Index

UNIT I

Characterization of Distributed Systems: Introduction, Examples of Distributed systems, Resource Sharing and Web, Challenges.

System Models: Introduction, Architectural models, Fundamental models.

Characterization of Distributed Systems: Introduction

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

A distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This simple definition covers the entire range of systems in which networked computers can usefully be deployed.

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 has the following significant consequences:

***Concurrency*:** In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network.

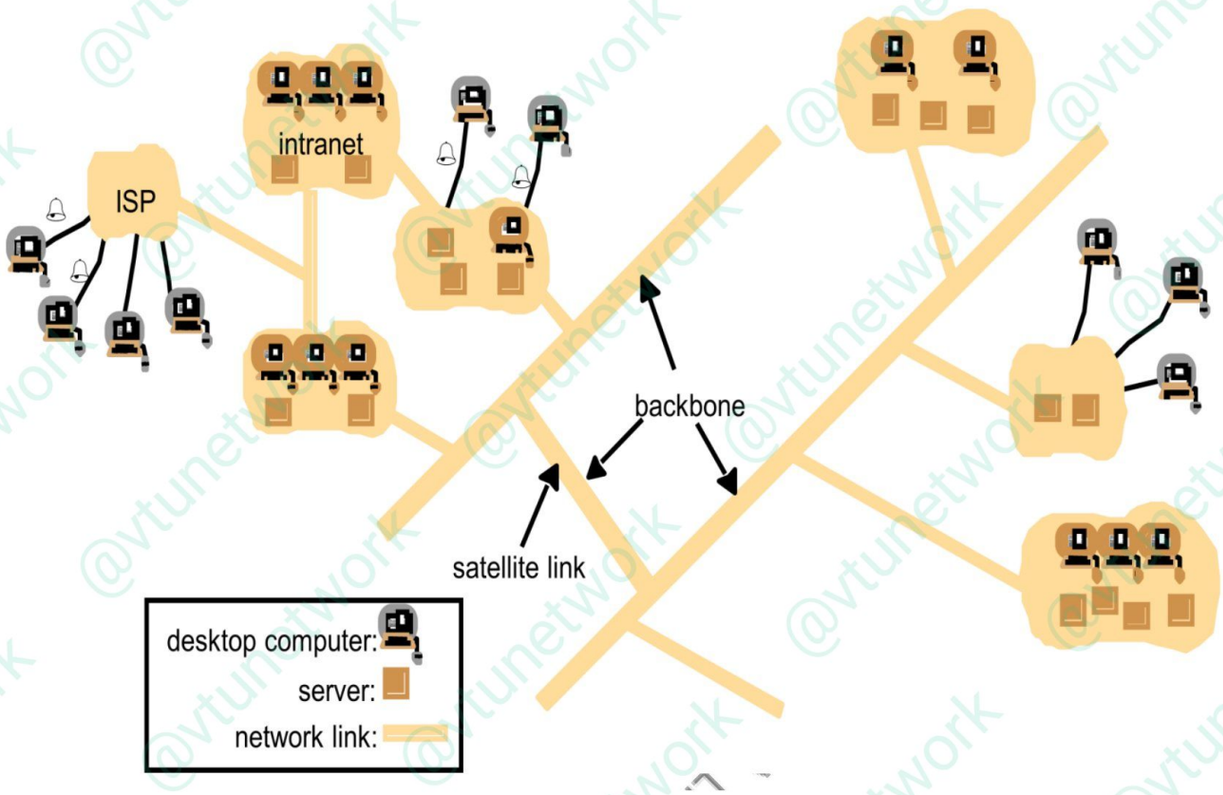
***No global clock*:** When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network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

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

Examples of Distributed systems

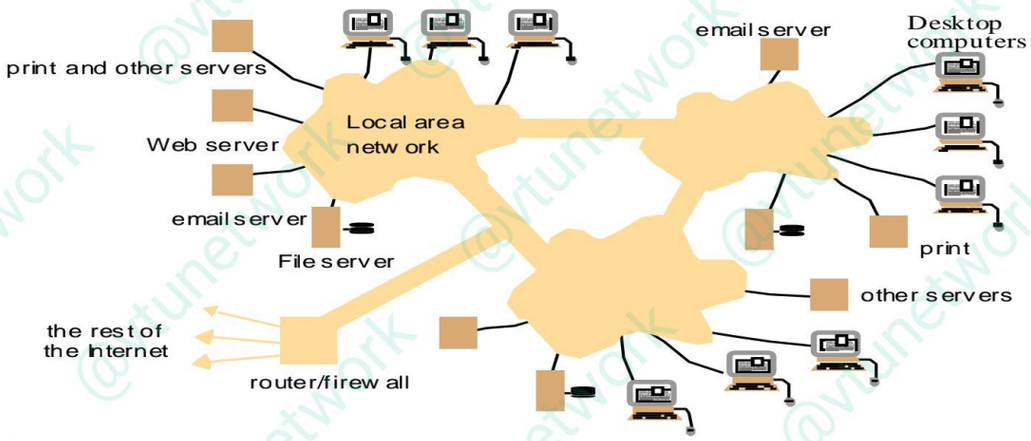
To place distributed systems in a realistic context through examples: the Internet, an intranet and mobile computing.

1. The Internet (Figure 1) :
   - A vast interconnected collection of computer networks of many different types.
   - Passing message by employing a common means of communication (Internet Protocol).
   - The web is not equal to the Internet.



2. Intranets (Figure 2):

- An intranet is a private network that is contained within an enterprise.
- It may consist of many interlinked local area networks and also use leased lines in the Wide Area Network.
- It separately administrated and enforces local security policies.
- It is connected to the Internet via a router
- It uses firewall to protect an Intranet by preventing unauthorized messages leaving or entering
- Some are isolated from the Internet
- Users in an intranet share data by means of file services.



3. **Mobile and Ubiquitous Computing (Figure 1.3)**

   a. Distributed systems techniques are equally applicable to mobile computing involving laptops, PDAs and wearable computing devices.

   b. Mobile computing (nomadic computing) - perform of computing tasks while moving (nomadic computing)

   c. Ubiquitous computing - small computers embedded in appliances

      i. harness of many small, cheap computation devices

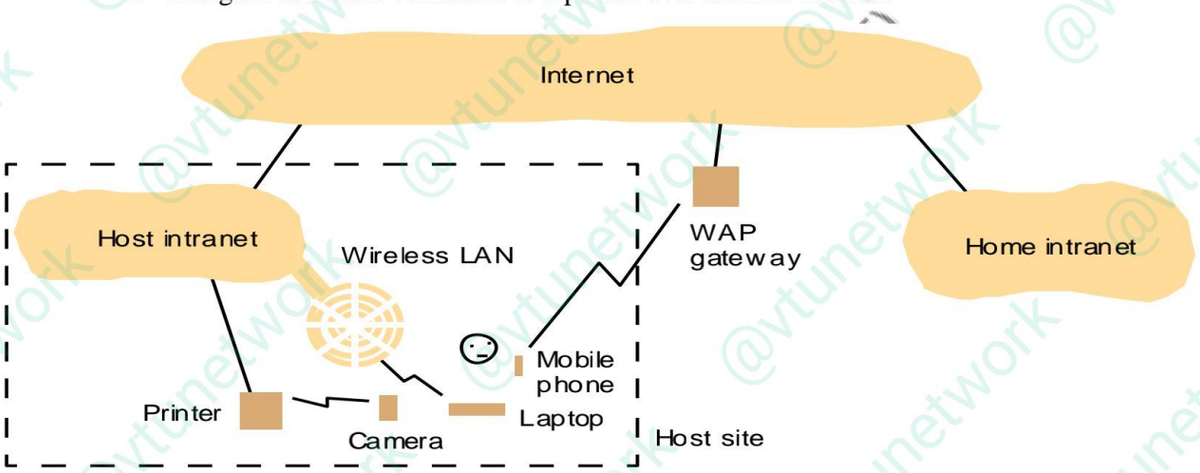
      ii. It benefits users while they remain in a single environment such as home.

   Distributed In Figure 3 user has access to three forms of wireless connection**:**

   d. A laptop is connected to host's wireless LAN.

   e. A mobile (cellular) phone is connected to Internet using Wireless Application Protocol (WAP) via a gateway.

   f. A digital camera is connected to a printer over an infra-red link.



Resource Sharing and Web

- Equipments are shared to reduce cost. Data shared in database or web pages are high-level resources which are more significant to users without regard for the server or servers that provide these.
- Patterns of resource sharing vary widely in their scope and in how closely users work together:
  - Search Engine: Users need no contact between users
  - Computer Supported Cooperative Working (CSCW): Users cooperate directly share resources. Mechanisms to coordinate users' action are determined by the pattern of sharing and the geographic distribution.
- 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.
- Server is 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 a client.

- An executing web browser is a client. It communicates with a web server to request web pages from it.
- When a client invokes an operation upon the server, it is called the remote invocation.
- Resources may be encapsulated as objects and accessed by client objects. In this case a client object invokes a method upon a server object.

