SIR C R REDDY COLLEGE OF ENGINEERING, ELURU
Department of Information Technology

Subject Name: DISTRIBUTED SYSTEMS **Subject Code:** R1642051
Year-IV/IV B.Tech II Semester

Distributed Systems

Unit – I

Introduction: Now a days networks of computers are used in everywhere. The Internet is one, it provides the information of world as web or village. It means, know the information within fracation of seconds which we require. Different networks are composed and they used the internet. For example, Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks etc. All of these networks using '*Distributed Systems*' technology to communicate and coordinate their actions only by passing messages.

DS = N/W +Distributed Architecture +Distributed OS +Distributed Application.

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.*

(or)

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

This definition covers the entire range of systems in which networked computers can usefully be deployed.

Characterization of Distributed Systems:

Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents in the same building or in the same room. Our definition of distributed Systems have the following Characteristics of Distributed Systems or Significant Consequences:

1.Concurrency: In a network of computers, concurrent program execution is common. It means, " *I can do my work on my computer while you can do your work on your computer*", sharing resources of both work (such as web pages on files) when necessary. The capacity of sharing can be increased by adding more resources (for example computers) to the network.

2. No global clock: Programs coordinate actions by exchanging messages in distributed systems. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But there are limits to the accuracy with which the computers in a network can synchronize their clocks. In this case, "*there is no single global notion of the correct time*". This is a direct consequence of the fact that the only communication is by sending messages through network.

3. Independent failures: Every system can fial based on the responsibility of system designers. But Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected it, but they doesn't mean stop running. In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or unexpected termination of a program by a crash is not effect to other computer in distributed networks.

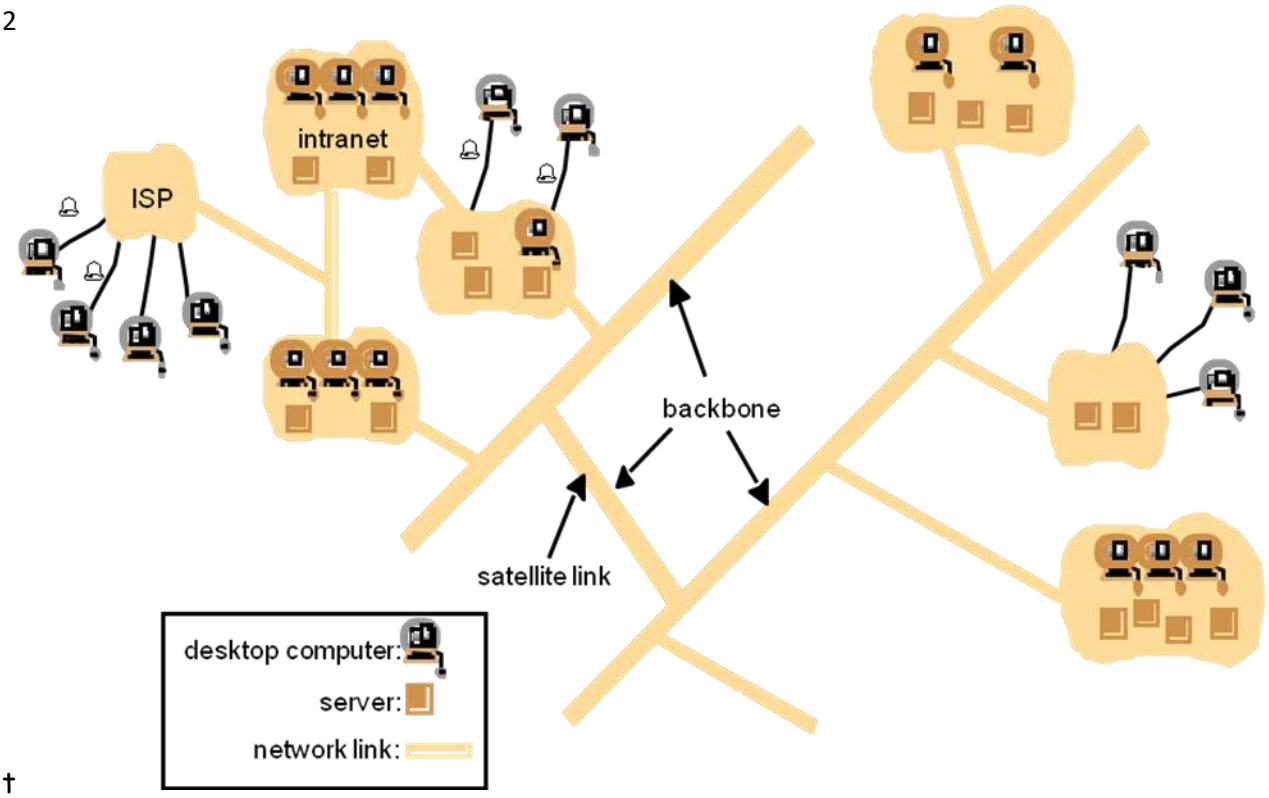
when some systems fail, others may not know and other system will provide the service goal of the distributed system is sharing the resource.

Examples of Distributed Systems: Examples of distributed systems which are familiar and widely used networks. The Internet, Intranet, Mobile Computing and Mobile and ubiquiting computing.

1. **Internet**: It is a very large distributed system that allows users throughout the world to make use of its services. For example, World Wide Web, web search, online gaming, email, social networks, ecommerce, etc.
   - WWW, email, FTP, VOD, etc

- Internet protocols is a major technical achievement.
  - TCP/IP: connect different type computer networks.

2



t

The figure illustrates a typical portion of the Internet. Programms running on the computers connected to it interact by passing messages in communication. The design and construction of the Internet communication mechanism (the Internet protocols) is a major technical achievement, enabling a program running anywhere to address messages to programs anywhere else.

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 figure shows a collection of intranets that subnetworks operated by compaines and other organizations.

Internet Service Providers (ISPs) are companies that provide modem links and other types of connection to individual users and small organizations, enabling them to access services anywhere in the Intranets are linked together by backbones.

A backbone is a network link with a high transmission capacity, employing satellite connections, fiber optic cables and other high-bandwidth circuits.

2. **Intranets:** 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.

The figure shows a typical intranet. It is composed of several local area networks (LANs) linked by backbone connections. The network configuration of a particular intranet is the responsibility of the organization that administers it and may vary widely.

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 also allows the users in other intranets to access the services it provides. Many organizations need to protect their own services from unauthorized use by possibly malicious users elsewhere.

For example, a company will not want secure information to be accessible to users in competing organizations, and a hospital will not want sensitive patient data to be revealed. Companies also want to protect themselves from harmful programs such as viruses entering and attacking the computers in the intranet and possibly destroying valuable data.

The role of a *firewall* is to protect an intranet by preventing unauthorized messages leaving or entering. A firewall is implemented by filtering incoming and outgoing messages, for example according to their source or destination. A firewall might for example allow only those messages related to email and web access to pass into or out of the intranet that it protects.

The main issues arising in the design of components for use in intranets are:

- File services are needed to enable users to share data.
- Firewalls tend to impede legitimate access to services
- The cost of software installation and support is an important issue.

3. **Mobile and Ubiquitous Computing:** Due to the technological advances in device efficiency and wireless networking have been increased to the integration of small and portable computing into distributed systems.

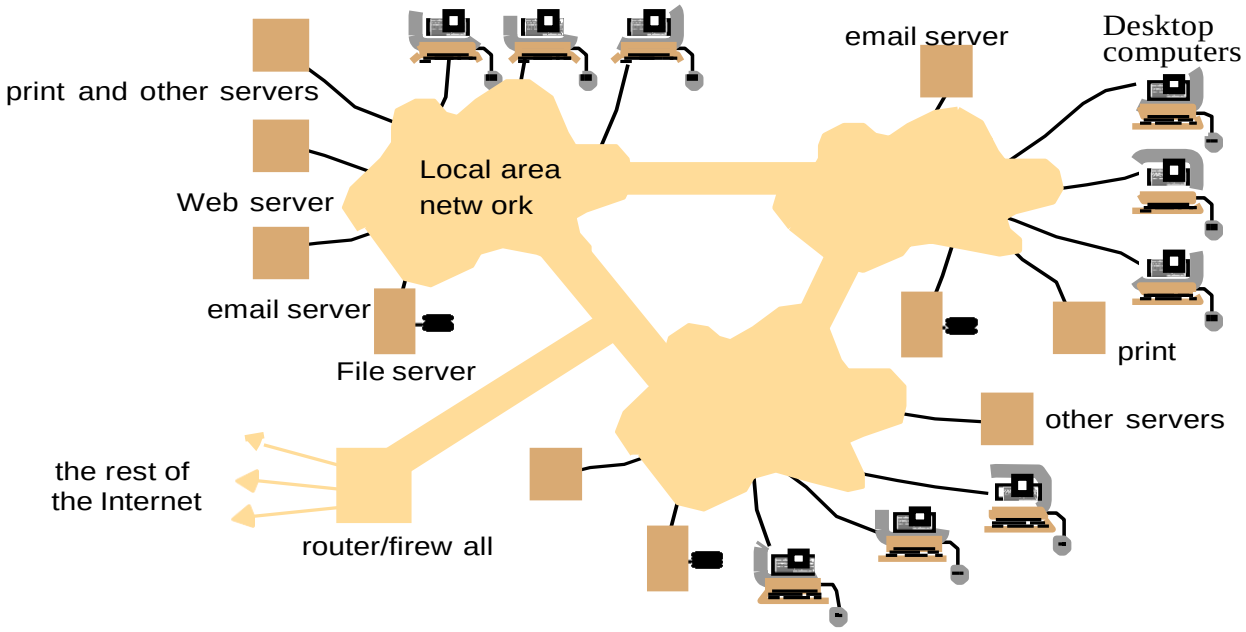
These devices include:

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

Mobile computing: Mobile computing is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their 'home' intranet are still provided with access to resources via the devices they carry with them. The mobile computing increasing the service for users to utilize resources such as printers or even sales points that are conveniently nearby as they move around. This mobility service is also known as *location-aware or context-aware computing.*

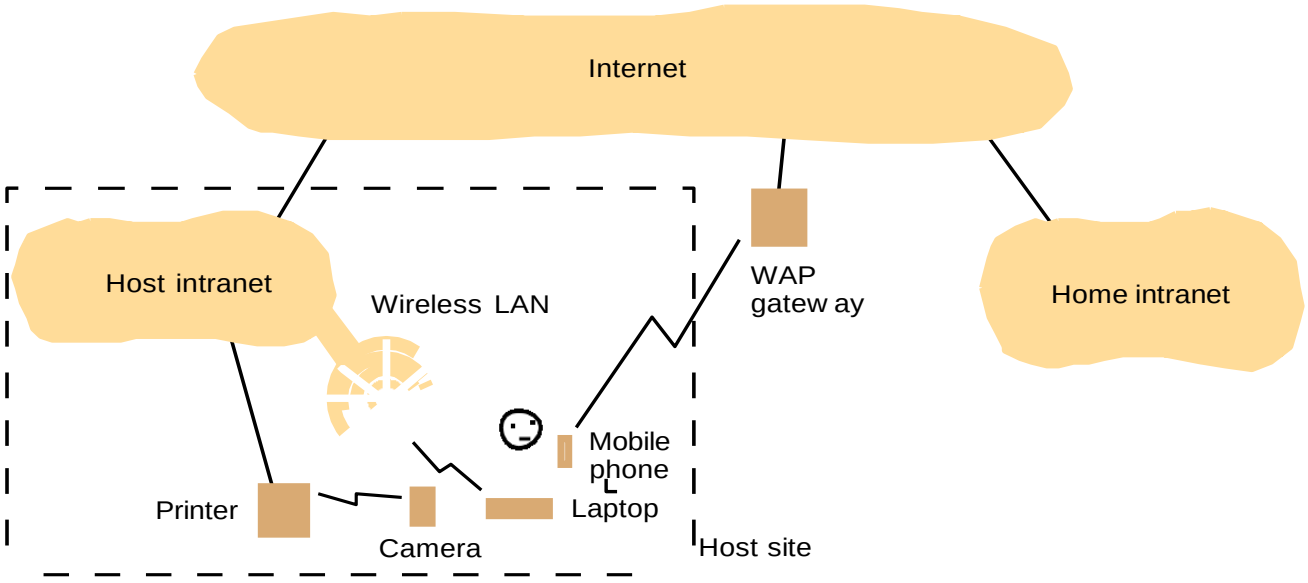
Thus, Mobility introduces a number of challenges for distributed systems including to maintain operation in the face of device mobility.

Ubiquitous computing is small, cheap computational device that 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 (persistent) in everyday objects that they are hardly noticed. It means, its computational behavior will be transparently and tied up with their physical function.


print and other servers
Web server
email server
File server
Local area
netw ork
email server
Desktop
computers
print
other servers
the rest of
the Internet
router/firew all

The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it may be convenient for users to control their washing machine or their entertainment system from their phone or a 'universal remote control' device in the home. Equally, the washing machine could notify the user via a smart badge or phone when the washing is done.
The following figure shows a user who is visiting a host organization. The figure shows the user's home intranet and host intranet at the site that the user is visiting. 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(a floor of a building). It connects to the rest of the host intranet via a gateway or access point. The user also has a mobile telphone, which is connected to the internet. The phone gives access to the web and other internet services. Finally, the user carries a digital camera, which can communicate over a personal area wireless network with a device such as a printer.



Focus on resource sharing and web

We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.

- Resources in a distributed system are physically encapsulated within computers and can only be accessed by communication.
- For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.

Service: *A service* is a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications.

For example,
→ We access shared files through a file service;
→ We send documents to printers through a printing service;
→ We buy goods through an electronic payment service.

The only access we have to the service is via the set of operations that it exports.
For example, a file service provides *read*, *write* and *delete* operations on files.

Server and Client: The term *server* is refers to a running program (a *process*) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately.

The requesting processes are referred to as *clients,* and the overall approach is known as *client-server computing*.

In this approach, requests are sent in messages from clients to a server and replies are sent in messages from the server to the clients. When the client sends a request for an operation to be carried out, we say that the client *invokes an operation* upon the server.

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*.

The same process may be both a client and a server, since servers sometimes invoke operations on other servers.

The terms 'client' and 'server' apply only to the roles played in a single request. In this case, Clients are active (making requests) and servers are passive (only waking up when they receive requests);

Servers run continuously, whereas clients last only as long as the applications of which they form a part.

World Wide Web The World Wide Web [www.w3.org I, Berners-Lee 1991] is an evolving system for publishing and accessing resources and services across the Internet. Through web browsers, users retrieve and view documents of many types, listen to audio streams and view video streams, and 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 between a community of physicists connected by the Internet [Berners-Lee 1999]. The feature of the Web is a *hypertext* structure among the documents that it stores, reflecting the users' requirement to organize their knowledge. This means that documents contain *links* (or *hyperlinks*) – references to other documents and resources that are also stored in the Web.

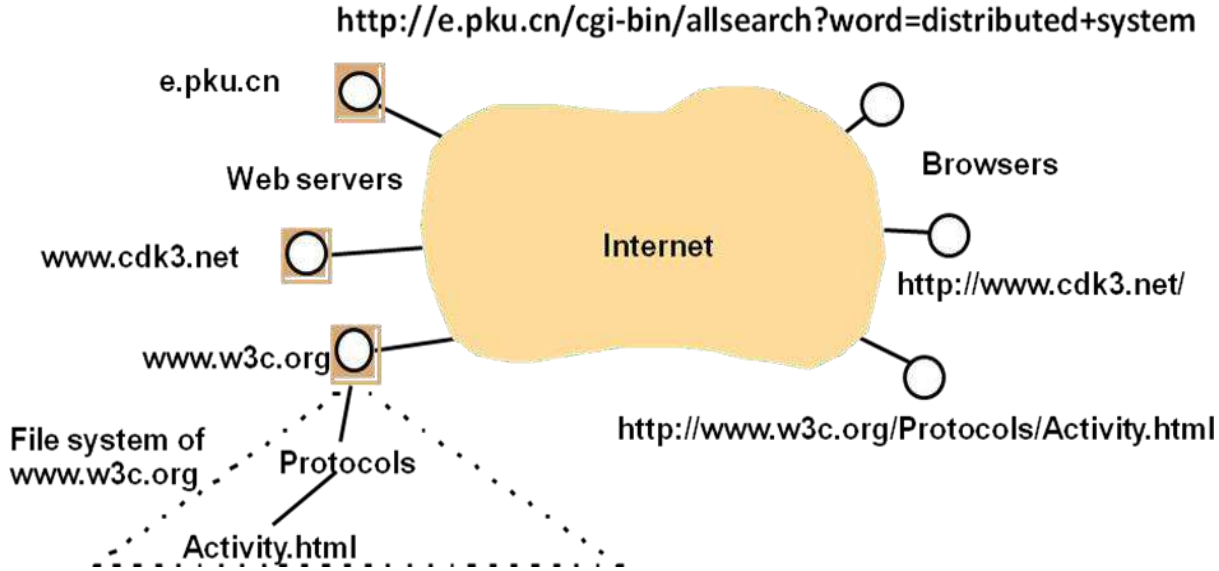
- **Definition**: Web is an open system that it can be extended and implemented in new ways without disturbing its existing functionality.

  - Its operation is based on communication standards and document standards
  - Respect to the types of 'resource' that can be published and shared on it.

The Web has moved beyond these simple data resources to encompass services, such as electronic purchasing of goods. It has evolved without changing its basic architecture. The Web is based on three main standard technological components:

- The ***HyperText Markup Language*** (HTML), a language for specifying the contents and layout of pages as they are displayed by web browsers;
- ***Uniform Resource Locators*** (URLs), also known as Uniform Resource Identifiers (URIs), which identify documents and other resources stored as part of the Web;
- A ***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. Figure shows some web servers, and browsers making requests to them. It is an important feature that users may locate and manage their own web servers anywhere on the Internet

6 **Web servers and web browsers:**



HTML : The HyperText Markup Language [www.w3.org II] 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.
Users may produce HTML by hand, using a standard text editor, but they more commonly use an HTML-aware 'wysiwyg' editor that generates HTML from a layout that they create graphically. 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

- This HTML text is stored in a file that a web server can access.
- A browser retrieves the contents of this file from a web server.
  -The browser interprets the HTML text
  -The server can infer the content type from the filename extension.

URLs : The purpose of a Uniform Resource Locator [www.w3.org III] is to identify a resource. It means, the term used in web architecture documents is Uniform Resource *Identifier* (URI),

```
Scheme: scheme-specific-location
e.g:
        mailto:joe@anISP.net
        ftp://ftp.downloadIt.com/software/aProg.exe
        http://net.pku.cn/
        ....
```

The 'scheme', declares which type of URL this is. URLs are required to identify a variety of resources. For example, *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.

Other examples of schemes are 'tel' (used to specify a telephone number to dial, which is particularly useful when browsing on a mobile phone) and 'tag' (used to identify an arbitrary entity).

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.

The above figure (web servers and web browser) 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.

In general, HTTP URLs are of the following form:

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

where 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
Consider the URLs:

http://www.cdk5.net http://www.w3.org/standards/faq.html#conformance
http://www.google.com/search?q=obama

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

Note that the HTML directives, known as *tags*, are enclosed by angle brackets, such as *<P>*. Line 1 of the example identifies a file containing an image for presentation. Its URL is *http://www.cdk5.net/WebExample/Images/earth.jpg*. Lines 2 and 5 are directives to begin and end a paragraph, respectively. Lines 3 and 4 contain text to be displayed on the web page in the standard paragraph format.

HTTP:The HyperText Transfer Protocol [www.w3.org IV] defines the ways in which browsers and other types of client interact with web servers.

- Main features
  - ***Request-replay interaction*** : 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 such as the familiar '404 Not Found'.
  - ***Content types*** : The strings that denote the type of content are called MIME (RFC2045,2046)
  - ***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.
  - ***Simple access control***: Any user with network connectivity to a web server can access any of its published resources. for example, by typing in a password.
- Dynamic content
  - Common Gateway Interface: a program that web servers run to generate content for their clients
- Downloaded code : Sometimes the designers of web services require some service-related code to run inside the browser, at the user's computer. In this particular way, need run one of the following code.
  - JavaScript
  - Applet

[8]Challenges

The main challenges are

1. **Heterogeneity:** The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity applies to all of the following:
   - Networks;
   - Computer hardware;
   - Operating systems;
   - Programming languages;
   - Implementations by different developers.

Networks: The Internet consists of many different networks and their differences are masked by the internet protocols to communicate with one another. For example, a computer attached to an Ethernet and the entire Internet protocols are implementation is over the Ethernet.

Computer hardware: Data types such as integers may be represented in different ways on different hardware. For example, there are two alternatives for the byte ordering of integers. These differences in representation must be done when messages are to be exchanged between programs running on different hardware.

Operating Systems: The Operating Systems of all computers on the Internet need to include an implementation of the Internet protocols and with different Application Programs on each. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

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.

Implementations by different developers: Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this, standards need to be agreed and adopted.

Middleware: The term *middleware* applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. The Common Object Request Broker (CORBA) is an example. Some middleware such as Java Remote Method Invocation (RMI) supports only a single programming language.

Middleware provides a uniform computational model for use by the programmers of servers and distributed applications.

Heterogeneity and mobile code: The term *mobile code* is used to refer to program code that can be transferred from one computer to another and run at the destination. Java applets are an example.

The *virtual machine* approach provides a way of making code executable on a variety of host computers. The compiler for a particular language generates code for a virtual machine instead of a particular hardware order code.

For example, the java compiler produces code for a java virtual machine, which executes it by interpretation. The java virtual machine needs to be implemented once for each type of computer to enable java programs to run.

Today, the most commonly used form of mobile code is JavaScript programs in some web pages loaded into client browsers.

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.

In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures, which are usually cumbersome and slow-moving.

However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people.

A benefit that is often cited for open systems is their independence from individual vendors.

To summarize:

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

3. **Security:** Many of the information resources are 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) Eg: ACL in UNIX File system,
- Integrity (protection against alteration or corruption) Eg: checksum, and
- Availability (protection against interference with the means to access the resources) Eg: Denial of service.

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

For example:

1. A doctor might request access to hospital patient data or send additions to that data.
2. In electronic commerce and banking, users send their credit card numbers across the Internet.

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

In the first example, the server needs to know that the user is really a doctor, and in the second example, the user needs to be sure of the identity of the shop or bank with which they are dealing.

The second challenge here is to identify a remote user or other agent correctly. Both of these challenges can be met by the use of encryption techniques developed for this purpose.

4. **Scalability:** A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The number of computers and servers in the Internet has increased dramatically.

Figure shows the increasing number of computers and web servers during the 12-year history of the Web up to 2005 [zakon.org]. It is interesting to note the significant growth in both computers and web servers in this period, but also that the relative percentage is flattening out.

10**Figure** Growth of the Internet (computers and web servers)

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

The design of scalable distributed systems presents the following challenges:

1. *Controlling the cost of physical resources*: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it. For example, the frequency with which files are accessed in an intranet is likely to grow as the number of users and computers increases.

   For example, if a single file server can support 20 users, then two such servers should be able to support 40 users.
2. *Controlling the performance loss*: Consider the management of a set of data whose size is proportional to the number of users or resources in the system.

   For example, the table with the correspondence between the domain names of computers and their Internet addresses held by the Domain Name System (DNS), which is used mainly to look up DNS names such as **www.amazon.com**.
3. *Preventing software resources running out*: An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet). In the late 1970s, it was decided to use 32 bits for this purpose, the supply of available Internet addresses is running out. For this reason, a new version of the protocol with 128-bit Internet addresses is being adopted, and this will require modifications to many software components.
4. *Avoiding performance bottlenecks*: In general, algorithms should be decentralized to avoid having performance bottlenecks. The Domain Name System removed this bottleneck by partitioning the name table between servers located throughout the Internet and administered locally.
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 proposed 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. For example, checksums can be used to detect corrupted data in a message or a file. That is, difficult or even impossible to detect some other failures, such as a remote crashed server in the Internet.

The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

Masking failures: Some failures that have been detected can be hidden or made less severe. Two examples of hiding failures:

1. Messages can be retransmitted when they fail to arrive.

2. File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

Tolerating failures: Most of the services in the Internet do exhibit failures.

For example, when a web browser cannot contact a web server, it does not make the user wait forever while it keeps on trying.

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.

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.

6. Concurrency: Both services and applications provide resources that can be shared by clients in a distributed system. Therefore several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time.

- Correctness
  - ensure the operations on shared resource correct in a concurrent environment

  e.g. records bids for an auction.
- Performance
  - Ensure the high performance of concurrent operations.

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. The implications of transparency are a major influence on the design of the system software.

Access transparency enables local and remote resources to be accessed using identical operations.

Location transparency enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.

Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.

Failure transparency enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

Mobility transparency allows the movement of resources and clients within a system without affecting the operation of users or programs.

Performance transparency allows the system to be reconfigured to improve performance as loads vary.

Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

SYSTEM MODELS

Introduction: Systems which are used in real-world environments should be designed to function correctly in the widest possible.

1. Architecture model
2. fundamental model

Difficulties for and threads to distributed System:

1. **Widely varying modes of use:** The components (or) parts of systems are wide variations in workload – for example, some web pages are accessed several million times a day. In this case, some parts of a system may be disconnected, or poorly connected some of the time.

For example, multimedia applications have special requirements for high communication bandwidth and low latency.

2. **Wide range of system environments:** A distributed system must accommodate heterogeneous hardware, operating systems and networks.
   → The networks may differ widely in performance i.e., wireless networks operate at a fraction of the speed of local networks.
   → In distributed systems, differing scales of system must be supported i.e., ranging from tens of computers to millions of computers.
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.

To overcome the above problems in distributed system, the properties and design issues can change with the use of **descriptive system models**. Each type of model is planned to provide an abstract, simplified but consistent description of a relevant aspect of distributed system design:

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

Fundamental models are concerned with properties that are common in all of the architectural models. It is addressed by three fundamental models that examine the important aspects of distributed systems:

→ ***interaction models,*** which consider the structure and sequencing of the communication between the elements of the system;
→ ***Failure models,*** which consider the ways in which a system may fail to operate correctly.
→ ***Security models***, which consider how the system is protected against attempts to interfere with its correct operation or to steal its data.

Architectural models: The architectural of a system has a collection of individually defined components. The complete objective is to make sure that the structure will satisfy the current and future demands on it. The vital responsibilities are to make the system cost effective, manageable, reliable and adaptable. The architectural design of a building has same features like determining architectural style gives a constant frame of reference for the design.

The central architectural models are constructed almost on the notion of process and object. The functioning of separate components of a distributed system is first simplified and then abstracted in an architectural model. This model later evaluates the following things.

a) The arrangement of components throughout a network of computers i.e., trying to specify beneficial patterns for the distribution of data and workload.
b) The interrelationships b/w the components i.e., the components functional roles and the pattern of communication.

By altering client server model, one can construct few dynamic systems.

a) A process can assign a task to another, since it is possible to the code from one process to another.
   Example: The code from servers is downloaded by the client processes and executed locally. To minimize access delays and communication traffic, the code and objects that accesses the process can be transferred.
b) Few of the distributed systems are designed to allow the computers and other mobile devices to be added or removed smoothly, enabling to find obtainable services and also to provide their services to others.

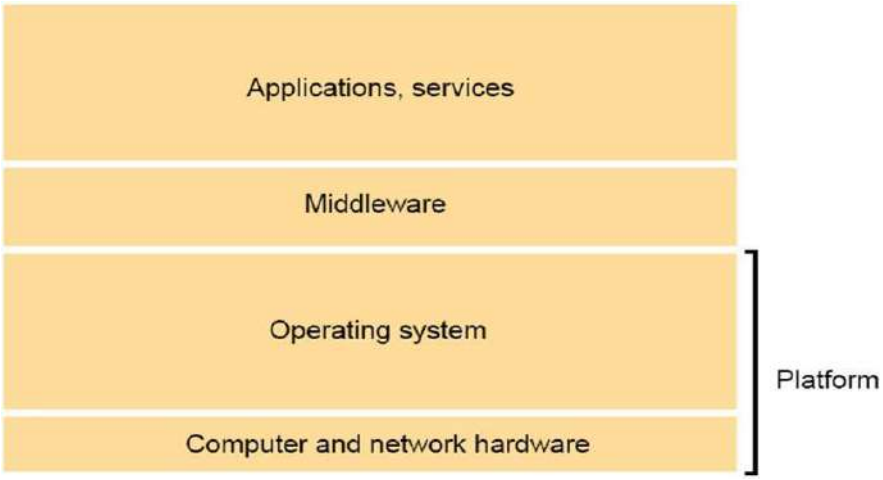
There are many patterns that are used fully for the allocation of work in a distributed system. These patterns have a major effect on the performance of the resulting system.
An architectural model of a distributed system is concerned with the placement of its parts and the relationships b/w them.
Example: Client-server, Peer-to-peer

1. Software Layers: The software architecture refers to services offered and requested b/w processes located in the same or different computers.

The structuring of software as modules or layers in a single computer and the services extended and appealed b/w processes situated in the same or different computers is actually referred to as software architecture. The disclosed form of this process and service is a layer of services in distributed systems. This layer of service included both software and hardware services. The process which accepts request from other processes is a server. One or more servers can supply a distributed service. To support a constant system-wide view of the service resources, these servers can communicate with one another and also with client processes.

The above figure introduced two important terms platform and middleware.

1. **Platform:** The bottom layers (i.e., operating system, computer and network hardware layers) provide a platform for distributed applications. The services provided by these bottom layers are utilized by higher layers. In every computer, operating system layer and computer network layer are implemented separately. This implementation facilitates communication and coordination b/w processes by transferring the system's programming interface to that level.
   Examples: Intel x86/Windows, Intel x86/Solaris, PowerPC/MAC OS, Intel x86/Linux



2. **Middleware:** A piece of software placed b/w the application and operating system is called "middleware". Its intension is to enclose heterogeneity and to supply an accessible programming model to application programmers. The concept of middleware can be easily explained in the context of objects or processes in a set of computers that interact with each other to carryout communication and resource-sharing support for distributed application. Middleware supplies helpful building blocks in the construction of software components. These components can interact in distributed applications.

   It provides useful building blocks. They are Remote method invocation, communication b/w a group of processes, notification of events, partitioning, placement and retrieval of data or objects, replication, transmission of multimedia data in real time.

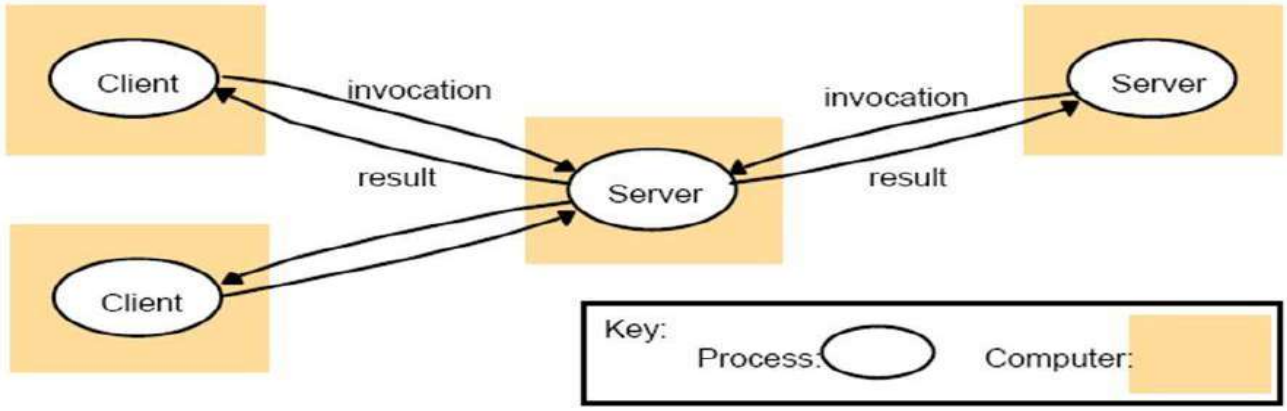
   The earliest examples of middleware is remote procedure calling packages (eg: sun RPC) and group communication system [eg: Isis]. Object-oriented middleware products and standards are

   a) CORBA b) java RMI c)Microsoft Distributed Component objects model (DCOM)
   d) Web services e) The ISO/ITU-T's Reference model for open distributed processing (RM-ODP).

