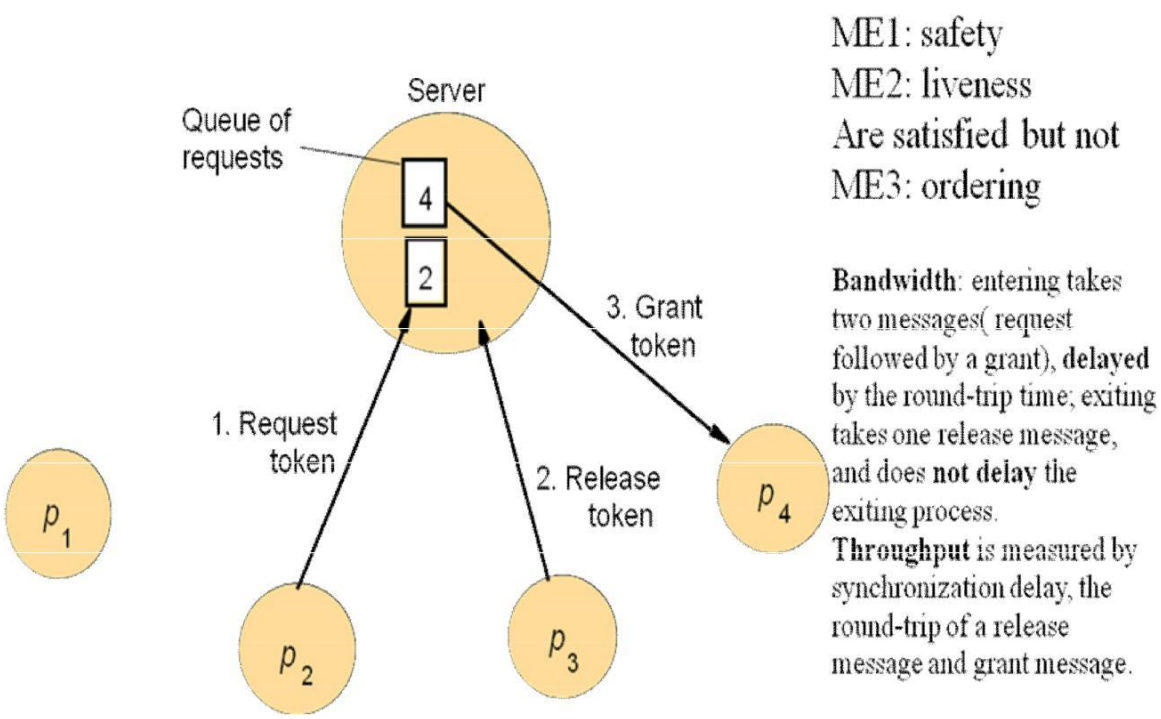


Figure: Central Server algorithm: managing a mutual exclusion token for a set of processes

Sub topic 1.3: Ring-based Algorithm

Simplest way to arrange mutual exclusion between N processes without requiring an additional process is arrange them in a logical ring.

Each process pi has a communication channel to the next process in the ring, p(i+1)/mod N.

The unique **token** is in the form of a message passed from process to process in a single direction clockwise.

If a process does not require to enter the CS when it receives the token, then it immediately forwards the token to its neighbor.

A process requires the token waits until it receives it, but retains it.

To exit the critical section, the process sends the token on to its neighbor.

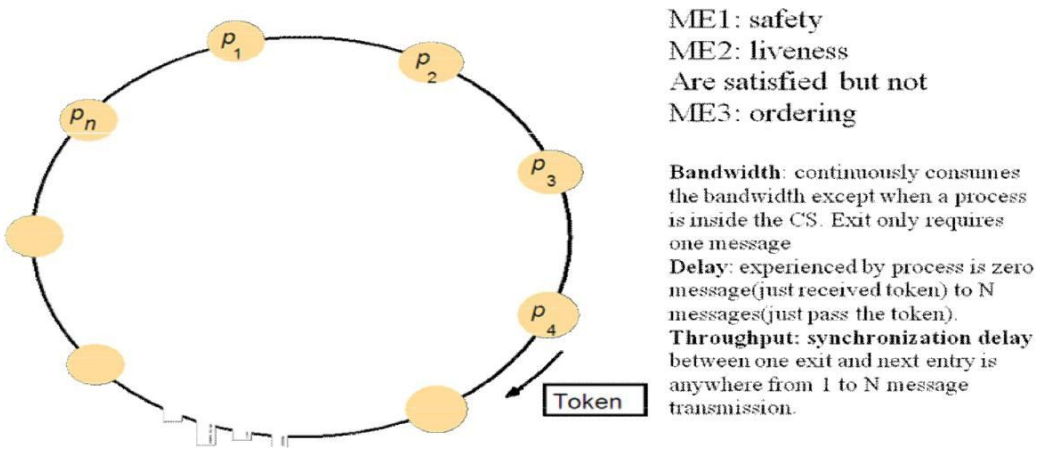


Fig: A ring of processes transferring a mutual exclusion token.

Sub topic 1.4 :Using Multicast and logical clocks

Mutual exclusion between N peer processes based upon multicast.

Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.

The condition under which a process replies to a request are designed to ensure ME1

ME2 and ME3 are met.

Each process pi keeps a Lamport clock. Message requesting entry are of the form<T, pi>.

Each process records its state of either RELEASE, WANTED or HELD in a variable state.

– If a process requests entry and all other processes is RELEASED, then all processes reply immediately.

– If some process is in state HELD, then that process will not reply until it is finished.

– If some process is in state WANTED and has a smaller timestamp than the incoming request, it will queue the request until it is finished.

– If two or more processes request entry at the same time, then whichever bears the lowest timestamp will be the first to collect N-1 replies.

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;

Fig: Ricart and Agrawala's algorithm

Sub topic 1.5 :Multicast synchronization

P1 and P2 request CS concurrently. The timestamp of P1 is 41 and for P2 is 34. When P3 receives their requests, it replies immediately. When P2 receives P1's request, it finds its own request has the lower timestamp, and so does not reply, holding P1 request in queue. However, P1 will reply. P2 will enter CS. After P2 finishes, P2 reply P1 and P1 will enter CS.

Granting entry takes 2(N-1) messages, N-1 to multicast request and N-1 replies.

Bandwidth consumption is high.

Client delay is again 1 round trip time

Synchronization delay is one message transmission time.

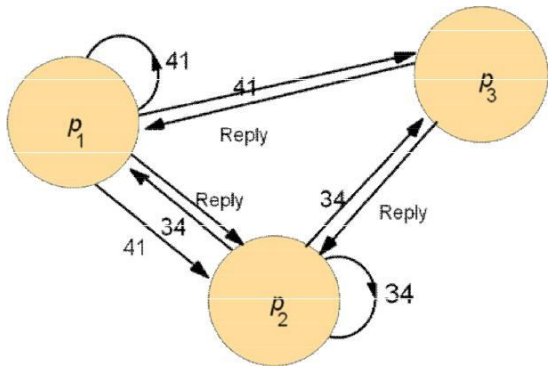


Figure: Multicast synchronization

Sub topic 1.6: Maekawa's voting algorithm

It is not necessary for all of its peers to grant access. Only need to obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.

Think of processes as voting for one another to enter the CS. A candidate process must collect sufficient votes to enter.

Processes in the intersection of two sets of voters ensure the safety property ME1 by casting their votes for only one candidate.

A voting set Vi associated with each process pi.

there is at least one common member of any two voting sets, the size of all voting set are the same size to be fair.

The optimal solution to minimizes K is K~sqrt(N) and M=K.

~ C *{Pl'P2,···,PN}*

such that for all i, j == 1,2, ... JV:

P . EV.

l *l*

*V.nV.°*0*

l *}*

V. l==K

I

Each process is contained in M of the voting set~

On initialization
stall' := RELEASED:
,·oted:=FALSE:
Forp, to enter the critical
section state := WANTED:
Multicast *request* to all processes in *V,:*
Hait until (number of replies received= J...):
state := HELD:
On receipt of a requestfront Pi at p1
if(state = HELD *or voted*= TRUE)
Ihm
queue *request* from *Pi* without
replying:
else
send *replv* to *Pi:*
,·oted: = TRUE:
end if

Forp, lo exit the critical section
stall' := RELEASED:
Multicast *re/east'* to all processes in V,:
011 receipt of a releasefrom Pi at p1
*if(*queue ofrequests is non-empty)
then
remove head of queue - from *Pk.·* say:
send *reply* to Pie:
,'oted :=
TRUE: *else*
rnted:=FALSE:
mdif

Fig: Maekawa's algorithm – part 1

Maekawa's algorithm:

ME1 is met. If two processes can enter CS at the same time, the processes in the intersection of two voting sets would have to vote for both. The algorithm will only allow a process to make at most one vote between successive receipts off a release message.

Deadlock prone. For example, p1, p2 and p3 with V1={p1,p2}, V2={p2, p3}, V3={p3,p1}.

If three processes concurrently request entry to the CS, then it is possible for p1 to reply to itself and hold off p2; for p2 rely to itself and hold off p3; for p3 to reply to itself and hold off p1. Each process has received one out of two replies, and none can proceed.

Bandwidth utilization is 2sqrt(N) messages per entry to CS and sqrt(N) per exit.

Client delay i s the sam e as Ric art and Agrawala's algori thm , one round -trip time.

Synchronization delay is one round-trip time.

Fault tolerance

What happens when messages are lost?

What happens when a process crashes?

None of the algorithm that we have described would tolerate the loss of messages if the channels were unreliable.

– The ring-based algorithm cannot tolerate any single process crash failure.

– Maekawa's algirithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required.

– The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token.

– The Ricart and Agrawala algorithm as we have described it can be adapted to tolerate the crash failure of such a process by taking it to grant all requests implicitly.

Topic 02 : Elections

Algorithm to choose a unique process to play a particular role is called an election algorithm. E.g. central server for mutual exclusion, one process will be elected as the server. Everybody must agree. If the server wishes to retire, then another election is required to choose a replacement.

Requirements:

– E1(safety): a participant pi has *elected*, = L *or elected*, = *P*:

Where P is chosen as the non-crashed process at the end of run with the largest identifier.

(concurrent elections possible.)

– E2(liveness): All processes Pi participate in election process and eventually set

Sub topic 2.1: A ring based election algorithm

All processes arranged in a logical ring.

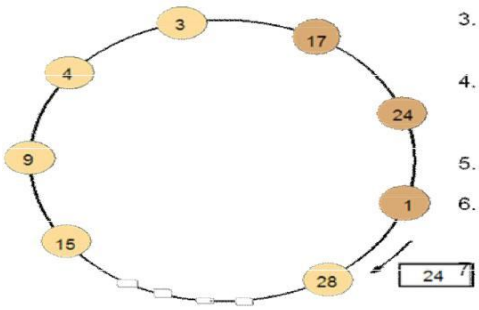
Each process has a communication channel to the next process.

All messages are sent clockwise around the ring.

Assume that no failures occur, and system is asynchronous.

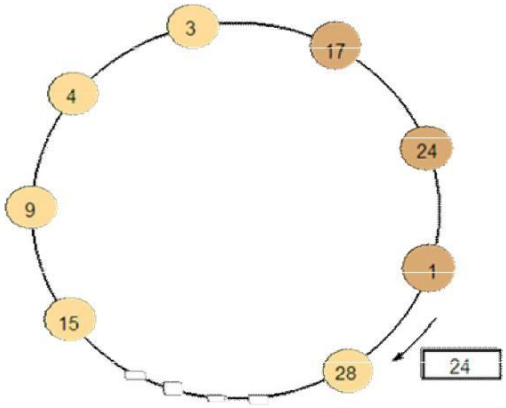
Goal is to elect a single process coordinator which has the largest identifier.