The World Wide Web (WWW)

- WWW is an evolving system for publishing and accessing resources and services across Internet. Web is an open system. Its operations are based on freely published communication standards and documents standards.
- Key feature: Web provides a hypertext structure among the documents that it stores. The documents contain links - references to other documents or resources. The structures of links can be arbitrarily complex and the set of resources that can be added is unlimited.
- Three main standard technological components:
    - HTML (Hypertext Makeup Language) specify the contents and layout of web pages.
        - Contents: text, table, form, image, links, information for search engine, ...;
        - Layout: text format, background, frame, ...
    - URL (Uniform Resource Location): identify a resource to let browser find it.
        - scheme : scheme-specific-location
        - http://web.cs.twsu.edu/ (HyperText Transfer Protocol)
    - URL (continued):
        - ftp://ftp.twsu.edu/ (File Transfer Protocol)
        - telnet://kirk.cs.twsu.edu (log into a computer)
        - mailto:chang@cs.twsu.edu (identify a user's email address)
    - HTTP (HyperText Transfer Protocol) defines a standard rule by which browsers and any other types of client interact with web servers. Main features:
        - Request-reply interaction
        - Content types may or may not be handled by browser - using plug-in or external helper One resource per request - Several requests can be made concurrently.
        - Simple access control
        - Services and dynamic pages
        - form - Common Gateway Interface program on server (Perl)
        - JavaScript (download from server and run on local computer)
        - Applet (download from server and run on local computer)

Challenges

As distributed systems are getting complex, developers face a number of challenges:

- Heterogeneity
- Openness
- Security
- Scalability
- Failure handling
- Concurrency
- Transparency
- Quality of service

Heterogeneity:

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

- Hardware devices: computers, tablets, mobile phones, embedded devices, etc.
- Operating System: Ms Windows, Linux, Mac, Unix, etc.
- Network: Local network, the Internet, wireless network, satellite links, etc.
- Programming languages: Java, C/C++, Python, PHP, etc.
- Different roles of software developers, designers, system managers

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

Middleware : The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the difference in operating systems and hardware

Heterogeneity and mobile code : The term mobile code is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

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. In other words, distributed systems designers must hide the complexity of the systems as much as they can.

- 8 forms of transparency:
  - Access transparency – access to local an remote resources using identical operations
  - Location transparency – access to resources without knowing the physical location of the machine
  - Concurrency transparency – several processes operate concurrently without interfering each other
  - Replication transparency – replication of resources in multiple servers. Users are not aware of the replication
  - Failure transparency – concealment of faults, allows users to complete their tasks without knowing of the failures

- Mobility transparency – movement of resources and clients within a system without affecting users operations
- Performance transparency – systems can be reconfigured to improve performance by considering their loads
- Scaling transparency – systems and applications can be expanded without changing the structure or the application algorithms

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. If the well-defined interfaces for a system are published, it is easier for developers to add new features or replace sub-systems in the future. Example: Twitter and Facebook have API that allows developers to develop their own software interactively.

Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

Security

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

Scalability

Distributed systems must be scalable as the number of user increases. The scalability is defined by B. Clifford Neuman as

A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity

Scalability has 3 dimensions:

- Size
  - Number of users and resources to be processed. Problem associated is overloading
- Geography
  - Distance between users and resources. Problem associated is communication reliability
- Administration
  - As the size of distributed systems increases, many of the system needs to be controlled. Problem associated is administrative mess

Failure Handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. The handling of failures is particularly difficult.

- Dealing with failures in distributed systems:

- Detecting failures – known/unknown failures
- Masking failures – hide the failure from become severe. E.g. retransmit messages, backup of file data
- Tolerating failures – clients can be designed to tolerate failures – e.g. inform users of failure and ask them to try later
- Recovery from failures - recover and rollback data after a server has crashed
- Redundancy- the way to tolerate failures – replication of services and data in multiple servers

Quality of service

- The main nonfunctional properties of distributed systems that affect the quality of service experienced by users or clients are: reliability, security, performance, adaptability.
- Reliability
- Security
- Performance
- Adaptability

System Models: Introduction

- Architectural Models
  - Client-Server Model
  - Peer-Peer Model
- Fundamental Models
  - Interaction Model
  - Failure Model
  - Security Model

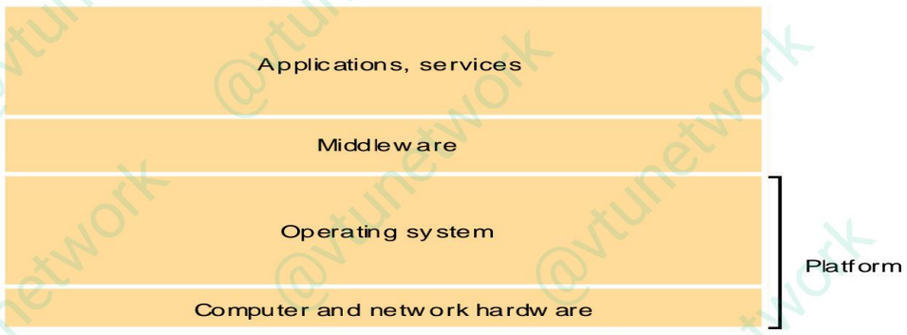
Architectural Models:

- An architectural model of a distributed system is concerned with the placement of its parts and the relationships between them.
- The architecture of a system is its structure in terms of separately specified components.
- The overall goal is to ensure that the structure will meet present and likely future demands on it.
- Major concerns are to make the system:
  - Reliable
  - Manageable
  - Adaptable
  - Cost-effective
- An architectural Model of a distributed system first simplifies and abstracts the functions of the individual components of a distributed system.
- An initial simplification is achieved by classifying processes as:
  - Server processes
  - Client processes
  - Peer processes
    - Cooperate and communicate in a symmetric manner to perform a task.

Software Layers

- Software architecture referred to:
  - The structure of software as layers or modules in a single computer.
  - The services offered and requested between processes located in the same or different computers.

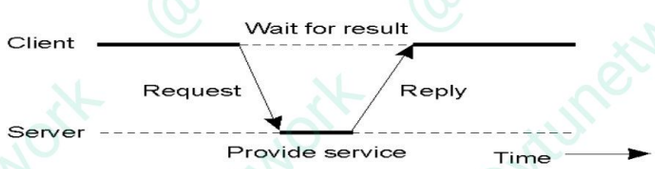
- Software architecture is breaking up the complexity of systems by designing them through layers and services.
  - Layer: a group of related functional components.
  - Service: functionality provided to the next layer.



- Platform
  - The lowest-level hardware and software layers are often referred to as a platform for distributed systems and applications.
    - These low-level layers provide services to the layers above them, which are implemented independently in each computer.
    - These low-level layers bring the system's programming interface up to a level that facilitates communication and coordination between processes.
- Middleware
  - A layer of software whose purpose is
    - to mask heterogeneity presented in distributed systems.
    - To provide a convenient programming model to application developers.
  - Major Examples of middleware are:
    - Sun RPC (Remote Procedure Calls)
    - OMG CORBA (Common Request Broker Architecture)
    - Microsoft D-COM (Distributed Component Object Model)
    - Sun Java RMI

Client-Server model

- Most often architecture for distributed systems.
- Client process interact with individual server processes in a separate host computers in order to access the shared resources
- Servers may in turn be clients of other servers.
  - E.g. a web server is often a client of a local file server that manages the files in which the web pages are stored.
  - E.g. a search engine can be both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers.



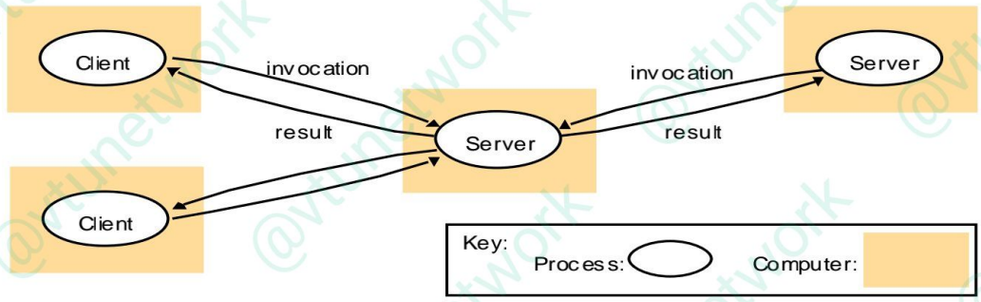
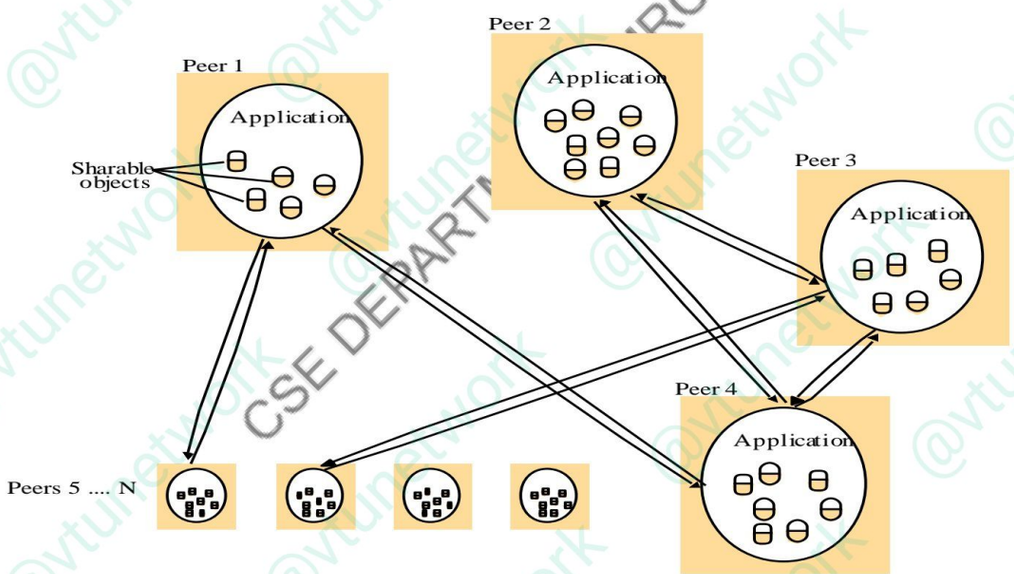