2. System Architecture: The architecture model has two types of environments in distributed systems. They are

1. Client-server model and 2. Multiple server model.

Client-server model: It is very frequently referred in distributed systems. This architecture is broadly utilized and significantly very important. The figure below shows the structure of client's server model. The figure shows the structure of client-server model.

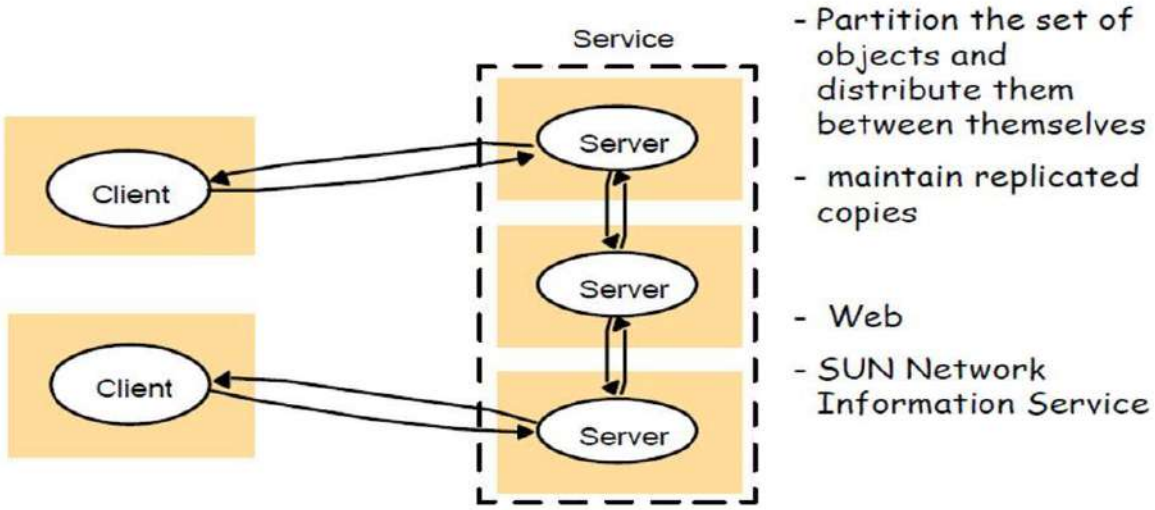


In order to access the shared resources managed by server processes in different host computers, the client processes interact with those server processes. The figure above specifies that the clients of some servers may also be consisting the web pages has a common client called web server. The DNS service has many clients such as web servers and many other internet services. The service provided by DNS is that it interprets Internet domain names to network address. Search engines are another example related to web. It allows users to search the sites all over the Internet for synopsis of information accessible on web pages. These synopses are created by programs called web crawlers.

At search engine site, these web crawlers are executed in the background, using HTTP requests to access web servers all over the Internet. A search engine can play the role of both a server and a client. It performs the following two tasks,

a) It replies to queries from browser clients.
b) Executing web crawlers that act as clients of other web servers.

Multiple server's model: In multiple server model, several servers implement the services as its processes in separate host computers. After implementation these computers interact with each other if required in order to provide the service either by partitioning the services into set of objects and distributing them among themselves or by maintaining replicate copies of these services on several hosts. The server may also provide the service either by partitioning the services into set of objects and distributing them among themselves by maintain replicate copies of these services on several hosts. The multiple server model is shown in fig.



The concept of this model is illustrated in the following examples.

1. On the web each server maintains their own resources, called partitioned data. Therefore, a user can access a resource of any one server.
2. Replication is an important concept of increasing availability and performance and for improving fault tolerance. The processes that run on different computers can have multiple

consistent copies of data due to replication. For an instance, data available at mittal elec.com might also be provided by some other web services. It is due to mapping of service onto several servers mainlining database replicated in memory.

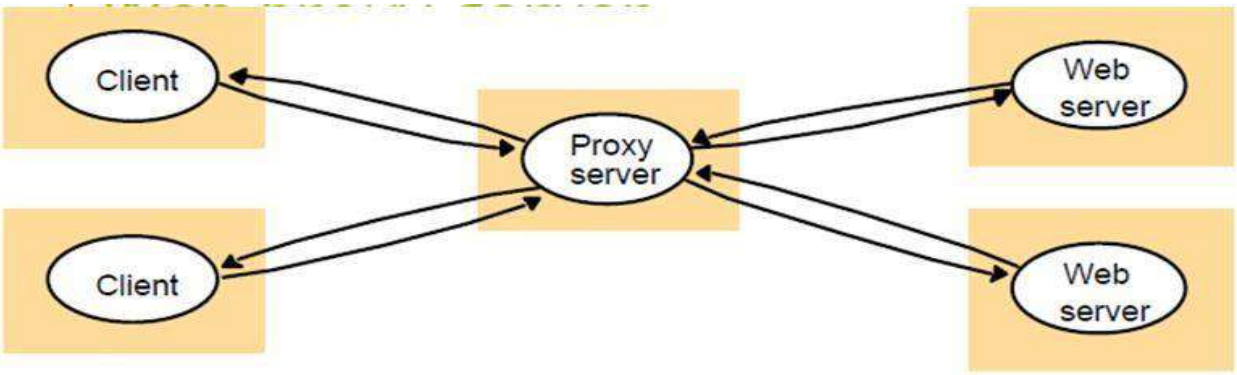
3. Another example of replication based service is the Sun NIS (Network Information Service). Each server of NIS maintains duplicate copies of password file that holds a list of user's login names along with encrypted passwords.

3.Proxy servers and caches: A cache is repository. A cache has storage facility of recently used objects that is closer to the objects themselves. A web proxy server provides a shared cache of web resources for the client machines at a site or across several sites.

→ Increase availability

→ Increase performance

→ Reduce the load on the wide-area network and web server.

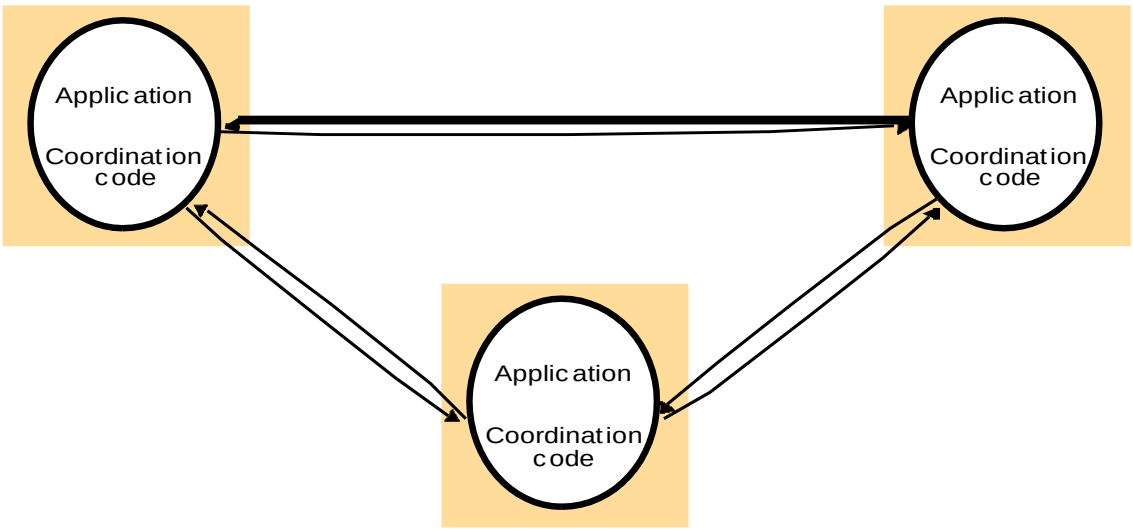


When a client process requests an object, the caching service starts examining the cache for updated copy. If the updated copy is available, then caching service provides the object from the repository. Otherwise, it retrieves an updates copy. In general caches are situated in a proxy server that is shared by multiple clients or available with each client.

The objective of a proxy server is to extend the performance and availability of the service by decreasing the load on wide-area network and web servers. Apart from this, they are used to get remote web servers through a firewall. For a client machine at a site or over multiple sites, a shared cache is supplied by web proxy server.

4.A Distributed application on Peer process: In peer processes architecture, all processes play similar roles and interact cooperatively with each other as peers, performing distributed activity or computation without making any differentiation among clients and servers.

The code in peer processes is responsible of maintaining consistency for application level resources and synchronizing application level actions based on the requirements. The peer processes architecture is shown in figure.



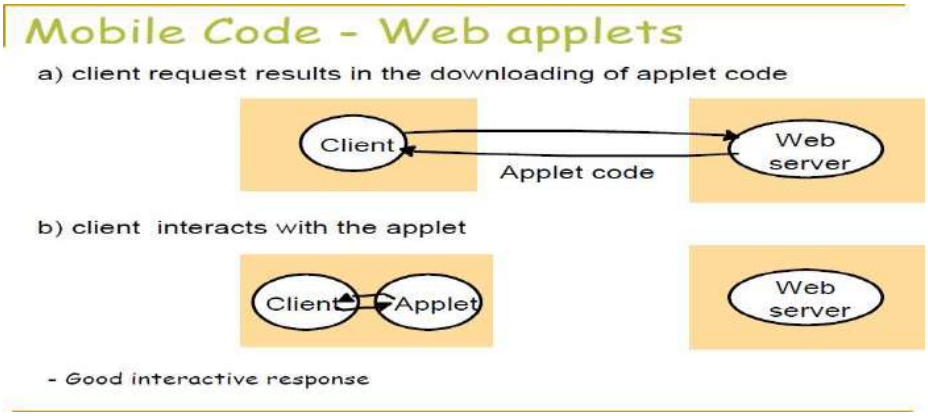
For example, assume an application of a distributed 'white board' that allows users to view and modify a picture shared b/w several computers. This can be made possible by implementing as an application process at each site relying on middleware layers. So that event notification and group communication can be performed in order to notify the changes to the picture for all application processes.

5. Client-server Model Variations for Mobile Code:

Several variations on the client server model can be derived from the consideration of the following factors:

- The use of mobile code and mobile agents
- Users need for low cost computer with limited H/W resources that are simple to manage.
- The requirement to add and remove mobile device in convenient manner.

- Example: Java applets
  - The user running a browser selects a link to an applets whose code is stored on a web server.
  - The code is downloaded to the browser and runs there.
- Advantage:
  - Good interactive response since.
  - Does not suffer from the delays or variability of bandwidth associated with network communication.
- Disadvantage:
  - Security threat to the local resources in the destination computer.



Mobile Agents: A mobile agent is a running program that travels from one system to another in a network. A network performs a task such as gathering information, providing results on behalf of someone. A mobile agent may make many invocations to local resources at each site it visits. Eg: access to individual database entries.

Mobile agents are possibly used in cases such as

a) Install software on the computers of an organization.
b) To compare the prices of products different vendors by visiting the site of every vendor and carryout a sequence of database operations.

A recent example is worm program developed at Xerox PARC. It is designed in such a way to transfer detailed computations by using of idle computers.

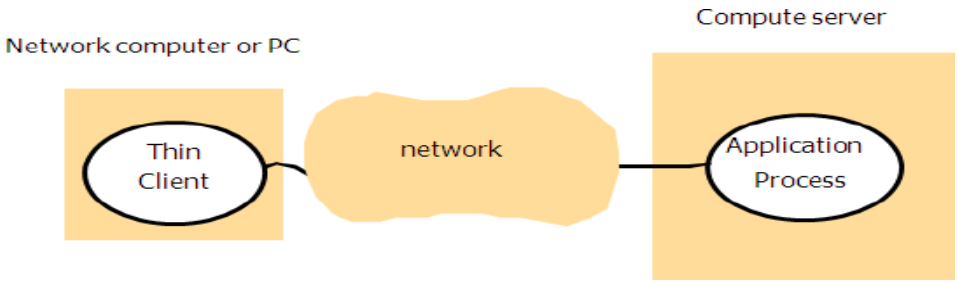
Thin Clients: A window-based user interface on a computer local to the user while running applications on a remote computer is supported by a software layer referred to as thin client.

A computer server has the capacity to run applications simultaneously. It is a powerful computer that runs a multiprocessor version of an operating system (UNIX or windows NT).

The highly interactive graphical activities such as CAD and image processing are the limitation of the tin client architecture. Since, the transfer of image and vector information b/w the thin

client and application process include both network and operating system potentials, the delay experience by users is increased.

Implementation: The implementation of thin client systems is uncomplicated. For examole, X-11 window system, a variant of UNIX. Other implementation includes The citrix Win Frame Product and the Tele-porting and Virtual Network Computer (VNC) systems.



Network Computers: The desktop computer local to a user is used to run the applications. For these computers, the operating system and application software generally needs data to be placed on a local disk and more active code. A more technical effort is needed to manage application files and to maintain local software but most of the users are not eligible to provide this effort.

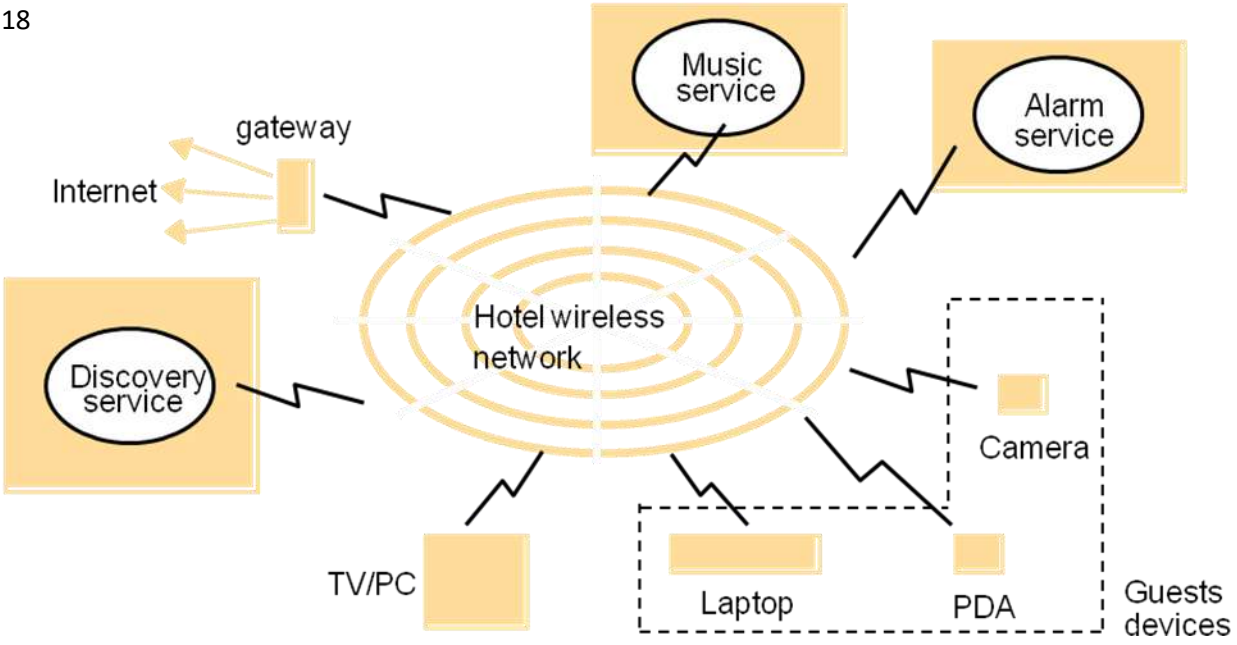
With the help of network computer, the user downloads application software and its operating system from remote file server. It takes response to the above problem. Although the files are administered by a remote file server, applications such as web browser also run locally. It is possible for a user to move from one network computer to other, because a file server stores all the data and code of an application.

Mobile devices: Mobile devices are the hardware computing components that carry software and are freely movable b/w physical locations and networks. Some examples of mobile devices include the following: laptops, handheld devices, mobile phones, PDAs (Personal Digital Assistants), digital cameras, and some wearable computers like smart watches.

Mostly mobile devices are supported by wireless networking. They can be oerated on greater ranges of networks such as GSM and 3G telecommunication networks. Some of them can operate over network upto hundreds of meters like Wifi (IEEE802.11) or upto ten meters like Bluetooth. On mobile devices there is a possibility of existing for both, client and server. The existences of clients are common, but in some cases it might be possible for a mobile device to have server as well.

6.Mobile devices and Spontaneous networking : Key Features of Spontaneous Networking are easy connection to a local network and Easy integration with local services

- Security problems from connectivity of mobile users:
  - Limited connectivity
  - Security and privacy
- The purpose of a **discovery service** is to accept and store details of services that are available on the network and to respond to queries from clients.
- A discovery service offers two interfaces:
  - A registration service accepts registration requests from servers and records the details in the discovery service's database.
  - A lookup service accepts queries concerning available services and searches its database for registered services that match the queries.



Interface and Objects:

- Interface definitions specify the set of functions available for invocation in a server (or peer) processes.
- In OO paradigm, distributed processes can be constructed as objects whose methods can be accessed by remote invocation (COBRA approach).
- Many objects can be encapsulated in server or peer processes.
- Number, types, and locations (in some implementation) of objects may change dynamically as system activates require.
- Therefore, new services can be instantiated and immediately be made available for invocation.

Design Requirements for Distributed Architectures:

1. **Performance Issues:**
   - Considered under the following factors:
     - Responsiveness:
       - Fast and consistent response time is important for the users of interactive applications.
       - Response speed is determined by the load and performance of the server and the network and the delay in all the involved software components.
       - If the process is in the same computer then also data transfer rate b/w processes and associated control switching is low. This can be overcome if the system must be composed of relatively few software layers and small quantities of transferred data to achieve good response times.
       - In case of remote text pages, good response time can be achieved due to their small size.
       - In case of graphical images that contain large volume of data, interactive response time is very slow.
     - Throughput:
       - It can be defined as the rate of completing the computational task. It is the traditional performance measure for computer systems.
       - It determines the ability of performing the work for all users in a distributed system.
       - Data present on remote server if needed to be passed from a server process to a client process, then it has to pass through all software layers present in both computers.

- Load balancing:
  - Enable applications and service processes to proceed concurrently without competing for the same resources.
  - Exploit available processing resources.

2. **Quality of Service provided in distributed systems:**
   - The main system properties that affect the service quality are:
     - **Reliability:** related to failure fundamental model (discussed later).
     - **Performance**: ability to meet timeliness guarantees.
     - **Security**: related to security fundamental model (discussed later).
     - **Adaptability**: ability to meet changing resource availability and system configurations.
3. **Dependability issues:**
   - The dependability issues can be defined as correctness, security and fault tolerance.
   - Dependability is the crucial requirement in most application domains.
   - The issues achieved by:
     - **Fault tolerance**: The ability to continue its function in the presence of failures that may arise in software, hardware and networks. Reliability of an application can be obtained by means of redundancy.
     - **Security:** locate sensitive data only in secure computers. For example, consider the database of a bank containing customer records with sensitive and widely used items. The sensitive issues must be visible to only some specific bank authorities.
     - Correctness of distributed concurrent programs: research topic.

Fundamental Models: A model contains only the essential ingredients needed to understand and reason about some aspects of a system's behavior.

System Model: System model is a model used to create accurate assumptions regarding the systems that are to be modeled. It is used to create generalizations about the systems that are to be modeled based on the assumptions made. Here, generalizations are considered as efficient general purpose algorithm or the properties which are required and guaranteed.

The factors that are to be considered for the construction of fundamentals models:

a) **Security:** The distributed systems have large attacks because of their openness and modular nature. The security in fundamental model defines attack front forms with the threat analysis and system design that are resistant to threats.

b) **Interaction:** Interaction is the communication and coordination b/w the processes with the exchange of messages. The interaction model in distributed systems must reflect the following facts:
   i) Communication involves delay of specific duration.
   ii) The delays limit the accuracy of process coordination.
   iii) Difficult to maintain same time conventions across the computers.

c) **Failure:** The operation of a distributed system is large to failure with the occurrence of fault in the network or in the computer which is connected to it. Fundamental model describes the faults and analyses their effects. It also designs fault tolerable systems capable of running even in the presence of faults.

1.Interaction Model: Interacting processes are responsible for the execution of activities in the distributed systems. Each process has a state that cannot be accessed or updated by another process. A state consists of variables and a set of data that can be updated and accessed by its state.

Interacting processes in a distributed system are affected by two significant factors:

a) Performance of communication channels: is characterized by:

i) **Latency**: It is the delay that occurs when the message is transmitted from one process to another. The delay includes the time in between sending and receipt of a message including

1. Network access time.
1. Time for first bit transmitted through a network to reach its destination.
2. Processing time within the sending and receiving processes.

ii. **Throughput/ latency** : number of units (e.g., packets) delivered per time unit.

- It can be defined as the rate of completing the computational task. It is the traditional performance measure for computer systems.
- It determines the ability of performing the work for all users in a distributed system.
- Data present on remote server if needed to be passed from a server process to a client process, then it has to pass through all software layers present in both computers.

iii. **Bandwidth**: It is the amount of data that can be transmitted over a network in a given time period. If same network is being used by many communication channels, then they have to share the bandwidth.

iv. **Jitter**: It is the difference in time during the transmission of a message. This difference in time can be seen when a series of audio data samples with variant time intervals are played resulting into distortion in time.

b) Computer clocks:

- Each computer in a distributed system has its own internal clock to supply the value of the current time to local processes.
- Therefore, two processes running on different computers read their clocks at the same time may take different time values.
- *Clock drift rate* refers to the relative amount a computer clock differs from a perfect reference clock.
- Several approaches to correcting the times on computer clocks are proposed.
- Clock corrections can be made by sending messages; from a computer has an accurate time to other computers, which will still be affected by network delays.

Types of Interaction model: Setting time limits for process execution, as message delivery, in a distributed system is hard.

Two opposing extreme positions provide a pair of simple interaction models:

***Synchronous* distributed systems:**

- A system in which the following bounds are defined:
  1. Time to execute each step of a process has known lower and upper bounds.
  2. Each message transmitted over a channel is received within a known bounded time.
  3. Each process has a local clock whose drift rate from perfect time has a known bound.
- Easier to handle, but determining realistic bounds can be hard or impossible.

***Asynchronous* distributed systems:**

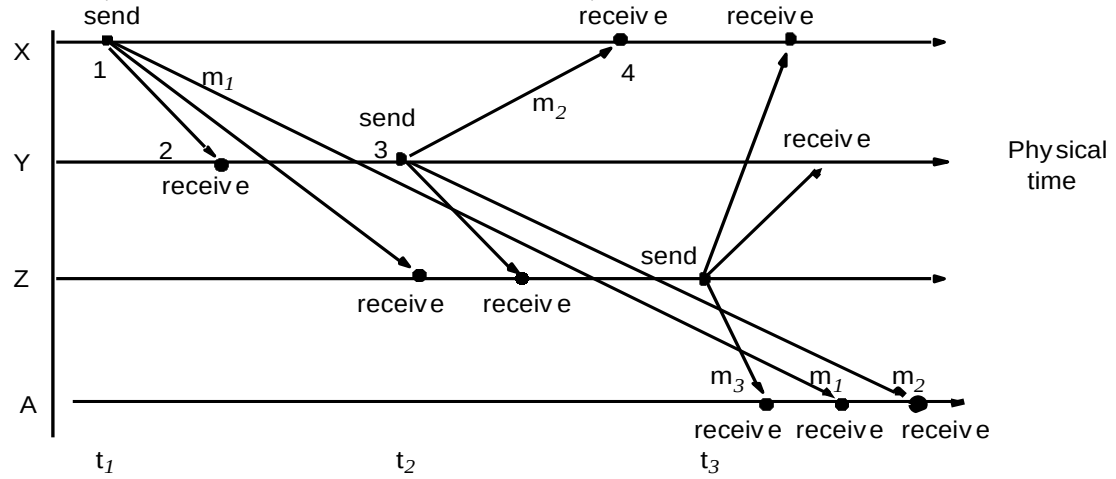
- A system in which there are no bounds on:
  1. Process execution times.
  2. Message delivery times.
  3. Clock drift rate.
- Allows no assumptions about the time intervals involved in any execution.
- Exactly models the Internet.
  - Browsers are designed to allow users to do other things while they are waiting.

➢ More abstract and general:

- A distributed algorithm executing on one system is likely to also work on another one.

Event ordering: if an event at one process (sending or receiving a message) occurred before, after, or concurrently with another event at another process, then DS algorithms keep them in order.

- Why because it is impossible for any process in a distributed system to have a view on the current global state of the system.
- The execution of a system can be described in terms of events and their ordering even though the lack of accurate clocks.
- Logical clocks define some event order based on causality.
- Logical time can be used to provide ordering among events in different computers in a distributed system (since real clocks cannot be synchronized).



2. Failure Model: Defines the ways in which failure may occur in order to provide an understanding of its effects.

- The classification of failures and differences between the failures of processes and communication channels is provided as:
  - *Omission* failures
    - Process or channel failed to do something.
    - Communication omission failures.
  - *Arbitrary* failures
    - Any type of error can occur in processes or channels (worst).
  - *Timing* failures
    - Applicable only to synchronous distributed systems where time limits may not be met.

I. *Omission* failures:

- Process or channel failed to do something.
- Communication omission failures.

➔ Omission failures are classified into

1. Process omission failure 2. Communication omission failures

1. Process Omission Failures: The primary reason in processes for omission failure is its crash. That means a process has been terminated completely. The design services that can resist the faults can be simplified on the assumption that the services crash clearly. In a clean crash, a process terminates or executes correctly. Due to a crash, the process does not respond to invocation messages repeatedly. This favors other processes in identifying the crash using timeout implies that a process is not responding. The reasons for this may be its crash or slow performance, or undelivered of messages.

A crash that is identified by other processes is called fail-stop. This can be introduced in a synchronous system as:

→ If messages delivery is correct.

→ If processes employ timeouts identify failure of their processes.

Consider the example, if the reply to a process is not delivered within the specified time limit, then the process that is waiting for the reply concludes that sender process has been failed.

2. Communication Omission Failures: Send and receive are two communication primitives. Consider the example where send operation is performed by process A. It inserts a message in its outgoing message buffer. Communication channel transmits this message to process B's incoming message buffer. Receive operation is performed by process B that takes the message from its incoming message buffer and delivers it.

The communication channel raises an omission failure if the message from A's outgoing message buffer is not transmitted to B's incoming message buffer. This is called a dropped message which is due to insufficient buffer space at the receiver site or at the intervening gateway, or by error at network transmission. These consequences are identified by the checksum that is attached with the message content.

Following are the failures resulting due to loss of messages:

a) **Send-omission failure:** Occur in b/w sending process and outgoing message buffer.
b) **Receive-Omission failures:** Occur in b/w incoming message buffer and receiving process.
c) **Channel-Omission failures:** Occur in b/w outgoing message buffer and incoming message buffer.

II. *Arbitrary Failures:* Arbitrary failure describes the semantics of a worst failure in which there is a possibility of occurrence of any type of error. In a process an arbitrary failure is caused when it skips any of its execution phases or adopts a new phase for execution. For example, a process may set incorrect values in its data items or it may return an incorrect value.

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

III. *Timing Failures:* Timing failure is applicable in synchronous distributed system whereas in asynchronous distributed system, it is not guaranteed.

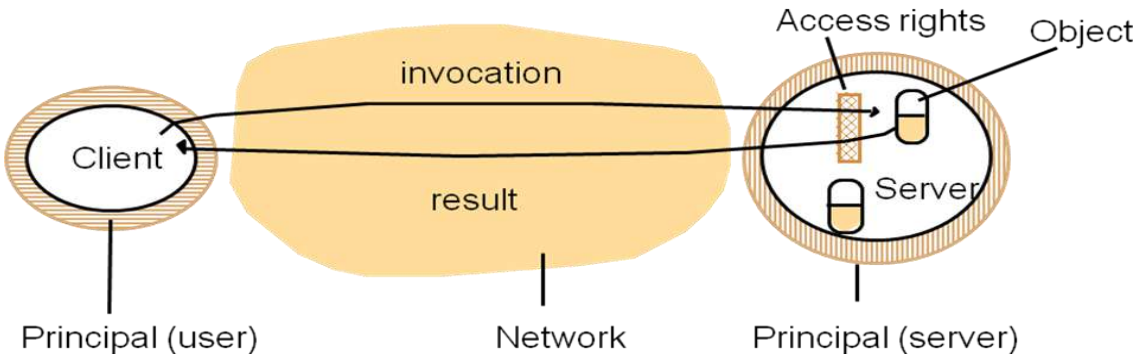
a) **Synchronous Distributed System:** In synchronous distributed system, time limits are fixed on process execution time, message delivery time and clock drift rate. A process is affected if it exceeds its time bounds. A channel is affected if the time taken for the transmission exceeds the specified bounds. A clock failure affects its process if its local time exceeds its drift rate bounds from real time. These three failures fail to respond within the specified bounds.

b) **Asynchronous Distributed system:** In asynchronous distributed system, timing failure cannot be guaranteed and there is a possibility of a heavily loaded server to respond late.

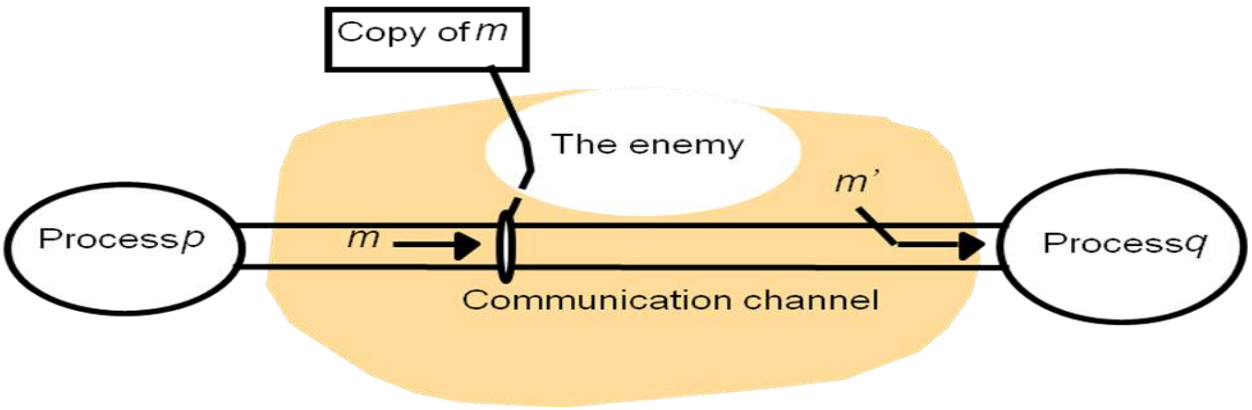
Class of Failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Security Model: Secure processes and channels and protect objects encapsulated against unauthorized access.