A ring-based election in progress



1. **Initially**, every process is marked as non-participant. Any process can begin an election.
2. The **starting** process marks itself as participant and place its identifier in a message to its neighbour.
3. A process receives a message and **compare** it with its own. If the arrived identifier is **larger**, it passes on the message.
4. If arrived identifier is **smaller** and receiver is not a participant, substitute its own identifier in the message and forward if. It does not forward the message if it is already a participant.
5. On forwarding of any case, the process marks itself as a participant.
6. If the received identifier is that of the receiver itself, then this process' s identifier must be the greatest, and it becomes the **coordinator**.
7. The coordinator marks itself as non-participant set elected_i and sends an **elected** message to its neighbour enclosing its ID.
8. When a process receives elected message, marks itself as a non-participant, sets its variable elected_i and forwards the message.

- Note: The election was started by process 17.
- The highest process identifier encountered so far is 24.
- Participant processes are shown darkened



- ⌘ E1 is met. All identifiers are compared, since a process must receive its own ID back before sending an elected message.
- ⌘ E2 is also met due to the guaranteed traversals of the ring.
- ⌘ Tolerate no failure makes ring algorithm of limited practical use.

- ⌘ If only a single process starts an election, the worst-performance case is then the anti-clockwise neighbour has the highest identifier. A total of N-1 messages is used to reach this neighbour. Then further N messages are required to announce its election. The elected message is sent N times. Making **3N-1 messages in all**.
- ⌘ **Turnaround time** is also 3N-1 sequential message transmission time

Sub Topic 2.2: The Bully Algorithm

Allows process to crash during an election, although it assumes the message delivery between processes is reliable.

Assume system is synchronous to use timeouts to detect a process failure.

Assume each process knows which processes have higher identifiers and that it can communicate with all such processes.

Three types of messages:

– **Election** is sent to announce an election message. A process begins an election

when it notices, through **timeouts**, that the coordinator has failed.

T=2Ttrans+Tprocess From the time of sending

– **Answer** is sent in response to an election message.

– **Coordinator** is sent to announce the identity of the elected process.

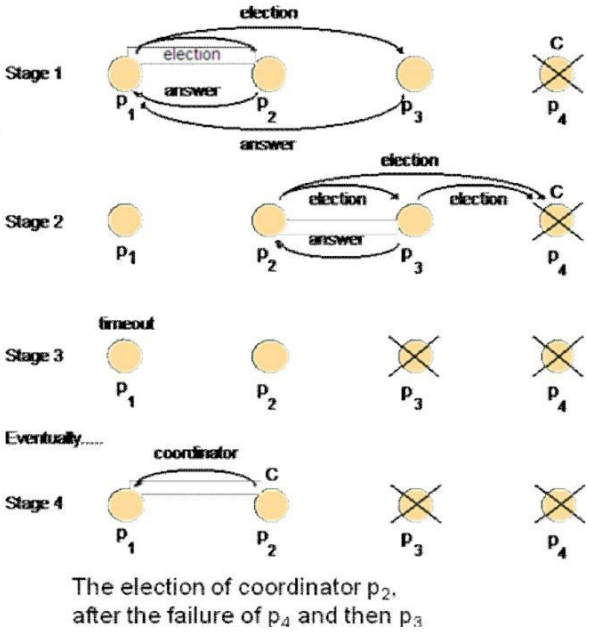
Best case the process with the second highest ID notices the coordinator's failure. Then

it can immediately elect itself and send N-2 coordinator messages.

The bully algorithm requires O(N^2) messages in the **worst case** - that is, when the process with the least ID first detects the coordinator's failure. For then N-1 processes altogether begin election, each sending messages to processes with higher ID.

1. The process begins a election by sending an election message to these processes that have a higher ID and awaits an answer in response. If none arrives within time T, the process considers itself the coordinator and sends coordinator message to all processes with lower identifiers. Otherwise, it waits a further time T' for coordinator message to arrive. If none, begins another election.
2. If a process receives a coordinator message, it sets its variable elected_i to be the coordinator ID.
3. If a process receives an election message, it send back an answer message and begins another election unless it has begun one already.

E1 may be broken if timeout is not accurate or replacement. (suppose P3 crashes and replaced by another process. P2 set P3 as coordinator and P1 set P2 as coordinator)
E2 is clearly met by the assumption of reliable transmission.



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

Figure: The bully Algorithm

Topic 03 : Multicast communication

IP multicast :

IP multicast – an implementation of group communication

– built on top of IP (note IP packets are addressed to computers)

– allows the sender to transmit a single IP packet to a set of computers that form a multicast group (a class D internet address with first 4 bits 1110)

– Dynamic membership of groups. Can send to a group with or without joining it

– To multicast, send a UDP datagram with a multicast address

– To join, make a socket join a group *(s.joinGroup*(*group*) enabling it to receive

messages to the group

Multicast routers

– Local messages use local multicast capability. Routers make it efficient by choosing other routers on the way.

Failure model

– Omission failures Þ some but not all members may receive a message. e.g.

a recipient may drop message, or a multicast router may fail

– IP packets may not arrive in sender order, group members can receive messages in different orders

Introduction to multicast:

Multicast communication requires coordination and agreement. The aim is for members of a group to receive copies of messages sent to the group

Many different delivery guarantees are possible

– e.g. agree on the set of messages received or on delivery ordering

A process can multicast by the use of a single operation instead of a send to each member

– For example in IP multicast *aSocket.send*(*aMessage*)

– The single operation allows for:

efficiency I.e. send once on each link, using hardware multicast when available, e.g. multicast from a computer in London to two in Beijing

delivery guarantees e.g. can't make a guarantee if multicast is implemented as multiple sends and the sender fails. Can also do ordering

System model:

The system consists of a collection of processes which can communicate *reliably* over

1- 1 channels

Processes fail only by crashing (no arbitrary failures)

Processes are members of groups - which are the destinations of multicast

messages In general process *p* can belong to more than one group Operations

– *multicast*(*g*, *m*) sends message *m* to all members of process group *g*

– *deliver* (*m*) is called to get a multicast message delivered. It is different from

receive as it may be delayed to allow for ordering or reliability.

Multicast message *m* carries the id of the sending process *sender*(*m*) and the id of the destination group *group*(*m*)

We assume there is no falsification of the origin and destination of messages.

Open and closed groups:

Closed groups

– only members can send to group, a member delivers to itself

– they are useful for coordination of groups of cooperating servers

Open

– they are useful for notification of events to groups of interested processes

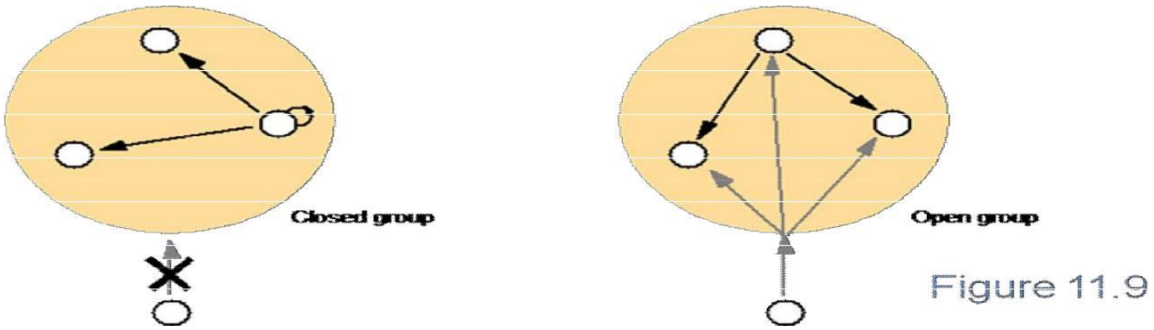


Figure 11.9

Reliability of one-to-one communication:

The term *reliable 1-1 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.

How do we achieve validity: *validity* - by use of acknowledgements and retries

How do we achieve integrity: *integrity*

by use checksums, reject duplicates (e.g. due to retries).

If allowing for malicious users, use security techniques

Sub Topic 3.1 : Basic multicast

A correct process will eventually deliver the message provided the **multicaster does not crash**

– note that IP multicast does not give this guarantee

The primitives are called *B-multicast* and *B-deliver*

A straightforward but ineffective method of implementation:

– use a reliable 1-1 *send* (i.e. with integrity and validity as above) To *B-multicast*(*g*,*m*)*:* for each process *p* e *g*, *send*(*p*, *m*);
On *receive* (*m)* at *p*: *B-deliver* (*m*) at *p*

Problem

– if the number of processes is large, the protocol will suffer from *ack-implosion*

What are ack-implosions: A practical implementation of Basic Multicast may be achieved over IP
multicast

Implementation of basic multicast over IP

Each process *p* maintains:

– a sequence number, S^p_g for each group it belongs to and

– R^q_g, the sequence number of the latest message it has delivered from process *q*

For process *p* to *B-multicast* a message *m* to group *g*

– it piggybacks Sg^p on the message *m*, using IP multicast to send it

– the piggybacked sequence numbers allow recipients to learn about messages they have not received

On *receive* (*g*,*m*, *S*) at *p*:

– if $S = R^q_g + 1$ *B-deliver* (*m*) and increment R^q_g by 1

– if $S < R^q_g + 1$ reject the message because it has been delivered before

– if $S > R^q_g + 1$ note that a message is missing, request missing message from sender. (will use a hold-back queue to be discussed later on)

If the sender crashes, then a message may be delivered to some members of the group but not others.

Sub topic 3.2: Reliable multicast

The protocol is correct even if the multicaster crashes

it satisfies criteria for *validity*, *integrity* and *agreement*

it provides operations *R-multicast* and *R-deliver*

Integrity - a correct process, *p* delivers *m* at most once.

Also *p* e *group*(*m*) and *m* was supplied to a multicast operation by *sender*(*m*)

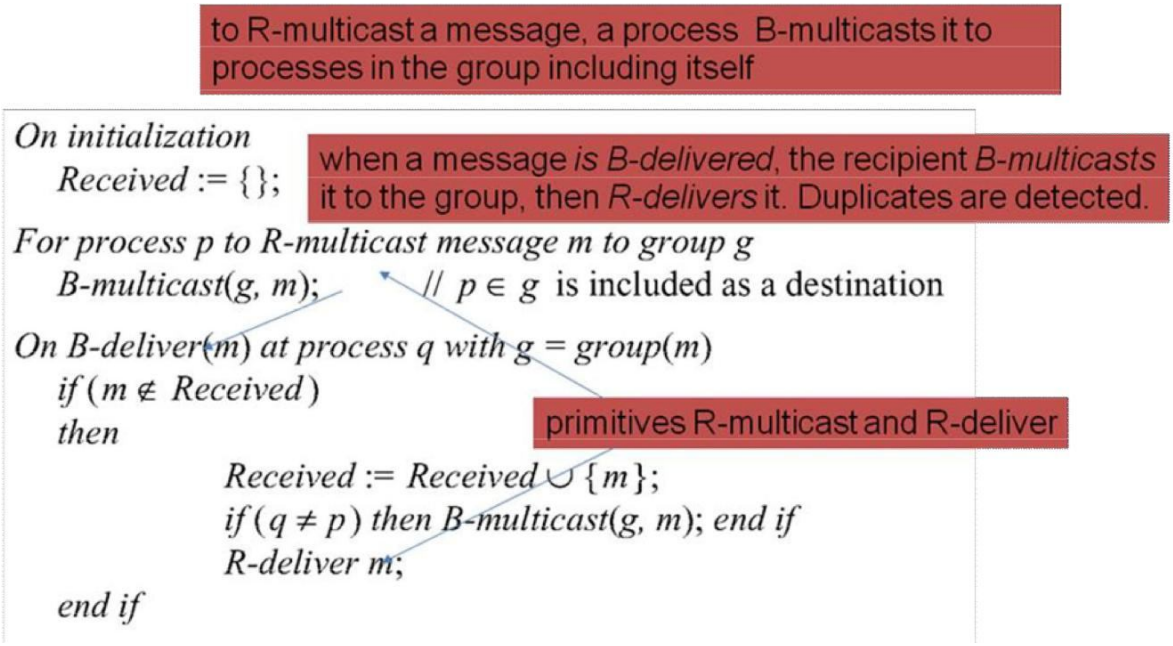
Validity - if a correct process multicasts *m*, it will eventually deliver *m*

Agreement - if a correct process delivers *m* then all correct processes in *group*(*m*) will eventually deliver *m*

Integrity as for 1-1 communication, validity - simplify by choosing sender as the one process , agreement - all or nothing - atomicity, even if multicaster

crashes. Reliable multicast algorithm over basic multicast

processes can belong to several closed groups



Validity - a correct process will B-deliver to itself,

Integrity - because the reliable 1-1 channels used for *B-multicast* guarantee integrity, Agreement - every correct process *B-multicasts* the message to the others. If *p* does not *R-deliver* then this is because it didn't *B-deliver* - because no others did either.