Figure 4. Clients invoke individual servers

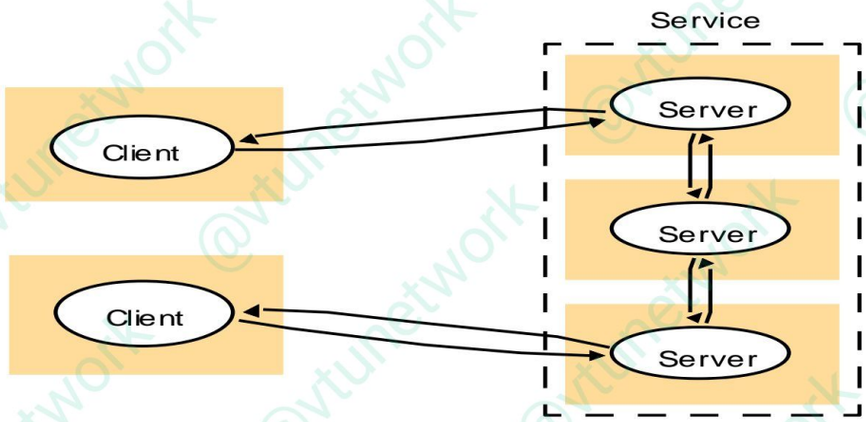
Peer-to-Peer model

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

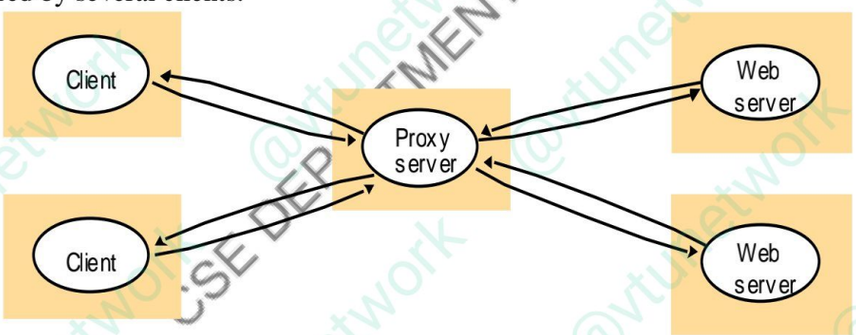


Variants of Client Sever Model

- The problem of client-server model is placing a service in a server at a single address that does not scale well beyond the capacity of computer host and bandwidth of network connections.
- To address this problem, several variations of client-server model have been proposed.
- Some of these variations are discussed in the next slide.
- **Services provided by multiple servers**
  - Services may be implemented as several server processes in separate host computers interacting as necessary to provide a service to client processes.
  - E.g. cluster that can be used for search engines.



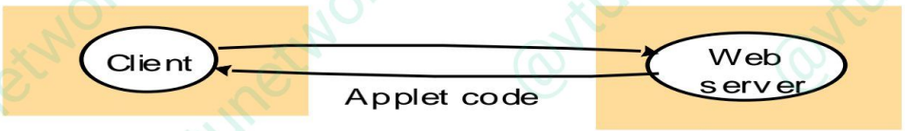
- **Proxy servers and caches**
  - A cache is a store of recently used data objects.
  - When a new object is received at a computer it is added to the cache store, replacing some existing objects if necessary.
  - When an object is needed by a client process the caching service first checks the cache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



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

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

a) client request results in the downloading of applet code



b) client interacts with the applet

- **Mobile agents**
  - A running program (code and data) that travels from one computer to another in a network carrying out of a task, usually on behalf of some other process.
  - Examples of the tasks that can be done by mobile agents are:
    - To collecting information.
    - To install and maintain software maintain on the Computers within an organization.
    - To compare the prices of products from a number of vendors.
    - Mobile agents are a potential security threat to the resources in computers that they visit.
    - The environment receiving a mobile agent should decide on which of the local resources to be allowed to use.
    - Mobile agents themselves can be vulnerable
    - They may not be able to complete their task if they are refused access to the information they need.

➔ **Network computers**
- It downloads its operating system and any application software needed by the user from a remote file server.
- Applications are run locally but the file are managed by a remote file server.
- Network applications such as a Web browser can also be run.

➔ **Thin clients**
- It is a software layer that supports a window-based user interface on a computer that is local to the user while executing application programs on a remote computer.
- This architecture has the same low management and hardware costs as the network computer scheme.
- Instead of downloading the code of applications into the user's computer, it runs them on a compute server.
- Compute server is a powerful computer that has the capacity to run large numbers of application simultaneously.
- The compute server will be a multiprocessor or cluster computer running a multiprocessor version of an operation system such as UNIX or Windows.

➔ **Performance Issues**
- Performance issues arising from the limited processing and communication capacities of computers and networks are considered under the following subheading:
  - Responsiveness

- ❖ E.g. a web browser can access the cached pages faster than the non-cached pages.
- ❖ Throughput
- ❖ Load balancing
  - ❖ E.g. using applets on clients, remove the load on the server.

➔ **Quality of service**

- ➢ The ability of systems to meet deadlines.
- ➢ It depends on availability of the necessary Computing and network resources at the appropriate time.
- ➢ This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time.
  - ❖ E.g. the task of displaying a frame of video

Fundamental Models:

- Fundamental Models deal with a more formal description of the properties that are common in all of the architectural models.
- Fundamental Models are concerned with a more formal description of the properties that are common in all of the architectural models.
- All architectural models are composed of processes that communicate with each other by sending messages over a computer networks.
- Aspects of distributed systems that are discussed in fundamental models are:

Interaction model:

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

- Distributed systems are composed of many processes, interacting in the following ways:
  - Multiple server processes may cooperate with one another to provide a service
    - E.g. Domain Name Service
  - A set of peer processes may cooperate with one another to achieve a common goal
    - E.g. voice conferencing
- Two significant factors affecting interacting processes in a distributed system are:
  - Communication performance is often a limiting characteristic.
  - It is impossible to maintain a single global notion of time.
- Performance of communication channels
  - The communication channels in our model are realized in a variety of ways in distributed systems, for example
    - By an implementation of streams
    - By simple message passing over a computer network
  - Communication over a computer network has the performance characteristics such as:
    - Latency
      - The delay between the start of a message's transmission from one process to the beginning of its receipt by another.

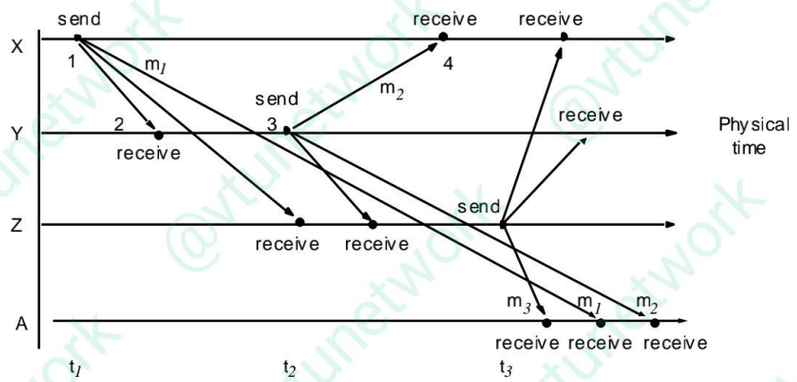
- Bandwidth
  - The total amount of information that can be transmitted over a computer network in a given time.
  - Communication channels using the same network, have to share the available bandwidth.
- Jitter
  - The variation in the time taken to deliver a series of messages.
  - It is relevant to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

- **Two variants of the interaction model**
  - In a distributed system it is hard to set time limits on the time taken for process execution, message delivery or clock drift.
  - Two models of time assumption in distributed systems are:
    - Synchronous distributed systems
      - It has a strong assumption of time
      - The time to execute each step of a process has known lower and upper bounds.
      - Each message transmitted over a channel is received within a known bounded time.
      - Each process has a local clock whose drift rate from real time has a known bound.
    - Asynchronous distributed system
      - It has no assumption about time.
      - There is no bound on process execution speeds.
        - Each step may take an arbitrary long time.
      - There is no bound on message transmission delays.
        - A message may be received after an arbitrary long time.
      - There is no bound on clock drift rates.
        - The drift rate of a clock is arbitrary.

- **Event ordering**
  - In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after, or concurrently with another event at another process.
  - The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.
  - For example, consider a mailing list with users X, Y, Z, and A.
  - User X sends a message with the subject Meeting.

1. Users Y and Z reply by sending a message with the subject RE: Meeting.

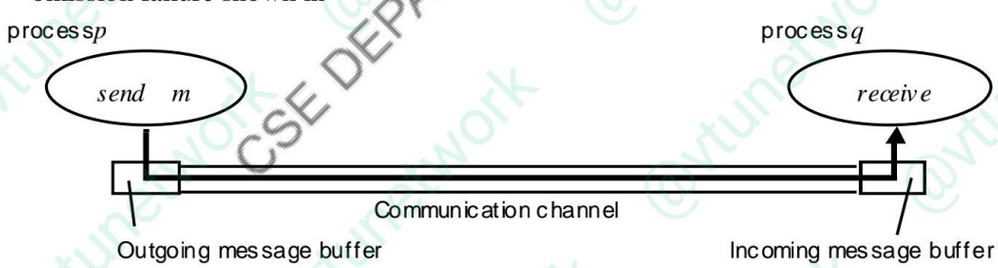
- In real time, X's message was sent first, Y reads it and replies; Z reads both X's message and Y's reply and then sends another reply, which references both X's and Y's messages.
- But due to the independent delays in message delivery, the messages may be delivered in the order is shown in figure 10.
- It shows user A might see the two messages in the wrong order.