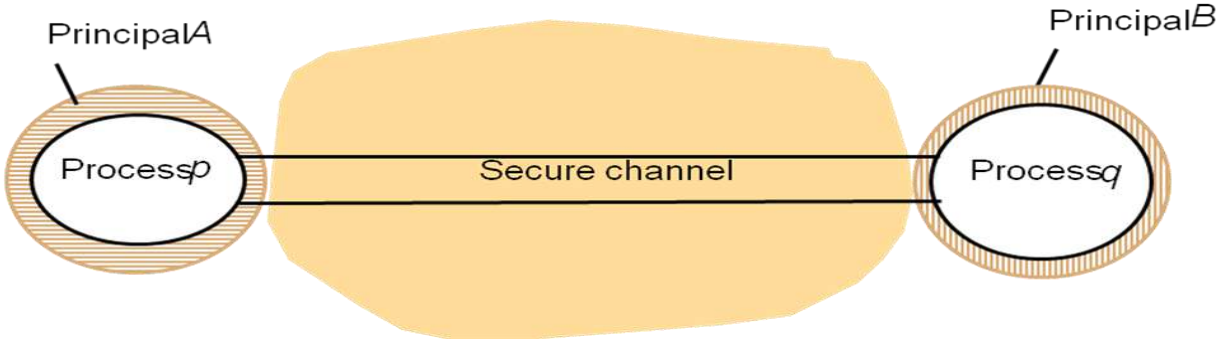
- **Protecting access to objects**
  - Access rights
  - In client server systems: involves authentication of clients.
- **Protecting processes and interactions**
  - Threats to processes: problem of unauthenticated requests / replies.
    - e.g., "man in the middle"
  - Threats to communication channels: enemy may copy, alter or inject messages as they travel across network.
    - Use of "secure" channels, based on cryptographic methods.



Objects and principals



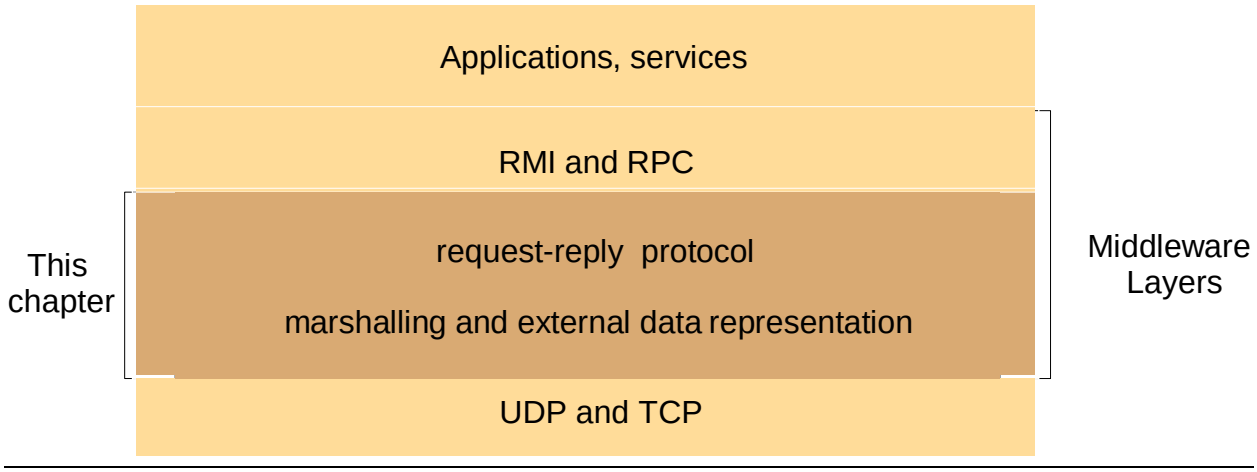
The enemy



Making failures – Reliability of one-to-one communication:

- A service masks a failure, either by hiding it all together or by converting it into a more acceptable type of failure.
  - Checksums are used to mask corrupting messages -> a corrupted message is handled as a missing message
  - Message omission failures can be hidden by re-transmitting messages.
- The term reliable communication is defined in terms of validity and integrity as follows:
  - Validity: any message in the outgoing buffer is eventually delivered to the incoming message buffer
  - Integrity: the message received is identical to one sent, and no messages are delivered twice.

UNIT – II INTERPROCESS COMMUNICATION



API for the internet protocol:

- Internet protocol as a example, explaining how programmers can use them, either by meaning of UDP message or through TCP streams.
- The application program interface **(API)** to **UDP** provides a message passing abstraction.
- The application program interface **(API)** to **TCP** provides the abstraction of a two-way stream between pairs of processes.
- The Java UDP and TCP APIs are discussed in the second section.

Characteristics of Inter process Communication:

- Message passing between a pair of processes can be supported by two message communication operations: ***send*** and ***receive***.
- In the **synchronous** form of communication, both **send** and **receive** are ***blocking*** operations.
- In the **asynchronous** form of communication, the ***send*** operation is **non-blocking** and the ***receive*** Operation can be **blocking and non-blocking.**
- Non-blocking receive is more efficient but more complex. So current systems do not generally provide the non-blocking form of receive.
- In the Internet protocols, messages are sent to (*Internet address, local port*) pairs. Any process that knows the number of a port can send a message to it.
- Using a fixed Internet address can be avoided by one of the following approaches to provide location transparency:
- Client program refer to services by name. This allows services to be relocated but not to migrate (be moved while running).
- The operating system, for example, Mach, provides location-independent identifiers for message destinations.

- Messages can be addressed to processes in the V system. However, ports provide several alternative points of entry to a receiving process. Chorus provides the ability to send messages to groups of destination.
    - **Ordering**: Some applications required that messages be delivered in send order-i.e; the order in which they were transmitted by sender. The delivery of messages out of sender order is regarded as failure by such applications.

Inter process communication in the Internet provides both datagram and stream communication. The Java APIs for these are presented, together with a discussion of their failure models. They provide alternative building blocks for communication protocols. This is complemented by a study of protocols for the representation of collections of data objects in messages and of references to remote objects.

Multicast is an important requirement for distributed applications and must be provided even if underlying support for IP multicast is not available. This is typically provided by an overlay network constructed on top of the underlying TCP/IP network. Overlay networks can also provide support for file sharing, enhanced reliability and content distribution.

Message destinations

- 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.

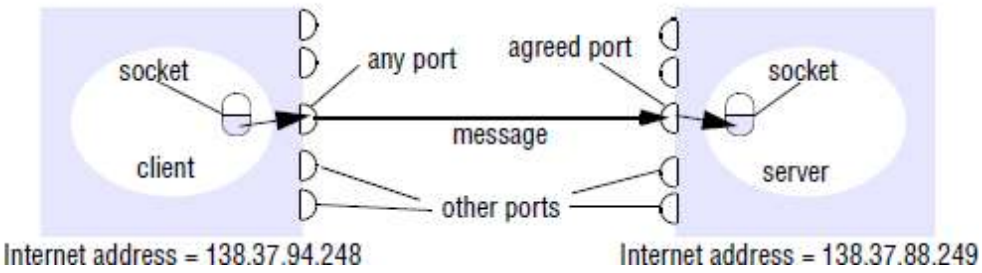
Reliability

- A reliable communication is defined in terms of **validity** and **integrity**.
- **Validity** is defined, a point-to-point message service is described as reliable if messages are guaranteed to be delivered without dropped or lost.
- For **integrity**, messages must arrive uncorrupted and without duplication.
- Some applications require that messages be delivered in ***send order***

Sockets

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an **endpoint for communication between processes**. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Inter process communication consists of transmitting a message between a socket in one process and a socket in another process, as illustrated in Figure below.

- A socket provides endpoints for communication between processes as shown in Figure
- A socket must be bound to a local port.
- A socket pair (local IP address, local port, foreign IP address, foreign port) uniquely identifies a coSockets and ports



Java API for Internet Address

As the IP packets underlying UDP and TCP are sent to internet addresses. Java provides a class, Inet Address that represents Internet addresses. Users of this class refer to computers by DNS hostnames.

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

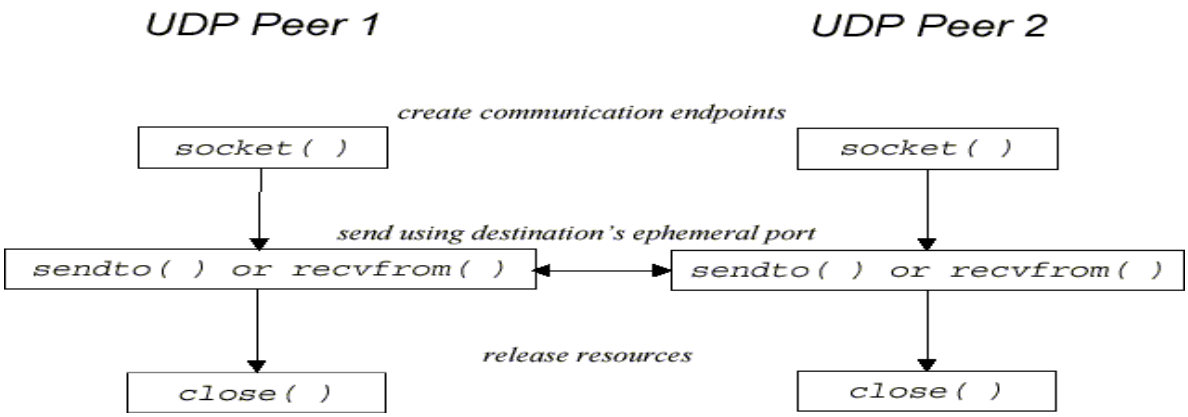
This method throws an UnknownHostexception.

UDP datagram communication

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

The receiving process needs to specify an array of bytes of a particular size in which to receive a message. The most size restriction is **8 kilobytes**.

Sockets normally provide **non-blocking** sends and **blocking receives** for datagram communication.



The following are some issues relating to datagram communication:

- ***Message size***: The receiving process needs to specify an array of bytes of a particular size in which to receive a message. The most size restriction is 8 kilobytes.
- ***Timeouts*:** The method receive blocks until a datagram is received, unless a timeout has been set on the socket
- **Receive from any:** The receive method does not specify an origin for messages. Instead ,an invocation of receive gets a message addressed to its socket from any origin.The receive method returns the Internetaddress and local port of the sender, allowing the recipient to check where the message came from.
- **Blocking** : Sockets normally provide non-blocking sends(Sender will perform all operations along with sending operation) and blocking receives(receiver stop the all operation except receiver) for datagram communication.

  Sockets normally provide blocking sends (Sender will perform all operations along with sending operation) and blocking receiver (receiver stop the all operation except receiver) for TCP communication.

- **Failure model / UDP datagram's suffer for following failure**:

Reliable communication in terms of two properties: integrity and validity. The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP data grams suffer from the following failures:

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

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

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

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

• The need to store state information at the source and destination;

• The transmission of extra messages;

• Latency for the sender.

Common Internet Applications that use UDP

- **Trace route** - print the route packets take to network host. Try /usr/sbin/traceroute www.microsoft.com.
- **RIP** (Routing Information Protocol) is a widely-used protocol for managing router information
- **BOOTP** (Bootstrap Protocol) is a protocol that lets a network user be automatically configured (receive an IP address) and have an operating system boot or initiated without user involvement. BOOTP is the basis for a more advanced network manager protocol, the Dynamic Host Configuration Protocol(DHCP).
- **Dynamic Host Configuration Protocol (DHCP)** is a communications protocol that lets network administrators manage centrally and automate the assignment of Internet Protocol (IP) addresses in an organization's network.)
- **Network Time Protocol (NTP)** is a protocol that is used to synchronize computer clock times in a network of computers.
- **Trivial File Transfer Protocol (TFTP)** is an Internet software utility for transferring files where user authentication and directory visibility are not required.
- **Simple Network Management Protocol (SNMP)** is the protocol governing network management and the monitoring of network devices and their functions.
- The **domain name system (DNS)** is the way that Internet domain names are located and translated into Internet Protocol addresses.
- The **Network File System (NFS)** is a client/server application that lets a computer user view and optionally store and update file on a remote computer as though they were on the user's own computer. UDP is used with the early version of NFS.
- **Remote Procedure Call (RPC)** is a protocol that one program can use to request a service from a

program located in another computer in a network without having to understand network details.

- There are several RPC models and implementations. A popular model and implementation is the Open Software Foundation's **Distributed Computing Environment (DCE).**
- **Open Network Computing (ONC)** RPC, sometimes referred to as Sun RPC, was one of the first commercial implementations of RPC.

The Java API provides datagram communication by means of two classes:

DatagramPacket - Datagram packets are used to implement a connectionless packet delivery service. DatagramSocket - A datagram socket is the sending or receiving point for a packet delivery service.

DatagramPacket contain the following methods

getData - Returns the data buffer.

getPort - Returns the port number on the remote host.

getAddress - Returns the IP address.

DatagramSocket contain the following methods:

send - Sends a datagram packet from this socket.
receive - Receives a datagram packet from this socket.

setSoTimeout - Enable/disable the specified timeout, in milliseconds.
connect - Connects the socket to a remote address for this socket.

Datagram packet:

arrayofbytescontainingmessage	lengthofmessage	Internetaddress	portnumber

UDP CLIENT

```
import java.net.*;
import java.io.*;

public class UDPClient{
   public static void main(String args[]){
    // args give message contents and server hostname
    DatagramSocket aSocket = null;
     try {
          aSocket = new DatagramSocket();
          byte [] m = args[0].getBytes();

          InetAddress aHost = InetAddress.getByName(args[1]);
          int serverPort = 6789;

          DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
          aSocket.send(request);

          byte[] buffer = new byte[1000];

          DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
          aSocket.receive(reply);
```

```
            System.out.println("Reply: " + new String(reply.getData()));
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        }catch (IOException e){System.out.println("IO: " + e.getMessage());}
      }finally {if(aSocket != null) aSocket.close();}
   } }
```

UDP SERVER

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

TCP stream communication

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read.

The following characteristics of the network are hidden by the stream abstraction:

***Message sizes*:**

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

***Lost messages*:**

The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges

all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required.

Flow control:

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

***Message duplication and ordering*:**

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

***Message destinations*:**

Once the connection between a pair of processes is established, the processes simply read from and write to the stream

TCP Abstractions

- *The data is the abstraction of a stream of bytes.*
- *A connection is established before messages are sent.*
- *It assumes one process is the client and one is the server in establishing a connection.*
- *The client create a socket bound to any local port and then makes a **connect** request for connecting the server at its server port.*
- *The server creates a listening socket bound to a server port and wait for (**accept**) the incoming requests.*
- *Messages are sent using handles (sockets) rather than source-destination addresses.*
- *An application **closes** a socket once it is done with the transmission.*
- ***The issues related to stream communication are:***
    - ***Matching of data items**: Two communicating processes need to agree on the contents of the transmitted data.*
    - ***Blocking**: When a process tries to read data from an input channel (stream), it will get the data or block until the data is available.*
    - ***Thread**: When a sever accepts a connection, it generally creates a new thread for the new client.*

TCP Failure Model

- ***TCP uses three mechanism to create a reliable communication**:*
    - ***Checksums and sequence numbers for integrity**: TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets.*
    - ***Timeouts** and retransmission for validity: TCP streams use timeouts and retransmissions to deal with lost packets.*
- *The problem is if the **network becomes congested**, no acknowledge is received and then the connection is broken. Thus TCP does not provide reliable communication.*

- ***When a connection is broken, it will has the following effects****:*
  - *The processes using the connection cannot distinguish between network failure and failure of the process.*
  - *The communication process cannot tell whether their recent messages have been received or not.*
- *The programmers need to deal with this situation in the program.*

Common Internet applications that use TCP

HTTP: The Hypertext Transfer Protocol is used for communication between web browsers and web servers;

FTP: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.

Telnet: Telnet provides access by means of a terminal session to a remote computer.

SMTP: The Simple Mail Transfer Protocol is used to send mail between computers.BGP (Border Gateway Protocol) is a protocol for exchanging routing information between gateway hosts (each with its own router) in a network of autonomous systems.

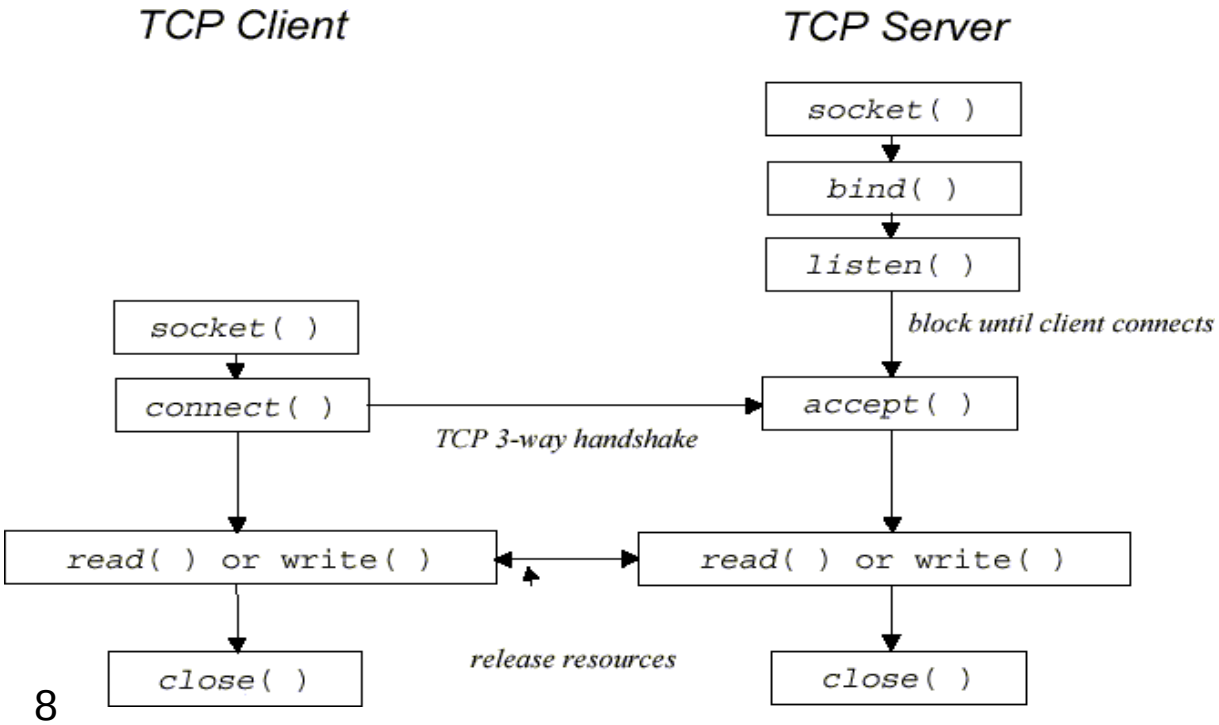
The domain name system (DNS) is the way that Internet domain names are located and translated into Internet Protocol addresses.

NNTP (Network News Transfer Protocol) is the predominant protocol used by computer clients and servers for managing the notes posted on Usenet newsgroups.

The Network File System (NFS) is a client/server application that lets a computer user view and optionally store and update file on a remote computer as though they were on the user's own computer.

Open Network Computing (ONC) RPC, sometimes referred to as Sun RPC, was one of the first commercial implementations of RPC.

The Open Software Foundation's Distributed Computing Environment (DCE) is a popular model and implementation of RPC.



8

TCP Client

```
import java.net.*;
import java.io.*;

public class TCPClient {

    public static void main (String args[]) {

    // arguments supply message and hostname of destination
    Socket s = null;

       try{

            int serverPort = 7896;

            s = new Socket(args[1], serverPort);

            DataInputStream in = new  DataInputStream( s.getInputStream());
            DataOutputStream out =

                  new        DataOutputStream(        s.getOutputStream());
            out.writeUTF(args[0]); // UTF is a string encoding see Sn 4.3
            String data = in.readUTF();

            System.out.println("Received: "+ data) ;

               }catch        (UnknownHostException        e){
                    System.out.println("Sock:"+e.getMessage());

       }catch (EOFException e){System.out.println("EOF:"+e.getMessage());

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

    } finally {if(s!=null) try {s.close();}catch (IOException e ) {System.out.println ("close:"
+e.getMessage()); }}}
```

TCP Server *import java.net.*; import java.io.*;*

```
public class TCPServer {

   public static void main (String args[]) { try{

            int serverPort = 7896;

            ServerSocket    listenSocket    =    new    ServerSocket(serverPort);
            while(true) {

                  Socket clientSocket = listenSocket.accept(); Connection c =
                  new Connection(clientSocket);

            }

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

```
    }
}
// this figure continues on the next slide


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

External data representation and marshalling

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

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

• The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. **An agreed standard for the representation of data structures and primitive values is called an *external data representation***.

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

Three alternative approaches to external data representation and marshalling are discussed

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

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

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

In the first two cases, the marshalling and un-marshalling activities are intended to be carriedoutbyamiddlewarelayerwithoutanyinvolvementonthepartoftheapplication programmer. In the first two approaches ,the primitive data types are marshaled into a binary form .In the third approach(XML),the primitive data types are represented textually.

CORBA's CommonDataRepresentation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0 (OMG).

CDR can represent all the data types that can be used as arguments and return values in remote invocations in CORBA.

CORBA CDR data types:

Primitive types:

1. char, boolean, 2. octet (1 byte), 3. short, 4. unsigned short,5. wchar (2 byte), 5. long, 6. unsigned long, 7. float (4 byte), 8. long long, 9. unsigned long long,10. Double 11.long double byte), any

Constructed types**:**

1. sequence,2. string, 3. array, 4. struct, 5. enumerated , 6. Union

Sun XDR (eXternal Data Representation) is another example of data exchange standard. It is developed by Sun for use in messages exchanged between clients and servers in Sun NFS.

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	1984	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

Marshalling in CORBA

Marshalling in CORBA is generated automatically from the specification of the data to be transmitted. This standard data item can be specified in CORBA IDL (Interface Definition Language) as follows:

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

The CORBA interface compiler generates appropriate marshalling and umarshalling operations for the arguments and results of remote method.

Java object serialization

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

```
public class Person implements Serializable {
private String name;
private String place;
private int year;
public Person (String n, String pl, int yr) {
        name = n;
        place = pl;
        year = yr;
} // followed by methods for accessing the instance variables. }
```

Java's object serialization

In Java, the term serialization refers to the activity of flattening an object or a connected set of objects into a serial form for storing or transmission.

Java's object de-serialization

Deserialization consists of restoring the state of an object or a set of objects from their serialized form.

- The information about a class consists of the name of the class and a version number, which is changed when major changes are made to the class.
- Java objects can contain references to other objects. These referred objects are serialized with the referring object.
- References are serialized as handles. The serialization must ensure 1-1 correspondence between object references and handles.
- To serialize an object is done as follows:
    - Its class name and version are written out, followed by its instance variables.
    - If the instance variables belong to new classes, their class information are also written out.
    - This recursive procedure continues until all necessary classes have been written out.
- The contents of primitive types are written in a portable binary format using methods of the ObjectOutputStream class.
- Strings and characters are written by its writeUTF method using Universal Transfer Format (UTF):
    - It enables ASCII characters to be represented unchanged (in one byte).
    - Unicode characters are represented by multiple bytes.
    - Strings are preceded by the number of bytes they occupy in the stream.
- The serialized form of Person p = new Person("Smith", "London", 1934) is illustrated in Figure .
- To make use of Java serialization, for example to serialize the Person object, create an instance of the class ObjectOutputStream and invoke its writeObject method, passing the Person object as argument.
- To deserialize an object from a stream of data, open an ObjectInputStream on the stream and use its readObject method to rebuild the original object.

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*
1984	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles.

Extensible Markup Lan1guage (XML)

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

- XML data items are tagged with 'markup' strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures.
- XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services.
- XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them.

XML elements and attributes • The following Figure shows the XML definition of the *Person* structure that was used to illustrate marshalling in CORBA CDR and Java.

Figure XML definition of the Person structure

<person id="123456789">

<name>Smith</name>

<place>London</place>

<year>1984</year>

<!-- a comment -->

</person >

It shows that XML consists of tags and character data. The character data, for example *Smith* or *1984*,is the actual data. As in HTML, the structure of an XML document is defined by pairs of tags enclosed in angle brackets. In above Figure, <*name*> and <*place*> are both tags.

Elements: An element in XML consists of a portion of character data surrounded by matching start and end tags. For example, one of the elements in Figure consists of the data *Smith* contained within the <*name*> ...

</*name*> tag pair. Note that the element with the <*name*> tag is enclosed in the element with the <*person id="123456789"*> ...</*person* > tag pair.

Attributes: A start tag may optionally include pairs of associated attribute names and values such as *id="123456789"*, as shown above as attributes. An element is generally a container for data, whereas an attribute issued for labeling that data. In our example, *123456789* might be an identifier used by the application, whereas *name, place* and *year* might be displayed.

Names: The names of tags and attributes in XML generally start with a letter, but can also start with an underline or a colon. The names continue with letters, digits, hyphens, underscores, colons or full stops. Letters are case-sensitive. Names that start with *xml*are reserved.

Binary data: All of the information in XML elements must be expressed as character data.

XML namespaces • Traditionally, namespaces provide a means for scoping names. An XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL. Any other

XML document can use an XML namespace by referring to its URL.

Any element that makes use of an XML namespace can specify that namespace asan attribute called *xmlns*, whose value is a URL referring to the file containing the namespace definitions. For example: *xmlns:pers = http://www.cdk5.net/person*

The name after *xmlns*, in this case *pers*can be used as a prefix to refer to the elements

in a particular namespace, as shown in following Figure. The *pers*prefix is bound to*http://www.cdk4.net/person* for the *person* element.

Illustration of the use of a namespace in the Person structure

<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">

<pers:name> Smith </pers:name>

<pers:place> London </pers:place>

<pers:year> 1984 </pers:year>

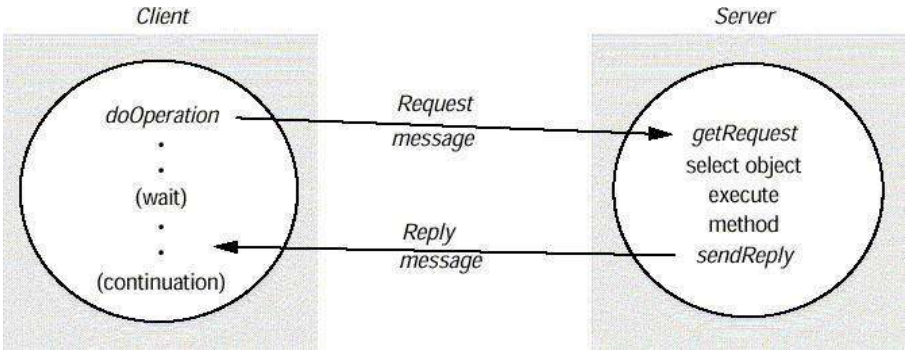
</person>

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.

1.The request-reply protocol

The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in following Figure.



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.

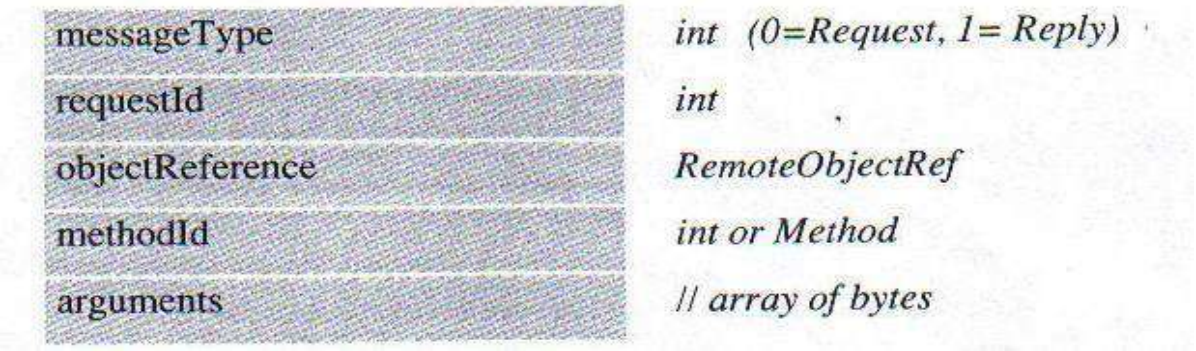
public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

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.

The following are the operations of Request-reply protocol

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



The Request-reply protocol message structure contains the following.

- 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 fourth 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 some communication failures.

- They suffer from omission failure
- Messages are not guaranteed to be delivered in sender order.

- Timeouts: There are various options as to what doOperation can do after a timeout. The simplest option is to return immediately from doOperation with an indication to the client that the doOperation has failed.The timeout may have been due to the request or reply messages getting lost and in the latter case, the operation will have been performed.
- Discarding duplicate request messages: In cases when the request message is retransmitted, the server may receive it more than once. To avoid this the protocol is designed to recognize successive messages with the same request identifier and to filter out duplicates.
- Lost reply messages: If the server has already sent the reply when it receives a duplicate request it will

need to execute the operation again to obtain the result, unless it has stored the result of the original execution. Some servers can execute their operations more than once and obtain the same results each time.

RPC exchange protocols

Three protocols are used for implementing various types of RPC.

- The request (R) protocol.
- The request-reply (RR) protocol.

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

- The request-reply-acknowledge (RRA) protocol.

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. Special acknowledgement messages are not required, because a server reply message is considered as an acknowledgement of the client's request message.

RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply. The acknowledgement reply message contains the requested from the reply message being acknowledged. This will enable the server to discard entries from its history.

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.

- Content Negotiation: Clients request can include information as to what data representation they can accept, enabling the server to choose the representation that is most appropriate for the user.
- Authentication: Credentials and challenges are used to support password style authentication on the first attempt to access a password protected area, the server reply contains a challenge applicable to the resource.

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 message to the server
- server sends a reply message to the client
- Connection is closed

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

Figure: HTTP Request message

HTTP methods

1.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.

2.HEAD

- This method is similar to GET, but only meta data on resource is returned (like date of last modification, type, and size)

3. 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

4. PUT

- Supplied data to be stored in the given URL as its identifier.

5. DELETE

- The server deletes an identified resource by the given URL on the server.

6. OPTIONS

- A server supplies the client with a list of methods.
- It allows to be applied to the given URL

7. TRACE

- The server sends back the request message

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

HTTP reply message is shown below.

Above reply message specifies

Group Communication

The pair wise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary, for example, when a service is implemented as a number of different processes in different computers, perhaps to provide fault tolerance or to enhance availability. A

multicast operation is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired

Behavior of a multicast. The simplest multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1 .*Fault tolerance based on replicated services*: A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.

2. Finding the *Discovering services 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.

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

4. *Propagation of event notifications*: Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers.

IP multicast – An implementation of multicast communication

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

Being a member of a multicast group allows a computer to receive IP packets sent to the group. The membership of multicast **groups is dynamic**, allowing computers to join or leave at any time and to join an arbitrary number of groups. It is possible to send datagram's to a multicast group without being a member.

When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

***Multicast routers*:** IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single datagram's to routers on other networks, where they are again multicast to local members. To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass – called the *time to live*, or **TTL** for short.

***Multicast address allocation*:**

Class D addresses in IPV4 (that is,addresses in the range (**224.0.0.0 to 239.255.255.255)** are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA) and address start with first 4 bit 1110 .

Failure model for multicast datagrams

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

Java API to IP multicast

The Java API provides a datagram interface to IP multicastthrough the class *MulticastSocket*, which is a subclass of *DatagramSocket* with theadditional capability of being able to join multicast groups. The class *MulticastSocket*provides two alternative constructors, allowing sockets to be created to use either aspecified local port (6789, in following figure) or any free local port. A process can join amulticast group with a given multicast address by invoking the *joinGroup()*method of itsmulticast socket. Effectively, the socket joins a multicast group at a given port and it willreceive datagrams sent by processes on other computers to that group at that port. Aprocess can leave a specified group by invoking the *leaveGroup()*method of its multicastsocket.

Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;

public class MulticastPeer{

public static void main(String args[]){

// args give message contents & destination multicast group (e.g. "228.5.6.7")
MulticastSocket s =null;

try {

InetAddress group = InetAddress.getByName(args[1]);
s = new MulticastSocket(6789);

s.joinGroup(group);

byte [] m = args[0].getBytes();

DatagramPacket messageOut =

new DatagramPacket(m, m.length, group, 6789);
s.send(messageOut);

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();}
}}
```

Reliability and ordering of multicast

A datagram sent from one multicast router to another may be lost, thus preventing all recipients beyond that router from receiving the message. Also, when a multicast on a local area network uses the multicasting capabilities of the network to allow a single datagram to arrive at multiple recipients, any one of those recipients may drop the message because its buffer is full.

Another factor is that any process may fail. If a multicast router fails, the group members beyond that router will not receive the multicast message, although local members may do so. Ordering is another issue. IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent, with the possible effect that some group members receive datagrams from a single sender in a different order from other group members. In addition, messages sent by two different processes will not necessarily arrive in the same order at all the members of the group.

Some examples of the effects of reliability and ordering • We now consider the effect of the failure semantics of IP multicast as follows

1. ***Fault tolerance based on replicated services*:** Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others. In most cases, this service would require that all members receive request messages in the same order as one another.
2. ***Discovering services in spontaneous networking*:** One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services.
3. ***Better performance through replicated data*:** Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.
4. ***Propagation of event notifications*:** The particular application determines the qualities required of multicast.

Some applications require a multicast protocol that is more reliable than IP multicast. In particular, there is a need for *reliable multicast*, in which any message transmitted is either received by all members of a group or by none of them. The examples also suggest that some applications have strong requirements for ordering, the strictest of which is called *totally ordered multicast*, in which all of the messages transmitted to a group reach all of the members in the same order.

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

Distributed Objects and Remote Invocation:

Introduction communication between distributed objects

Distributed objects[are objects that are distributed across different address spaces, either in multiple computers connected via a network or even indifferent processes on the same computer, but which work together by sharing data and invoking methods. This often involves location transparency, where remote objects appear the same as local objects.

The main method of distributed object communication is with remote method invocation

Invoking a method on a remote object is known as **remote method invocation**) generally by message-passing

Message-passing: one object sends a message to another object in a remote machine or process to perform some task. The results are sent back to the calling object.

The remote procedure call (RPC) approach extends the common programming Abstraction of the procedure call to distributed environments, allowing a calling Process to call a procedure in a remote node as if it is local.
Remote method invocation (RMI) is similar to RPC but for distributed objects,with
Added benefits in terms of using object-oriented programming concepts in Distributed systems and also extending the concept of an object reference to the Global distributed environments, and allowing the use of object references as Parameters in remote invocations
Remote procedure call – client calls the procedures in a server program that is running in a different process

Remote method invocation (RMI) – an object in one process can invoke methods of objects in another process

Event notification – objects receive notification of events at other objects for which they have registered
Middleware Roles

provide high-level abstractions such as RMI enable location transparency free from specifics of communication protocols

operating systems and communication hardware

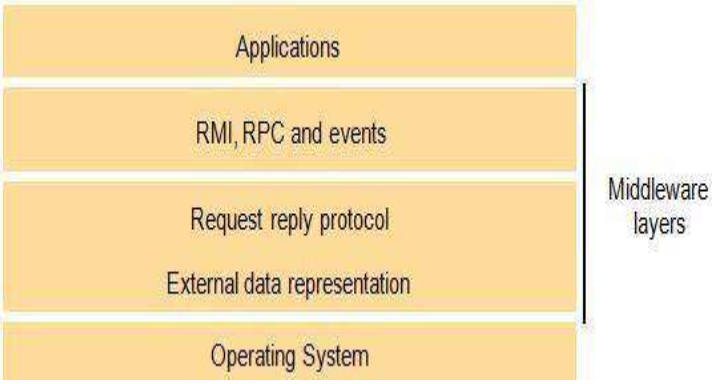


Fig middle ware layer

Communication between distributed objects and other objects

Life cycle : Creation, migration and deletion of distributed objects is different from local objects

Reference : Remote references to distributed objects are more complex than simple pointers to memory addresses

Request Latency : A distributed object request is orders of magnitude slower than local method invocation

Object Activation : Distributed objects may not always be available to serve an object request at any point in time

Parallelism: Distributed objects may be executed in parallel.

Communication : There are different communication primitives available for distributed objects requests

Failure: Distributed objects have far more points of failure than typical local objects.

Security: Distribution makes them vulnerable to attack.

Distributed object model:

The term **distributed objects** usually refers to software modules that are designed to work together, but reside either in multiple computers connected via a network or in different processes inside the same computer.

Distributed objects

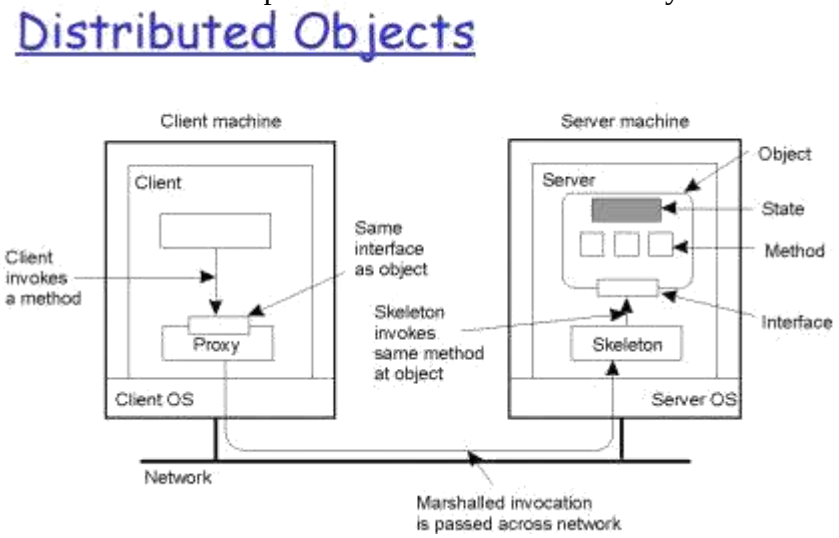
The state of an object consists of the values of its instance variables since object-based programs are logically partitioned, the physical distribution of objects into different

processes or computers in a distributed system. Distributed object systems may adopt the client-server architecture. objects are managed by servers and their clients invoke their methods using remote method invocation.

In RMI, the client‘s request to invoke a method of an object is sent in a message to the server managing the object. The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message

Distributed objects can assume other architectural models. For example, objects can be replicated in order to obtain the usual benefits of fault tolerance and enhanced performance, and objects can be migrated with a view to enhancing their performance and availability.

Another advantage of treating the shared state of a distributed program as a collection of objects is that an object may be accessed via RMI, or it may be copied into a local cache and accessed directly, provided that the class implementation is available locally.



Design issues of RMI

RMI Invocation Semantics:

Invocation semantics depend upon implementation of Request Reply Protocol used by RMI

It maybe, used At-least-once, At-most-once

Transparency:

Partial failure, higher latency, Different semantics for remote objects,

For e.g. wait/notify Current consensus: remote invocations should be made transparent in the sense that syntax of a remote invocation is the same as the syntax of local invocation (access

transparency) but programmers should be able to distinguish between remote and local objects by looking at their interfaces, e.g. in Java RMI, remote objects implement the Remote interface

Issues in implementing RMI

Parameter passing

Request reply protocol (handling failures at client and server)

Supporting constant objects, object adapters, dynamic invocations, etc

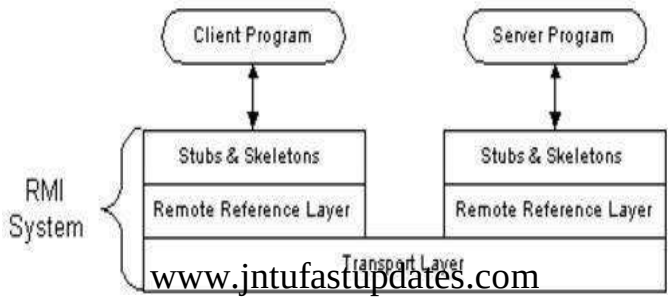
The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model. RMI architects have succeeded; creating a system that extends the safety and robustness of the Java architecture to the distributed computing world.

The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.
This fits nicely with the needs of a distributed system where clients are concerned about the definition of a service and servers are focused on providing the service.
Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class.

Therefore, the key to understanding RMI is to remember that interfaces define behavior and classes

Implementation of RMI:

The RMI implementation is essentially built from three abstraction layers. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.
The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via *Remote Object Activation*.
The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



Distributed Garbage collection

Distributed garbage collection (DGC) in computing is a particular case of **garbage collection** where references to an object can be held by a remote client.

One of the joys of programming for the Java platform is not worrying about memory allocation. The JVM has an automatic garbage collector that will reclaim the memory from any object that has been discarded by the running program.
One of the design objectives for RMI was seamless integration into the Java programming language, which includes garbage collection. Designing an efficient single-machine garbage collector is hard; designing a distributed garbage collector is very hard.
The RMI system provides a reference counting distributed garbage collection algorithm based on Modula-3's Network Objects.
This system works by having the server keep track of which clients have requested access to remote objects running on the server. When a reference is made, the server marks the object as "dirty" and when a client drops the reference; it is marked as being "clean."

DGC uses some combination of the classical garbage collection (GC) techniques, tracing and reference counting. It has to cooperate with local garbage collectors in each process in order to keep global counts, or to globally trace accessibility of data.

In general, remote processors do not have to know about internal counting or tracing in a given process, and the relevant information is stored in interfaces associated with each process.

DGC is complex and can be costly and slow in freeing memory. One cheap way of avoiding DGC algorithms is typically to rely on a time lease set or configured on the remote object; it is the stub's task to periodically renew the lease on the remote object.

If the lease has expired, the server process (the process owning the remote object) can safely assume that either the client is no longer interested in the object, or that a network partition or crash obstructed lease renewal, in which case it is "hard luck" for the client if it is in fact still interested.

Hence, if there is only a single reference to the remote object on the server representing a remote reference from that client, that reference can be dropped, which will mean the object will be garbage collected by the local garbage collector on the server at some future point in time.

Distributed systems typically require distributed garbage collection. If a client holds a proxy to an object in the server, it is important that the server does not garbage-collect that object until the client releases the proxy. Most third-party distributed systems, such as RMI, handle the distributed garbage collection, but that does not necessarily mean it will be done efficiently. The overhead of distributed garbage collection and remote reference maintenance in RMI can slow network communications by a significant amount when many objects are involved.

Of course, if you need distributed reference maintenance, you cannot eliminate it, but you can

reduce its impact. You can do this by reducing the number of temporary objects that may have

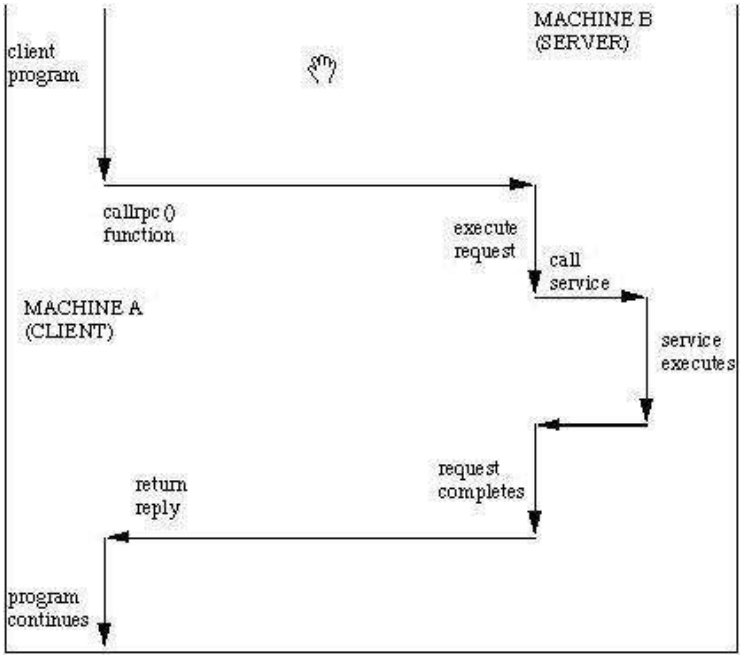
distributed references. The issue is considerably more complex in a multiuser distributed environment, and here you typically need to apply special optimizations related to the products you use in order to establish your multiuser environment.

Remote Procedure Call (**RPC**) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. (A procedure call is also sometimes known as a function call or a subroutine call.) **RPC** uses the client/server model.

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.

The flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out.

When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.



Remote Procedure Calling Mechanism A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) the program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously. Each version contains a number of procedures that can be called remotely. Each procedure has a procedure number.

Events and notification

Events of changes/updates...

notifications of events to parties interested in the events

publish events to send

subscribe events to receive

main characteristics in distributed event-based systems:

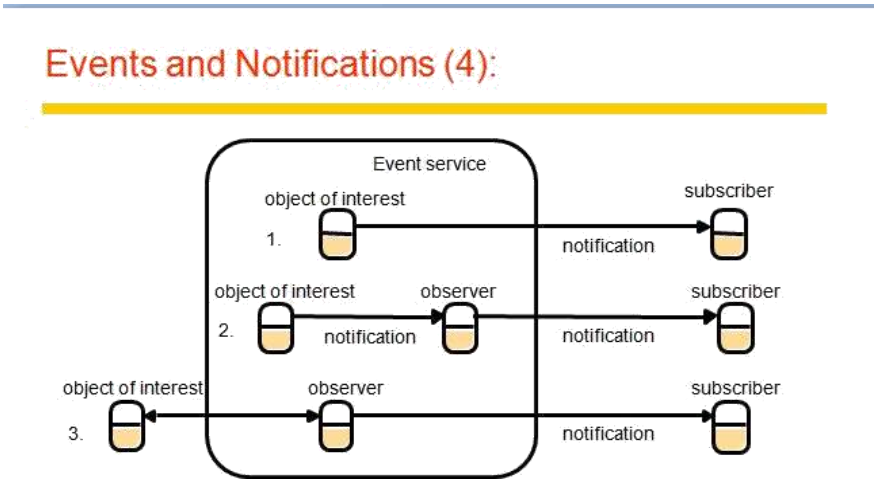
a way to standardize communication in heterogeneous systems (not designed to communicate directly)

asynchronous communication (no need for a publisher to wait for each subscriber-- subscribers come and go)

event types

each type has attributes (information in it)

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



Publish-subscribe paradigm: publisher sends notifications, i.e. objects representing events

‹ Subscriber registers interest to receive notifications

The object of interest: where events happen, change of state as a result of its operations being invoked „ Events: occurs in the object of interest „

Notification: an object containing information about an event

Subscriber: registers interest and receives notifications „

publisher: generate notifications, usually an object of interest

Observer objects: decouple an object of interest from its subscribers (not important)

Case study JAVA RMI

server program main program: binding instances of servant classes

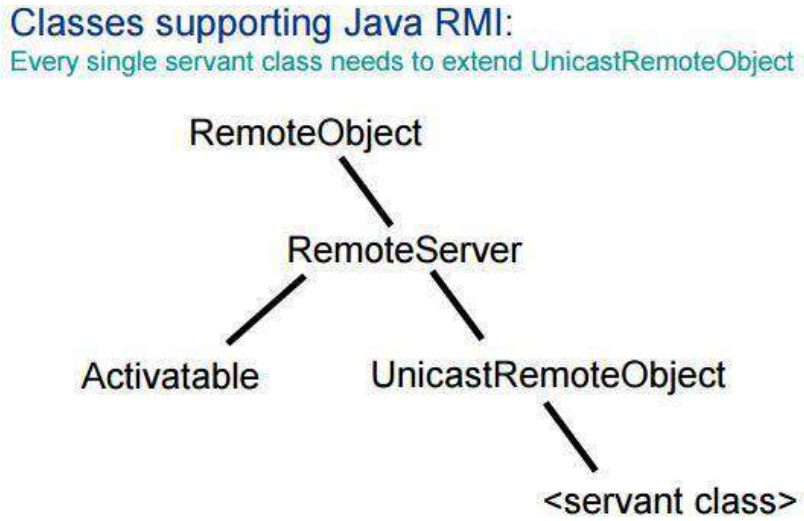
main method needs to create a security manager to enable Java security. A default security manager, RMISecurityManager, is provided

Note: if an RMI server sets no security manager, proxies and classes can only be loaded from the local classpath, in order to protect the program from code that is downloaded as a result of remote method invocations.

servant classes: ShapeList Servant and Shape Servant, implementing ShapeList and Shape interfaces respectively ‹ servant classes need to extend

UnicastRemoteObject, which provides remote object that live only as long as the process in which they are created

implementation of servant classes are straightforward, no concern of communication details



UnicastRemote Object:

automatically creates socket and listens for network requests, and make its services available by exporting them.

RMISecurityManager (): Needed to download objects from network. The downloaded objects are allowed to communicate only with sites they came from.

Default security manager, when none is explicitly set, allows only loading from local file system

Reflection: the class of an object can be determined at runtime, and this class can be examined to determine which methods are available, and even invoke these methods with dynamically created arguments „

The key to reflection is the java.lang.Class, which allows much information to be determined about a class. This leads onto the other reflection classes such as java.lang.reflect.Method

Heterogeneity is an important challenge to designers: ‹ Distributed systems must be constructed from a variety of different networks, operating systems, computer hardware and programming languages. The Internet communication protocols mask the difference in networks and middleware can deal with the other differences. „

External data representation and marshalling ‹
CORBA marshals data for use by recipients that have prior knowledge of the types of its components. It uses an IDL specification of the data types

Java serializes data to include information about the types of its contents, allowing the recipient to reconstruct it. It uses reflection to do this. „ RMI ‹

Each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely. ‹

local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once ‹

Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers

Unit – IV – Operating System Support

Introduction: In distributed systems, the middleware applies to a software layer that providesa programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. Some middleware, such as java Remote Method Invocation (RMI) supports only a single programming language. But all middleware deals with the operating system and hardware.

The middleware layer uses protocols based on messages b/w processes to provide its higher-level abstractions. The main important aspect of middleware is the provision of location transparency and independence from the details of communication protocols, operating systems and the computer hardware.

The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources such as the processors, memory, communications and storage media. An operating system such as UNIX or Windows-XP provides the programmer with files rather than disk blocks, and with sockets rather than raw network access. It takes over the physical resources on a single node and manages them to present these resources abstractions through the system-call interface.

Both UNIX and Windows-XP are examnples of *network operating systems*. They have a networking capability built into them and so can be used to access remote resources. With the network operating system, a user can remotely log in to another computer, using *rlogin* or *telnet*, and run processes there.

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

Middleware and network operating system: In fact, there is no distributed operating systems in general use, only network operating system such as UNIX, Mac OS, and Windows-XP. Because of these following two reasons.

The first reason is that the users have much invested in their application software. So they refused to adopt new operating system, whatever efficiency advantages it offers.

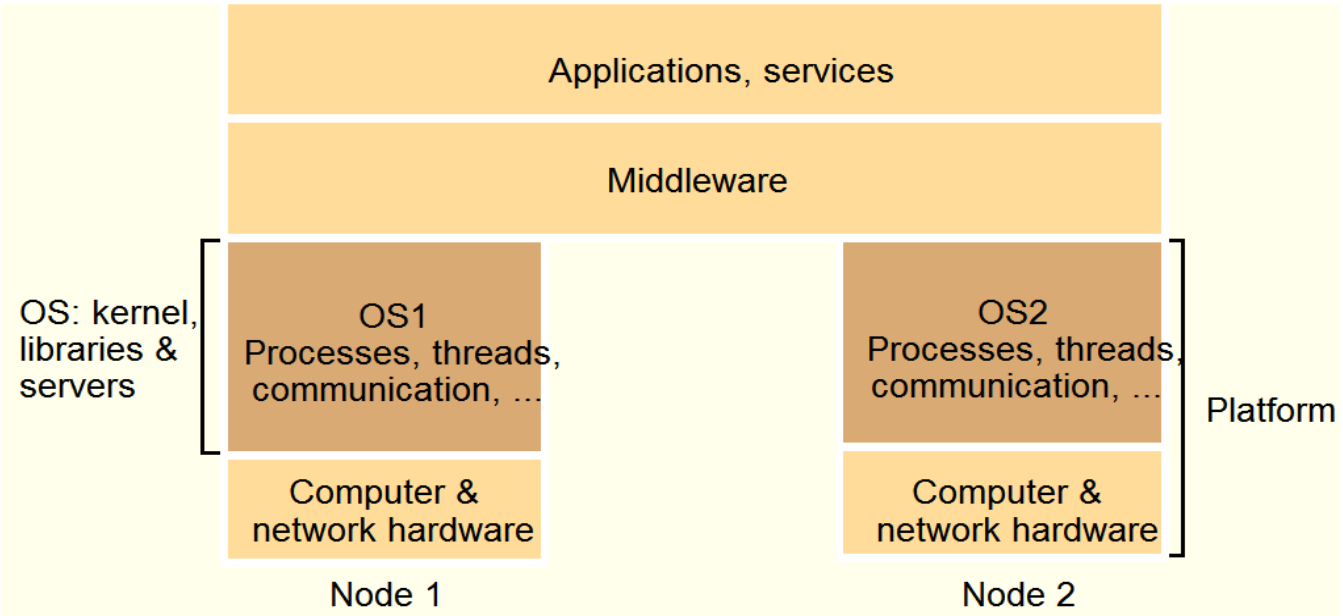
The second reason is that users tend to prefer to have a degree of autonomy for their machines because of its performance.

For example, vasanthi needs good interactive responsiveness while she writes her documents and would dislike it if srilatha's programs were slowing her down.

The combination of middleware and network operating systems provides an acceptable balance b/w the requirement for autonomy on the one hand, and network-transparent resource access on the other.

The network operating system enables users to run their favourite word processor and other standalone applications. Middleware enables them to take advantage of services that become available in their deistributed system.

The Operating system layer: The good performance of middleware and OS combination, the users are satisfied to distribute the data in distributed networking technology. In this case, middleware runs on a variety of OS platforms at the nodes of a distributed system. The OS running at a node with a librariy service which provides local hardware resources for processing, storage and communication.



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

→Middleware and the Operating System

- Middleware implements abstractions that support network-wide programming. Examples:
    - RPC and RMI (Sun RPC, Corba, Java RMI)
    - event distribution and filtering (Corba Event Notification, Elvin)
    - resource discovery for mobile and ubiquitous computing
    - support for multimedia streaming
- Traditional OS's (e.g. early Unix, Windows 3.0)
    - simplify, protect and optimize the use of local resources
- Network OS's (e.g. Mach, modern UNIX, Windows NT)
    - do the same but they also support a wide range of communication standards and enable remote processes to access (some) local resources (e.g. files).

→ Combination of middleware and network OS

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

→ The relationship between OS and Middleware

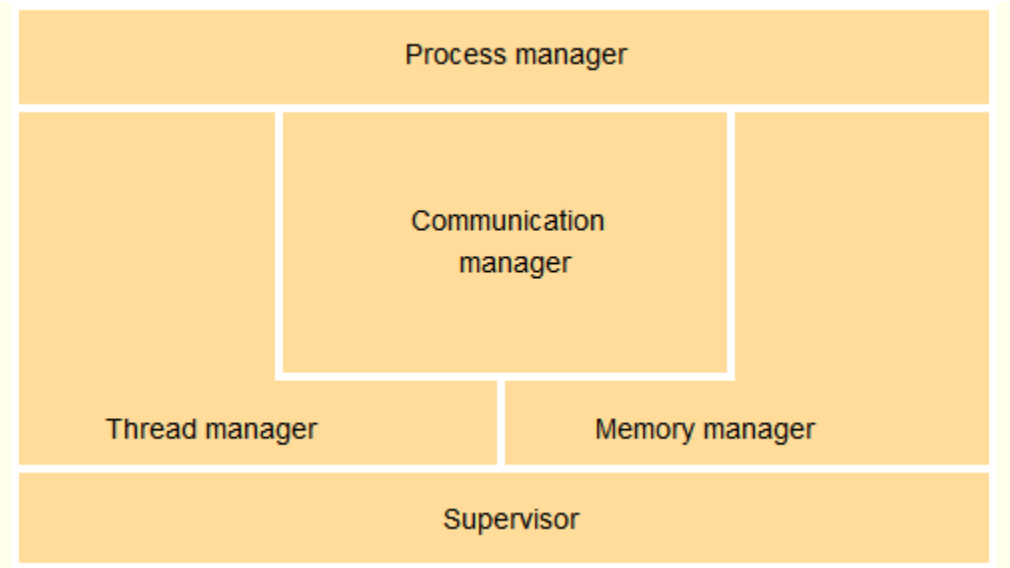
- The tasks of Operating System are processing, storage and communication
- The Components of Operating System are kernel, library, user-level services

- Middleware
    - It runs on a variety of OS-hardware combinations
    - remote invocations

→ **Functions that OS should provide for middleware**

- Encapsulation
  - provide a set of operations that meet their clients' needs
- Protection
  - Protect resource from illegitimate access. For example, files are protected from being read by users without read permissions, and devices registers are protected from application processes.
- Concurrent processing
  - support clients access resource concurrently
- Invocation mechanism: a means of accessing an encapsulated resource
  - Communication
    - Pass operation parameters and results between resource managers
  - Scheduling
    - Schedule the processing of the invoked operation

Core OS functionality:



- **Process manager**
  - Handles the creation of and operations upon processes. A process is unit of resource management, including an address space and one or more threads.
- **Thread manager**
  - Thread creation, synchronization and scheduling. Threads are schedulable activities attached to processes and are fully described.
- **Communication manager**
  - Communication between threads attached to different processes on the same computer. Why because, some kernels also support communication b/w threads in remote processes. Other kernels require additional service for external communication.

- **Memory manager**
  - Management of physical and virtual memory.
- **Supervisor**