(Reliable multicast can be implemented efficiently over IP multicast by holding back messages until every member can receive them.)

Reliable multicast over IP multicast

This protocol assumes groups are closed. It uses:

– piggybacked acknowledgement messages

– negative acknowledgements when messages are missed

Process *p* maintains:

– S^p_g a message sequence number for each group it belongs to and

– R^q_g sequence number of latest message received from process *q* to *g*

(the piggybacked values in a message allow recipients to learn about messages they have not yet received)

For process *p* to *R-multicast* message *m* to group *g*

– piggyback S_g^p

and +ve acks for messages received in the form $<q, R^q_g>$

– IP multicasts the message to *g*, increments S^p_g by 1

A process on receipt by of a message to *g* with *S* from *p*.

– Iff $S=R^p_g+1$ *R-deliver* the message and increment R^p_g by 1

– If $S \leq R^p_g$ discard the message

– If $S > R^p_g + 1$ or if $R<R^q_g$ (for enclosed ack $<q,R>$)

then it has missed messages and requests them with negative acknowledgements

puts new message in hold-back queue for later delivery

The hold-back queue for arriving multicast messages

The hold back queue is not necessary for reliability as in the implementation using IP muilticast, but it simplifies the protocol, allowing sequence numbers to represent sets of messages. Hold-back queues are also used for ordering protocols.

Figure 11.11

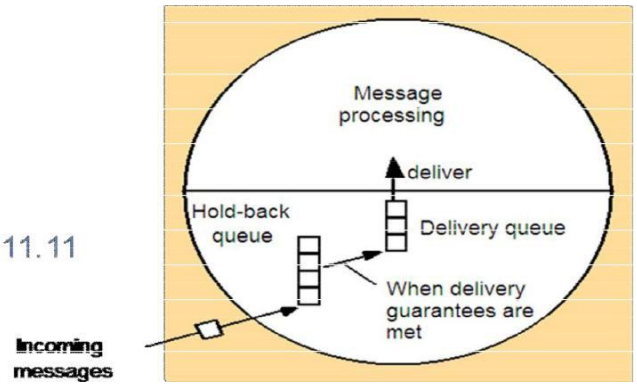


Figure: The hold-back queue for arriving multicast messages

Reliability properties of reliable multicast over IP

Integrity - duplicate messages detected and rejected.

IP multicast uses checksums to reject corrupt messages

Validity - due to IP multicast in which sender delivers to itself

Agreement - processes can detect missing messages. They must keep copies of messages they have delivered so that they can re-transmit them to others.

discarding of copies of messages that are no longer needed :

– when piggybacked acknowledgements arrive, note which processes have received messages. When all processes in *g* have the message, discard it.

– problem of a process that stops sending - use 'heartbeat' messages.

This protocol has been implemented in a practical way in Psynch and Trans

Sub topic 3.3: Ordered multicast

The basic multicast algorithm delivers messages to processes in an arbitrary order.

A variety of orderings may be implemented:

FIFO ordering

– If a correct process issues *multicast*(*g*, *m*) and then *multicast*(*g*,*m'*), then every correct process that delivers *m'* will deliver *m* before *m'* .

Causal ordering

– If *multicast*(*g*, *m*) ® *multicast*(*g*,*m'*), where ® is the happened-before relation between messages in group *g*, then any correct process that delivers *m'* will deliver *m* before *m'* .

Total ordering

– If a correct process delivers message *m* before it delivers *m'*, then any other correct process that delivers *m'* will deliver *m* before *m'*.

Ordering is expensive in delivery latency and bandwidth consumption

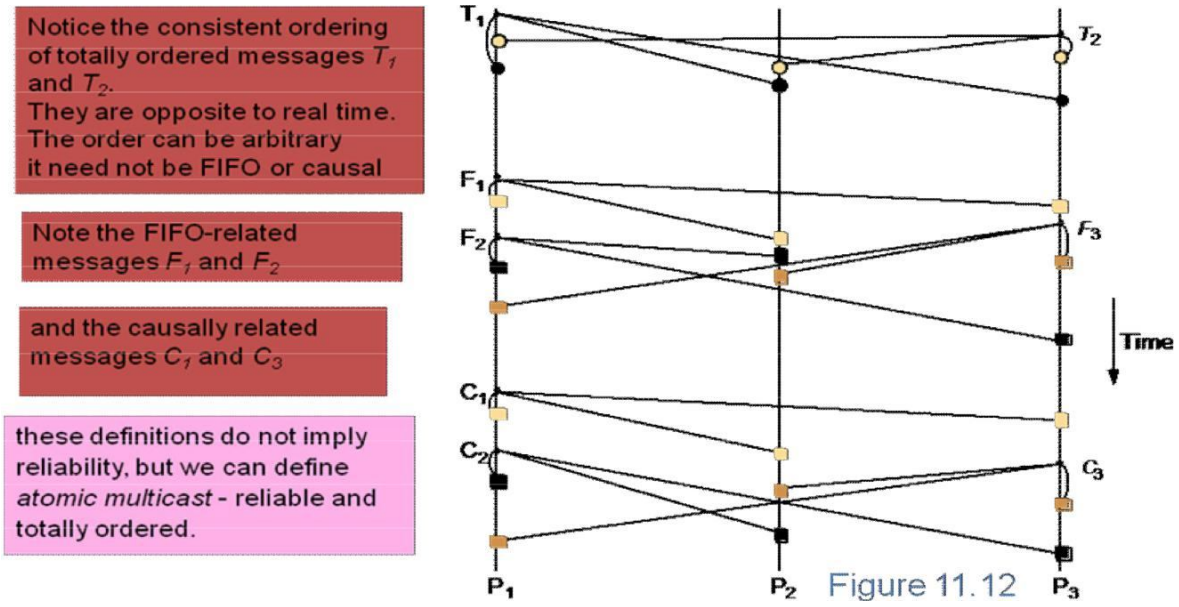


Figure 11.12

Ordered multicast delivery is expensive in bandwidth and latency. Therefore the less expensive orderings (e.g. FIFO or causal) are chosen for applications for which they are suitable

Figure: Total, FIFO and causal ordering of multicast messages

Display from a bulletin board program

Users run bulletin board applications which multicast messages

One multicast group per topic (e.g. *os.interesting*)

Require reliable multicast - so that all members receive messages

Ordering:

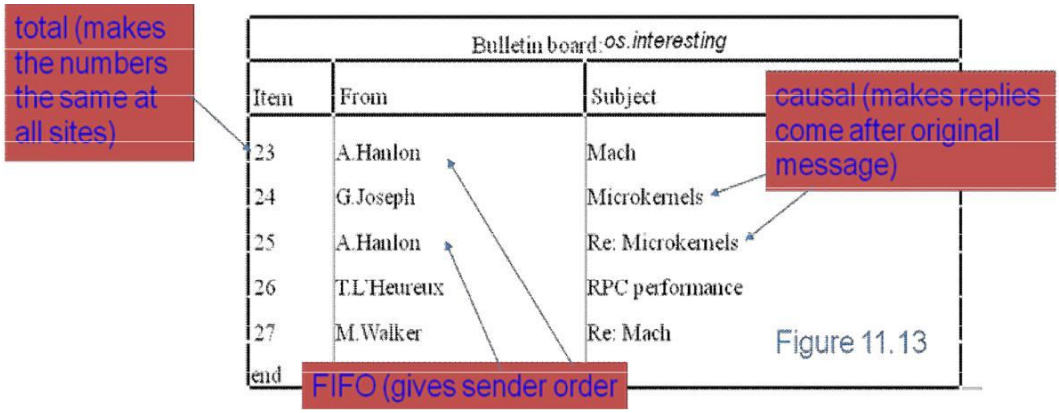


Figure: Display from a bulletin board program

Implementation of FIFO ordering over basic multicast

We discuss FIFO ordered multicast with operations *FO-multicast* and *FO-deliver* for non-overlapping groups. It can be implemented on top of any basic multicast

Each process p holds:

- S^p_g a count of messages sent by *p* to *g* and
- R^q_g the sequence number of the latest message to *g* that *p* delivered from *q*

For *p* to *FO-multicast* a message to *g*, it piggybacks S^p_g on the message, *B-multicasts* it and increments S^p_g by 1

On receipt of a message from *q* with sequence number *S*, *p* checks whether $S = R^q_g + 1$. If so, it *FO-delivers* it.

if $S > R^q_g + 1$ then *p* places message in hold-back queue until intervening messages have

been delivered. (note that *B-multicast* does eventually deliver messages unless the sender crashes)

Implementation of totally ordered multicast

The general approach is to attach *totally ordered identifiers* to multicast messages

- each receiving process makes ordering decisions based on the identifiers
- similar to the FIFO algorithm, but processes keep group specific sequence numbers
- operations *TO-multicast* and *TO-deliver*

we present two approaches to implementing total ordered multicast over basic multicast

- using a sequencer (only for non-overlapping groups)
- the processes in a group collectively agree on a sequence number for each message.

Total ordering using a sequencer

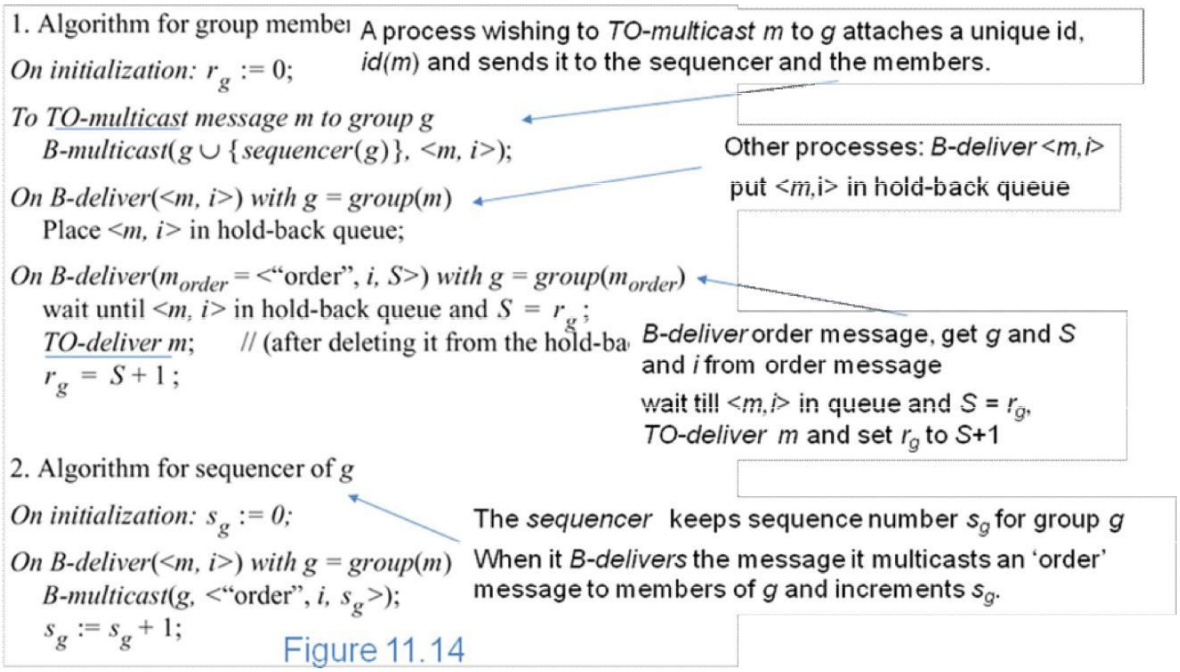


Figure 11.14

Discussion of sequencer protocol

Since sequence numbers are defined by a sequencer, we have total ordering.