Failure model

- Failure model defines and classifies the faults.

- In a distributed system both processes and communication channels may fail – That is, they may depart from what is considered to be correct or desirable behavior.
- Types of failures:
  - Omission Failures
  - Arbitrary Failures
  - Timing Failures
- **Omission failure**
  - Omission failures refer to cases when a process or communication channel fails to perform actions that it is supposed to do.
  - The chief omission failure of a process is to crash. In case of the crash, the process has halted and will not execute any further steps of its program.
  - Another type of omission failure is related to the communication which is called communication omission failure shown in



  - ➢ The communication channel produces an omission failure if it does not transport a message from "p"s outgoing message buffer to "q"'s incoming message buffer.
  - ➢ This is known as "dropping messages" and is generally caused by lack of buffer space at the receiver or at an gateway or by a network transmission error, detected by a checksum carried with the message data.

- **Arbitrary failure**
  - ➢ Arbitrary failure is used to describe the worst possible failure semantics, in which any type of error may occur.
    - ❖ E.g. a process may set a wrong values in its data items, or it may return a wrong value in response to an invocation.
  - ➢ Communication channel can suffer from arbitrary failures.

- ❖ E.g. message contents may be corrupted or non-existent messages may be delivered or real messages may be delivered more than once.
- ❖ The omission failures are classified together with arbitrary failures shown in

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

- **Timing failure**
  - ➢ Timing failures are applicable in synchronized distributed systems where time limits are set on process execution time, message delivery time and clock drift rate.

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.

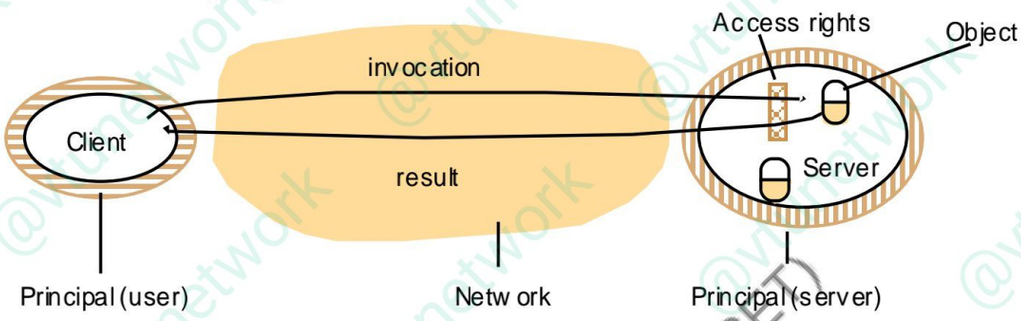
- **Masking failure**
  - ➢ It is possible to construct reliable services from components that exhibit failure.
    - ❖ E.g. multiple servers that hold replicas of data can continue to provide a service when one of them crashes.
  - ➢ A service masks a failure, either by hiding it altogether or by converting it into a more acceptable type of failure.
    - ❖ E.g. checksums are used to mask corrupted messages- effectively converting an arbitrary failure into an omission failure.

Security model

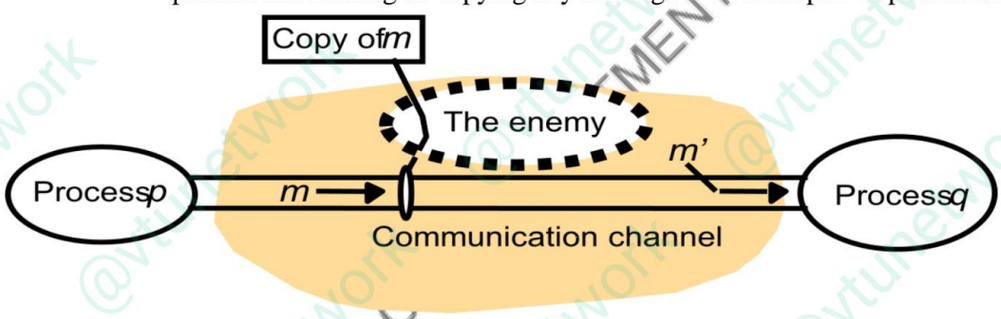
- Security model defines and classifies the forms of attacks.
- It provides a basis for analysis of threats to a system
- It is used to design of systems that are able to resist threats.

- The security of a distributed system can be achieved by securing the processes and the channels used in their interactions.
- Also, by protecting the objects that they encapsulate against unauthorized access.
- Protecting Objects
  - Access rights
    - Access rights specify who is allowed to perform the operations on a object.
      - Who is allowed to read or write its state.

- Principal
  - Principal is the authority associated with each invocation and each result.
  - A principal may be a user or a process.
  - The invocation comes from a user and the result from a server.
- The sever is responsible for
  - Verifying the identity of the principal (user) behind each invocation.
  - Checking that they have sufficient access rights to perform the requested operation on the particular object invoked.
  - Rejecting those that do not.



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



  - Threats from a potential enemy are classified as:
    - Threats to processes
    - Threats to communication channels
    - Denial of service
  - Defeating security threats
  - Secure systems are based on the following main techniques:
    - Cryptography and shared secrets
      - Cryptography is the science of keeping message secure.
      - Encryption is the process of scrambling a message in such a way as to hide its contents.
    - Authentication
      - The use of shared secrets and encryption provides the basis for the authentication of messages.
    - Secure channels
      - Encryption and authentication are use to build secure channels as a service layer on top of the existing communication services.

- A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal.
- VPN (Virtual Private Network) and secure socket layer (SSL) protocols are instances of secure channel.

- A secure channel has the following properties:
  - Each of the processes knows the identity of the principal on whose behalf the other process is executing.
  - In a secure channel, the server knows the identity of the principal behind the invocations and can check their access rights before performing an operation.
  - A secure channel ensures the privacy and integrity of the data transmitted across it.
  - Each message includes a physical or logical time stamp to prevent messages from being replayed or reordered.

- Other possible threats from an enemy
  - Denial of service
    - This is a form of attack in which the enemy interferes with the activities of authorized users by making excessive and pointless invocations on services of message transmissions in a network.
    - It results in overloading of physical resources (network bandwidth, server processing capacity).
  - Mobile code
    - Mobile code is security problem for any process that receives and executes program code from elsewhere, such as the email attachment.
    - Such attachment may include a code that accesses or modifies resources that are available to the host process but not to the originator of the code

UNIT II

Time and Global States: Introduction, Clocks, Events and Process states, Synchronizing Physical clocks, Logical time and Logical clocks, Global states.

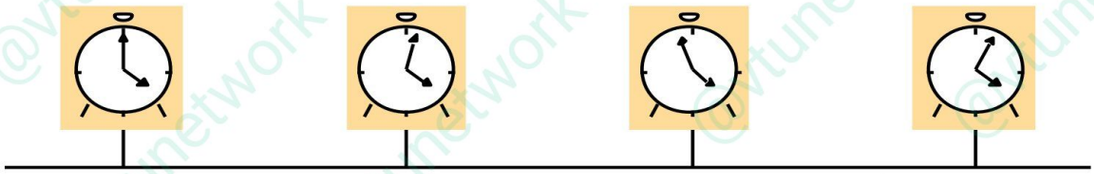
Coordination and Agreement: Introduction, Distributed mutual exclusion, Elections, Multicast Communication, Consensus and Related problems.

Time and Global States: Introduction

- Time is an Important and interesting issue in distributes systems.
- One we can measure accurately.
- Can use as a metric.
- Example: e-commerce transaction timestamp for auditing and accountability purposes.
- Need time for maintaining consistency of databases.
- According to Einstein's theory of relativity a stationary observer on earth and an observer moving away from earth, if they observed an event at the same time, the event may "happen" at different times for them
- Relative order of two events can be reversed unless one caused the other.
- Thus the notion of physical time is problematic in distributed systems.
- We will examine synchronization of clocks using message passing;
- We will study logical clocks: vector clocks.
- We will also look at algorithms to capture global states of distributed systems as they execute.

Clocks, event and process states

- History (pi) = hi = $<e_i^0, e_i^1, e_i^2 \ldots>$
- $\rightarrow_i$ represents relation
- A processor clock is derived from a hardware clock:
- $C_i(t) = \alpha H_i(t) + \beta$
- May result in clock skew and drift
- Coordinated universal time: most accurate clock use atomic oscillators with very low drift rates of 1 in 10^{13}
- In use since 1967: standard second is defined as 9,192,631,770 periods of transitions between the two hyperfine levels of the ground state of Caesium-133 (Cs^{133})
- Days, hours, years are rooted in astronomical time.
- Japanese earthquake should have caused Earth to rotate a bit faster, shortening the length of the day by about 1.8 microseconds (a microsecond is one millionth of a second).
- Currents within earth's core also affect decreases and increases.
- Abbreviated UTC (is equivalent to Greenwich Mean Time (GMT)).
- This is based on atomic time (leap second is inserted, rarely leap second is deleted to keep up astronomical time).
- UTC signal are regularly synchronized and broadcast.
- In USA the radio station WWV broadcasts time signals on several shortwave frequencies.
- Satellite sources for time include Global Positioning Systems (GPS).
- NIST is another source.



➔ Each node maintain a physical clock. However, they tend to drift even after an accurate initial setting.