  - Dispatching of interrupts, system call traps and other exceptions
  - control of memory management unit and hardware caches
  - Processor and floating point unit register manipulations.

Shared-memory multiprocessors: Shared-memory multiprocessor computers are equipped with several processors that share one or more modules of memroy (RAM). The processors may also have their own private memory. Multiprocessor computers can be constructed in a variety of forms.

In the common *symmetric processing architectre*, each processor executes the same kernel and the kernels paly largely equivalent roles in managing the hardware resources. The kernels share key data structures such as the queue of runnable threads, but some of their working data is private. Each processor can execute a thread simultaneously, accessing data in the shared memory, which may be private or shared with other threads.

Multiprocessors can be used for many high-performance computing tasks. In distributed systems, they are particularly useful for the implementation of high-performance servers because the server can run a single program with serval threads that handle several requests from clients simultaneously. For example providing access to a shared database.

Protection: The resources require protection from illegitimate (illegal) access. For example,the file can opened in two operations, such as *read* and *write*.

Therefore files are protected from being read by users without read permissions, and devices registers are protected from application processes.

The other type of illegitimate access is address of file, called file pointer. **For example,** smith managed to access file pointer variable directly. So she constructs a (*setFilePointerRandomly*) operation, which sets the file pointer to a random number.

Thus, Access the file pointer variable directly (*setFilePointerRandomly*) i.e Non-type-safe language vs. type-safe language.

- Type–safe language, e.g. Java or Modula-3.
- Non-type-safe language, e.g. C or C++

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. In particular, it can control the memory management unit and set the processor registers so that no other code may access the machine's physical resource 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.

- **Different execution mode**
  - *An address space:* a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, e.g.: read only or read-write
  - *supervisor* mode (*kernel process*) / *use*r mode (*user process*)

- Interface between kernel and user processes: system call trap
- The 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.
- 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.

- **The price for protection**
  - switching between different processes take many processor cycles
  - a *system call trap* is a more expensive operation than a simple method call
  - 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.

Processes and Threads:

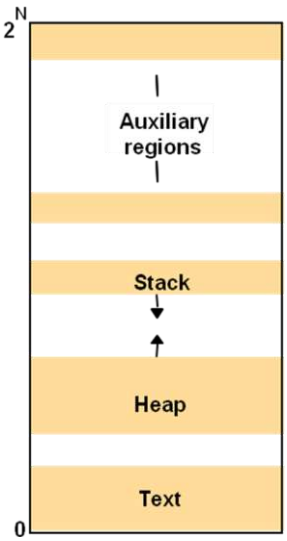
- **Process:** A process is nothing but an execution of program.
  - Problem: sharing between related activities is awkward and expensive.
  - Solution: To enhance the notion of process so that it could be associated with multiple activities.
  - Nowadays, a process consists of an *execution environment* together with one or more *threads*
- **Execution environment**
  - An execution environment is the unit of resource management. It means, a collection of local kernel managed resources to which its threads have access.
  - An execution environment represents the protection domain in which its threads execute.
  - Execution environment consist of
    - An address space
    - Thread synchronization and communication resources such as semaphores and communication interfaces (e.g. sockets)
    - Higher-level resources such as open files and windows
  - Shared by threads within a process
- **Thread:** A *process* is divided into number of smaller tasks; each task is called a "*Thread*". A *thread* is the operating system abstraction of an activity. Threads can be created and destroyed dynamically as needed. The central aim of multiple threads execution is to maximize the degree of concurrent execution b/w operations such as enabling the overlap of

computation with input and output, and enabling concurrent processing on multiprocessors. This can be helpful within servers.

For example one thread can process a client's request while a second thread servicing another request waits for a disk access to complete.

- Benefits
  - Responsiveness, Resource sharing, Economy, Utilization of MP architectures

Address spaces



- An address space is a unit of management of *a process*'s virtual memory
- It is large up to 2^{32} bytes and sometimes up to 2^{64} bytes
- It consists of one or more regions
- **Region**
- an area of continuous virtual memory that is accessible by the threads of the owning process
- Each region is specified by the following Properties:
  - Its extent i.e. lowest virtual address and size.
  - Read/write/execute permissions for the process's threads.
  - Whether it can be grown upwards or downwards.
- **The number of regions is indefinite**
  - One of these is the need to support a separate stack for each thread. Allocating a separate stack region to each thread makes it possible to detect and exceed the stack limits and control each stack's growth.
  - Another one is to map the address space. Access the map as array of bytes in memory is called *mapped file.*
  - Share memory between processes or b/w processes and the kernel is another factor leading to extra regions in the address space.
- **Region can be shared with the following uses:**
  - **Libraries**: Library code can be shared by being mapped as a region in the address space of processes that required it.
  - **Kernel**: Kernel code and data are mapped into every address space at the same location.
  - **Shared data and communication**: It is more efficient for the data to be shared by being mapped as regions in both address spaces than by being passed in messages b/w them.

    Creation of new process in distributed system: The creation of a new process has traditionally been an indivisible operation provided by the operating system. For example, the UNIX *fork* system call creates a process with an execution environment copied from the caller. The UNIX *exec* system call transforms the calling process into one executing the code of a named program.

For a distributed system, the design of the process creation mechanism has to take account of the utilization of multiple computers.

The creation of 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 computer server.
- The creation of an execution environment.

***Choice of* Process host:** Process allocation policies range from always -

- running new processes at their originator's computer
- sharing processing load between a set of computers

- Policy categories for load sharing:

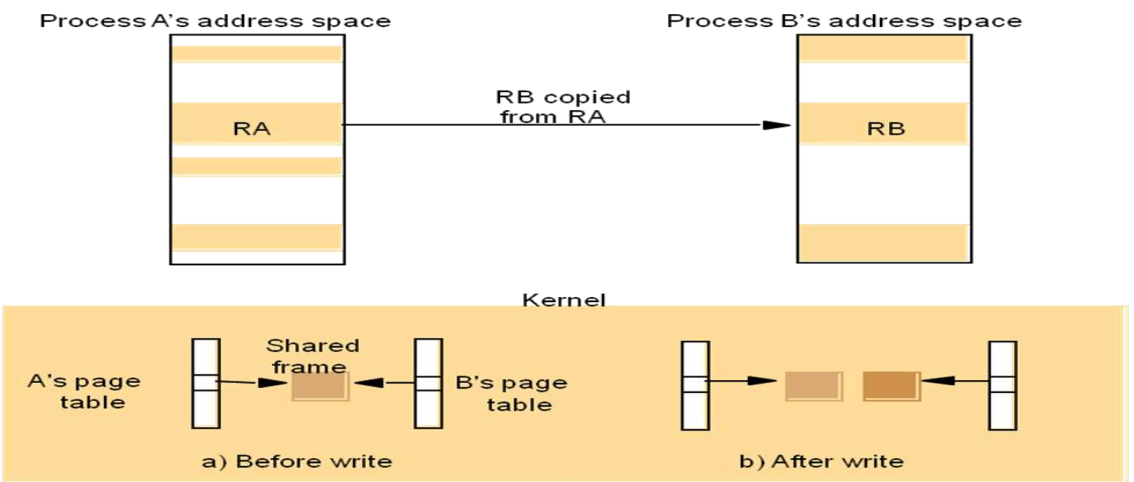
- The ***transfer policy*** determines whether to situate a new process locally or remotely. This may depend on whether the local node is lightly or heavily loaded.
- The ***location policy*** determines which node should host a new process selected for transfer. This decision may depend on the relative loads on their machine architectures and on any specialized resources they may possess.
- The ***migratory*** load-sharing systems can shift load at any time, not just when a new process is created. Use of this mechanism is called *process migration.*

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 *statically defined format*. For example, it could contain just a program text region, heap region and stack region.
- The second approach is the address space can be with respect to an existing execution environment, e.g. *fork*

Copy-on-write: It is a general technique. It is also used in copying large messages etc. For example, regions RA and RB's memory is shared copy-on-write b/w two processes, A and B. This is shown in fig. Let us assume that process A set region RA to be copy- inherited by its child, process B, and that the region RB was created in process B.



Threads: A process is divided into number of smaller tasks, each task is called a "*Thread*". Number of threads with in a process execute at a time is called "multithreading". Supporting of multithreading is the additional capability of an operating system. Based on the functionality, threads are divided into 4 categories:

1. Responsiveness: It increases responsiveness to the user when a multithreaded web browser are using.

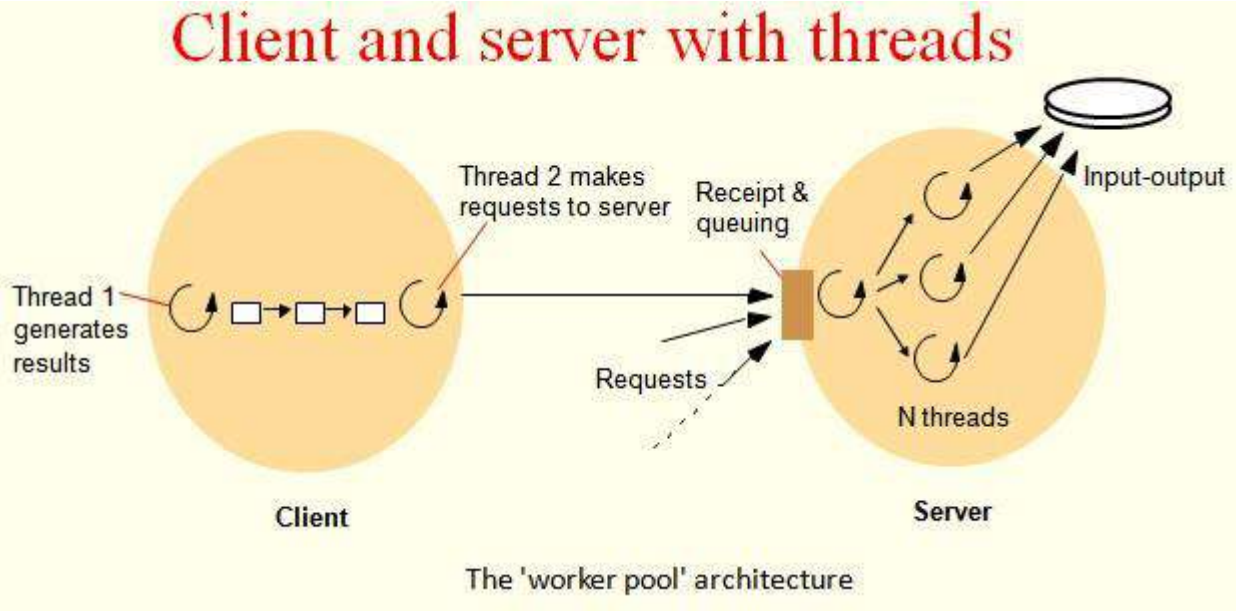
2. Resource sharing: Threads share the memory and the resources of the process to execute the application. In this case, the address space of shared memory allows several different threads of activity within the same address space.

3. Economy: Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong. For example, creating a process is about thirty times slower than is creating a thread and context switching is about five time slower.

4. Utilization of multiprocessor architectures: The benefits of multithreading can be greatly increased in a multiprocessor architecture. A single threaded process can only run on one CPU, but not more. Multithreading on a multi-CPU machine increases concurrency.

Architectures for multi-threaded servers:

Consider the server shown in fig that the server has a pool of one or more threads, each of which repeatedly removes a request from a queue of received requests and processes it.



We assume that each thread applies the same procedure to process the requests. Let us assume that each request takes 2 milliseconds of processing and 8 milliseconds of I/O (input/output) delay when the server reads from a disk. Let us further assume for the moment that the server executes at a single-processor computer.

Consider the maximum server throughput is measured in client requests handled per second for different numbers of threads. Therefore if a single thread has to perform all processing, then the turnaround time for handling any request is on average 2 + 8 = 10 milliseconds. So this server can handle 100 client requests per second.

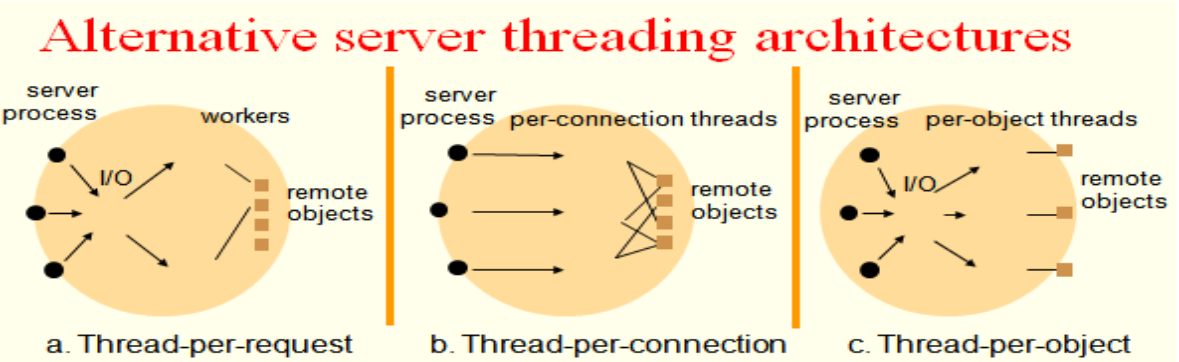
Now consider if the server pool contains two threads. Let us assume that one thread can be scheduled when another becomes blocked for I/O. This increases the server throughput. For

example, if all disk requests are serialized and take 8 milliseconds each, then the maximum throughput is 1000/8 = 125 requests per second.

The figure shows one of the possible threading architectures and that described how multi-threading enables servers to maximize their throughput and measured as the number of requests processed per second.

The figure shows one of the possible threading architecture is the *worker pool architecture.* In it's, the server creates a fixed pool of 'worker' threads to process the requests when it starts up. The module marked 'receipt and queuing' in figure is implemented by an 'I/O' thread which receives requests from a collection of sockets or ports and places them on a shared request queue for retrieval by the workers.

Alternative server threading architectures:



In the ***thread-per-request architecture*** the I/O thread spawns a new worker thread for each request, and that worker destroys it when it has processed the request against its designated remote object. This architecture has the advantage that the threads do not attend for a shared queue, and throughput is potentially maximized because I/O thread can create many workers by outstanding requests. Its disadvantage is the overhead of the thread creation and destruction operation.

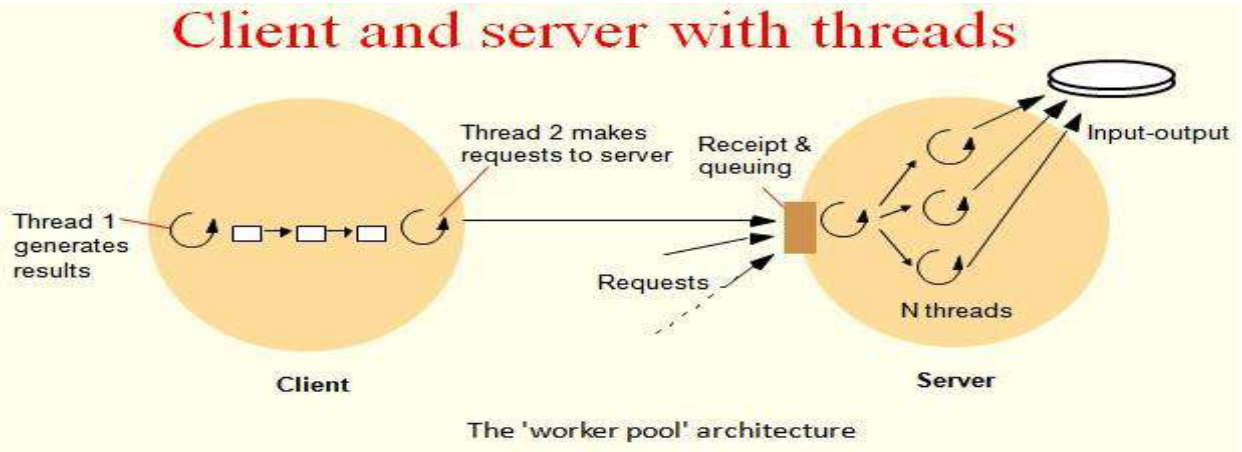
The ***thread-per-connection architecture*** associates a thread with each connection. The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection. In b/w, the client may make many requests over the connection, targeted at one or more remote objects.

The ***thread-per-object architecture*** associates a thread with each remote object. An I/O thread receives requests and queues them for the workers, but this time there is a per-object queue.

In each of these last two architectures the server benefits from lowered thread-management overheads compared with the thread-per-request architecture. The disadvantage is that clients may be delayed while a worker thread has several outstanding requests but another thread has no work to perform.

Threads within clients: Threads can be useful for clients as well as servers. The figure shows a client process with two threads.

- The first thread generates results to be passed to a server by remote method invocation, but does not require a reply and able to continue computing further results.
- The second thread incorporates the client process and performs the remote method invocation and blocks.
- The first thread places its results in buffers, which are emptied by the second thread. It is only blocked when all the buffers are full.



Threads versus multiple processes: Multi-threaded clients are evident in the examples of web browsers.

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

State associated with execution environments and threads

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

Threads Programming: Threads programming is concurrent programming. Much threads programming is done in a conventional language such as C with thread for library. The C threads package developed for the Mach operating system, the POSIX thread of IEEE 1003, pthreads, etc.

Thread(ThreadGroup group, runnable target, string name)

run()

start()

Thread lifetimes: A new thread is created on the same Java virtual machine (JVM) as its creator, in the SUSPENDED state. After it is made RUNNABLE with the ***start()*** method, it executes the ***run()*** method of an object designated in its constructor. The JVM and the threads on top of it all execute in a process on top of the underlying operating system. A thread ends its life when it returns from he **run()** method, or when its ***destroy()*** method is called.

Programs can manage threads in groups. Every thread belongs to one group, which it assigned at the time of its creation. Thread groups are useful when several applications coexist on the same JVM.

Thread groups also facilitate control of the relative priorities of threads. This is useful for browsers running applets, and for web servers running programs called *servlets*, which create dynamic web pages.

Java Thread Constructors and management methods

Thread (ThreadGroup group, runnable target, string name)

Creates a new thread in the SUSPENDED state, which will belong to group and be identifier as name the thread will execute the run() method of target.

setPriority (int newPriorty), getPriority()- *Sets and returns the thread priority.*

run() - *A thread executes the run() method of its target object if it has one and otherwise its own* ***run()-*** *method i.e; thread implements runnable.*

start()- *Changes the state of the thread from SUSPENDED to RUNNABLE.*

sleep(int milisecs)*-Causes the thread to enter the SUSPENDED state for the specified time.*

yield () *- Causes the thread to enter the READY state and invokes the scheduler.*

destroy()*- Destroys the thread.*

Thread synchronization: Programming a multi-threaded process requires great care. The main difficult issues are sharing of objects and the techniques used for thread coordination and cooperation.

Java provides the *synchronized* keyword for programmers to designate the well-known monitor construct for thread coordination. Programmers designate either entire methods or arbitrary blocks of code as belonging to a monitor associated with an individual object. The monitor's guarantee is that at most one thread can execute within it at any time.

Java thread synchronization calls:

Thread.join(int millisecs)

Blocks the calling thread for up to the specified time until thread has terminated.

Thread.interrupt()

Interrupts thread: causes it to return from a blocking method call such as sleep()

Object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to notify() or notifyAll() on object wakes the thread, or the thread is interrupted, or the specified time has elapsed.

Object-notify(),object.notifyAll()

Wakes, respectively, one or all of any threads that have called wait() on object.

Thread scheduling: The thread scheduling are

1. *Preemptive scheduling****:*** In this, a thread may be suspended at any point to make way for another thread, even when the preempted thread would otherwise continue running.

2. *non-preemptive scheduling*: In this a thread runs until it makes a call to the threading system, when the system may de-schedule it and schedule another thread to run.

The advantage of non-preemptive scheduling is that any section of code that does not contain a call to the threading system is automatically a critical section. But it cannot allow in multiprocessor.

Threads Implementation: Threads can be implemented in many kernels to support multi- threaded processes. For example, windows, Linux, Solaris, Mach and Chorus. These kernels provide thread creation and management system calls, and they schedule individual threads. The multi-threaded processes must be implemented in a library of procedures linked to application programs. In this case, user-level threads cannot schedule them independently.

Scheduler activations:

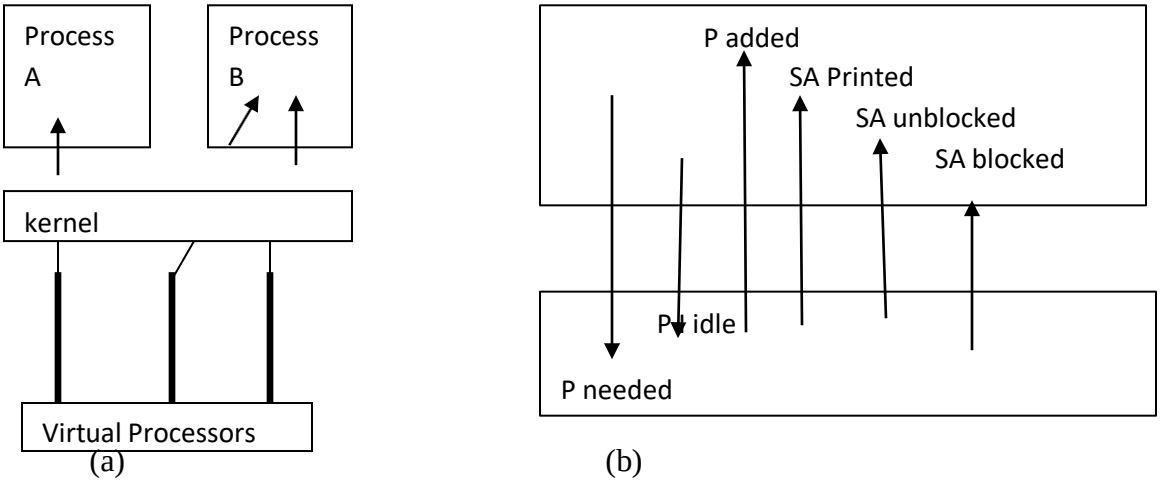


Fig (a): Assignment of Virtual processors to processes

Fig (b): Events between user level scheduler and kernel key: P=Processor; SA= Scheduler Activation

Figure (a) shows that as process notifies the kernel when either of two types of event occurs: when a virtual processor 'idle' and no longer needed ,or an extra virtual processor is required.

Figure(b) shows that the kernel notifies the process when any of four types of events occurs. A scheduler activation(SA) is a call from kernel to the process, which notifies the process's schedulers of an event. Entering the body of code from a lower layer(kernel) in this way is some times called an upcall.

The four types of event that kernel notifies user level scheduler are as follows:

Virtual processor allocated: The kernel has assigned a new virtual processor to the process, and this is the first timeslice upon it, the scheduler can load SA with the context of READY thread, which can thus recommend execution.

SA blocked : An SA has blocked in the kernel, and the kernel is using a fresh SA to notify the scheduler, scheduler sets the state if the corresponding thread to BLOCKED and can allocate a READY thread to notifying SA.

SA unblocked: An SA that was blocked in the kernel has become unblocked and is ready to execute at user level again, the scheduler can now return the corresponding thread to the READY list.

SA preempted: The kernel has taken away the specified SA from the process , the scheduler place the preempted thread in the READY list and revaluates the thread allocations.

Communication and invocation: Communication is the part of the implementation of an invocation. It means, a construct such as a remote method invocation, remote procedure call or event notification, whose purpose is to bring an operation on a resource in a different address space.

Communication:

Communication primitives: Some kernels designed for distributed systems have provided communication primitives to the all types of invocation. For example,

- TCP(UD
- P) Socket in Unix and Windows
- *DoOperation*, *getRequest*, *sendReply* in Amoeba
- Group communication primitives in V system

For example, if middleware provides RMI over UNIX's connected (TCP) sockets, then a client must make two communication system calls for each remote invocation. The principal reasons for using sockets are portability and interoperability.

Protocols and openness: One of the main requirements of the operating system is to

- Provide standard protocols that enable internetworking between middleware implementations on different platforms.
- integrate novel low-level protocols without upgrading their application
- Protocols are normally arranged in a stack of layers.
    - Static Stack: Many operating systems allow new layers to be integrated statically and permanently installed protocol as a "driver"
    - Dynamic stack
        - protocol stack be composed on the fly
        - E.g. web browser utilize wide-area wireless link on the road and faster Ethernet connection in the office

Invocation performance: Invocation performance is a critical factor in distributed system design. The more designers separate functionality b/w address spaces, the more remote invocations are required. Clients and servers may make many millions of invocation-related operations in their lifetimes, so that small fractions of milliseconds count in invocation costs.

- **Invocation costs:** Calling a conventional procedure or method, making a system call, sending a message, remote procedure calling and remote method invocations are all examples of invocation mechanisms. Each mechanism code to be executed out of scope of the calling procedure or object. This code can be helped to communicate the arguments and return the data values to the caller. This mechanism can be either synchronous or can be asynchronous.
- **Invocation over the network:** A *null* RPC is defined as an RPC without parameters that executes a null procedure and returns no values. Its execution involves an exchange of messages carrying some system data but no user data.
    - Delay: the total RPC call time experienced by a client
    - Latency: the fixed overhead of an RPC is measured by null RPC
    - Throughput: the rate of data transfer between computers in a single RPC
    - An example
        - Threshold: one extra packet to be sent, might be an extra acknowledge packet is needed