Like B-multicast, if the sender does not crash, all members receive the message

Kaashoek's protocol uses hardware-based multicast The sender transmits one message to sequencer, then the sequencer multicasts the sequence number and the message but IP multicast is not as reliable as B-multicast so the sequencer stores messages in its history buffer for retransmission on request members notice messages are missing by inspecting sequence numbers

What can the sequencer do about its history buffer becoming full?

Sol: Members piggyback on their messages the latest sequence number they have seen

What happens when some member stops multicasting?

Sol: Members that do not multicast send heartbeat messages (with a sequence number)

<u>The ISIS algorithm for total ordering</u>

this protocol is for open or closed groups

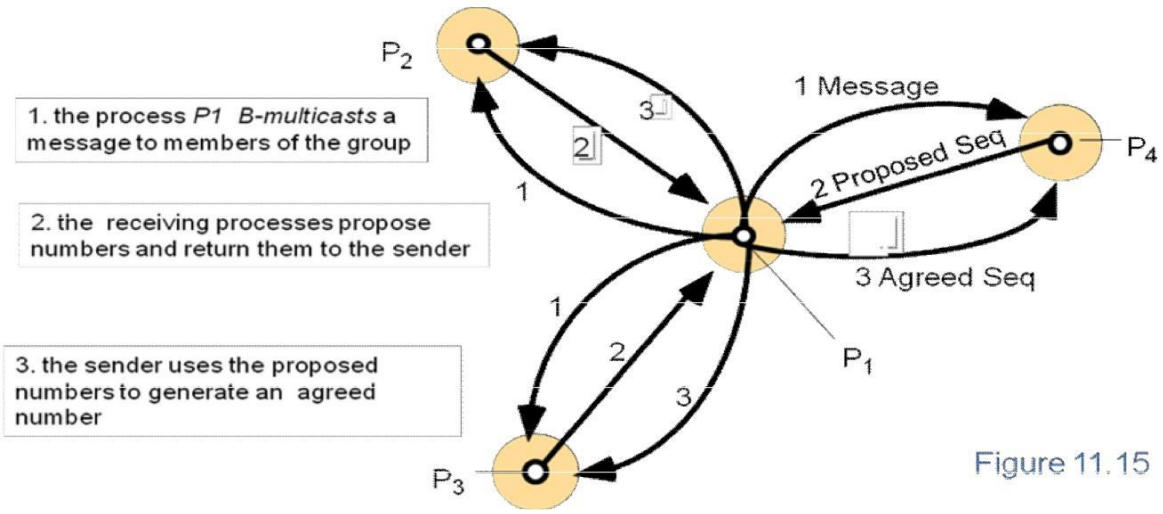


Figure 11.15

ISIS total ordering - agreement of sequence numbers

Each process, q keeps:

- A^q_g the largest agreed sequence number it has seen and
- P^q_g its own largest proposed sequence number

1. Process *p 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

– $P^q_g := \text{Max}(A^q_g) + 1$.

– assigns the proposed sequence number to the message and places it in its hold-back queue

3. *p* collects all the proposed sequence numbers and selects the largest as the next agreed sequence number, *a*. It *B-multicasts* <*i*, *a*> to *g*. Recipients set $A^q_g := \text{Max}(A^q_g, a)$, attach *a* to the message and re-order hold-back queue.

Discussion of ordering in ISIS protocol

Hold-back queue

ordered with the message with the smallest sequence number at the front of the queue when the agreed number is added to a message, the queue is re-ordered

when the message at the front has an agreed id, it is transferred to the delivery queue

– even if agreed, those not at the front of the queue are not transferred every

process agrees on the same order and delivers messages in that order, therefore

we have total ordering.

Latency

– 3 messages are sent in sequence, therefore it has a higher latency than sequencer method
– this ordering may not be causal or FIFO Causally ordered multicast

We present an algorithm of Birman 1991 for causally ordered multicast in non-overlapping, closed groups. It uses the *happened before* relation (on multicast messages only)

– that is, ordering imposed by one-to-one messages is not taken into account

It uses vector timestamps - that count the number of multicast messages from each process that happened before the next message to be multicast

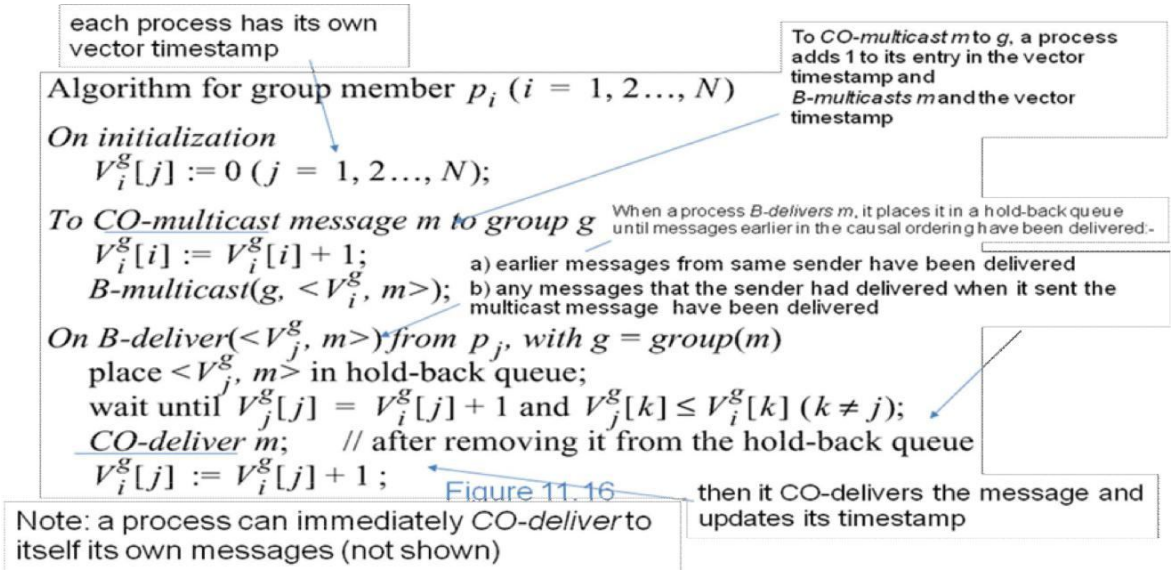


Figure: Causal ordering using vector timestamps

Comments

after delivering a message from *pj*, process *pi* updates its vector timestamp

– by adding 1 to the *j*th element of its timestamp

compare the vector clock rule where

Vi[*j*] := max(*Vi*[*j*], *t*[*j*]) for *j*=1, 2, ...N

– in this algorithm we know that only the *j*th element will increase

for an outline of the proof see page 449

if we use *R-multicast* instead of *B-multicast* then the protocol is reliable as well as causally ordered.

If we combine it with the sequencer algorithm we get total and causal ordering

Comments on multicast protocols

we need to have protocols for overlapping groups because applications do need to subscribe to several groups

definitions of 'global FIFO ordering' etc on page 450 and some references to papers on them

multicast in synchronous and asynchronous systems

– all of our algorithms do work in both

reliable and totally ordered multicast

– can be implemented in a synchronous system

– but is impossible in an asynchronous system (reasons discussed in consensus section - paper by Fischer et al.)

Transactions and replications:

Introduction to replication

Replication of data: - the maintenance of copies of data at multiple computers

performance enhancement

– e.g. several web servers can have the same DNS name and the servers are selected in turn. To share the load.

– replication of read-only data is simple, but replication of changing data has overheads

fault-tolerant service

– guarantees correct behaviour in spite of certain faults (can include timeliness)

– if f of f+1 servers crash then 1 remains to supply the service

– if f of 2f+1 servers have byzantine faults then they can supply a correct service

availability is hindered by

– server failures

Replicate data at failure- independent servers and when one fails, client may use another. Note that caches do not help with availability(they are incomplete).

– network partitions and disconnected operation

Users of mobile computers deliberately disconnect, and then on re-connection, resolve conflicts

e.g. : a user on a train with a laptop with no access to a network will prepare by copying data to the laptop, e.g. a shared diary. If they update the diary they risk missing updates by other people.

Requirements for replicated data

Replication transparency

– clients see logical objects (not several physical copies)

they access one logical item and receive a single result

Consistency

– specified to suit the application,

e.g. when a user of a diary disconnects, their local copy may be inconsistent with the others and will need to be reconciled when they connect again. But connected clients using different copies should get consistent results. These issues are addressed in Bayou and Coda.

Topic 02: System model:

each *logical* object is implemented by a collection of *physical* copies called *replicas*

– the replicas are not necessarily consistent all the time (some may have received updates, not yet conveyed to the others)

we assume an asynchronous system where processes fail only by crashing and generally assume no network partitions

replica managers

– an RM contains replicas on a computer and access them directly

– RMs apply operations to replicas recoverably

i.e. they do not leave inconsistent results if they crash

– objects are copied at all RMs unless we state otherwise

– static systems are based on a fixed set of RMs

– in a dynamic system: RMs may join or leave (e.g. when they crash)

– an RM can be a *state machine*, which has the following properties:

State machine

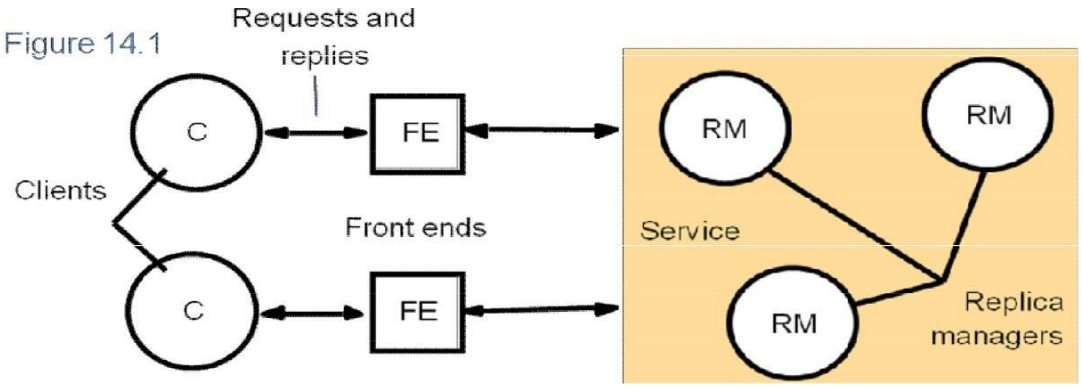
applies operations atomically

its state is a deterministic function of its initial state and the operations applied

all replicas start identical and carry out the same operations

Its operations must not be affected by clock readings etc.