- **Skew:** the difference between the readings of any two clocks.
- **Clock drift**: the crystal-based clock count time at different rates. Oscillator has different frequency. Drift rate is usually used to measure the change in the offset per unit of time. Ordinary quartz crystal clock, 1second per 11.6 days.

Synchronizing Physical clocks

➔ External synchronization: Ci is synchronized to a common standard.

- $|S(t) - Ci(t)| < D$, for i = 1,2,…N and for all real time t, namely clock Ci are accurate to within the bound D. S is standard time.

➔ Internal synchronization: Ci is synchronized with one another to a known degree of accuracy.

- $|Ci(t) - Cj(t)| < D$ for i,j=1,2,…N, and for all real time t, namely, clocks Ci agree with each other within the bound D.

Synchronization in a synchronous system

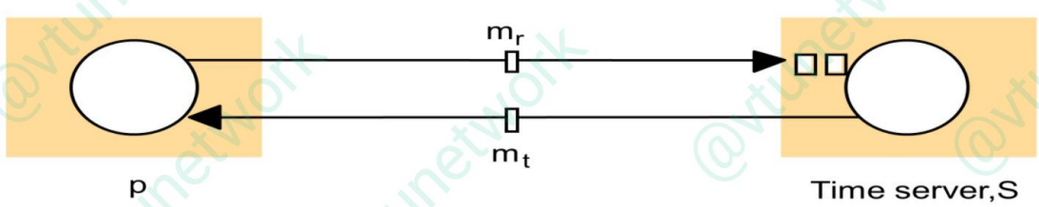
➔ In a synchronous system, bounds exist for clock drift rate, transmission delay and time for computing of each step.

➔ One process sends the time t on it local clock to the other in a message m. The receiver should set its clock to t+Ttrans. It doesn't matter whether t is accurate or not

- Synchronous system: Ttrans could range from min to max. The uncertainty is u=(max-min). If receiver set clock to be t+min or t+max, the skew is as much as u. If receiver set the clock to be t+(min+max)/2, the skew is at most u/2.
- Asynchronous system: no upper bound max. only lower bound.

Cristian's method

- Cristian's method: Time server, connected to a device receiving signals from UTC. Upon request, the server S supplies the time t according to its clock.
- The algorithm is probabilistic and can achieve synchronization only if the observed round trip time are short compared with required accuracy.
- From p's point of view, the earliest time S could place the time in mt was min after p dispatch mr. The latest was min before mt arrived at p.



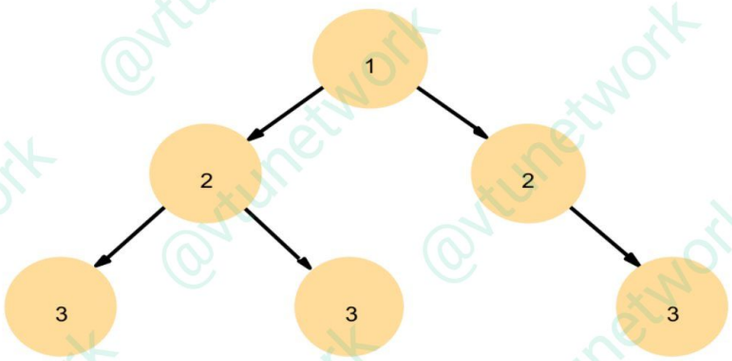
- The time of S by the time p receives the message mt is in the range of [t+min, t+Tround –min].
- P can measure the roundtrip time then p should set its time as (t + Tround/2) as a good estimation.
- The width of this range is (Tround -2min). So the accuracy is +-(Tround /2-min)
- Suffers from the problem associated with single server that single time server may fail.
- Cristian suggested to use a group of synchronized time servers. A client multicast is request to all servers and use only the first reply.
- A faulty time server that replies with spurious time values or an imposter time server with incorrect times.

Berkeley Algorithm

- Internal synchronization when developed for collections of computers running Berkeley UNIX.
- A coordinator is chosen to act as the master. It periodically polls the other computers whose clocks are to be synchronized, called slave. The salves send back their clock values to it. The master estimate their local clock times by observing the round-trip time similar to Cristian's method. It averages the values obtained including its own.
- Instead of sending the updated current time back to other computers, which further introduce uncertainty of message transmission, the master sends the amount by which each individual slave's clock should adjust.
- The master takes a fault-tolerant average, namely a subset of clocks is chosen that do not differ from one another by more than a specified bound.
- The algorithm eliminates readings from faulty clocks. Such clocks could have a adverse effect if an ordinary average was taken.

The Network Time Protocol

- Cristian's method and Berkeley algorithm are primarily for Intranets. The Network Time Protocol(NTP) defines a time service to distribute time information over the Internet.
  - Clients across the Internet to be synchronized accurately to UTC. Statistical techniques
  - Reliable service that can survive lengthy losses of connectivity. Redundant servers and redundant paths between servers.
  - Clients resynchronized sufficiently frequently to offset the rates of drift.
  - Protection against interference with time services. Authentication technique from claimed trusted sources.



Note: Arrows denote synchronization control, numbers denote strata.

Hierarchical structure called synchronization subnet

- Primary server: connected directly to a time source.
- Secondary servers are synchronized with primary server.
- Third servers are synchronized with secondary servers.

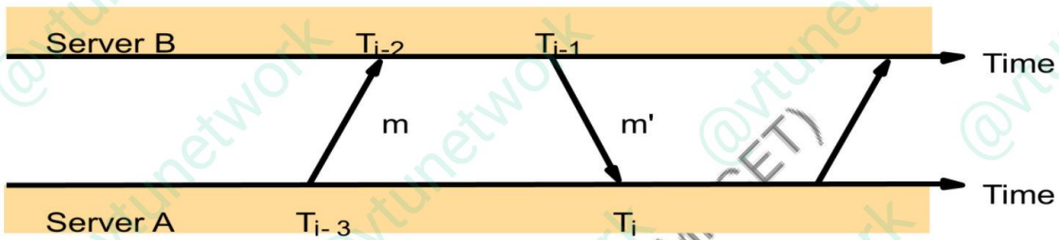
Such subnet can reconfigure as servers become unreachable or failures occur.

NTP servers synchronize in one of three modes:

1. Multicast mode: for high-speed LAN. One or more servers periodically multicasts the time to servers connected by LAN, which set their times assuming small delay. Achieve low accuracy.

2. Procedure call: similar to Cristian's algorithm. One server receives request, replying with its timestamp. Higher accuracy than multicast or multicast is not supported.

3. Symmetric mode: used by servers that supply time in LAN and by higher level of synchronization subnet. Highest accuracy. A pair of servers operating in symmetric mode exchange messages bearing timing information.



Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted. The recipient of the NTP message notes the local time when it receives the message.

Logical time and Logical clocks

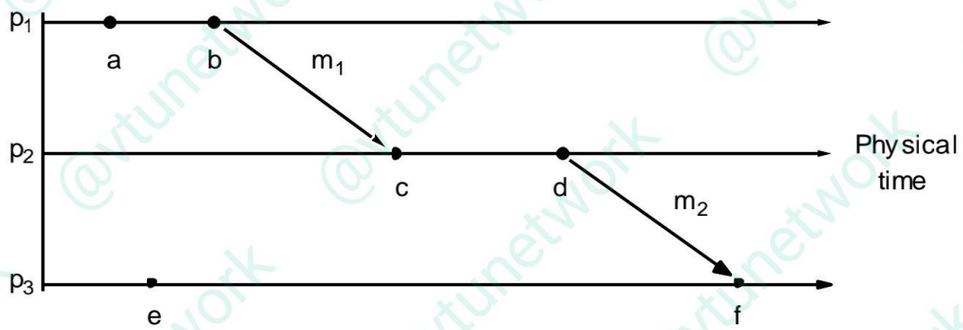
- In single process, events are ordered by local physical time. Since we cannot synchronize physical clocks perfectly across a distributed system, we cannot use physical time to find out the order of any arbitrary pair of events.
- We will use logical time to order events happened at different nodes. Two simple points:
  - If two events occurred at the same process, then they occurred in the order in which pi observes them
  - Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.
- Lamport (1978) called the partial ordering by generalizing these two relationships the happened-before relation.

$$HB1: If \exists\ process\ p_i : e \rightarrow_i e', then\ e \rightarrow e'$$

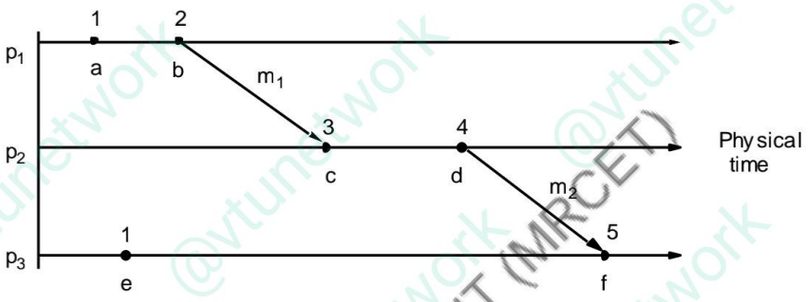
$$HB2: For\ any\ message\ m, send(m) \rightarrow receive(m)$$

$$HB3: If\ e, e', and\ e''\ are\ events, such\ that\ e \rightarrow e'\ and\ e' \rightarrow e'', then\ e \rightarrow e''$$

Events occurred at three processes



Lamport timestamps for the events



$a \rightarrow_1 b$ and $c \rightarrow_2 d$ $\quad a \not\rightarrow e$ and $e \not\rightarrow a$