The following are the main components accounting for remote invocation delay.

- **Marshalling:** It involves on copying and converting data.

- **Data copying:** Potentially message data is copied several times in the course of an RPC

The performance of RPC and RMI mechanisms is critical for effective distributed systems.

- Typical times for 'null procedure call':
  - Local procedure call< 1 microseconds
  - Remote procedure call ~ 10 milliseconds
- 'Network time' (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only .01 millisecond; the remaining delays must be in OS and middleware - latency, not communication time.

- **Factors affecting RPC/RMI performance**
  - marshalling/unmarshalling + operation despatch at the server
  - data copying:- application -> kernel space -> communication buffers
  - thread scheduling and context switching:- including kernel entry
  - protocol processing:- for each protocol layer
  - network access delays:- connection setup, network latency

Improve the performance of RPC

-Memory sharing: Shared regions may be used for rapid communication between processes and the kernel or b/w user processes. Data is communicated by *writing to* and *reading from* the shared region.

-Choice of protocol: If delay occurs during the request-reply interaction, then increase the performance of RPC for TCP.

- TCP/UDP E.g. Persistent connections: several invocations during one
- OS's buffer collect several small messages and send them together

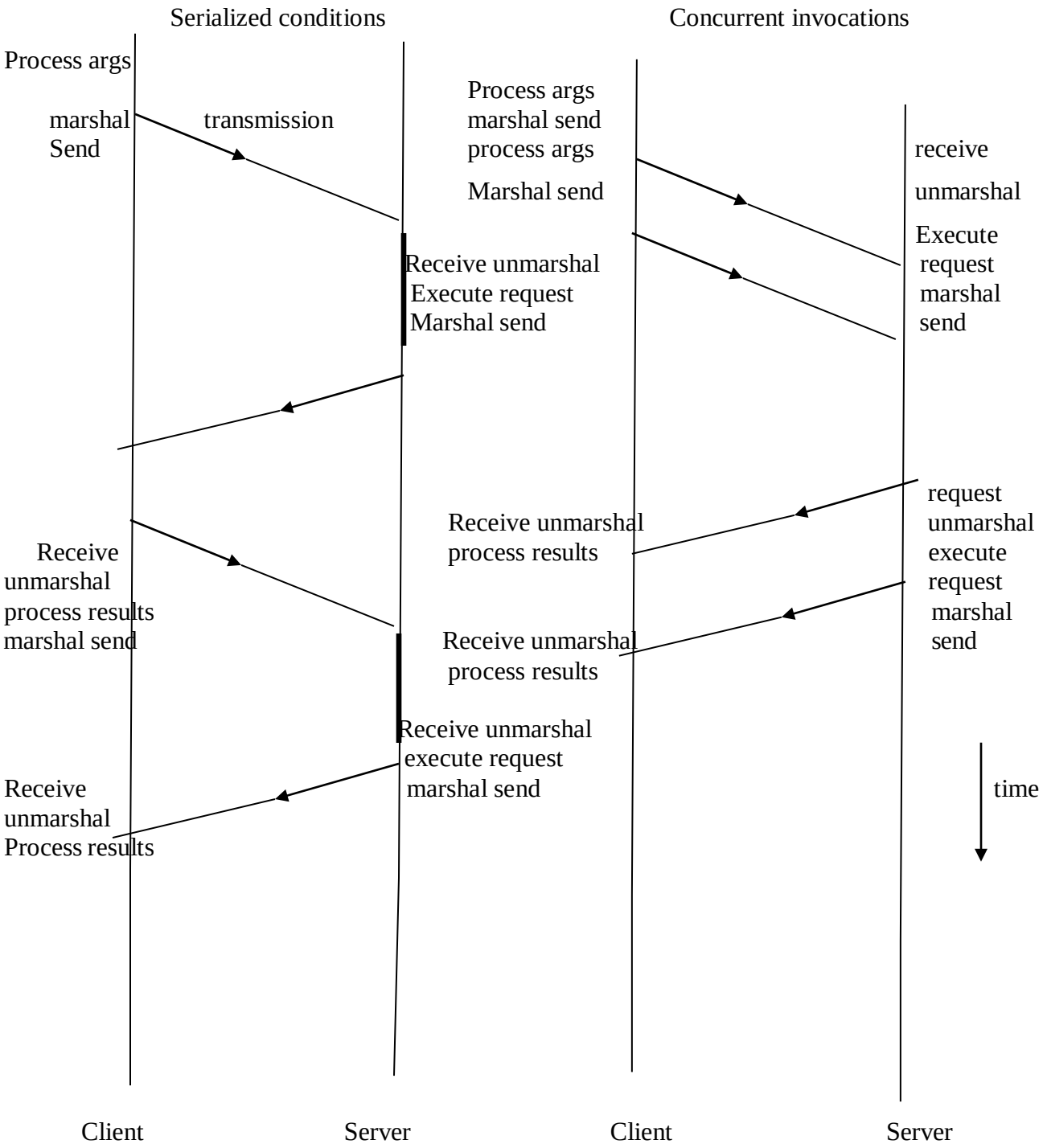
-**Invocation within a computer:** Most cross-address-space invocation takes place within a computer at time of installation of OS. The cost of RPC within a computer is growing in importance as a system performance parameter. A cross-address-space invocation is implemented within a computer exactly as it b/w computers, except that the underlying message passing happens to be local.

Asynchronous operation: A common technique to high latencies is asynchronous operations, which arises in two programming models. 1. Concurrent invocations and 2. Asynchronous invocations. 3.Persistent asynchronous invocations.

- ***Concurrent invocations:*** In this model the middleware provides only blocking invocations, but the application spawns multiple threads to perform blocking invocations concurrently.

> *E.g., A good example this is a web browser. A web page contains several images. The browser has to fetch each of these images in a separate HTTP GET.*

Times for serialized and concurrent invocation



- ***Asynchronous invocation***: An *asynchronous invocation* is performed asynchronously with respect to the caller. That is, it is made with a non-blocking call, which returns as soon as the invocation request message has been created and is ready for dispatch.

- E.g., CORBA *one-way* invocations have maybe semantics, otherwise, the client uses a separate call to collect the results of the invocation. For example, the Mercury communication system and supports asynchronous invocations. In this case, an asynchronous operation returns an object called a *promise*. The caller uses the claim operation to obtain the result from the promise.

- ***Persistent asynchronous invocations:***

A system for persistent asynchronous invocation tries indefinitely to perform the invocation, until it is known to have succeeded or failed. For example, QRPC (Queued RPC) is used keep the requests of all clients in queue which applied to access the server. In this case, persistent asynchronous invocation tries to establish the connection on server if TCP stream is in difficult.

Unit – V – Distributed File Systems

DFS: A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disk.

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.
- Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet.
- A well designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks.
- The sharing of resources is a key goal for distributed systems. The sharing of stored information is the most important aspect of distributed resource sharing.

- The requirements for sharing within local networks and intranets lead to a need for a different type of service – one that supports the persistent storage of data and programs of all types on behalf of clients and the consistent distribution of up-to-date data.

- A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet.

- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	✓	OceanStore

Types of Consistency:

1. Strict One copy- Slightly weaker guarantees
2. Considerably weaker guarantees

Characteristics of file systems:

- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media.
- Files contain both *data and attributes. The data consist of a sequence of data items that* accessible by operations to read and write any portion of the sequence.
- The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists.

File System Module: In this, defined an abstract model for a basic distributed file service, including a set of programming interfaces.

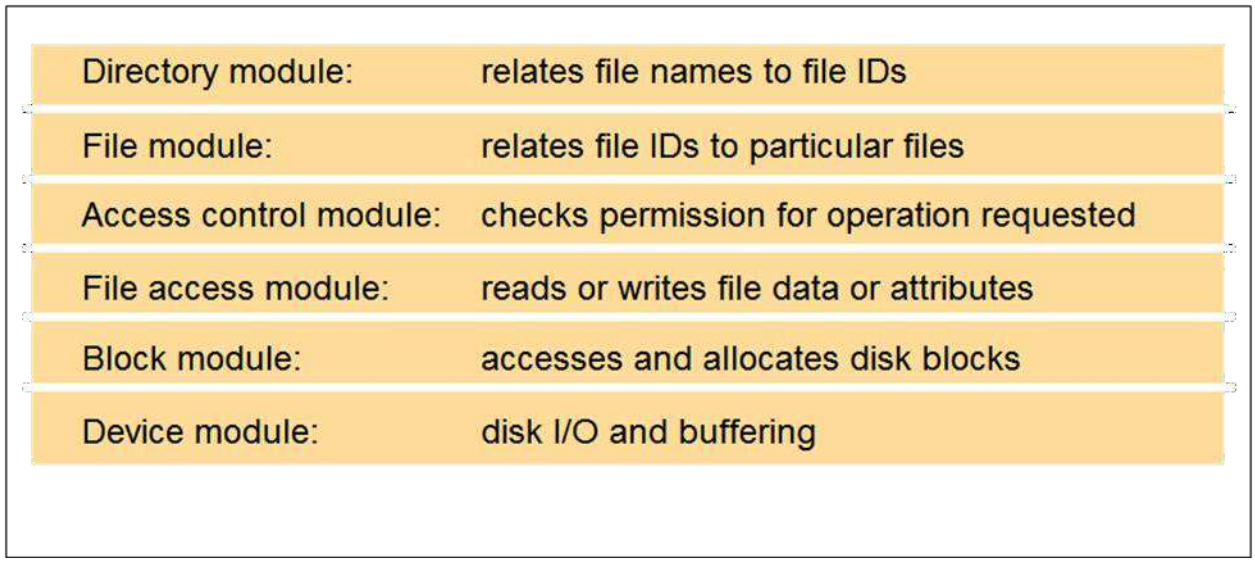
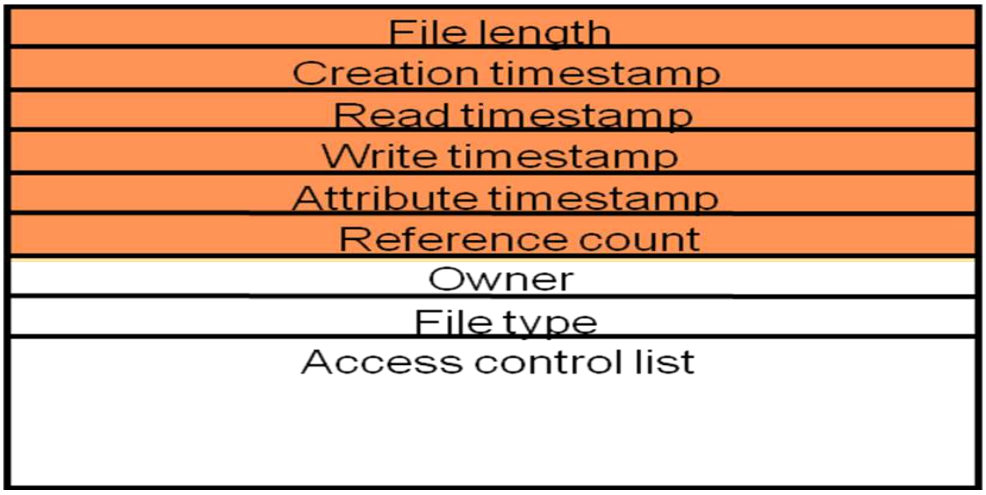


Figure shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system. Each layer depends only on the layers below it. The implementation of a distributed file service requires all of the components shown there, with additional components to deal with client-server communication and with the distributed naming and location of files

File attributes record structure:

Files contain both *data* and *attributes*. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. A typical attribute record structure is illustrated in Figure. The shaded attributes are managed by the file system and are not normally updatable by user programs.



File system operations:

Figure summarizes the main operations on files that are available to applications in UNIX systems. These are the *system calls* implemented by the kernel. Application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes.

Figure 4. UNIX file system operations

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

Distributed file system requirements:

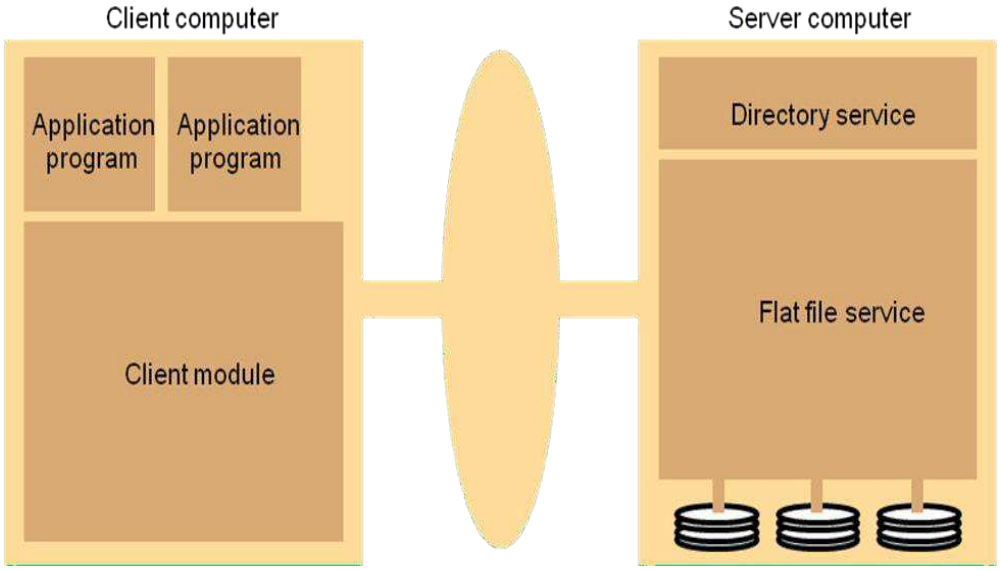
- Related requirements in distributed file systems are:
  - ***Transparency*:** The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems. They are **Access transparency, Location transparency, Mobility transparency, Performance transparency, Scaling transparency**.
  - ***Concurrency*:** Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control.
  - ***Replication*:** A file may be represented by several copies of its contents at different locations.
  - ***Heterogeneity*:** Cclient and server software can be implemented for different operating systems and computers.

- ***Fault tolerance*:** The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures.
- ***Consistency*:** See same directory and file contents on different clients at same time.
- **S*ecurity :*** Secure communication and user authentication
- **Efficiency:** A distributed file system should be accessed high performance.

File Service Architecture: An architecture that offers a clear separation of the main concerns inproviding access to files is obtained by structuring the file service as three components:

- A flat file service
- A directory service
- A client module.

- The relevant modules and their relationship is shown in Figure.



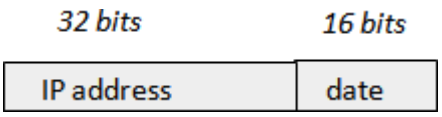
- **A flat file service** is concerned with implementing operations on the contents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. With the use of UFIDs each file has a unique among all of the files in a distributed system.
  - Table contains a definition of the interface to a flat file service.

Read(FileId, i, n) ->	*:* Reads a sequence of data upto n items from a file starting at item *i* and returns it in *Data.*
Write(FileId, i, Data)	: Write a sequence of *Data* to a file, starting item i, extending the file if necessary.
Create() -> FileId	Creates a new file of length0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure (i.e. File attribute record structure.)

- The Directory service provides mapping between text names for the files and their UFIDs. Clients may obtain the UFID of a file by its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.

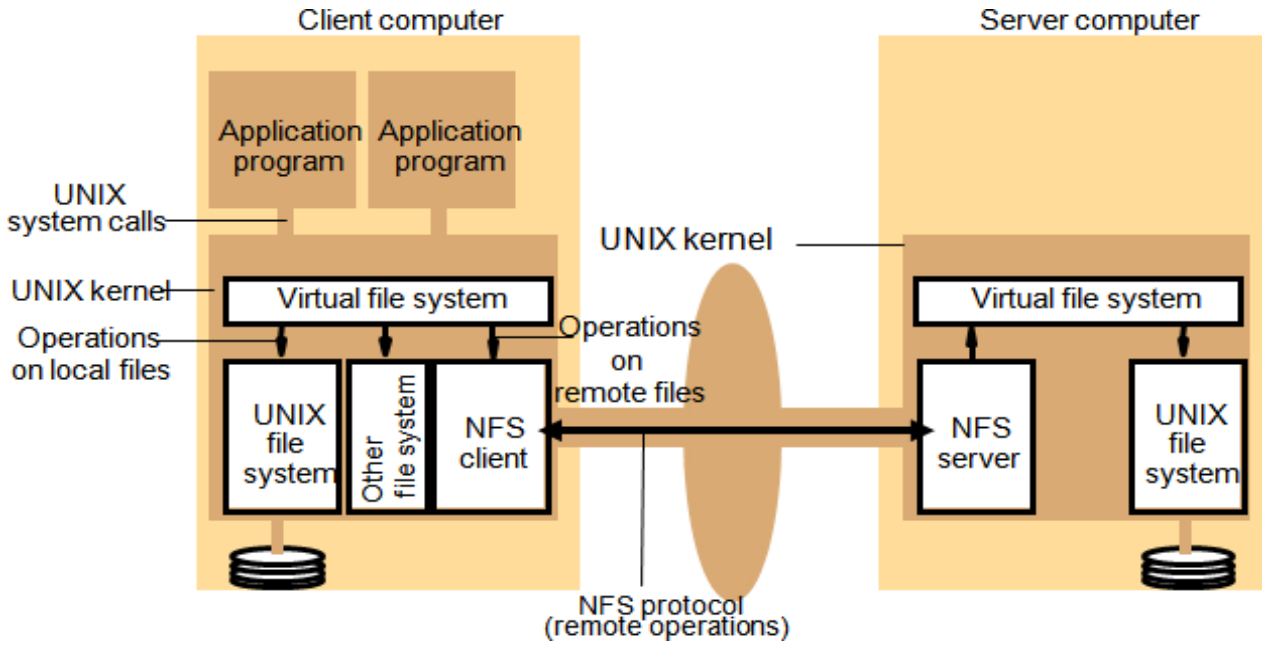
Lookup(Dir, Name) -> FileId -throws NotFound	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, File) -throws NameDuplicate	If *Name* is not in the directory, adds(*Name,File*) to the directory and updates the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

- **Client module** runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.
    - A client model also holds information about the network locations of flat-file and directory server processes and achieves better performance through implementation of a cache of recently used file blocks at the client.
- Access control: In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.
- **Directory service interface**: Figure contains a definition of the RPC interface to a directory service.
    - The primary purpose of the directory service is to provide a service for translating text names to UFIDs.
- **Hierarchic file system :**
    - A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.
- **File Group :**
    - A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.
        - A similar construct is used in a UNIX file system.
        - It helps with distributing the load of file serving between several servers.
        - File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).
    - To construct a globally unique ID we use some unique attribute of the machine on which it is created, e.g. IP number, even though the file group may move subsequently.



Examples for File Service architecture are NFS, AFS

: Network File System (NFS) architecture:



The NFS protocol was originally developed for use in networks of UNIX systems. The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system. The NFS client and server modules communicate using remote procedure calls.

- The file identifiers used in NFS are called file handles. In UNIX implementations of NFS, the file handle is derived from the file's *i-node number* by adding two extra fields as follows (the i-node number of a UNIX file is a number that serves to identify and locate the file within the file system in which the file is stored):

File handle:	Filesystem identifier	i-node number of file	i-node generation number

Sun Network File System

All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems, and we shall describe the UNIX implementation the NFS protocol

The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.

The NFS client and server modules communicate using remote procedure calls. Sun's RPC system, was developed for use in NFS. It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.

Virtual File System

The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file

identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the file systems that are currently available both locally and remotely, and it passes each request to the appropriate local system module.

Client integration • The NFS client module plays the role described for the client module in our architectural model, supplying an interface suitable for use by conventional application programs. But unlike our model client module, it emulates the semantics of the standard UNIX file system primitives precisely and is integrated with the UNIX kernel. It is integrated with the kernel and not supplied as a library for loading into client processes so that:

- user programs can access files via UNIX system calls without recompilation or reloading;
- a single client module serves all of the user-level processes, with a shared cache of recently used blocks.

NFS server operations

lookup(dirfh, name) → fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) → *newfh, attr*	Creates a new file *name* in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) → *status*	Removes file *name* from directory *dirfh*.
getattr(fh) → *attr*	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) → *attr*	Sets the attributes (mode, user ID, group ID, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) → *attr, data*	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) → *attr*	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) → *status*	Changes the name of file *name* in directory *dirfh* to *toname* in directory *todirfh*.
link(newdirfh, newname, fh) → *status*	Creates an entry *newname* in the directory *newdirfh* that refers to the file *or* directory *fh*.
symlink(newdirfh, newname, string) → *statu*	Creates an entry *newname* in the directory *newdirfh* of type *symbolic link* with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) → *string*	Returns the string that is associated with the symbolic link file identified by *fh*.

mkdir(dirfh,nam, attr) → *newfh, attr*	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) → *status*	Removes the empty directory name from the parent directory dirfh. Fails if the directory is empty
readdir(dirfh, cookie, count) → *entries*	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) → *fsstats*	Returns file system information (such as block size, number of free blocks and so on) for the file system containing the file fh.

Access control and authentication • Unlike the conventional UNIX file system, the NFS server is stateless and does not keep files open on behalf of its clients. So the server must check the user's identity against the file's access permission attributes afresh on each request, to see whether the user is permitted to access the file in the manner requested.

NFS server interface : A simplified representation of the RPC interface provided by NFS version 3 servers. The NFS file access operations *read*, *write*, *getattr* and *setattr* are almost identical to the *Read*, *Write*, *GetAttributes* and *SetAttributes* operations defined for our flat file service model. The other NFS operations on directories are *create*, *remove*, *rename*, *link*, *symlink*, *readlink*, *mkdir*, *rmdir*, *readdir* and *statfs*. They resemble their UNIX counterparts with the exception of *readdir*, which provides a representation- independent method for reading the contents of directories, and *statfs*, which gives the status information on remote file systems.

Mount service • The mounting of subtrees of remote filesystems by clients is supported by a separate *mount service* process that runs at user level on each NFS server computer. On each server, there is a file with a well-known name (*/etc/exports*) containing the names of local filesystems that are available for remote mounting. An access list is associated with each filesystem name indicating which hosts are permitted to mount the filesystem. Clients use a modified version of the UNIX *mount* command to request mounting of a remote filesystem, specifying the remote host's name, the pathname of a directory in the remote filesystem and the local name with which it is to be mounted.

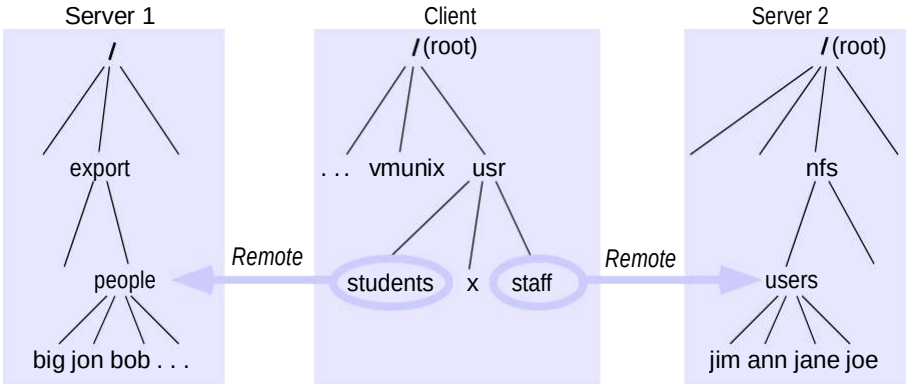


Figure: Local and remote file system accessible on an NFS client

Figure 12.10 illustrates a *Client* with two remotely mounted file stores. The nodes *people* and *users* in filesystems at *Server 1* and *Server 2* are mounted over nodes *students* and *staff* in *Client*'s local file store. The meaning of this is that programs running at *Client* can access files at *Server 1* and *Server 2* by using pathnames such as

/usr/students/jon and /usr/staff/ann.

Pathname translation • UNIX file systems translate multi-part file pathnames to i-node references in a step-by-step process whenever the *open*, *creat* or *stat* system calls are used. In NFS, pathnames cannot be translated at a server, because the name may cross a 'mount point' at the client – directories holding different parts of a multi-part name may reside in filesystems at different servers. So pathnames are parsed, and their translation is performed in an iterative manner by the client. Each part of a name that refers to a remote-mounted directory is translated to a file handle using a separate *lookup* request to the remote server. The *lookup* operation looks for a single part of a pathname in a given directory and returns the corresponding file handle and file attributes.

Automounter • The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client. The original implementation of the automounter ran as a user- level UNIX process in each client computer. Later versions (called *autofs*) were implemented in the kernel for Solaris and Linux.

Server caching • Caching in both the client and the server computer are indispensable features of NFS implementations in order to achieve adequate performance.

In NFS Protocol write operation offers two points

1. Data in *write* operations received from clients is stored in the memory cache at the server *and* written to disk before a reply is sent to the client. This is called *write- through* caching. The client can be sure that the data is stored persistently as soon as the reply has been received.
2. Data in *write* operations is stored only in the memory cache. It will be written to disk when a *commit* operation is received for the relevant file. The client can be sure that the data is persistently stored only when a reply to a *commit* operation for the relevant file has been received. Standard NFS clients use this mode of operation, issuing a *commit* whenever a file that was open for writing is closed.

Client caching • The NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations in order to reduce the number of requests transmitted to servers.

Each data or metadata item in the cache is tagged with two timestamps:

Tc is the time when the cache entry was last validated.

Tm is the time when the block was last modified at the server.

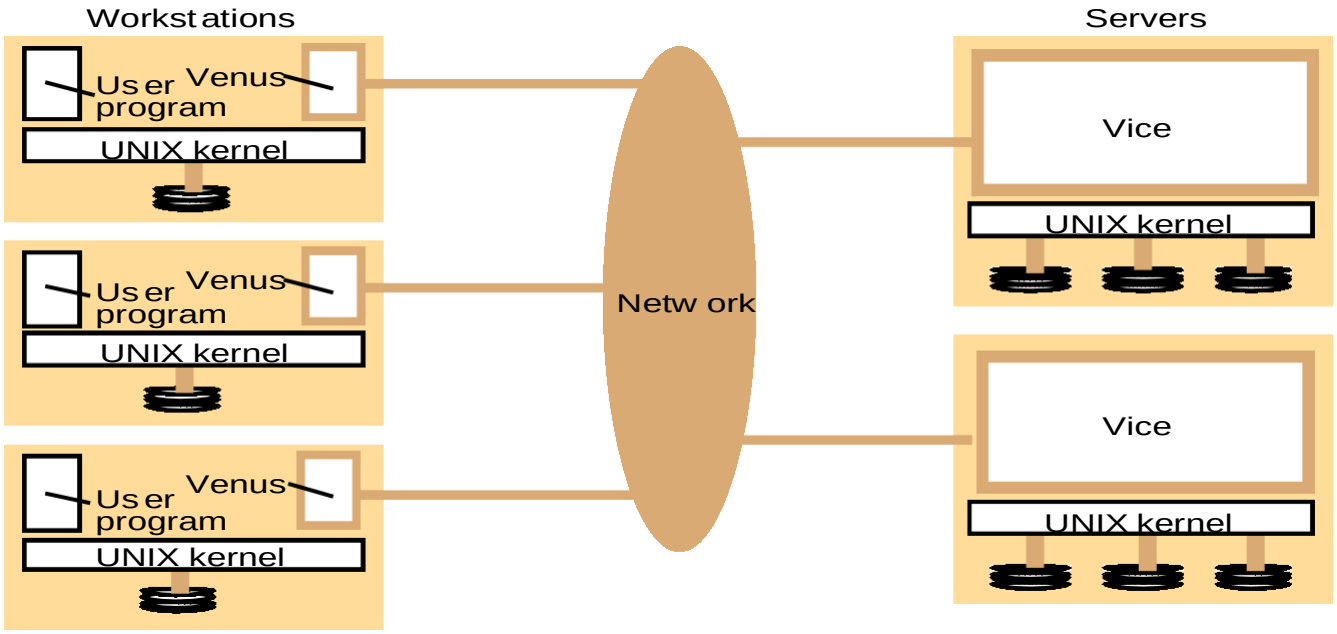
$$(T-Tc<t) \vee (Tm_{client} = Tm_{server})$$

: AFS architecture:

- Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations.
- AFS is implemented as two software components that exist at UNIX processes called Vice and Venus.
- Figure shows the distribution of Vice and Venus processes. ***Vice*** is the name given to the server software that runs as a user-level UNIX process in each server computer, and ***Venus*** is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.
- The files available to user processes running on workstations are either local or shared.

- Local files are handled as normal UNIX files.
- They are stored on the workstation's disk and are available only to local user processes.
- Shared files are stored on servers, and copies of them are cached on the local disks of workstations.

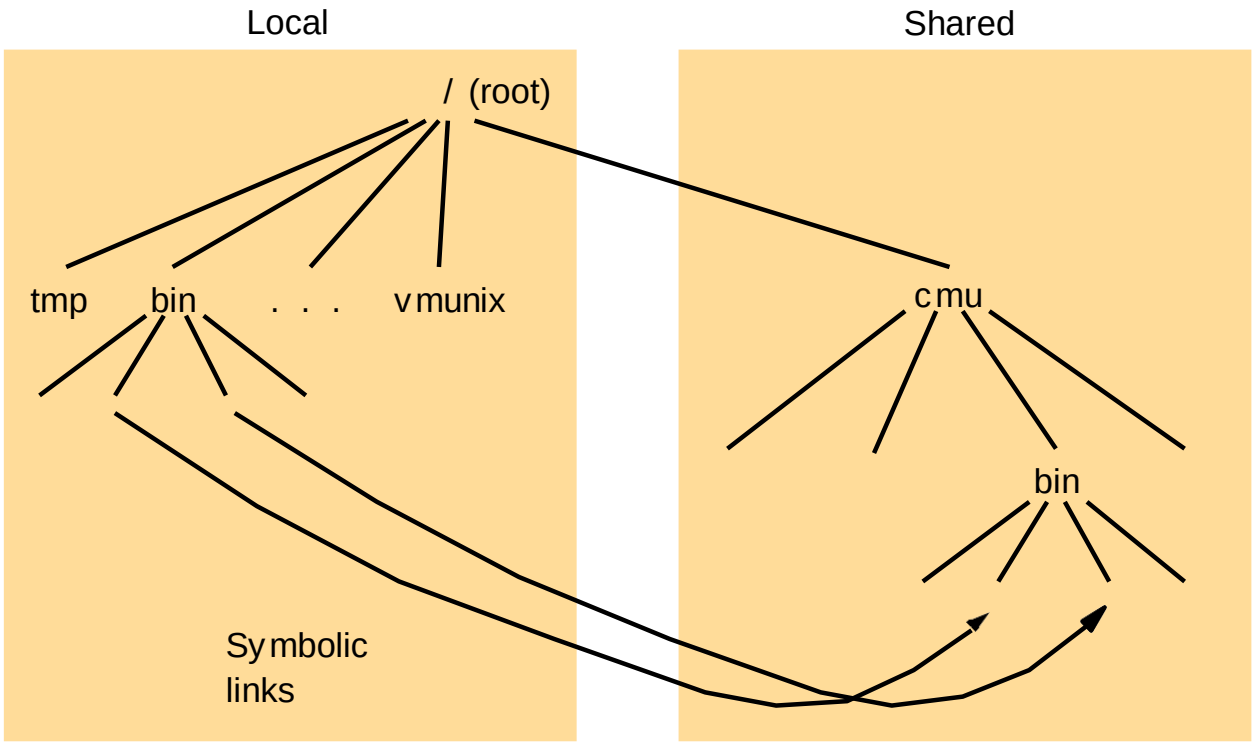
Implementation



AFS is implemented as two software components that exist as UNIX processes called *Vice* and *Venus*.

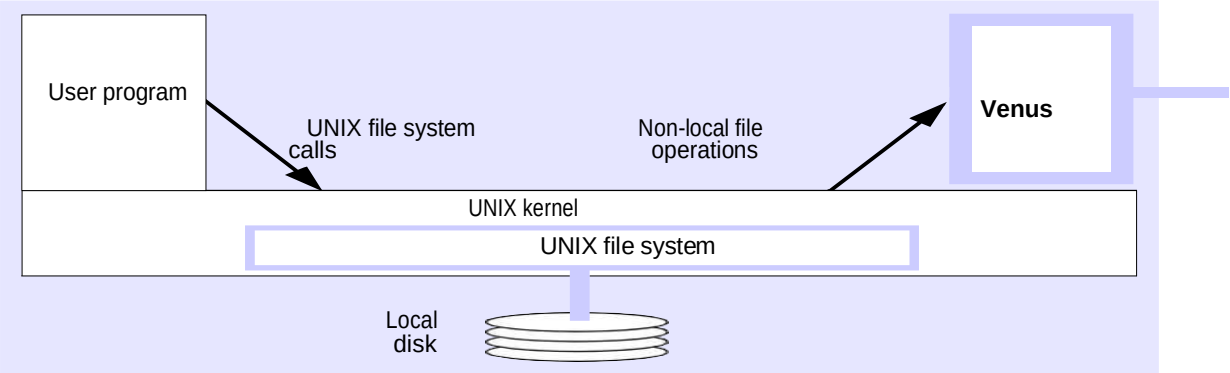
Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.

- The name space seen by user processes is illustrated in Figure.



- It is a conventional UNIX directory hierarchy, with a specific sub-tree (called *cmu*) containing all of the shared files. This splitting of the file name space into local and shared files leads to some loss of location transparency, but this is hardly noticeable to users other than system administrators.

System call interception in AFS



Files are grouped into *volumes* for ease of location and movement. Volumes are generally smaller than the UNIX filesystems, which are the unit of file grouping in NFS. For example, each user's personal files are generally located in a separate volume. Other volumes are allocated for system binaries, documentation and library code.

The representation of *fid*s includes the volume number for the volume containing the file (*cf.* the *file group identifier* in UFIDs), an NFS file handle identifying the file within the volume (*cf.* the *file number* in UFIDs) and a *uniquifier* to ensure that file identifiers are not reused:

File Identifier(fid's)

32 bits	32 bits	32 bits
Volume number	File handle	Uniquifier

Other Aspects

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

UNIX kernel modifications • We have noted that the Vice server is a user-level process running in the server computer and the server host is dedicated to the provision of an

AFS service. The UNIX kernel in AFS hosts is altered so that Vice can perform file operations in terms of file handles instead of the conventional UNIX file descriptors. This is the only kernel modification required by AFS, and it is necessary if Vice is not to maintain any client state (such as file descriptors).

Location database • Each server contains a copy of a fully replicated location database giving a mapping of volume names to servers. Temporary inaccuracies in this database may occur when a volume is moved, but they are harmless because forwarding information is left behind in the server from which the volume is moved.

Threads • The implementations of Vice and Venus make use of a non-preemptive threads package to enable requests to be processed concurrently at both the client (where several user processes may have file access requests in progress concurrently) and the server. In the client, the tables describing the contents of the cache and the volume database are held in memory that is shared between the Venus threads.

Read-only replicas • Volumes containing files that are frequently read but rarely modified, such as the UNIX */bin* and */usr/bin* directories of system commands and */man* directory of manual pages, can be replicated as read-only volumes at several servers. When this is done, there is only one read-write replica and all updates are directed to it. The propagation of the changes to the read-only replicas is performed