A basic architectural model for the management of replicated data



A collection of RMs provides a service to clients

Clients see a service that gives them access to logical objects, which are in fact replicated at the RMs

Clients request operations: those without updates are called *read-only* requests the others are called *update* requests (they may include reads)

Clients request are handled by front ends. A front end makes replication transparent. Five phases in performing a request (What can the FE hide from a client?) issue request

– the FE either

sends the request to a single RM that passes it on to the others

or multicasts the request to all of the RMs (in state machine approach)

coordination

– the RMs decide whether to apply the request; and decide on its ordering relative to other requests (according to FIFO, causal or total ordering)

execution

– the RMs execute the request (sometimes tentatively)

agreement

– RMs *agree* on the effect of the request, .e.g perform 'lazily' or immediately

response

– one or more RMs reply to FE. e.g.

for high availability give first response to client.

to tolerate byzantine faults, take a vote

FIFO ordering: if a FE issues r then *r*', then any correct RM handles *r* before *r*' Causal ordering: if r ® *r*', then any correct RM handles *r* before *r*' Total ordering: if a correct RM handles r before *r*', then any correct RM handles *r* before *r*' Bayou sometimes executes responses tentatively so as to be able to reorder them

RMs agree - I.e. reach a consensus as to effect of the request. In Gossip, all RMs eventually receive updates.

Topic 03: Group Communication

We require a membership service to allow dynamic membership of groups

process groups are useful for managing replicated data

– but replication systems need to be able to add/remove RMs

group membership service provides:

– interface for adding/removing members

create, destroy process groups, add/remove members. A process can generally belong to several groups.

– implements a failure detector (section 11.1 - not studied in this course)

which monitors members for failures (crashes/communication),

and excludes them when unreachable

– notifies members of changes in membership

– expands group addresses

multicasts addressed to group identifiers,

coordinates delivery when membership is changing

e.g. IP multicast allows members to join/leave and performs address expansion, but not the other features

Services provided for process groups.

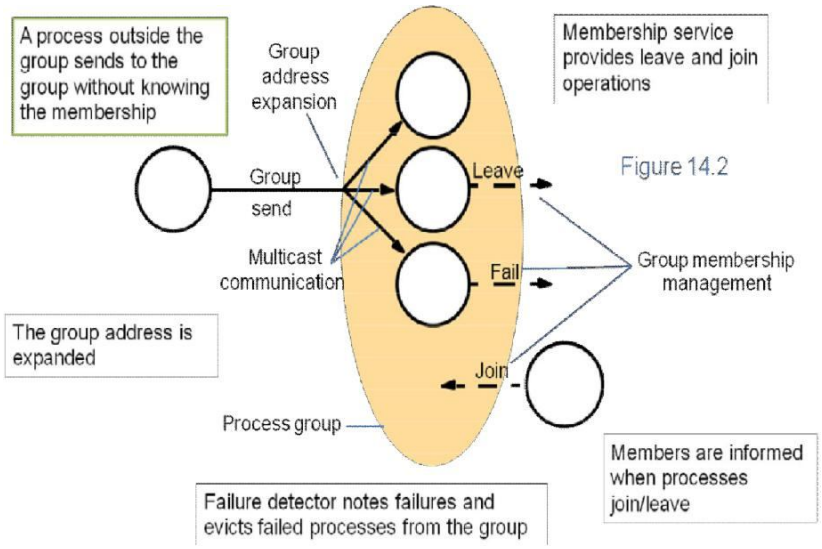


Figure 14.2

We will leave out the details of view delivery and view synchronous group communication

A full membership service maintains *group views*, which are lists of group members, ordered e.g. as members join group.

A new group view is generated each time a process joins or leaves the group.

View delivery p 561. The idea is that processes can 'deliver views' (like delivering multicast messages).

– ideally we would like all processes to get the same information in the same order relative to the messages.

view synchronous group communication (p562) with reliability.

– Illustrated in Fig below

- all processes agree on the ordering of messages and membership changes,
- a joining process can safely get state from another member.
- or if one crashes, another will know which operations it had already performed
- This work was done in the ISIS system (Birman)

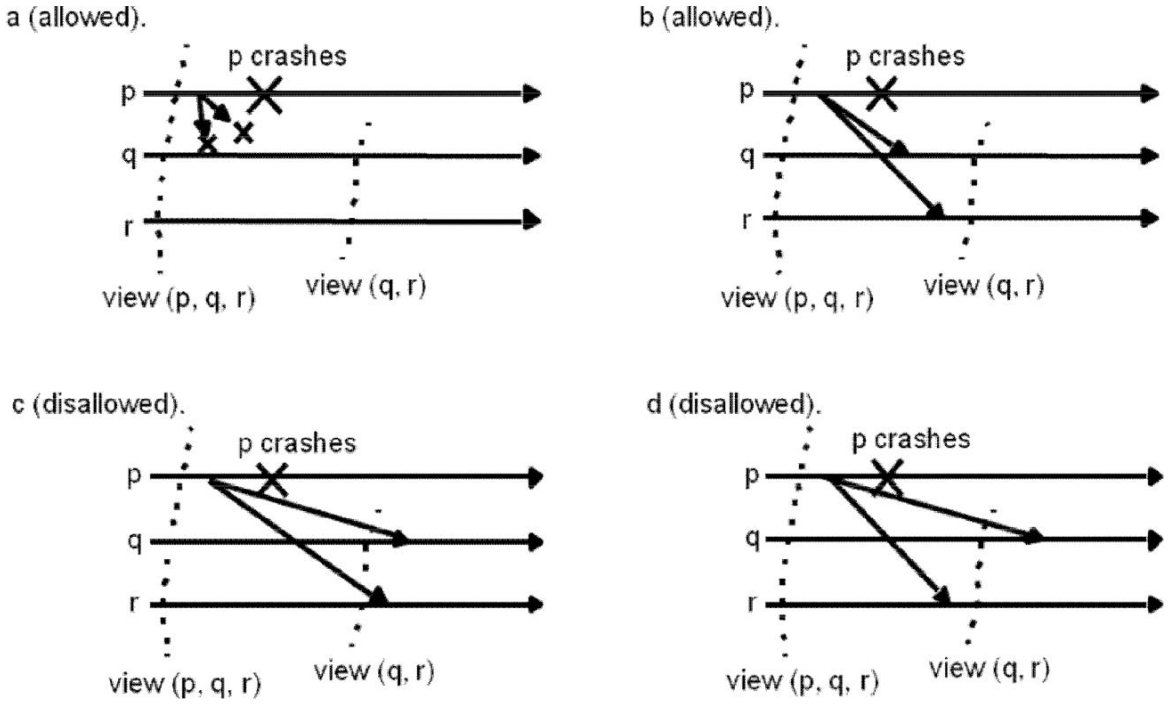


Figure : View-synchronous group communication

Topic 04: Distributed transactions – introduction

a *distributed transaction* refers to a flat or nested transaction that accesses objects managed by multiple servers

When a distributed transaction comes to an end

- the either all of the servers commit the transaction
- or all of them abort the transaction.

one of the servers is *coordinator*, it must ensure the same outcome at all of the servers.

the 'two-phase commit protocol' is the most commonly used protocol for achieving this

Concurrency control in distributed transactions

Each server manages a set of objects and is responsible for ensuring that they remain consistent when accessed by concurrent transactions

- therefore, each server is responsible for applying concurrency control to its own objects.
- the members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner

– therefore if transaction *T* is before transaction *U* in their conflicting access to objects at one of the servers then they must be in that order at all of the servers whose objects are accessed in a conflicting manner by both *T* and *U*

Sub Topic 4.1 : Locking

In a distributed transaction, the locks on an object are held by the server that manages it.

– The local lock manager decides whether to grant a lock or make the requesting transaction wait.

– it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.

– the objects remain locked and are unavailable for other transactions during the atomic commit protocol

an aborted transaction releases its locks after phase 1 of the protocol. Interleaving of transactions T and U at servers X and Y in the example on page 529, we have

– *T* before *U* at server X and *U* before *T* at server Y

different orderings lead to cyclic dependencies and distributed deadlock

– detection and resolution of distributed deadlock in next section

	T			*U*	
Write(A)	at *X*	locks *A*			
			Write(B)	at *Y*	locks *B*
Read(B)	at *Y*	waits for *U*			
			Read(A)	at *X*	waits for T

Sub topic 4.2 :Timestamp ordering concurrency control

Single server transactions

– coordinator issues a unique timestamp to each transaction before it starts

– serial equivalence ensured by committing objects in order of timestamps

Distributed transactions

– the first coordinator accessed by a transaction issues a globally unique timestamp

– as before the timestamp is passed with each object access

– the servers are jointly responsible for ensuring serial equivalence

that is if T access an object before U, then T is before U at all objects

– coordinators agree on timestamp ordering

a timestamp consists of a pair <*local timestamp, server-id*>.

the agreed ordering of pairs of timestamps is based on a comparison in

which the server-id part is less significant – they should relate to time

The same ordering can be achieved at all servers even if their clocks are not synchronized

– for efficiency it is better if local clocks are roughly synchronized

– then the ordering of transactions corresponds roughly to the real time order in which they were started

Timestamp ordering

– conflicts are resolved as each operation is performed

– if this leads to an abort, the coordinator will be informed

it will abort the transaction at the participants

– any transaction that reaches the client request to commit should always be able to do so

participant will normally vote *yes*

unless it has crashed and recovered during the transaction Optimistic concurrency control

Use backward validation

1. write/read, 2. read/write, 3. write/write

- each transaction is validated before it is allowed to commit
  - transaction numbers assigned at start of validation
  - transactions serialized according to transaction numbers
  - validation takes place in phase 1 of 2PC protocol

1. satisfied
2. checked
3. paralllel

- consider the following interleavings of *T* and *U*
  - *T* before *U* at *X* and *U* before *T* at *Y*

Suppose T & U start validation at about the same time

T		*U*	
Read(A)	at *X*	*Read(B)*	at *Y*
Write(A)		*Write(B)*	
Read(B)	at *Y*	*Read(A)*	at *X*
Write(B)		*Write(A)*	

X does T first
Y does U first

No parallel Validation –. commitment deadlock

Commitment deadlock in optimistic concurrency control

servers of distributed transactions do parallel validation

– therefore rule 3 must be validated as well as rule 2

the write set of *Tv* is checked for overlaps with write sets of earlier transactions

– this prevents commitment deadlock

– it also avoids delaying the 2PC protocol

another problem - independent servers may schedule transactions in different orders

– e.g. T before U at X and U before T at Y
– this must be prevented - some hints as to how on page 531

Topic 05: Distributed deadlocks

Single server transactions can experience deadlocks

– prevent or detect and resolve
– use of timeouts is clumsy, detection is preferable.

it uses wait-for graphs.