$b \rightarrow c$ and $d \rightarrow f$ $\quad$ concurrent $a \| e$

combing them, $a \rightarrow f$

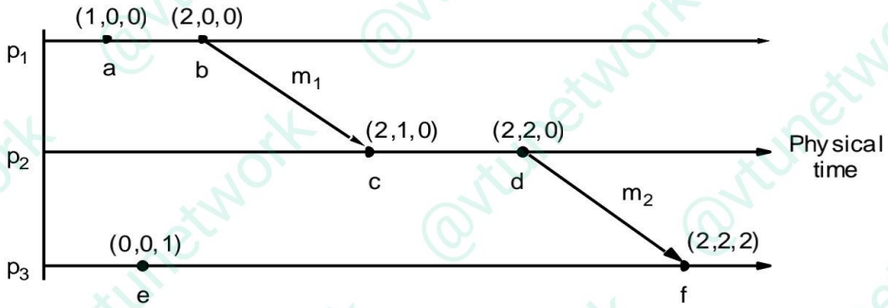
Logical Clocks

- Lamport invented a logical clock Li, which is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process pi keeps its own logical clock.
- LC1: Li is incremented before each event is issued at process pi: Li = Li +1
- LC2: a) Pi sends a message m, it piggybacks on m the value t = Li
  b) On receiving (m,t), a process pj computes Lj=max(Lj,t) and then applies LC1 before timestamping the event receive(m).
- It can be easily shown that:
- If e->e' then L(e) < L(e').
- However, the converse is not true. If L(e) < L(e'), then we cannot infer that e->e'. E.g b and e
- L(b)>L(e) but b||e
- How to solve this problem?

Vector Clock

- Lamport's clock: L(e)<L(e') we cannot conclude that e->e'.
- Vector clock to overcome the above problem.
- N processes is an array of N integers. Each process keeps its own vector clock Vi, which it uses to timestamp local events.

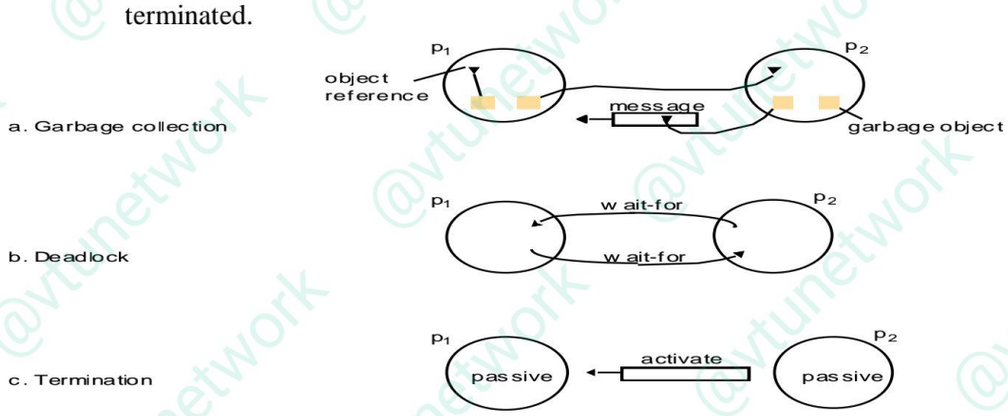
- VC1: initially, Vi[j] = 0, for i,j = 1,2…N
- VC2: just before pi timestamps an event, it sets Vi[i] = vi[i]+1
- VC3: pi includes the value t= Vi in every message it sends
- VC4: when pi receives a timestamp t in a message, it sets Vi[j]=max(Vi[j], t[j])for j =1,2…,N. Merge operation.



- To compare vector timestamps, we need to compare each bit. Concurrent events cannot find a relationship.
- Drawback compared with Lamport time, taking up an amount of storage and message payload proportional to N.

Global states

- We want to find out whether a particular property is true of a distributed system as it executes.
- We will see three examples:
  - Distributed garbage collection: if there are no longer any reference to objects anywhere in the distributed system, the memory taken up by the objects should be reclaimed.
  - Distributed deadlock detection: when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this "wait-for" relationship.
  - Distributed termination detection: detect if a distributed algorithm has terminated. It seems that we only need to test whether each process has halted. However, it is not true. E.g. two processes and each of which may request values from the other. It can be either in passive or active state. Passive means it is not engaged in any activity but is prepared to respond. Two processes may both in passive states. At the same time, there is a message in on the way from P2 to P1, after P1 receives it, it will become active again. So the algorithm has not terminated.



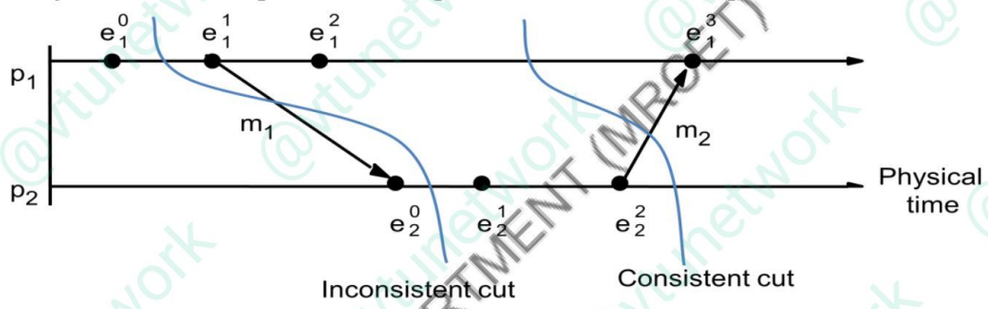
Global States and consistent cuts

- It is possible to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes is much harder.
- The essential problem is the absence of global time. If we had perfectly synchronized clocks at which processes would record its state, we can assemble the global state of the system from local states of all processes at the same time.
- The question is: can we assemble the global state of the system from local states recorded at different real times?
- The answer is "YES".

$$history(p_i) = h_i = < e_i^0, e_i^1, e_i^2, ... >$$

$$finite\ prefix\ of\ history: h_i^k = < e_i^0, e_i^1, e_i^2, ... e_i^k >$$

- A series of events occurs at each process. Each event is either an internal action of the process (variables updates) or it is the sending or receipt of a message over the channel.
- S^k_i is the state of process Pi before kth event occurs, so S^0_i is the initial state of Pi.
- Thus the global state corresponds to initial prefixes of the individual process histories.



- A cut of the system's execution is a subset of its global history that is a union of prefixes of process histories

$$C = h_1^{c_1} \cup h_2^{c_2} \cup ... \cup h_N^{c_N}$$

- The state of each process is in the state after the last event occurs in its own cut. The set of last events from all processes are called frontier of the cut.
- Inconsistent cut: since P2 contains receiving of m1, but at P1 it does not include sending of that message. This cut shows the an effect without a cause. We will never reach a global state that corresponds to process state at the frontier by actual execution under this cut.
- **Consistent cut**: it includes both the sending and receipt of m1. It includes the sending but not the receipt of m2. It is still consistent with actual execution.
- A cut C is **consistent** if, for each event it contains, it also contains all the events that happened-before that event.

$$for\ all\ events\ e \in C,\ f \rightarrow e \Rightarrow f \in C$$

- A consistent global state is one that corresponds to a consistent cut.
- A run is a total ordering of all the events in a global history that is consistent with each local history's ordering.
- A linearization or consistent run is an ordering of the events in a global history that is consistent with this happened-before relation.

Global state predicate

- Global state predicate is a function that maps from the set of global states of processes n the system to true or false.
- Stable characteristics associated with object being garbage, deadlocked or terminated: once the system enters a state in which the predicate is True. It remains True in all future states reachable from that state.
- Safety (evaluates to deadlocked false for all states reachable from S0)
- Liveness (evaluate to reaching termination true for some of the states reachable from S0)

Chandy and Lamport's 'snapshot' algorithm

- Chandy and Lamport(1985) describe a "snapshot" algorithm for determining global states of distributed system.
- Record a set of process and channel states for a set of processes Pi such that even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.
- The algorithm records state locally at processes without giving a method for gathering the global state.
- Assumptions:
  1. Neither channels nor processes fail; communication is reliable so that every message sent is eventually received intact, exactly once;
  2. Channel are unidirectional either incoming or outgoing and provide FIFO order message delivery;
  3. The graph of processes and channels is strongly connected (there is a path between any two processes).
  4. Any process may initiate a global snapshot at any time.
  5. The processes may continue their normal execution and send and receive normal massages while the snapshot takes place.
- Each process records its own state and also for each incoming channel a set of messages sent to it.
- Allow us to record process states at different times but to account for the differential between process states in terms of message transmitted but not yet received.
- If process pi has sent a message m to process pj, but pj has not received it, then we account for m as belong to the state of the channel between them.
- Use of special marker message. It has a dual role, as a prompt for the receiver to save its own state if it has not done so; and as a means of determining which messages to include in the channel state.
- **********

On p_i's receipt of a *marker* message over channel c:
if (p_i has not yet recorded its state) it
records its process state now;
records the state of c as the empty set;
turns on recording of messages arriving over other incoming channels;
else
p_i records the state of c as the set of messages it has received over c since it saved its state.
end if

Marker sending rule for process p_i
After p_i has recorded its state, for each outgoing channel c:
p_i sends one marker message over c
(before it sends any other message over c).