after the update by an explicit operational procedure. Entries in the location database for volumes that are replicated in this way are one-to-many, and the server for each client request is selected on the bases of server loads and accessibility.

Bulk transfers • AFS transfers files between clients and servers in 64-kilobyte chunks. The use of such a large packet size is an important aid to performance, minimizing the effect of network latency. Thus the design of AFS enables the use of the network to be optimized.

Partial file caching • The need to transfer the entire contents of files to clients even when the application requirement is to read only a small portion of the file is an obvious source of inefficiency. Version 3 of AFS removed this requirement, allowing file data to be transferred and cached in 64-kbyte blocks while still retaining the consistency semantics and other features of the AFS protocol.

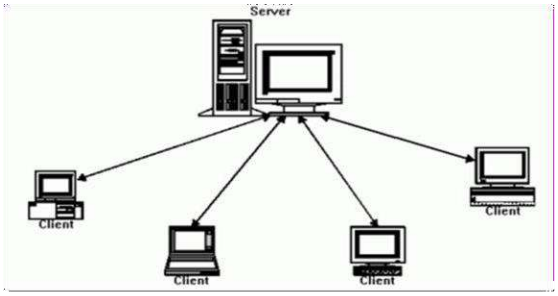
Performance • The primary goal of AFS is scalability, so its performance with large numbers of users is of particular interest. Howard *et al.* [1988] give details of extensive comparative performance measurements, which were undertaken using a specially developed *AFS benchmark* that has subsequently been widely used for the evaluation of distributed file systems. Not surprisingly, whole-file caching and the callback protocol led to dramatically reduced loads on the servers. Satyanarayanan [1989a] states that a server load of 40% was measured with 18 client nodes running a standard benchmark, against a load of 100% for NFS running the same benchmark. Satyanarayanan attributes much of the performance advantage of AFS to the reduction in server load deriving from the use of callbacks to notify clients of updates to files, compared with the timeout mechanism used in NFS for checking the validity of pages cached at clients.

Wide area support: • Version 3 of AFS supports multiple administrative cells, each with its own servers, clients, system administrators and users. Each cell is a completely autonomous environment, but a federation of cells can cooperate in presenting users with a uniform, seamless file name space. The resulting system was widely deployed by the Transarc Corporation, and a detailed survey of the resulting performance usage patterns was published [Spasojevic and Satyanarayanan 1996]. The system was installed on over 1000 servers at over 150 sites. The survey showed cache hit ratios in the range of 96 – 98% for accesses to a sample of 32,000 file volumes holding 200 Gbytes of data.

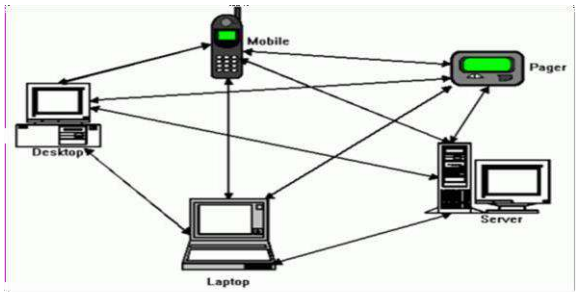
> **Peer-to-Peer Systems: Peer to peer** is an approach to computer networking where all computers share equivalent responsibility for processing data. Peer-to-peer networking (also known simply as *peer networking*) differs from client-server networking, where certain devices have responsibility for providing or "serving" data and other devices consume or otherwise act as "clients" of those servers.

The goal of peer-to-peer systems is to enable the sharing of data and resources on a very large scale by eliminating any requirement for separately managed servers and their associated infrastructure.

- Peer-to-peer systems aim to support useful distributed services and applications using data and computing resources available in the personal computers and workstations that are present in the Internet and other networks in ever-increasing numbers.
- An alternative to the client/server model of distributed computing is the peer-to-peer model.
- Client/server is naturally hierarchical, with resources centralized on a limited number of servers.
- In peer-to-peer networks, both resources and control are widely distributed among nodes that are theoretically equals. (A node with more information, better information, or more power may be "*more equal,*" but that is a function of the node, not the network controllers.)
- A key feature of peer-to-peer networks is decentralization. This has many implications. Robustness, availability of information and fault-tolerance tends to come from redundancy and shared responsibility instead of planning, organization and the investment of a controlling authority.
- Peer-to-peer applications provide better communication for 'the applications which exploit resources available at the edges of the Internet such as storage, cycles, content, human presence'. Each type of resource sharing mentioned in that definition is already represented by distributed applications available for most types of personal computer.



The Client/Server Model



The Peer-to-Peer Model

Characteristics:

1. Each user contributes resources to the system.
2. All the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.
3. Correct operation does not depend on the existence of any centrally administered systems.
4. Limited degree of anonymity to the providers and users of resources.
5. Efficient operation is the choice of an algorithm.

Advantages of Peer-to-peer networking over Client –Server networking are :-

1) It is easy to install and so is the configuration of computers on this network,
2) All the resources and contents are shared by all the peers, unlike server-client Architecture where Server shares all the contents and resources.
3) P2P is more reliable as central dependency is eliminated. Failure of one peer doesn't affect the functioning of other peers. In case of Client –Server network, if server goes down whole network gets affected.
4) There is no need for full-time System Administrator. Every user is the administrator of his machine. User can control their shared resources.
5) The over-all cost of building and maintaining this type of network is comparatively very less.

Disadvantages (drawbacks) of Peer to peer architecture over Client Server are:-

1) In this network, the whole system is decentralized thus it is difficult to administer. That is one person cannot determine the whole accessibility setting of whole network.

2) Security in this system is very less viruses, spywares, trojans, etc malwares can easily transmitted over this P-2-P architecture.

3) Data recovery or backup is very difficult. Each computer should have its own back-up system.

4) Lot of movies, music and other copyrighted files are transferred using this type of file transfer. P2P is the technology used in torrents.

Note: Peer to peer networks are good to connect small number (around 10) of computer and places where high level of security is not required. In case of business network where sensitive data can be present this type of architecture is not advisable or preferred.

Applications:

- Theory
  - Dynamic discovery of information
  - Better utilization of bandwidth, processor, storage, and other resources
  - Each user contributes resources to network
- Practice examples
  - Sharing browser cache over 100Mbps lines
  - Disk mirroring using spare capacity
  - Deep search beyond the web

Peer-to-peer middleware: The third generation is characterized by the emergence of middleware layers for the application-independent management of distributed resources on a global scale. Several research teams have now completed the development, evaluation and refinement of peer-to-peer middleware platforms and demonstrated or deployed them in a range of application services.

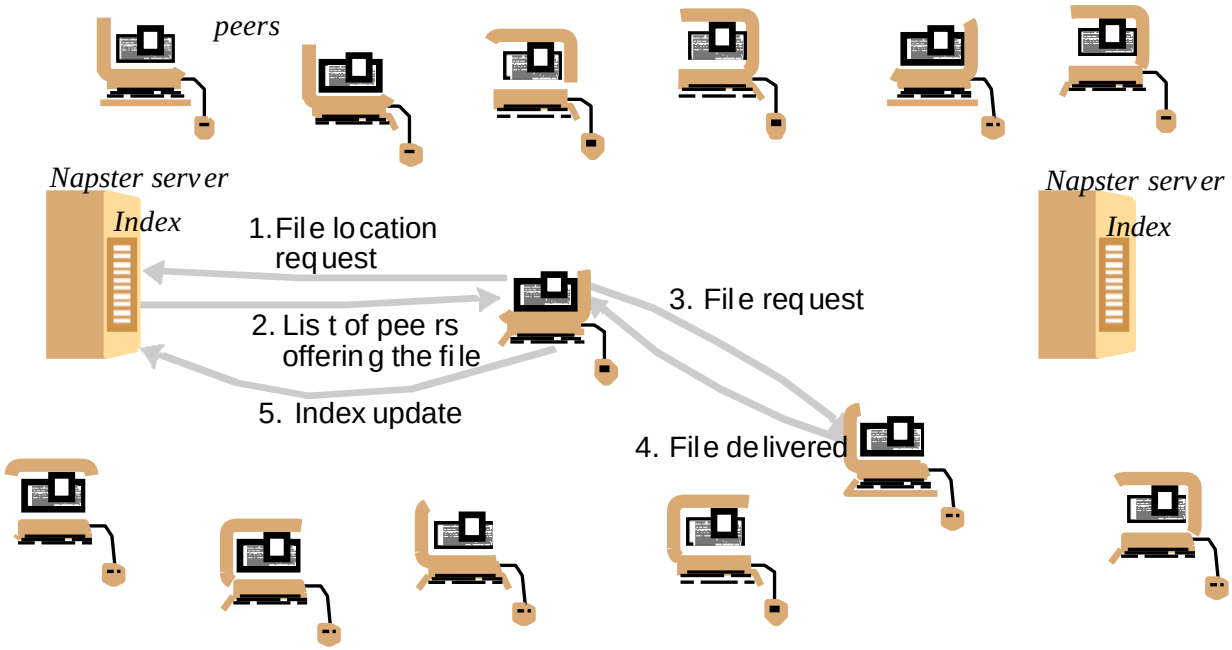
Routing Overlays: Routing overlays share many characteristics with the IP packet routing infrastructure that constitutes the primary communication mechanism of the Internet. It is therefore legitimate to ask why an additional application-level routing mechanism is required in peer-to-peer systems.

Napster and its legacy: Downloading the digital music files is the first application which globally scalable storage and retrieval in service. Both need and feasibility of a peer-to-peer solution was first demonstrated by the Napster file sharing system for users to share files. Napster became very popular for music exchange soon after its launch in 1999. At its peak, several million users were registered and thousands were swapping music files simultaneously.

Napster's architecture included centralized indexes, but users supplied the files, which were stored and accessed on their personal computers. Napster's method of operation is illustrated by the sequence of steps shown in Figure. Note that in step 5 clients are expected to add their own music files to the pool of shared resources by transmitting a link to the Napster indexing service for each available file.

Thus the motivation for Napster and the key to its success was the making available of a large, widely distributed set of files to users throughout the Internet and providing access to 'shared resources at the edges of the Internet'.

Napster: peer-to-peer file sharing with a centralized, replicated index



Lessons learned from Napster: Napster demonstrated the feasibility of building a useful large-scale service that depends almost wholly on data and computers owned by ordinary Internet users. To avoid swamping the computing resources of individual users (for example, the first user to offer a chart-topping song) and their network connections, Napster took account of network locality – the number of hops between the client and the server – when allocating a server to a client requesting a song. This simple load distribution mechanism enabled the service to scale to meet the needs of large numbers of users.

Limitations: Napster used a (replicated) unified index of all available music files. For the application in question, the requirement for consistency between the replicas was not strong, so this did not hamper performance, but for many applications it would constitute a limitation. Unless the access path to the data objects is distributed, object discovery and addressing are likely to become a bottleneck.

Application dependencies: Napster took advantage of the special characteristics of the application for which it was designed in other ways:

• Music files are never updated, avoiding any need to make sure all the replicas of files remain consistent after updates.

• No guarantees are required concerning the availability of individual files – if a music file is temporarily unavailable, it can be downloaded later. This reduces the requirement for dependability of individual computers and their connections to the Internet.

Peer-to-peer middleware:

Peer-to-Peer Middleware is to provide mechanism to access data resources anywhere in network. A key problem in Peer-to-Peer applications is to provide a way for clients to access data resources efficiently. Similar needs in client/server technology led to solutions like NFS. However, NFS relies on pre-configuration and is not scalable enough for peer-to-peer.

For example, Sun NFS addresses this need with the aid of a virtual file system abstraction layer at each client that accepts requests to access files stored on multiple servers in terms of virtual file references

Peer clients need to locate and communicate with any available resource, even though resources may be widely distributed and configuration may be dynamic, constantly adding and removing resources and connections.

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

- Functional Requirements :
  - Simplify construction of services across many hosts in wide network
  - Add and remove resources at will
  - Add and remove new hosts at will
  - Interface to application programmers should be simple and independent of types of distributed resources
- Non-Functional Requirements :
  - **Global Scalability**: peer-to-peer applications are to exploit the hardware resources of very large numbers of hosts connected to the Internet. So it that access millions of objects on tens of thousands or hundreds of thousands of hosts.
  - **Load Balancing**: The performance of any system is designed to exploit a large number of computers depends upon the balanced distribution of workload across them. This will be achieved by a random placement of resources together with the use of replicas of heavily used resources.
  - **Optimization for local interactions between neighboring peers:** The 'network distance' between nodes that interact has a substantial impact on the latency of individual interactions, such as client requests for access to resources. The

middleware should aim to place resources close to the nodes that access them the most.

- **Accommodation to highly dynamic host availability:** Most peer-to-peer systems are constructed from host computers that are free to join or leave the system at any time.
- ***Security of data in an environment with heterogeneous trust*:** Security of data in an environment simplify construction of services across many hosts in wide network
- Anonymity, deniability and resistance to restrict.

Routing Overlays: Routing overlays share many characteristics with the IP packet routing infrastructure that constitutes the primary communication mechanism of the Internet. It is therefore legitimate to ask why an additional application-level routing mechanism is required in peer-to-peer systems. This is shown figure (table).

- A routing overlay is a distributed algorithm for a middleware layer responsible for routing requests from any client to a host that holds the object to which the request is addressed.
- Any node can access any object by routing each request through a sequence of nodes, exploiting knowledge at each of them to locate the destination object.

- Global User IDs (GUID) also known as opaque identifiers are used as names, but do not contain location information.
- A client wishing to invoke an operation on an object submits a request including the object's GUID to the routing overlay, which routes the request to a node at which a replica of the object resides.

Figure Distinctions between IP and overlay routing for peer-to-peer applications

	IP	*Application-level routing overlay*
Scale	IPv4 is limited to 2^{32} addressable nodes. The IPv6 namespace is much more generous (2^{128}), but addresses in both versions are hierarchically structured and much of the space is preallocated according to administrative requirements.	Peer-to-peer systems can address more objects. The GUID namespace is very large and flat (>2^{128}), allowing it to be much more fully occupied.
Load balancing	Loads on routers are determined by network topology and associated traffic patterns.	Object locations can be randomized and hence traffic patterns are divorced from the network topology.
Network dynamics (addition/deletion of objects/nodes)	IP routing tables are updated asynchronously on a best-effort basis with time constants on the order of 1 hour.	Routing tables can be updated synchronously or asynchronously with fractions-of-a-second delays.

Fault tolerance	Redundancy is designed into the IP network by its managers, ensuring tolerance of a single router or network connectivity failure. *n*-fold replication is costly.	Routes and object references can be replicated *n*-fold, ensuring tolerance of *n* failures of nodes or connections.
Target identification	Each IP address maps to exactly one target node.	Messages can be routed to the nearest replica of a target object.
Security and anonymity	Addressing is only secure when all nodes are trusted. Anonymity for the owners of addresses is not achievable.	Security can be achieved even in environments with limited trust. A limited degree of anonymity can be provided.

The main task of a routing overlay is the following:

***Routing of requests to objects*:** A client wishing to invoke an operation on an object submit a request including the object's GUID to the routing overlay, which routes the request to a node at which a replica of the object resides.

But the routing overlay must also perform some other tasks:

***Insertion of objects*:** A node wishing to make a new object available to a peer-to-peer service computes a GUID for the object and announces it to the routing overlay, which then ensures that the object is reachable by all other clients.

***Deletion of objects*:** When clients request the removal of objects from the service the routing overlay must make them unavailable.

***Node addition and removal*:** Nodes (i.e., computers) may join and leave the service.

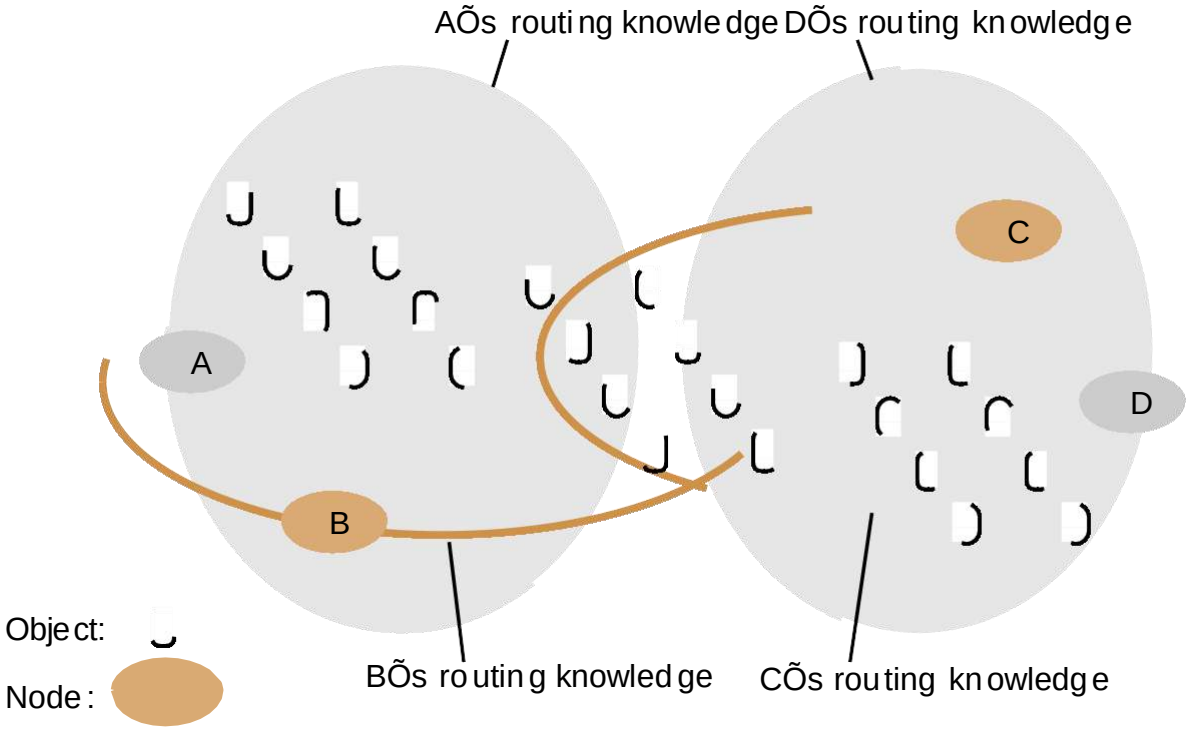


Figure: **Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry**

***p**ut(GUID, data)*

The *data* is stored in replicas at all nodes responsible for the object identified by *GUID*.

remove(GUID)

Deletes all references to *GUID* and the associated data.

value = get(GUID)

The data associated with *GUID* is retrieved from one of the nodes responsible it.

Figure: **Basic programming interface for distributed object location and routing (DOLR) as implemented by Tapestry.**

publish(GUID)

GUID can be computed from the object (or some part of it, e.g. its name). This function makes the node performing a *publish* operation the host for the object corresponding to *GUID*.

unpublish(GUID)

Makes the object corresponding to *GUID* inaccessible.

sendToObj(msg, GUID, [n])

The invocation message is sent to an object in order to access it. This might be a request to open a TCP connection for data transfer or to return a message containing all or part of the object's state. The final optional parameter *[n],* if present, requests the delivery of the same message to n replicas of the object.

The interfaces in Figures 1 & 2 are based on a set of abstract to show that most peer-to-peer routing overlay implementations developed to date provide very similar functionality.

Coordination and Agreement

Introduction: Collection of algorithms are used to provide a communication with the help of peer-to- peer process in distributed systems. In view of that,

- A set of processes to coordinate their actions or to agree on one or more values.

- For example, a complex piece of machinery such as a spaceship is essential that the computers need control the operation for proceeding or has been aborted. In this case, the computers must coordinate their actions correctly with respect to shared resources (the spaceship's sensors and actuators). The computers must be able to proceeding their process if there is no fixed master-slave relationship between the components.

- The reason for avoiding fixed master-slave relationships is that we often require our systems to keep working correctly even if failures occur, so we need to avoid single points of failure, such as fixed masters.

- The distributed system can using either asynchronous or synchronous for communication. In an asynchronous system we can make no timing assumptions. In a synchronous system, we shall assume that there are bounds on the maximum message transmission delay, on the time taken to execute each step of a process, and on clock drift rates. The synchronous assumptions allow us to use timeouts to detect process crashes.
- A failure model is another important in distributed system. This begin by considering some algorithms that tolerate no failures and progress through benign failures before exploring how to tolerate arbitrary failures. Along the way, we encounter a fundamental result in the theory of distributed systems: even under surprisingly benign failure conditions, it is impossible to guarantee

in an asynchronous system that a collection of processes can agree on a shared value – for example, for all of a spaceship's controlling processes to agree 'mission proceed' or 'mission abort'.

Coordination and agreement related to group communication is the ability to multicast a message to a group is a very useful communication paradigm, with applications from locating resources to coordinating the updates to replicated data. It examines multicast reliability and ordering semantics, and gives algorithms to achieve the variations.

7.1.1 Failure assumptions and failure detectors:

Failure assumptions: The fundamental network components may suffer failures by a reliable communication protocol – for example, by retransmitting missing or corrupted messages. Also for the sake of simplicity, we assume that no process failure implies a threat to the other processes' ability to communicate. This means that none of the processes depends upon another to forward messages.

In any particular interval of time, communication between some processes may succeed while communication between others is delayed. For example, the failure of a router between two networks may mean that a collection of four processes is split into two pairs, such that intra-pair communication is possible over their respective networks; but inter-pair communication is not possible while the router has failed. This is known as a *network partition.*

Failure detectors: A *failure detector* is a service that processes queries about whether a particular process has failed. It is often implemented by an object local to each process (on the same computer) that runs a failure-detection algorithm in conjunction with its counterparts at other processes. The object local to each process is called a *local failure detector*. We outline how to implement failure detectors shortly, but first we concentrate on some of the properties of failure detectors.

A failure 'detector' is categorized into two types, they are

1. *Unreliable failure detectors* and 2. *Reliable failure detectors.*

An ***unreliable failure*** detector may produce one of two values to identity the process: *Unsuspected* or *Suspected*. Both of these results are May or may not accurately reflect whether the process has actually failed.

A result of *Unsuspected* signifies that the detector has recently received evidence suggesting that the process has not failed. For example, a message was recently received from it. But of course, the process may have failed since then.