Distributed transactions lead to distributed deadlocks

– in theory can construct global wait-for graph from local ones
– a cycle in a global wait-for graph that is not in local ones is a distributed deadlock

sub topic 5.1: **Interleavings of transactions *U, V* and *W***

- objects *A, B* managed by *X* and *Y* ; *C* and *D* by *Z*
  - next slide has global wait-for graph

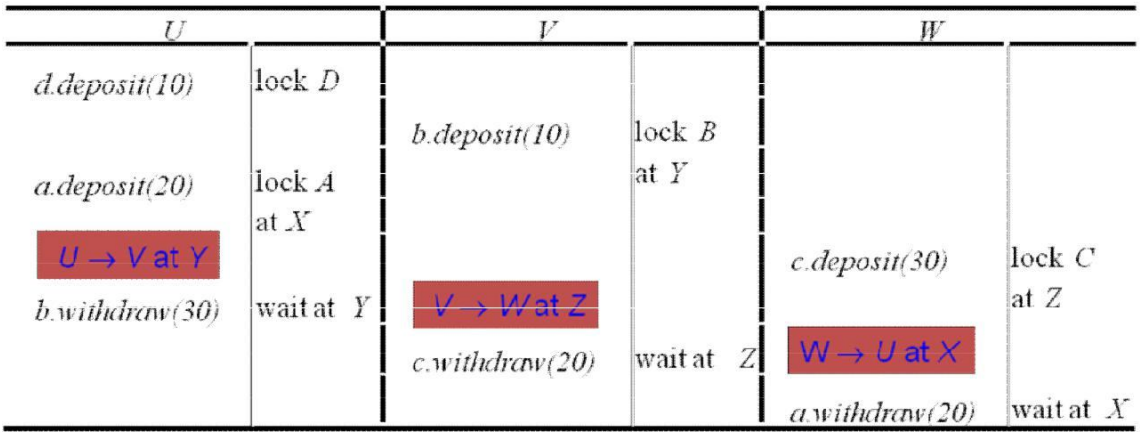


Figure: Interleavings of transactions *U, V* and *W*

Sub topic 5.2 Distributed deadlock

Deadlock detection - local wait-for graphs

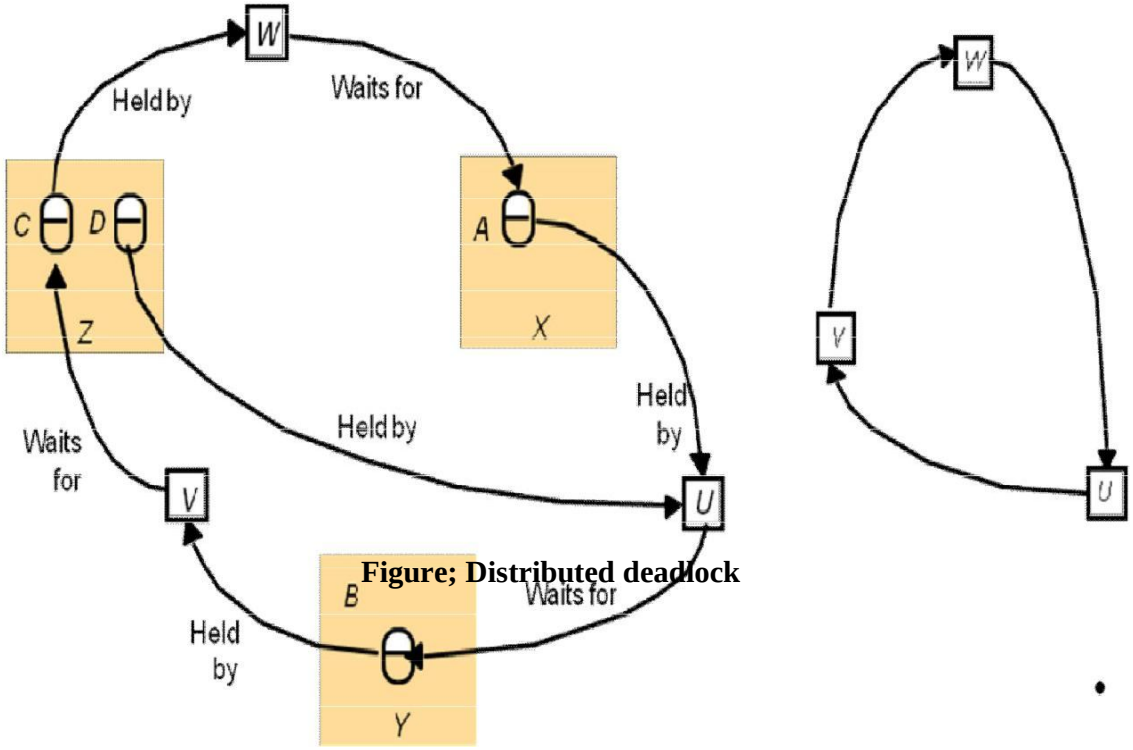
Local wait-for graphs can be built, e.g.

– *server Y: U ® V added when U requests b.withdraw(30)*
– *server Z: V ® W added when V requests c.withdraw(20)*
– *server X: W ® U added when W requests a.withdraw(20)*

to find a global cycle, communication between the servers is needed centralized deadlock detection

- *one server takes on role of global deadlock detector*
- *the other servers send it their local graphs from time to time*
- *it detects deadlocks, makes decisions about which transactions to abort and informs the other servers*
- *usual problems of a centralized service - poor availability, lack of fault tolerance and no ability to scale*

- a deadlock cycle has alternate edges showing wait-for and held-by
- wait-for added in order: $U \rightarrow V$ at Y; $V \rightarrow W$ at Z and $W \rightarrow U$ at X



Figure; Distributed deadlock

Subtopic 5.3: Local and global wait-for graphs

Phantom deadlocks

- a 'deadlock' that is detected, but is not really one
- happens when there appears to be a cycle, but one of the transactions has released a lock, due to time lags in distributing graphs
- in the figure suppose U releases the object at X then waits for V at Y and the global detector gets Y's graph before X's (T ® U ® V ® T)

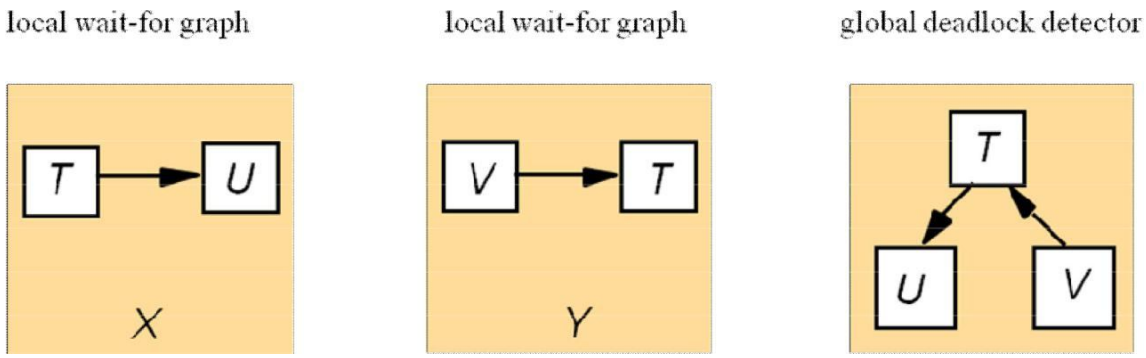


Figure: Local and global wait-for graphs

Edge chasing - a distributed approach to deadlock detection

a global graph is not constructed, but each server knows about some of the edges

– servers try to find cycles by sending *probes* which follow the edges of the graph through the distributed system

– when should a server send a probe (go back to Fig 13.13)

– edges were added in order *U* ® *V* at *Y; V* ® *W* at Z and *W* ® *U* at *X*

when *W* ® *U* at *X* was added, U was waiting, but

when *V* ® *W* at *Z*, *W* was not waiting

– send a probe when an edge *T*1 ® *T*2 when *T*2 is waiting

– each coordinator records whether its transactions are active or waiting

the local lock manager tells coordinators if transactions start/stop waiting

when a transaction is aborted to break a deadlock, the coordinator tells the participants, locks are removed and edges taken from wait-for graphs

Edge-chasing algorithms

Three steps

– Initiation:

When a server notes that T starts waiting for U, where U is waiting at another server, it initiates detection by sending a probe containing the edge < T ® U > to the server where U is blocked.

If U is sharing a lock, probes are sent to all the holders of the lock.

– Detection:

Detection consists of receiving probes and deciding whether deadlock has occurred and whether to forward the probes.

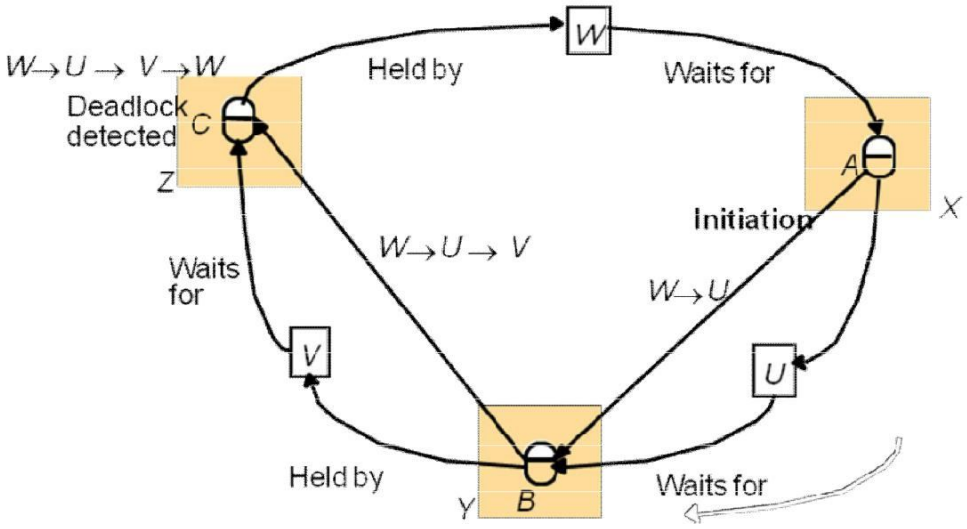
- e.g. when server receives probe < T ® U > it checks if U is waiting, e.g. U ® V, if so it forwards < T ® U ® V > to server where V waits
- when a server adds a new edge, it checks whether a cycle is there

– Resolution:

When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

Probes transmitted to detect deadlock

- example of edge chasing starts with *X* sending <*W* → *U*>, then *Y* sends <*W* → *U* → *V* >, then *Z* sends <*W* → *U* → *V* → *W*>



Edge chasing conclusion

probe to detect a cycle with *N* transactions will require 2(*N*-1) messages.

– Studies of databases show that the average deadlock involves 2 transactions.

the above algorithm detects deadlock provided that

– waiting transactions do not abort

– no process crashes, no lost messages

– to be realistic it would need to allow for the above failures

refinements of the algorithm

– to avoid more than one transaction causing detection to start and then more than one being aborted

– not time to study these now

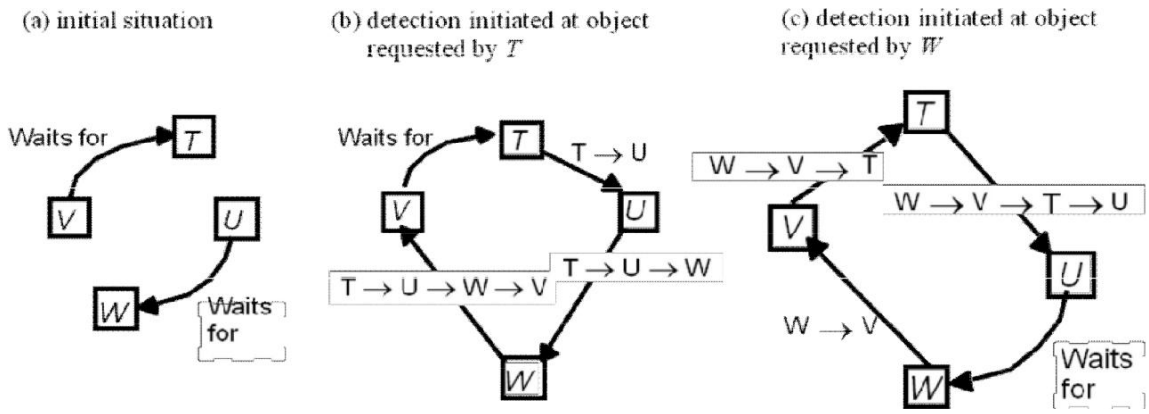


Figure: Two probes initiated

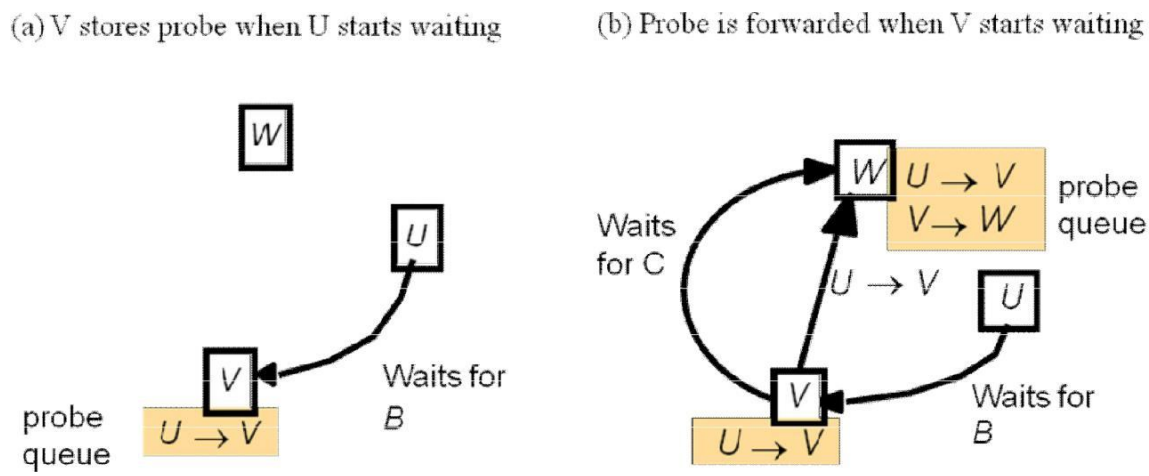


Figure: Probes travel downhill

Topic 06: Transaction recovery

Atomicity property of transactions

- durability and failure atomicity
- durability requires that objects are saved in permanent storage and will be available indefinitely
- failure atomicity requires that effects of transactions are atomic even when the server crashes