Coordination and Agreement: Introduction

- A set of processes coordinate their actions. How to agree on one or more values
  - Avoid fixed master-salve relationship to avoid single points of failure for fixed master.
- Distributed mutual exclusion for resource sharing
- A collection of process **share resources**, mutual exclusion is needed to prevent interference and ensure consistency. (critical section)
- No shared variables or facilities are provided by single local kernel to solve it. Require a solution that is based solely on message passing.
- Important factor to consider while designing algorithm is the failure

Distributed mutual exclusion

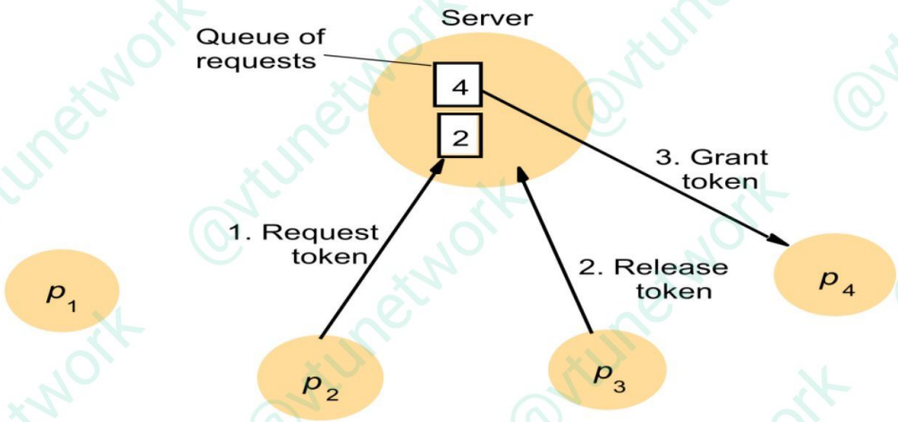
- Application level protocol for executing a critical section
  - enter() // enter critical section-block if necessary
  - resrouceAccess() //access shared resoruces
  - exit() //leave critical section-other processes may enter
- Essential requirements:
  - ME1: (safety) at most one process may execute in the critical section
  - ME2: (liveness) Request to enter and exit the critical section eventually succeed.
  - ME3(ordering) One request to enter the CS happened-before another, then entry to the CS is granted in that order.
- ME2 implies freedom from both deadlock and starvation. Starvation involves fairness condition. The order in which processes enter the critical section. It is not possible to use the request time to order them due to lack of global clock. So usually, we use happen-before ordering to order message requests.

Performance Evaluation

- Bandwidth consumption, which is proportional to the number of messages sent in each entry and exit operations.
- The client delay incurred by a process at each entry and exit operation.
- throughput of the system. Rate at which the collection of processes as a whole can access the critical section. Measure the effect using the synchronization delay between one process exiting the critical section and the next process entering it; the shorter the delay is, the greater the throughput is.

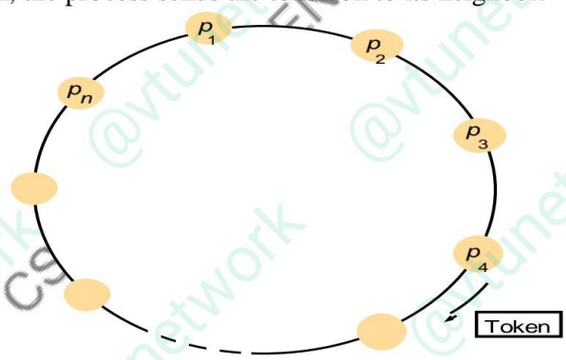
Central Server Algorithm

- The simplest way to grant permission to enter the critical section is to employ a server.
- A process sends a request message to server and awaits a reply from it.
- If a reply constitutes a token signifying the permission to enter the critical section.
- If no other process has the token at the time of the request, then the server replied immediately with the token.
- If token is currently held by other processes, the server does not reply but queues the request.
- Client on exiting the critical section, a message is sent to server, giving it back the token.
  - ME1: safety
  - ME2: liveness Are satisfied but not
  - ME3: ordering



Ring-based Algorithm

- Simplest way to arrange mutual exclusion between N processes without requiring an additional process is arrange them in a logical ring.
- Each process pi has a communication channel to the next process in the ring, p(i+1)/mod N.
- The unique token is in the form of a message passed from process to process in a single direction clockwise.
- If a process does not require to enter the CS when it receives the token, then it immediately forwards the token to its neighbor.
- A process requires the token waits until it receives it, but retains it.
- To exit the critical section, the process sends the token on to its neighbor.



ME1: safety ME2: liveness Are satisfied but not ME3: ordering

Using Multicast and logical clocks

- Mutual exclusion between N peer processes based upon multicast.
- Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.
- The condition under which a process replies to a request are designed to ensure ME1 ME2 and ME3 are met.
- Each process pi keeps a Lamport clock. Message requesting entry are of the form<T, pi>.
- Each process records its state of either RELEASE, WANTED or HELD in a variable state.
    - If a process requests entry and all other processes is RELEASED, then all processes reply immediately.
    - If some process is in state HELD, then that process will not reply until it is finished.
    - If some process is in state WANTED and has a smaller timestamp than the incoming request, it will queue the request until it is finished.

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

Ricart and Agrawala's algorithm

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

On receipt of a request <Ti, pi> at pj (i ≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
        queue request from pi without replying;
    else
        reply immediately to pi;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

Maekawa's voting algorithm

- It is not necessary for all of its peers to grant access. Only need to obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.
- Think of processes as voting for one another to enter the CS. A candidate process must collect sufficient votes to enter.
- Processes in the intersection of two sets of voters ensure the safety property ME1 by casting their votes for only one candidate
- A voting set Vi associated with each process pi.
- there is at least one common member of any two voting sets, the size of all voting set are the same size to be fair.
- The optimal solution to minimizes K is K~sqrt(N) and M=K.

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

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

$$p_i \in V_i$$

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

$$|V_i| = K$$

$$Each\ process\ is\ contained\ in\ M\ of\ the\ voting\ set\ V_i$$

Maekawa's algorithm

On initialization
  state := RELEASED;
  voted := FALSE;
For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = *K*);
  state := HELD;
On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;
On receipt of a release from p_i at p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

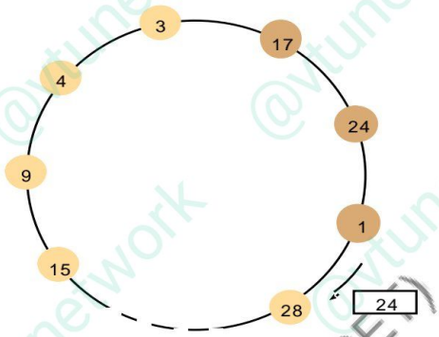
Elections

- Algorithm to choose a unique process to play a particular role is called an election algorithm. E.g. central server for mutual exclusion, one process will be elected as the server. Everybody must agree. If the server wishes to retire, then another election is required to choose a replacement.
- Requirements:
  - E1(safety): a participant pi has $elected_i = \perp \; or \; elected_i = P$
    Where P is chosen as the non-crashed process at the end of run with the largest identifier. (concurrent elections possible.)
  - E2(liveness): All processes Pi participate in election process and eventually set $elected_i \neq \perp \; or \; crash$

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.
  - **Initially**, every process is marked as non-participant. Any process can begin an election.
  - The **starting** process marks itself as participant and place its identifier in a message to its neighbour.
  - A process receives a message and **compare** it with its own. If the arrived identifier is **larger**, it passes on the message.
  - 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.

- On forwarding of any case, the process marks itself as a participant.
- If the received identifier is that of the receiver itself, then this process' s identifier must be the greatest, and it becomes the **coordinator**.
- The coordinator marks itself as non-participant set elected_i and sends an **elected** message to its neighbour enclosing its ID.
- When a process receives elected message, marks itself as a non-participant, sets its variable elected_i and forwards the message.



- E1 is met. All identifiers are compared, since a process must receive its own ID back before sending an elected message.
- E2 is also met due to the guaranteed traversals of the ring.
- Tolerate no failure makes ring algorithm of limited practical use.
- If only a single process starts an election, the worst-performance case is then the anti-clockwise neighbour has the highest identifier. A total of N-1 messages is used to reach this neighbour. Then further N messages are required to announce its election. The elected message is sent N times. Making 3N-1 messages in all.
- Turnaround time is also 3N-1 sequential message transmission time