A result of *Suspected* signifies that the failure detector has some indication that the process may have failed. For example, it may be that no message from the process has been received for more than a nominal maximum length of silence (even in an asynchronous system, practical upper bounds can be used as hints).

A ***reliable failure detector*** is one that is always accurate in detecting a process's failure. A result of *Failed* means that the detector has determined that the process has crashed.

Thus, a failure detector may sometimes give different responses to different processes, since communication conditions vary from process to process.

Distributed mutual exclusion:

A collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources. This is the *critical section* problem in the domain of operating systems. In a distributed system, a solution to *distributed mutual exclusion*: one that is based solely on message passing.

In some cases shared resources are managed by servers that also provide mechanisms for mutual exclusion. But in some practical cases, a separate mechanism for mutual exclusion is required.

Algorithms for mutual exclusion

Consider a system of *N* processes *pi*, $i = 1, 2, \ldots\ldots\ldots N$, that do not share variables. The processes access common resources, but they do so in a critical section. For the sake of simplicity, we assume that there is only one critical section. It is straightforward to extend the algorithms we present to more than one critical section.

Assume that the system is asynchronous, that processes do not fail and that message delivery is reliable, so that any message sent is eventually delivered intact, exactly once.

The application-level protocol for executing a critical section is as follows:

enter() // enter critical section – block if necessary
resourceAccesses() // access shared resources in critical section.

exit() // leave critical section – other processes may now enter

Our essential requirements for mutual exclusion are as follows:

ME1: (safety) At most one process may execute in the critical section (CS) at a time.

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

Condition ME2 implies freedom from both ***deadlock*** and ***starvation***. A *deadlock* would involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the critical section by their mutual interdependence. But even without a deadlock, a poor algorithm might lead to *starvation*.

The absence of starvation is a *fairness* condition. Another fairness issue is the order in which processes enter the critical section. It is not possible to order entry to the critical section by the times that the processes requested it, because of the absence of global clocks. But a useful fairness requirement that is sometimes made makes use of the happened-before ordering between messages that request entry to the critical section.

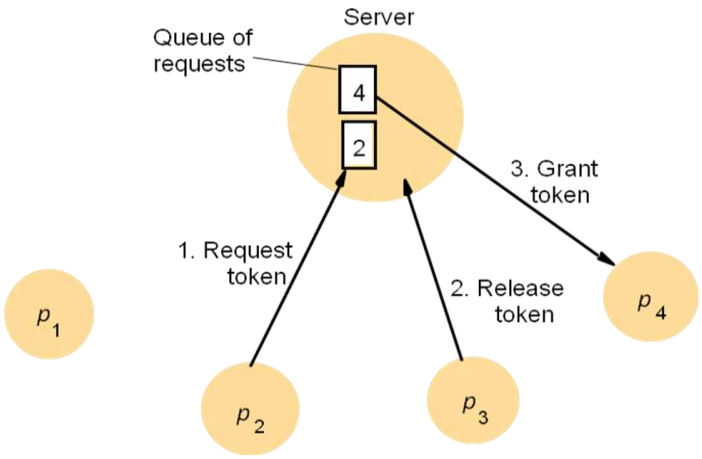
The performance of algorithms for mutual exclusion according to the following criteria:

- **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.

The central server algorithm: The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the critical section.

- 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 shows the Central Server algorithm that managing a mutual exclusion token for a set of processes.

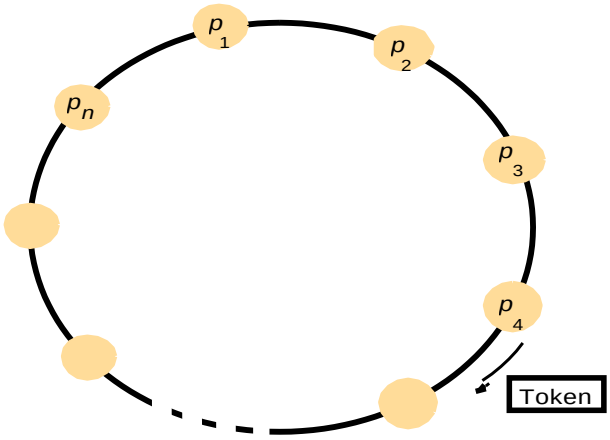


- ME1: safety
- ME2: liveness
- Are satisfied but not
- ME3: ordering
- **Bandwidth**: entering takes two messages(request followed by a grant), **delayed** by the round-trip time; exiting takes one release message, and does **not delay** the exiting process.
- **Throughput** is measured by synchronization delay, the round-trip of a release message and grant message.

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 entering 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.

A ring of processes transferring a mutual exclusion token is shown in fig.



ME1: safety

ME2: liveness

Are satisfied but not

ME3: ordering

Bandwidth: continuously consumes the bandwidth except when a process is inside the CS. Exit only requires one message

Delay: experienced by process is zero message(just received token) to N messages(just pass the token).

Throughput: synchronization delay between one exit and next entry is anywhere from 1 to N message transmission.

Ricart and Agrawala's algorithm:

On initialization

state := RELEASED;

To enter the section

state := WANTED;

Multicast *request* to all processes; request processing deferred here

T := request's timestamp;

Wait until (number of replies received = (*N* – 1));

state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$

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

then

queue *request* from p_i without replying;

else

reply immediately to p_i;

end if

To exit the critical section

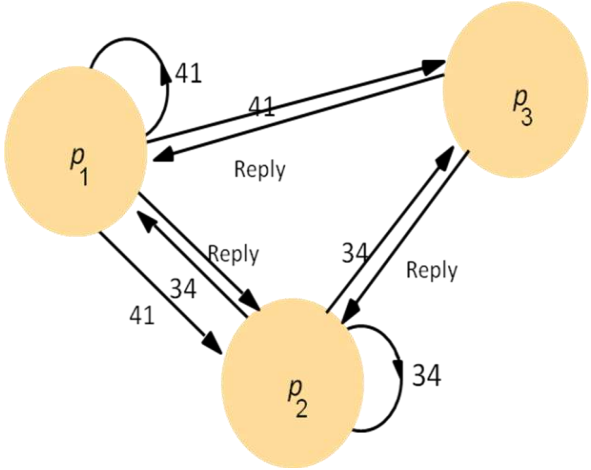
state := RELEASED;

reply to any queued requests;

An algorithm using multicast and logical clocks: Ricart and Agrawala developed an algorithm to implement

- 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 that conditions 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.

To illustrate the algorithm, consider a situation involving three processes, *p*1 , *p*2 and *p*3 , shown in Figure.



→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.

Maekawa's voting algorithm: In 1985, Maekawa observed that in order for a process to enter a critical section,

- 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.

Maekawa associated a *voting* set Vi associated with each process pi. i.e., *pi* ($i = 1, 2, \ldots N$),

$$V_i \subseteq \{p_1, p_2, \ldots, p_N\}$$

$$such\ that\ for\ all\ i, j = 1,2,\ldots N:$$

$$p_i \in V_i$$

$$V_i \cap V_j \neq \varnothing$$

$$|V_i| = K$$

- 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.

Maekawa's algorithm is shown below to obtain entry to the critical section, a process *pi* sends *request* messages to all *K* members of *Vi* (including itself). *pi* cannot enter the critical section until it has received all *K reply* messages. When a process *pj* in *Vi* receives *pi* 's *request* message, it sends a *reply* message immediately.

On initialization
    state := RELEASED;
    voted := FALSE;
For p_i *to enter the critical section*
    state := WANTED;
    Multicast *request* to all processes in V_i;
    Wait until (number of replies received = *K*);
    state := HELD;
On receipt of a request from p_i *at* p_j
    if (*state* = HELD *or voted* = TRUE)

then
queue *request* from p_i without replying;
else
send *reply* to p_i;
voted := TRUE;
end if

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

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

- 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** is the same as Ricart and Agrawala's algorithm, one round-trip time.
- **Synchronization delay** is one round-trip time.

Fault tolerance: The main points to consider when evaluating the above algorithms with respect to fault tolerance are:

→ 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.

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 *electedi* = ⊥ or *electedi* = *P*,
  Where P is chosen as the non-crashed process at the end of run with the largest

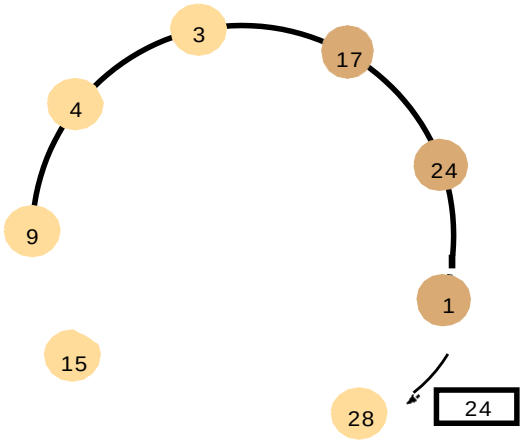
identifier.

- E2(liveness): All processes Pi participate in election process and eventually set *electedi* ‡ ⊥ – or crash.

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

Note:

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

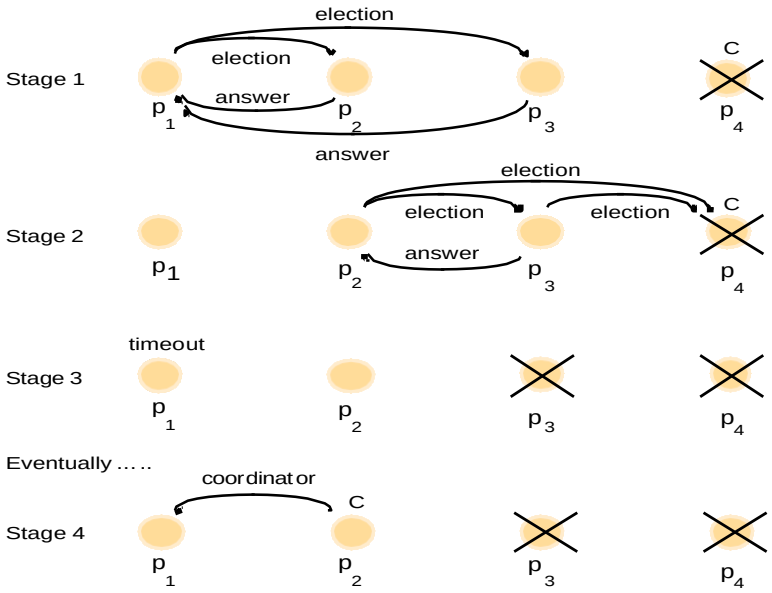
→ If only a single process starts an election, the worst-performance case is then the anti-clockwise neighbour has the highest identifier. A total of N-1 messages is used to reach this neighbour. Then further N messages are required to announce its election.

→ The elected message is sent N times. Making **3N-1 messages in all**.

- **Turnaround time** is also 3N-1 sequential message transmission time

The bully algorithm: The bully algorithm

- Allows processes to crash during an election, although it assumes that 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.



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

- 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.
- If a process receives a coordinator message, it sets its variable elected_i to be the coordinator ID.
- 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.

➔ **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.

Multicast communication: A *multicast operation* is an operation that sends a single message from oneprocess to each of the members of a group of processes. In such a way, the membership of the group is transparent to the sender. There is a range of possibilities in the desired behavior of a multicast. The simplest multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1. ***Fault tolerance based on replicated services*:** A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.
2. ***Discovering services in spontaneous networking***: It defines service discovery in the context of spontaneous networking. Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
3. ***Better performance through replicated data*:** Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value are multicast to the processes managing the replicas.
4. ***Propagation of event notifications*:** Multicast to a group may be used to notify processes when something happens. For example, in Face book, when someone changes their status, all their friends receive notifications. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers.

IP multicast – An implementation of multicast communication

For group communication, different group communication protocols were used. In addition to these IP multicast is provided which presents Java's API to it through the ***Multicast Socket*** class.

IP multicast: *IP multicast* is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers i.e., ports belonging to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group.

A *multicast group* is specified by a Class D Internet address – that is, an address whose first 4 bits are 1110 in IPv4.

Being a member of a multicast group allows a computer to receive IP packets sent to the group. The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups. It is possible to send data grams to a multicast group without being a member.

An application program performs multicasts by sending UDP data grams with multicast addresses and ordinary port numbers. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group.

At the IP level, a computer belongs to a multicast group when one or more of its processes have sockets that belong to that group. When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

- ***Multicast routers*:** IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single data grams to routers on other networks, where they are again multicast to local members.
- ***Multicast address allocation***: Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the ***Internet Assigned Numbers Authority*** (IANA). The management of this address space is reviewed annually, with current practice documented in RPC 3171. This document defines a partitioning of this address space into a number of blocks, including:

  → Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.

  → Internet Control Block (224.0.1.0 to 224.0.1.225).

  → Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.

  → Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic (to constrain propagation).

Multicast addresses may be permanent or temporary. Permanent groups exist even when there are no members i.e., their addresses are assigned by IANA and span the various blocks mentioned above.

For example, 224.0.1.1 in the Internet block is reserved for the Network Time Protocol (NTP), as discussed in Chapter 14, and the range 224.0.6.000 to 224.0.6.127 in the ad hoc block is reserved for the ISIS project.

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

Java API to IP multicast : The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*, which is a subclass of *DatagramSocket* with the additional capability of being able to join multicast groups. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a specified local port or any free local port.

A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port. A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket.

Multicast peer joins a group and sends and receives datagrams

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

→ In the example, the arguments to the *main* method specify a message to be multicast and the multicast address of a group (for example, *"228.5.6.7").*

→ After joining that multicast group, the process makes an instance of *DatagramPacket* containing the message and sends it through its multicast socket to the multicast group address at port 6789.

→ After that, it attempts to receive three multicast messages from its peers via its socket, which also belongs to the group on the same port.

→ When several instances of this program are run simultaneously on different computers, all of them join the same group, and each of them should receive its own message and the messages from those that joined after it.

→ 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.

Reliability and ordering: 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. In most cases, this service would require that all members receive request messages in the same order as one another.

2. *Discovering services in spontaneous networking*: One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond.

An occasional lost request is not an issue when discovering services. In fact, Jini uses IP multicast in its protocol for discovering services.

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

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

- → These examples suggest that some applications require a multicast protocol that is more reliable than IP multicast. In particular, there is a need for *reliable multicast*, in which any message transmitted is either received by all members of a group or by none of them.
- → The examples also suggest that some applications have strong requirements for ordering, the strictest of which is called *totally ordered multicast*, in which all of the messages transmitted to a group reach all of the members in the same order.

Unit-6 Transactions & Replications

Syllabus: Introduction, System Model and Group Communication, Concurrency Control in Distributed Transactions, Distributed Dead Locks, Transaction Recovery; Replication-Introduction, Passive (Primary) Replication, Active Replication

Topic 01: INTRODUCTION

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 behavior 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 2*f*+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:

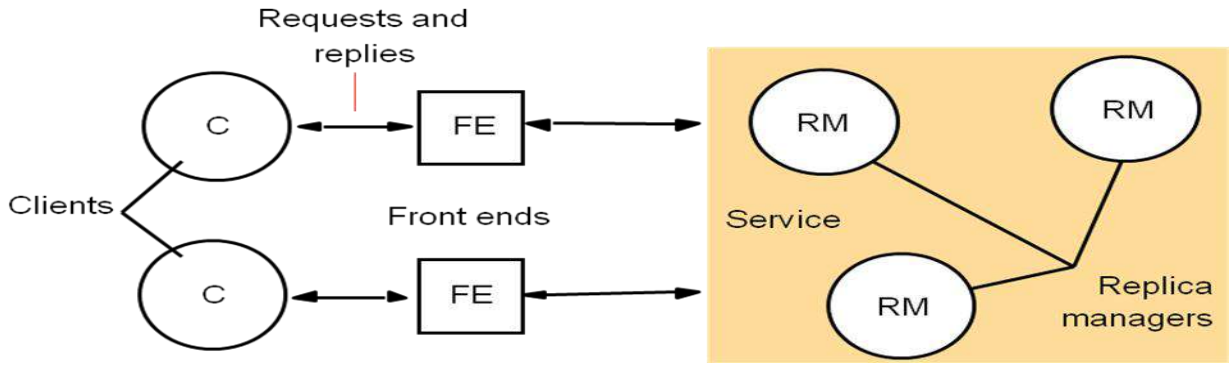
- each *logical* object is implemented by a collection of *physical* copies called *replicas*
  - the replicas are not necessarily consistent all the time (some may have received updates, not yet conveyed to the others)
- we assume an asynchronous system where processes fail only by crashing and generally 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?)

1. 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)
2. 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)
3. execution
   - the RMs execute the request (sometimes tentatively)
4. agreement
   - RMs *agree* on the effect of the request, .e.g perform 'lazily' or immediately
5. 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*'

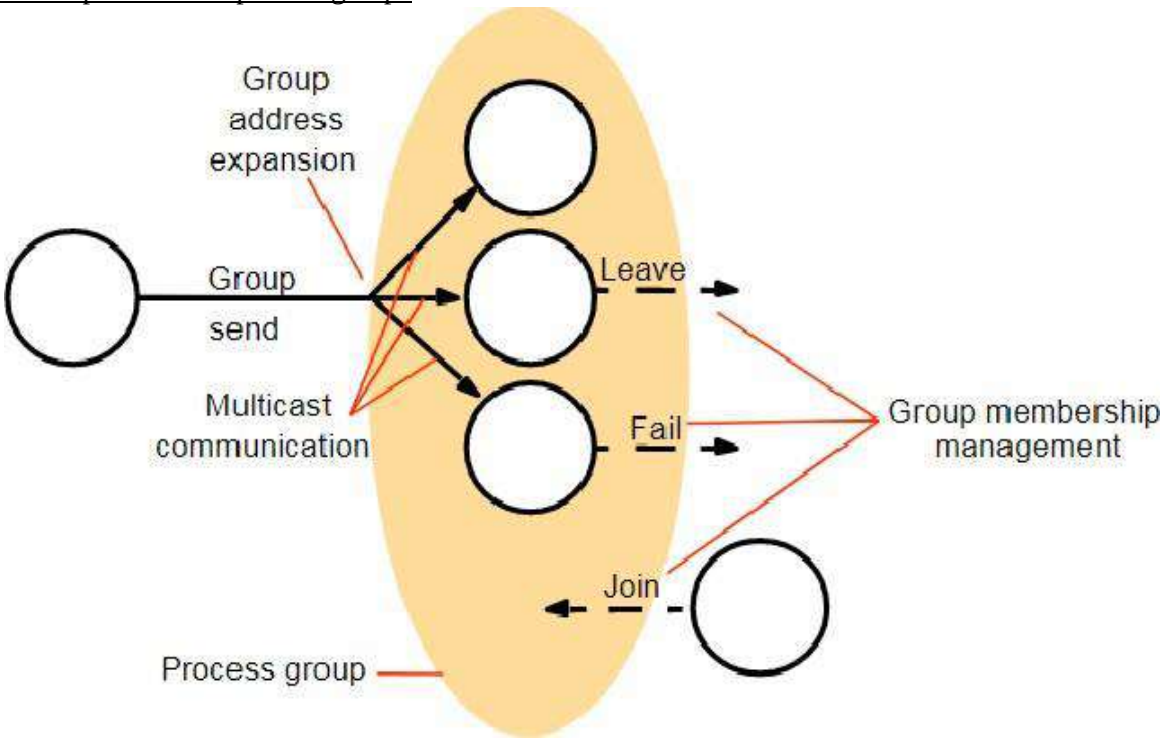
Total ordering: if a correct RM handles r before *r*', then any correct RM handles *r* before *r*' Bayou sometimes executes responses tentatively so as to be able to reorder them

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

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

<u>Services provided for process groups</u>



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)

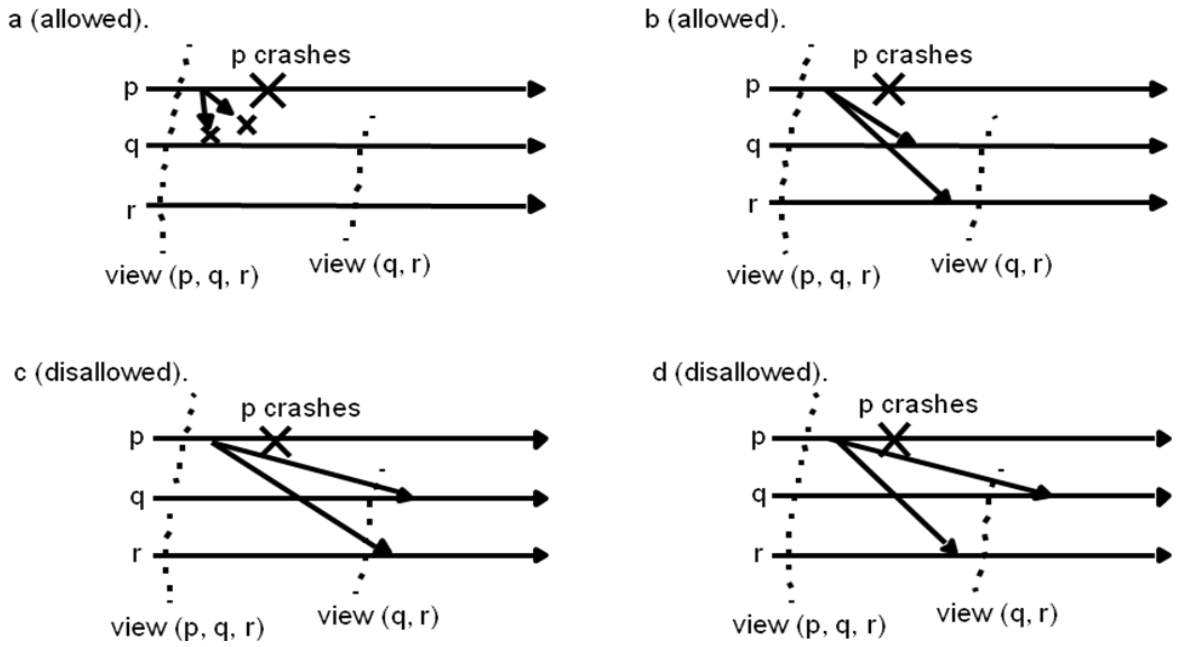


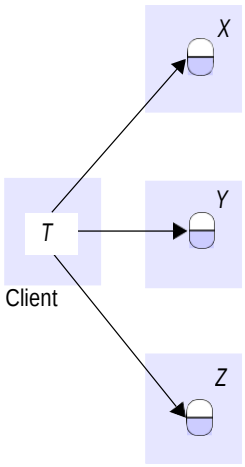
Figure : View-synchronous group communication

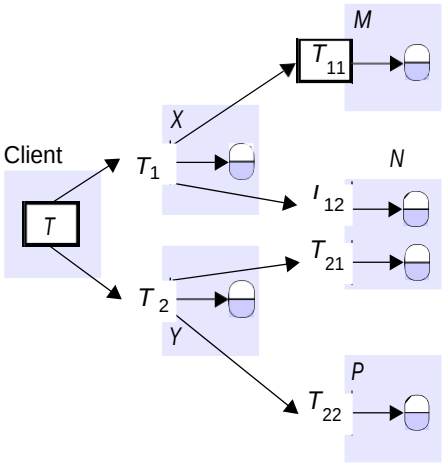
Topic 03: Distributed transactions – introduction

- 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

Flat and nested distributed transaction: The distributed transactions are two types:

1. Flat Transaction
2. Nested Transaction

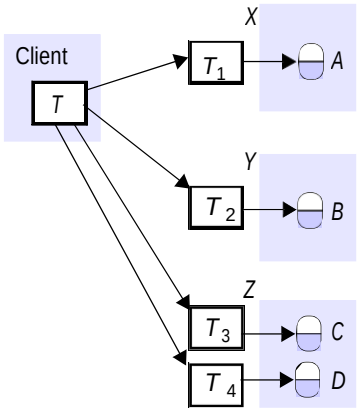




Flat Transaction **Nested Transaction**

- In a flat transaction, each transaction is decoupled from and independent of other transactions in the system. Another transaction cannot start in the same thread until the current transaction ends. Flat transactions are the most prevalent model and are supported by most commercial database systems.
- A nested transaction is a database transaction that is started by an instruction within the scope of an already started transaction.This means that a commit in an inner transaction does not necessarily persist updates to the system.

Nested Banking Transaction:



a.withdraw(10)

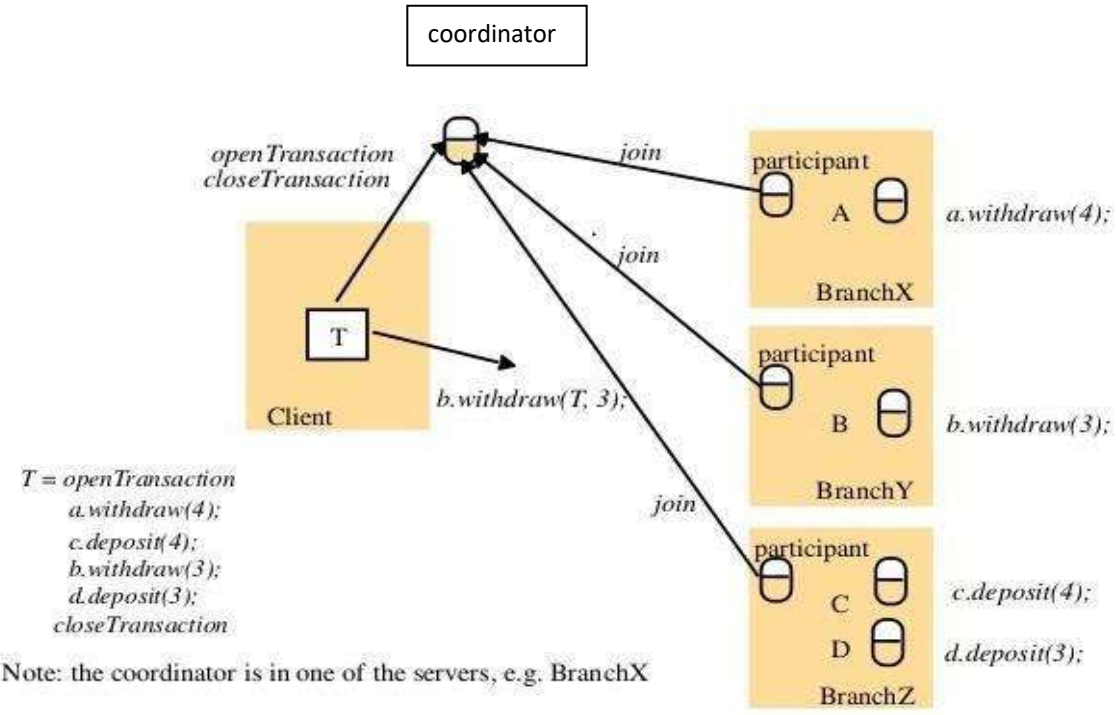
b.withdraw(20)

c.deposit(10)

d.deposit(20)

T = openTransaction
openSubTransaction
a.withdraw(10);
openSubTransaction
b.withdraw(20);
openSubTransaction
c.deposit(10);
openSubTransaction
d.deposit(20);
closeTransaction

Distributed Banking Transaction:



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**	**Wait for U**			
			Read(A)	**at X**	**Wait 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

- Each transaction is validated before it is allowed to commit
  - I. Transaction numbers assigned at start of validation
  - II. Transactions serialized according to transaction numbers
  - III. Validation takes place in phase 1 of 2PC protocol
- Consider the following interleaving of T and U
  - I. T before U at X and U before T at Y

    Suppose T & U start validation at about the same time

    X does T first

    Y does U first

    No parallel Validation – Commitment deadlock.

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)	

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

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

U		***V***		***W***	
D .deposit(10)	***Lock D***				
		b.deposit(10)	***Lock B at Y***		
U->V at Y				***c.deposit(30)***	***Lock C at Z***
b.withdraw(30)	***Wait at Y***	***V->W at Z***			
		c.withdraw(20)	***Wait at Z***	***W->U at X***	
				a.withdraw(20)	***Wait at X***

Figure: Interleavings of transactions U,V and W

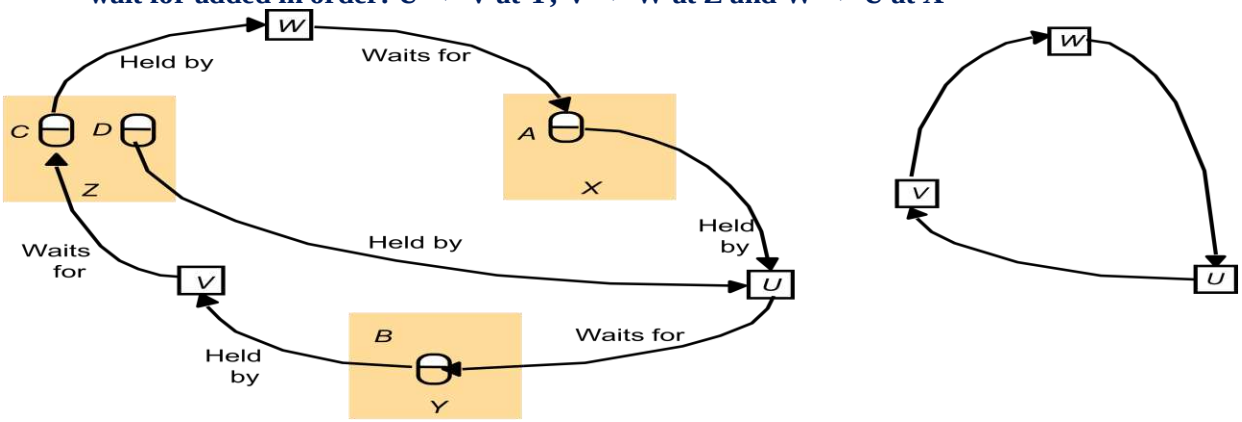
Sub topic 5.2 Distributed deadlock

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 dead lock cycle has alternate edges showing wait for and held by**
- **wait for added in order: U - > V at Y; V - > W at Z and W - > U at X**



Figure; Distributed deadlock

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)

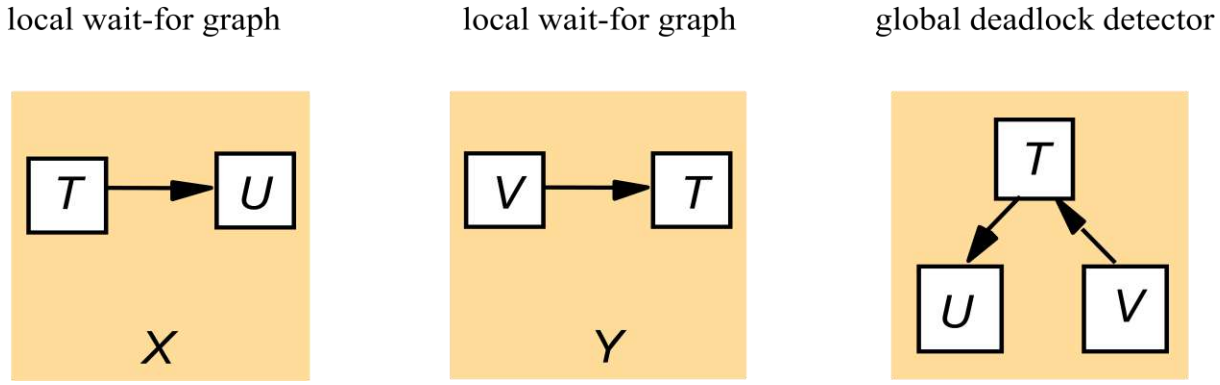


Figure: Local and global wait-for graphs

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

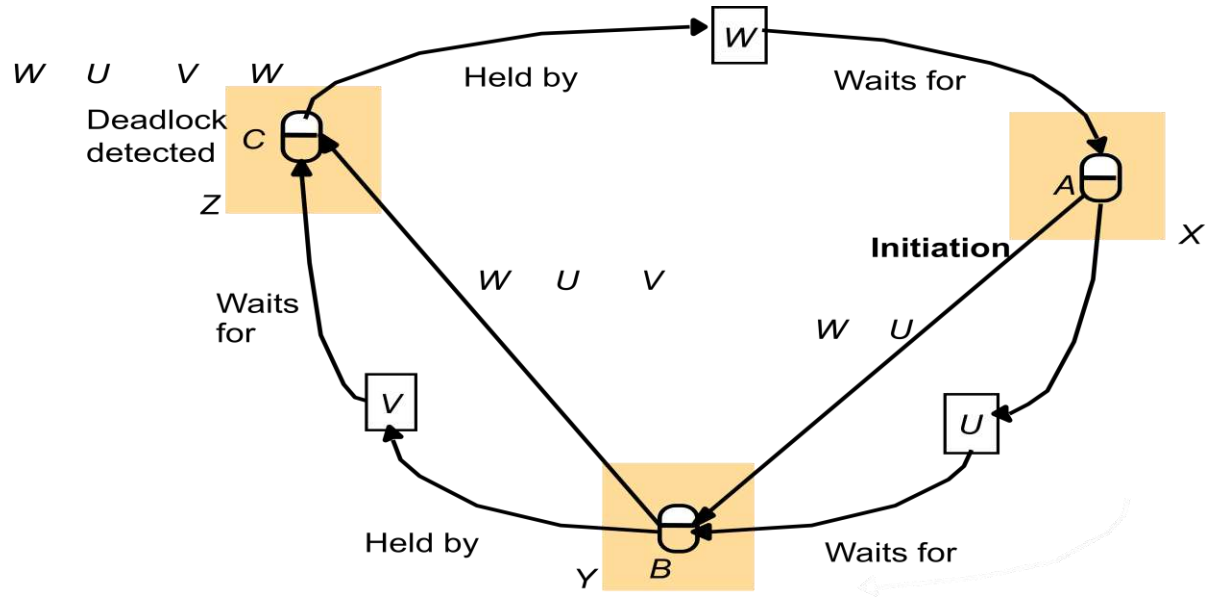
- each coordinator records whether its transactions are active or waiting
  - the local lock manager tells coordinators if transactions start/stop waiting
  - when a transaction is aborted to break a deadlock, the coordinator tells the participants, locks are removed and edges taken from wait-for graphs

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

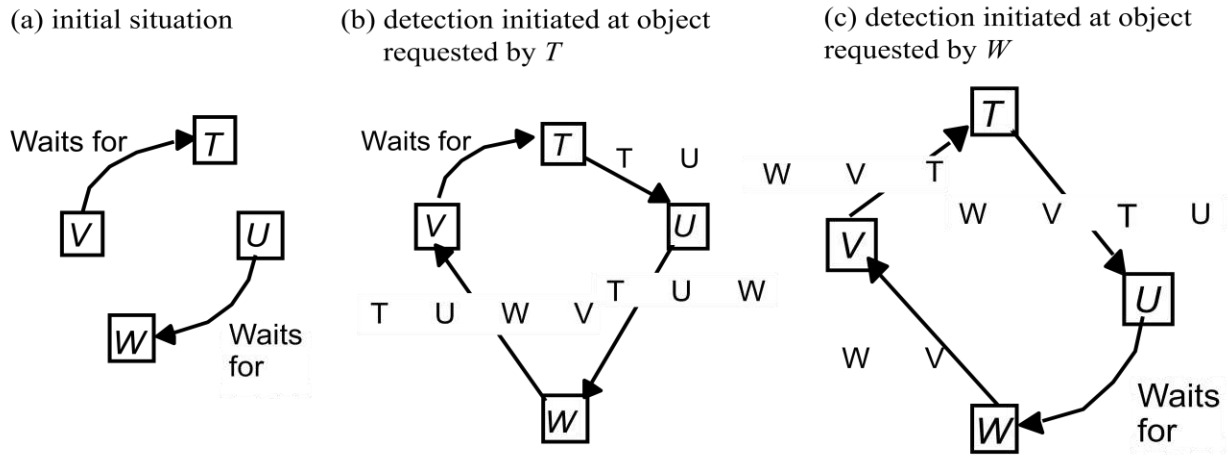


Figure: Two probes initiated

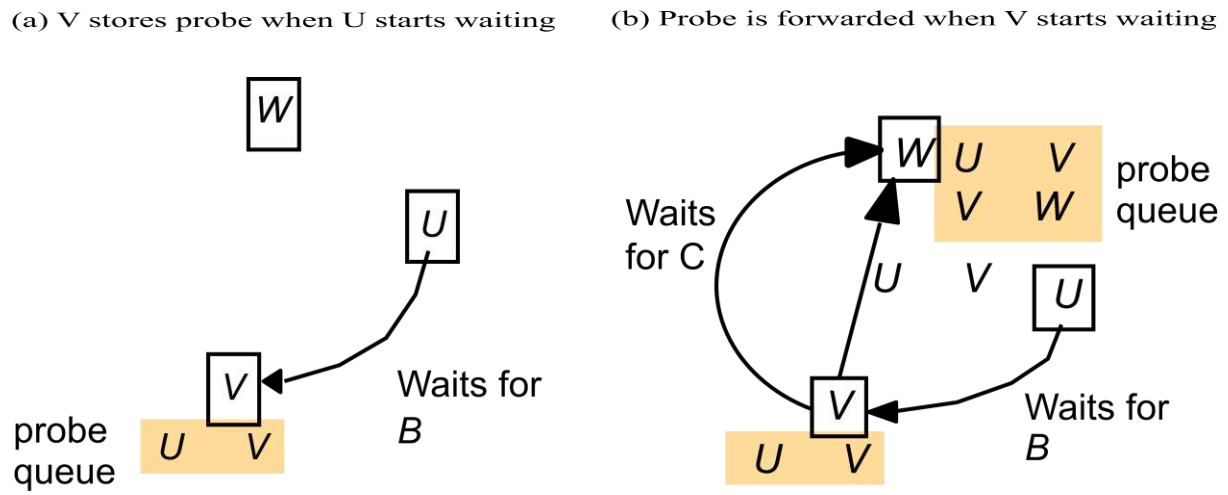


Figure: Probes travel downhill

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 <*objectID, Pi*>, where *Pi* is the position in the recovery file of the value of the object.

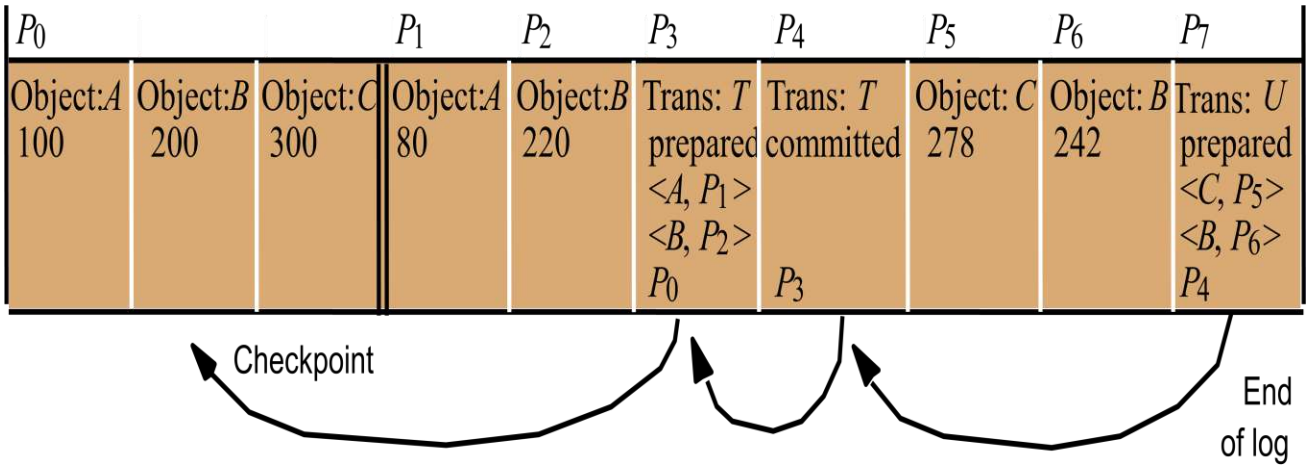
For distributed transactions we need information relating to the 2pc as well as object values, that is:

- Transaction status(committed, prepared or aborted)
- Intentions List

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



- 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 – Reorganizing 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*
$A \rightarrow P_0$	$A \rightarrow P_1$
$B \rightarrow P_0'$	$B \rightarrow P_2$
$C \rightarrow P_0''$	$C \rightarrow P_0''$

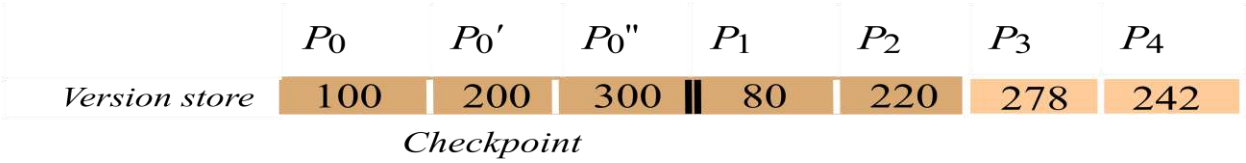


Figure: Shadow versions

Recovery of the two-phase commit protocol

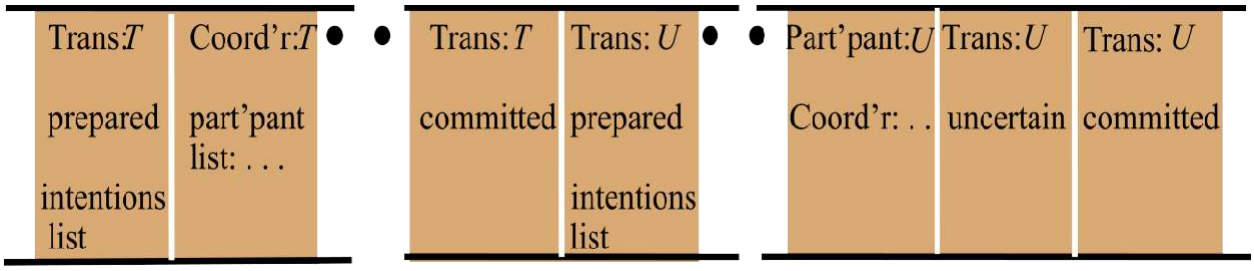
- The above recovery scheme is extended to deal with transactions doing the 2PC protocol when a server fails

- 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



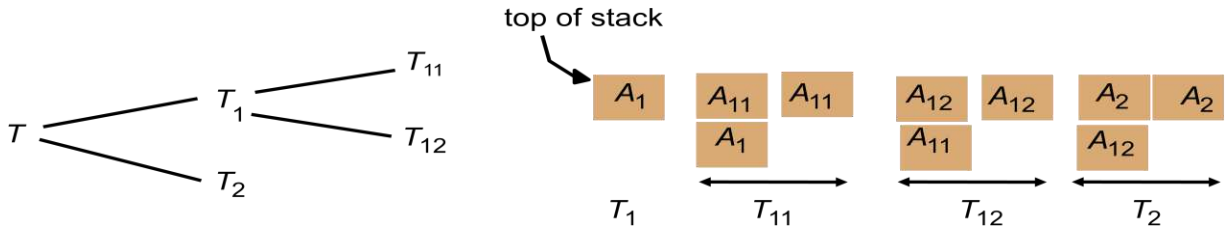
- entries in log for
  - T where server is coordinator (prepared comes first, followed by the coordinator entry, then committed – done is not shown)
  - And U where server is participant (prepared comes first followed by the participant entry, then uncertain and finally committed)
  - These entries will 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.

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

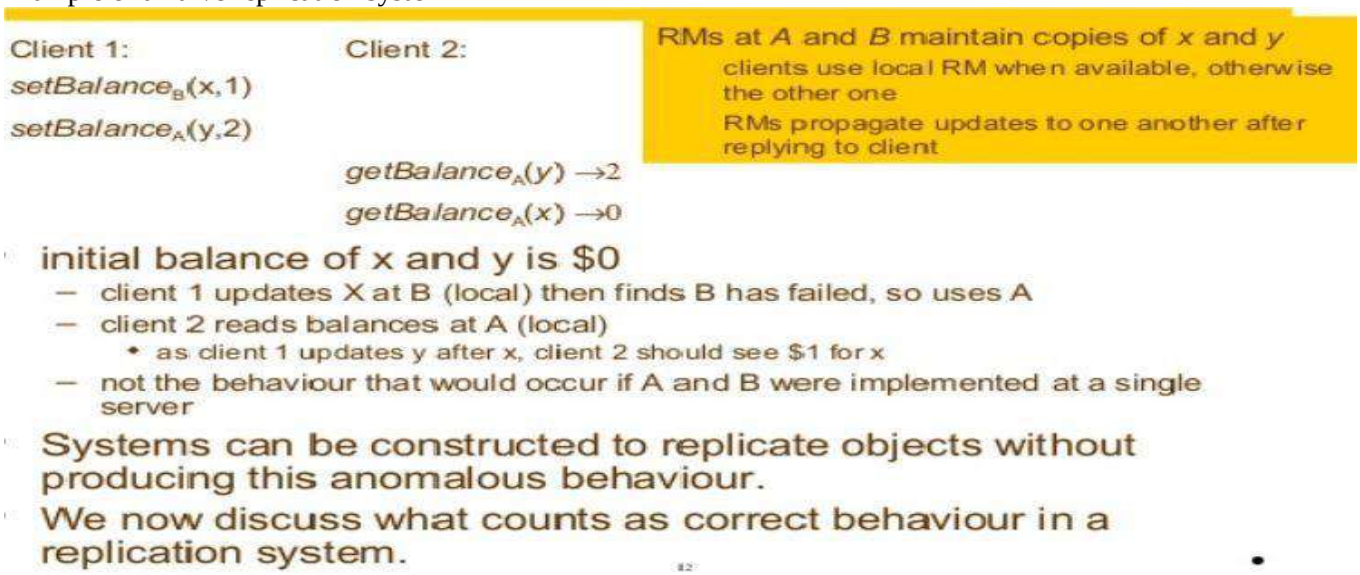


Figure: Native Replication System

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

<u>Sequential consistency</u>

- 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

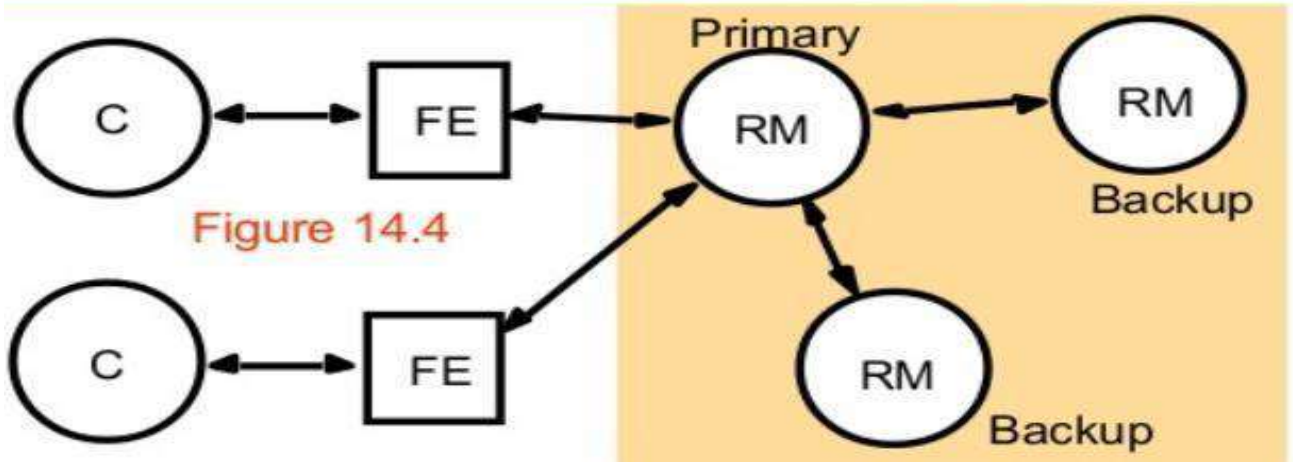


Figure 14.4

The FE has to find the primary, e.g. after it crashes and another takes over

Passive (primary-backup) replication. Five phases.

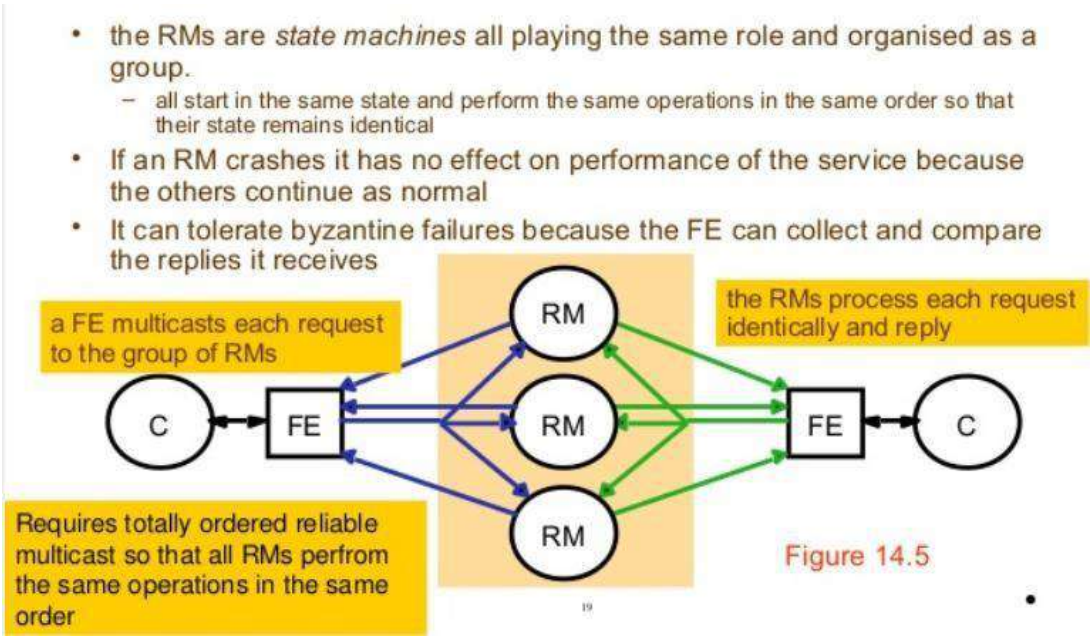
- The five phases in performing a client request are as follows:
- 1. Request:
  - a FE issues the request, containing a unique identifier, to the primary RM
- 2. Coordination:
  - the primary performs each request atomically, in the order in which it receives it relative to other requests
  - it checks 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



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