Recovery is concerned with

- ensuring that a server's objects are durable and
- that the service provides failure atomicity.
- for simplicity we assume that when a server is running, all of its objects are in volatile memory
- and all of its committed objects are in a *recovery file* in permanent storage
- recovery consists of restoring the server with the latest committed versions of all of its objects from its recovery file

Recovery manager

The task of the Recovery Manager (RM) is:

– to save objects in permanent storage (in a recovery file) for committed transactions;

– to restore the server's objects after a crash;

– to reorganize the recovery file to improve the performance of recovery;

– to reclaim storage space (in the recovery file).

media failures

– i.e. disk failures affecting the recovery file

– need another copy of the recovery file on an independent disk. e.g. implemented as stable storage or using mirrored disks

Recovery - intentions lists

Each server records an intentions list for each of its currently active transactions

– an intentions list contains a list of the object references and the values of all the objects that are altered by a transaction

– when a transaction commits, the intentions list is used to identify the objects affected

the committed version of each object is replaced by the tentative

one the new value is written to the server's recovery file

– in 2PC, when a participant says it is ready to commit, its RM must record its intentions list and its objects in the recovery file

it will be able to commit later on even if it crashes

when a client has been told a transaction has committed, the recovery files of all participating servers must show that the transaction is committed,

- even if they crash between *prepare* to commit and *commit*

Types of entry in a recovery file

Type of entry	*Description of contents of entry*
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared*, *committed* *aborted*) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <identifier of object>, <position in recovery file of value of object>.

Figure 13.18

Object state flattened to bytes

first entry says *prepared*

- For distributed transactions we need information relating to the 2PC as well as object values, that is:
  - transaction status (committed, prepared or aborted)
  - intentions list

Note that the objects need not be next to one another in the recovery file

Logging - a technique for the recovery file

the recovery file represents a log of the history of all the transactions at a server

– it includes objects, intentions lists and transaction status
– in the order that transactions prepared, committed and aborted
– a recent snapshot + a history of transactions after the snapshot
– during normal operation the RM is called whenever a transaction prepares, commits or aborts

prepare - RM appends to recovery file all the objects in the intentions list followed by status (prepared) and the intentions list

commit/abort - RM appends to recovery file the corresponding status

assume *append* operation is atomic, if server fails only the last write will be incomplete

to make efficient use of disk, buffer *writes*. Note: sequential *writes* are more efficient than those to random locations

committed status is forced to the log - in case server crashes

Log for banking service

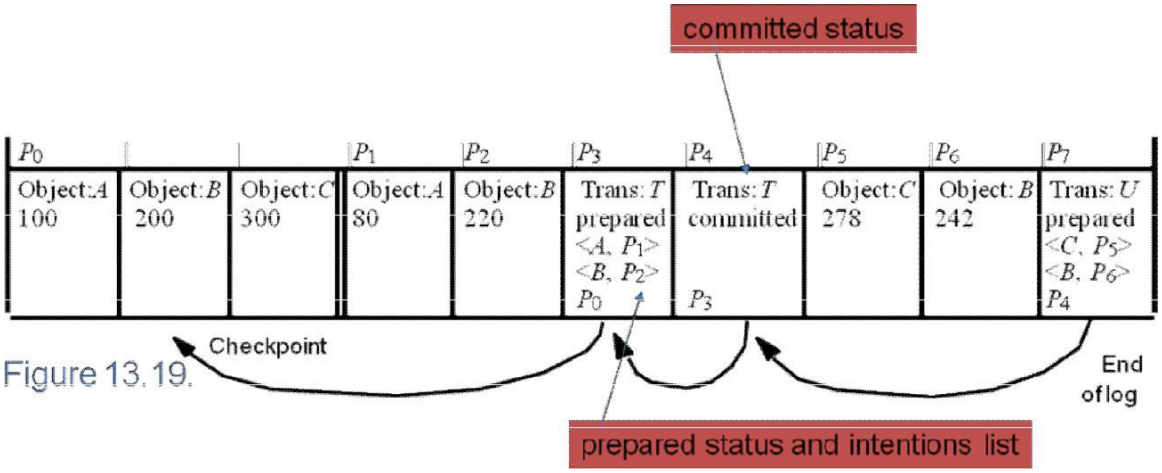


Figure 13.19.

Logging mechanism (there would really be other objects in log file)

– initial balances of *A*, *B* and *C* $100, $200, $300
– *T* sets *A* and *B* to $80 and $220. *U* sets *B* and *C* to $242 and $278
– entries to left of line represent a snapshot (checkpoint) of values of *A*, *B* and *C*

before *T* started. *T* has committed, but *U* is prepared.

– the RM gives each object a unique identifier (*A*, *B*, *C* in diagram)
– each status entry contains a pointer to the previous status entry, then the checkpoint can follow transactions backwards through the file

Recovery of objects - with logging

When a server is replaced after a crash

– it first sets default initial values for its objects

– and then hands over to its recovery manager.

The RM restores the server's objects to include

– all the effects of all the committed transactions in the correct order and

– none of the effects of incomplete or aborted transactions

– it 'reads the recovery file backwards' (by following the pointers).

restores values of objects with values from committed transactions

continuing until all of the objects have been restored

– if it started at the beginning, there would generally be more work to do

– to recover the effects of a transaction use the intentions list to find the value of the objects

e.g. look at previous slide (assuming the server crashed before T committed)

– the recovery procedure must be idempotent

Logging - reorganising the recovery file

RM is responsible for reorganizing its recovery file

– so as to make the process of recovery faster and

– to reduce its use of space

checkpointing

– the process of writing the following to a new recovery file

the current committed values of a server's objects,

transaction status entries and intentions lists of transactions that have not yet been fully resolved

including information related to the two-phase commit protocol (see later)

– checkpointing makes recovery faster and saves disk space

done after recovery and from time to time

can use old recovery file until new one is ready, add a 'mark' to old file do as above and then copy items after the mark to new recovery file replace old recovery file by new recovery file

Map at start	*Map when T commits*
$A \rightarrow P_0$	$A \rightarrow P_1$
$B \rightarrow P_0'$	$B \rightarrow P_2$
$C \rightarrow P_0''$	$C \rightarrow P_0''$

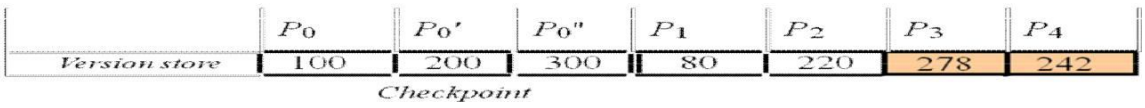


Figure: Shadow versions

Recovery of the two-phase commit protocol

The above recovery scheme is extended to deal with transactions doing the 2PC protocol when a server fails

it uses new transaction status values *done, uncertain*

the coordinator uses *committed* when result is *Yes;*

done when 2PC complete (if a transaction is *done* its information may be removed when reorganising the recovery file)

the participant uses *uncertain* when it has voted *Yes; committed* when told the result (*uncertain* entries must not be removed from recovery file)

It also requires two additional types of entry:

Type of entry	*Description of contents of entry*
Coordinator	Transaction identifier, list of participants added by RM when coordinator prepared
Participant	Transaction identifier, coordinator added by RM when participant votes yes

Log with entries relating to two-phase commit protocol

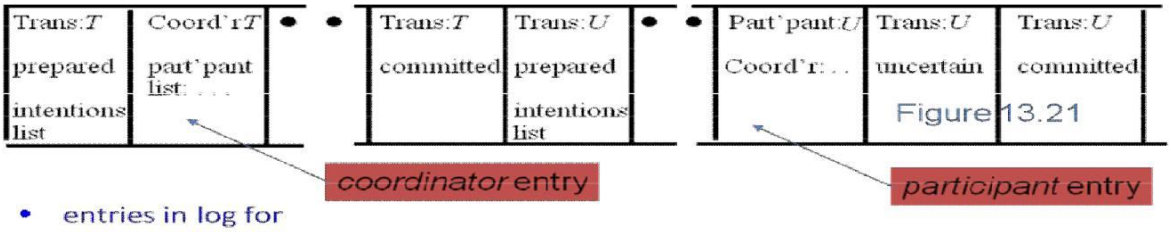


Figure 13.21

- entries in log for
  - *T* where server is coordinator (*prepared* comes first, followed by the coordinator entry, then *committed* – *done* is not shown)
  - and *U* where server is participant (*prepared* comes first followed by the participant entry, then *uncertain* and finally *committed*)
  - these entries will be interspersed with values of objects
- recovery must deal with 2PC entries as well as restoring objects
  - where server was coordinator find *coordinator* entry and status entries.
  - where server was participant find *participant* entry and status entries

Start at end, for U find it is committed and a participant , We have T committed and coordinator, But if the server has crashed before the last entry we have U *uncertain* and participant, or if the server crashed earlier we have U *prepared* and participant

.

Recovery of the two-phase commit protocol

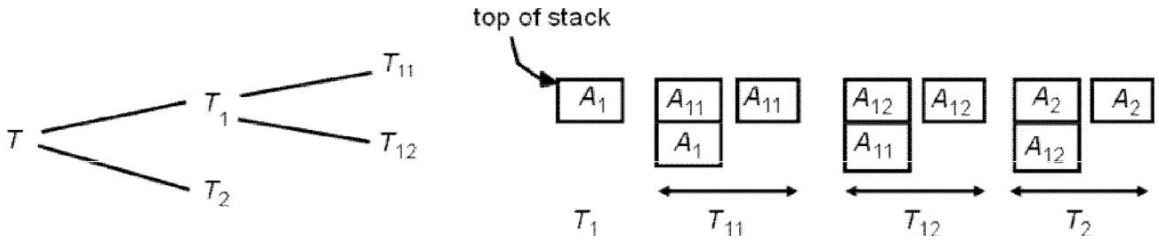
Role	*Status*	*Action of recovery manager*
Coordinator	*prepared*	No decision had been reached before the server failed. It sends *abortTransaction* to all the servers in the participant list and adds the transaction status *aborted* in its recovery file. Same action for state *aborted*. If there is no participant list, the participants will eventually timeout and abort the transaction.
Coordinator	*committed*	A decision to commit had been reached before the server failed. It sends a *doCommit* to all the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4 (Fig 13.5).
Participant	*committed*	The participant sends a *haveCommitted* message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	*uncertain*	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a *getDecision* to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	*prepared*	The participant has not yet voted and can abort the transaction.
Coordinator	*done*	No action is required.

Figure 13.22

the most recent entry in the recovery file determines the status of the transaction at the time of failure

the RM action for each transaction depends on whether server was coordinator or participant and the status

Nested transactions:



Summary of transaction recovery

Transaction-based applications have strong requirements for the long life and integrity of the information stored.

Transactions are made durable by performing checkpoints and logging in a recovery file, which is used for recovery when a server is replaced after a crash.

Users of a transaction service would experience some delay during recovery.

It is assumed that the servers of distributed transactions exhibit crash failures and run in an asynchronous system,

- but they can reach consensus about the outcome of transactions because crashed servers are replaced with new processes that can acquire all the relevant information from permanent storage or from other servers

Topic 07: Fault-tolerant services

provision of a service that is correct even if *f* processes fail

- by replicating data and functionality at RMs
- assume communication reliable and no partitions
- RMs are assumed to behave according to specification or to crash
- intuitively, a service is correct if it responds despite failures and clients can't tell the difference between replicated data and a single copy
- but care is needed to ensure that a set of replicas produce the same result as a single one would.

Example of a naive replication system

Client 1:	Client 2:
$setBalance_B(x,1)$	
$setBalance_A(y,2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

RMs at *A* and *B* maintain copies of *x* and *y*

clients use local RM when available, otherwise the other one

RMs propagate updates to one another after replying to client

- initial balance of x and y is $0
  - client 1 updates X at B (local) then finds B has failed, so uses A
  - client 2 reads balances at A (local)
    - as client 1 updates y after x, client 2 should see $1 for x
  - not the behaviour that would occur if A and B were implemented at a single server
- Systems can be constructed to replicate objects without producing this anomalous behaviour.
- We now discuss what counts as correct behaviour in a replication system.

Figure: Native Replication System

Linearizability the strictest criterion for a replication system

Consider a replicated service with two clients, that perform read and update operations. A client waits for one operation to complete before doing another. Client operations o10, o11, o12 and o20, o21, o22 at a single server are interleaved in some order e.g. o20, o21, o10, o22 , o11, o12 (client 1 does o10 etc)

The correctness criteria for replicated objects are defined by referring to a virtual interleaving which would be correct

a replicated object service is *linearizable* if *for any execution* there is some interleaving of clients'
operations such that:

– the interleaved sequence of operations meets the specification of a (single)
correct copy of the objects

– the order of operations in the interleaving is consistent with the real time at which they occurred

– For any set of client operations there is a virtual interleaving (which would be correct for a set of single objects).

– Each client sees a view of the objects that is consistent with this, that is, the results of clients operations make sense within the interleaving.

the bank example did not make sense: if the second update is observed,the first update should be observed too.

– linearizability is not intended to be used with transactional replication systems

– The real-time requirement means clients should receive up-to-date information

but may not be practical due to difficulties of synchronizing clocks a weaker criterion is sequential consistency

Sequential consistency

- a replicated shared object service is sequentially consistent if for any execution there is some interleaving of clients' operations such that:
  - the interleaved sequence of operations meets the specification of a (single) correct copy of the objects
  - the order of operations in the interleaving is consistent with the program order in which each client executed them

the following is sequentially consistent but not linearizable

Client 1:	Client 2:
$setBalance_B(x,1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y,2)$	

this is possible under a naive replication strategy, even if neither *A* or *B* fails - the update at *B* has not yet been propagated to *A* when client 2 reads it

but the following interleaving satisfies both criteria for sequential consistency :

$getBalance_A(y) \rightarrow 0$; $getBalance_A(x) \rightarrow 0$; $setBalance_B(x,1)$; $setBalance_A(y,2)$

it is not linearizable because client2's *getBalance* is after client 1's *setBalance* in real time.

The passive (primary-backup) model for fault tolerance

There is at any time a single primary RM and one or more secondary (backup, slave) RMs

FEs communicate with the primary which executes the operation and sends copies of the updated data to the result to backups

if the primary fails, one of the backups is promoted to act as the primary

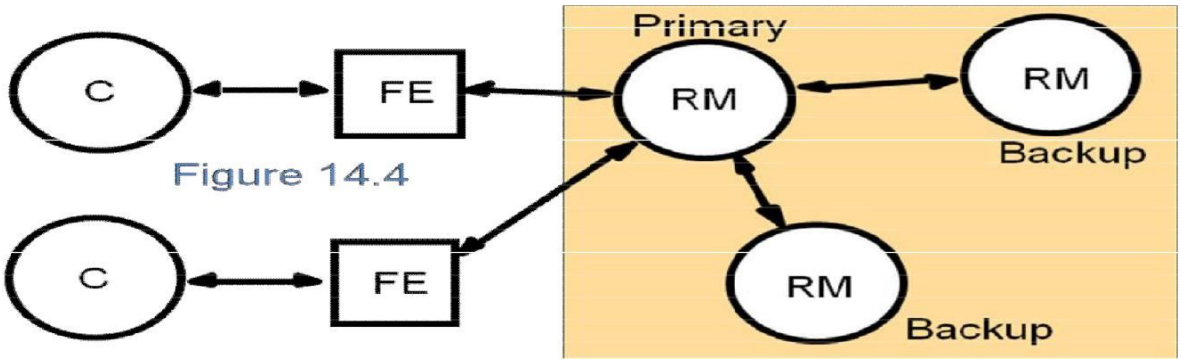


Figure 14.4

The FE has to find the primary, e.g. after it crashes and another takes over

Passive (primary-backup) replication. Five phases.

The five phases in performing a client request are as follows:

1. Request:
   – a FE issues the request, containing a unique identifier, to the primary RM
2. Coordination:
   – the primary performs each request atomically, in the order in which it receives it relative to other requests
   – it checks the unique id; if it has already done the request it re-sends the response.
3. Execution:
   – The primary executes the request and stores the response.
4. Agreement:
   – If the request is an update the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
5. Response:
   – The primary responds to the FE, which hands the response back to the client.

Discussion of passive replication

To survive *f* process crashes, *f*+1 RMs are required

– it cannot deal with byzantine failures because the client can't get replies from the backup RMs

To design passive replication that is linearizable

– View synchronous communication has relatively large overheads
– Several rounds of messages per multicast
– After failure of primary, there is latency due to delivery of group view

variant in which clients can read from backups

– which reduces the work for the primary

– get sequential consistency but not linearizability Sun

NIS uses passive replication with weaker guarantees

– Weaker than sequential consistency, but adequate to the type of data stored

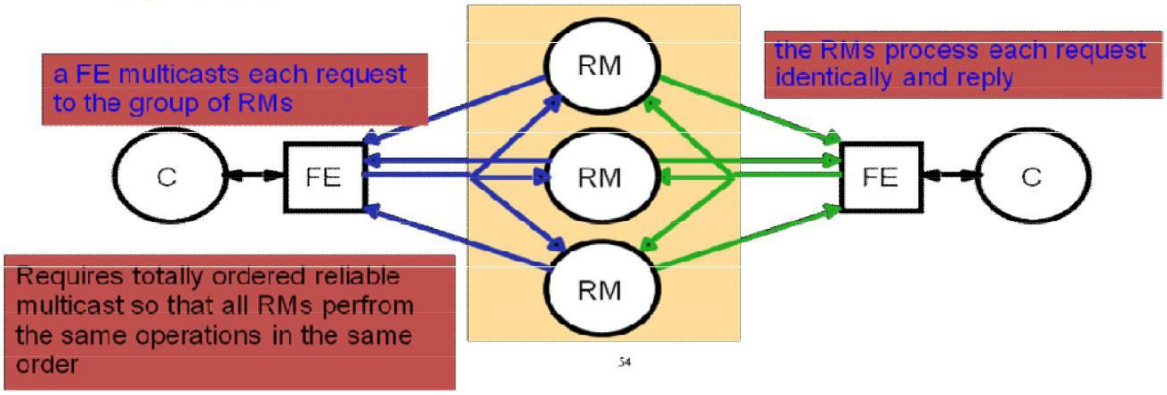
– achieves high availability and good performance

– Master receives updates and propagates them to slaves using 1-1 communication. Clients can uses either master or slave

– updates are not done via RMs - they are made on the files at the master.

Active replication for fault tolerance

- the RMs are *state machines* all playing the same role and organised as a group.
  - all start in the same state and perform the same operations in the same order so that their state remains identical
- If an RM crashes it has no effect on performance of the service because the others continue as normal
- It can tolerate byzantine failures because the FE can collect and compare the replies it receives



Active replication - five phases in performing a client request

Request

– FE attaches a unique *id* and uses *totally ordered reliable multicast* to send request to RMs.

FE can at worst, crash. It does not issue requests in parallel Coordination

– the multicast delivers requests to all the RMs in the same (total) order.

Execution

– every RM executes the request. They are state machines and receive requests in the same order, so the effects are identical. The *id* is put in the response

Agreement

– no agreement is required because all RMs execute the same operations in the same order, due to the properties of the totally ordered multicast.

Response

– FEs collect responses from RMs. FE may just use one or more responses. If it is only trying to tolerate crash failures, it gives the client the first response.

Active replication – discussion

As RMs are state machines we have sequential consistency

– due to reliable totally ordered multicast, the RMs collectively do the same as a single copy would do

– it works in a synchronous system

– in an asynchronous system reliable totally ordered multicast is impossible – but failure detectors can be used to work around this problem. How to do that is beyond the scope of this course.

this replication scheme is not linearizable

– because total order is not necessarily the same as real-time order

To deal with byzantine failures

– For up to f byzantine failures, use $2f+1$ RMs

– FE collects f+1 identical responses

To improve performance,

– FEs send read-only requests to just one RM.

UNIT-VI
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. In distributed systems, a logical clock is associated with [ ]
   A)each instruction B)each process
   C)each register D)none of the above
2. If time stamps of two events are same, then the events are [ ]
   A)concurrent B)non-concurrent C)monotonic D)non-monotonic
3. If a process is executing in its critical section
   A)any other process can also execute in its critical section
   B)no other process can execute in its critical section
   C)one more process can execute in its critical section
   D)none of the above
4. A process can enter into its critical section [ ]
   A)any time
   B)when it receives a reply message from its parent process
   C)when it receives a reply message from all other processes in the system
   D)none of the above
5. In case of failure, a new transaction coordinator can be elected by[ ]
   A)bully algorithm B)ring algorithm
   C)both(a)and(b) D)none of the above
6. According to the ring algorithm, links between processes are
   A)bidirectional B)unidirectional
   C)both(a)and(b) D)none of the above
7. A_____would involve two or more of the process becoming stock indefinetly while attempting to enter or exit the critical section.
8. The simplest way to achieve_____is to employ a server that grants permission to enter the critical section.
9. Each logical object is implemented by a collection of physical copies called_____.
10. The goal of the ring based election algorithm is to elect a single process called the_____which is the process with the largest identifier.
    A.Replica B.Coordinator C.Checkpoint D. xecution
11. Which of the following facility or capacity are required to provide support for mutual exclusion.
    i)A process that halts in its non critical section must do so with out interfering with other processes.
    ii)Assumption should made about relative process speeds or number of processors.
    iii)A process remains in side its critical section for a finite time only
    A) (i)&(ii) B) (ii)&(iii) C)(i) & (iii)

12. In the token passing approach of distributed systems, processes are organized in a ring structure

a)logically b)physically

c)both(a)and(b) d)none of the above

SECTION-B

SUBJECTIVE QUESTIONS

1. Illustrate bully algorithm and explain how it is different from other election algorithms.
2. Describe in detail about distributed deadlocks.
3. identify features required for election algorithms.
4. Differentiate active replication and passive replication.
5. What is meant by concurrency control? How it is important in distributed systems.
6. Summarize about coordination and agreement in group communication.
7. Explain about multicast communication in distributed systems?
8. Write the algorithm of distributed mutual exclusion.
9. Identify edge chasing in deadlock detection?
10. How does synchronization delay affect the throughput of a system? How can it be avoided?