The bully algorithm

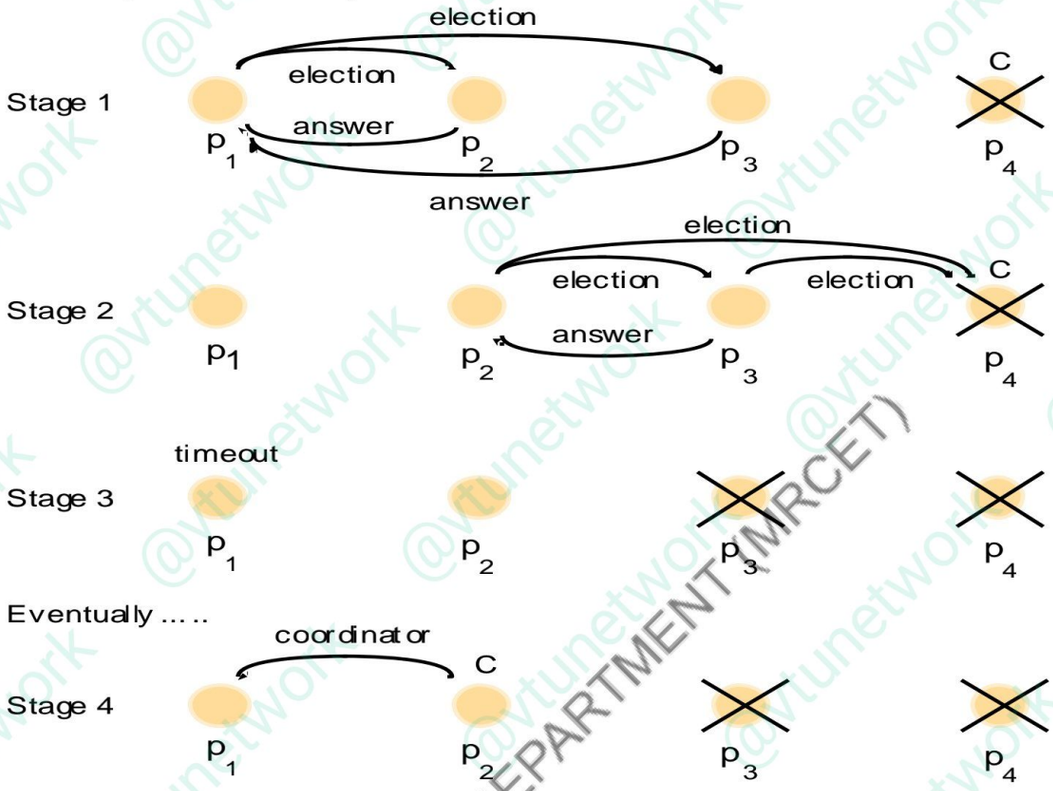
- Allows process to crash during an election, although it assumes the message delivery between processes is reliable.
- Assume system is synchronous to use timeouts to detect a process failure.
- Assume each process knows which processes have higher identifiers and that it can communicate with all such processes.
- Three types of messages:
  - Election is sent to announce an election message. A process begins an election when it notices, through timeouts, that the coordinator has failed. T=2Ttrans+Tprocess From the time of sending
  - Answer is sent in response to an election message.
  - Coordinator is sent to announce the identity of the elected process.

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

3. If a process receives an election message, it send back an answer message and begins another election unless it has begun one already.

E1 may be broken if timeout is not accurate or replacement. (suppose P3 crashes and replaced by another process. P2 set P3 as coordinator and P1 set P2 as coordinator)

E2 is clearly met by the assumption of reliable transmission.



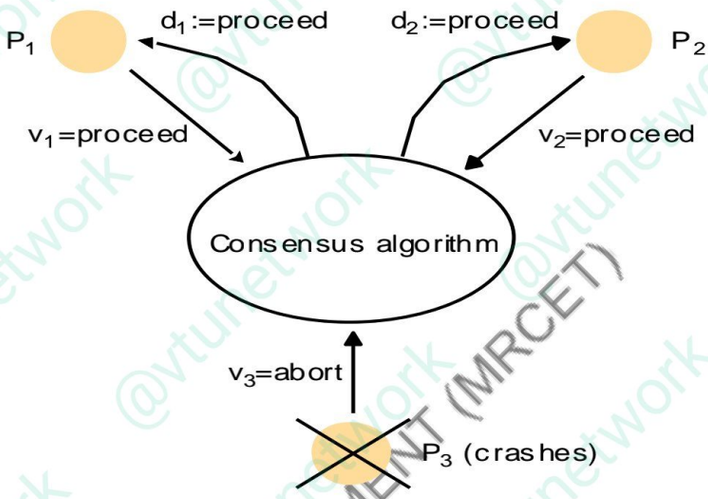
- Best case the process with the second highest ID notices the coordinator's failure. Then it can immediately elect itself and send N-2 coordinator messages.
- The bully algorithm requires O(N^2) messages in the worst case - that is, when the process with the least ID first detects the coordinator's failure. For then N-1 processes altogether begin election, each sending messages to processes with higher ID.

Consensus and Related Problems (agreement)

- The problem is for processes to agree on a value after one or more of the processes has proposed what that value should be. (e.g. all controlling computers should agree upon whether let the spaceship proceed or abort after one computer proposes an action.)
- Assumptions: Communication is reliable but the processes may fail (arbitrary process failure as well as crash). Also specify that up to some number f of the N processes are faculty.
- Every process pi begins in the undecided state and propose a single value vi, drawn from a set D. The processes communicate with one another, exchanging values. Each process then sets the value of a decision variable di. In doing so it enters the decided state, in which it may no longer change di.
- Requirements:
  - Termination: Eventually each correct process sets its decision variable.

- Agreement: The decision value of all correct processes is the same: if pi and pj are correct and have entered the decided state, then di=dj
- Integrity: if the correct processes all proposed the same value, then any correct process in the decided state has chosen that value. This condition can be loosen. For example, not necessarily all of them, may be some of them.

- It will be straightforward to solve this problem if no process can fail by using multicast and majority vote.
- Termination guaranteed by reliability of multicast, agreement and integrity guaranteed by the majority definition and each process has the same set of proposed value to evaluate. .



Byzantine general problem (proposed in1982)

- Three or more generals are to agree to attack or to retreat. One, the commander, issues the order. The others, lieutenants to the commander, are to decide to attack or retreat.
- But one or more of the general may be treacherous-that is, faulty. If the commander is treacherous, he proposes attacking to one general and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.
- A. Byzantine general problem is different from consensus in that a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value.
- Requirements:
  - Termination: eventually each correct process sets its decision variable.
  - Agreement: the decision value of all correct processes is the same.
  - Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.
- If the commander is correct, the integrity implies agreement; but the commander need not be correct.
- B. Interactive consistency problem : Another variant of consensus, in which every process proposes a single value. Goal of this algorithm is for the correct processes to agree on a decision vector of values, one for each process.
- Requirements: Termination: eventually each correct process sets it decision variable. Agreement: the decision vector of all correct processes is the same. Integrity: If pi is correct, then all correct processes decide on vi as the ith component of the vector.

Consensus in a synchronous system

- Basic multicast protocol assuming up to f of the N processes exhibit crash failures.
- Each correct process collects proposed values from the other processes. This algorithm proceeds in f+1 rounds, in each of which the correct processes Basic-multicast the values between themselves. At most f processes may crash, by assumption. At worst, all f crashes during the round, but the algorithm guarantees that at the end of the rounds all the correct processes that have survived have the same final set of values are in a position to agree.

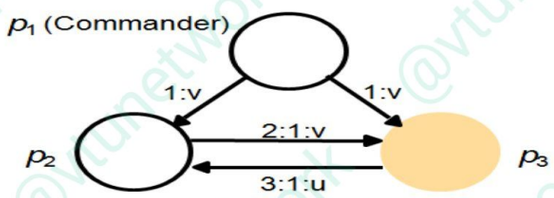
Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

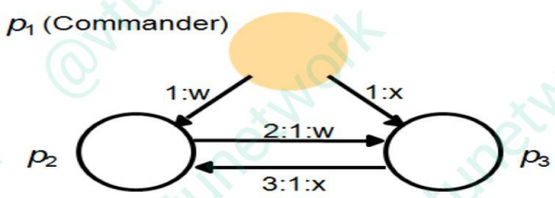
On initialization
$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

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

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

Three byzantine generals

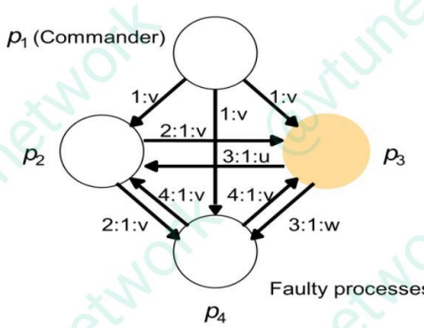


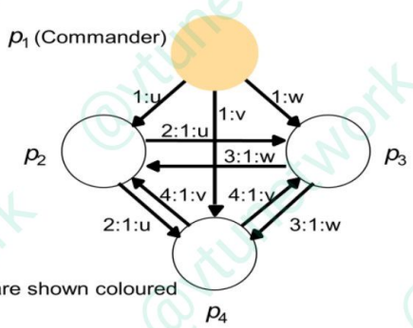


Faulty processes are shown coloured

3:1:u: first number indicates source, the second number indicates Who says. From P3, P1 says u.
If solution exists, P2 bound to decide on v when commander is correct. If no solution can distinguish between correct and faulty commander, p2 must also choose the value sent by commander. By Symmetry, P3 should also choose commander, p2 does the same thing. But it contradicts with agreement. No solution is N<=3f. All because that a correct general can not tell which process is faulty. Digital signature can solve this problem.

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4





Faulty processes are shown coloured

Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
  - The set of messages that every process of the group should receive
  - On the delivery ordering across the group members
- Challenges
  - Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
  - Delivery guarantees ensure that operations are completed
- Types of group
  - Static or dynamic: whether joining or leaving is considered
  - Closed or open
    - A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
    - A group is open if processes outside the group can send to it
- Simple basic multicasting **(B-multicast)** is sending a message to every process that is a member of a defined group
  - B-multicast(g, m) for each process p Î group g, send(p, message m)
  - On receive(m) at p: B-deliver(m) at p
- **Reliable multicasting (R-multicast)** requires these properties
  - Integrity: a correct process sends a message to only a member of the group and does it only once
  - Validity: if a correct processsends a message, it will eventually be delivered.
  - Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it


```
On initialization
    Received := {};

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

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

- Implementing reliable R-multicast over B-multicast
- When a message is delivered, the receiving process multicasts it
- Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering

Three types of message ordering

- ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message before another, every correct process will deliver the first message before the other
- ***Casual ordering*:** any correct process that delivers the second message will deliver the previous message first
- ***Total ordering*:** if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first

•Note that

- FIFO ordering and casual ordering are only partial orders
- Not all messages are sent by the same sending process
- Some multicasts are concurrent, not able to be ordered by happened- before
- Total order demands consistency, but not a particular order
-

•**Totally ordered** messages T_1 and T_2.

•**FIFO-related** messages F_1 and F_2.

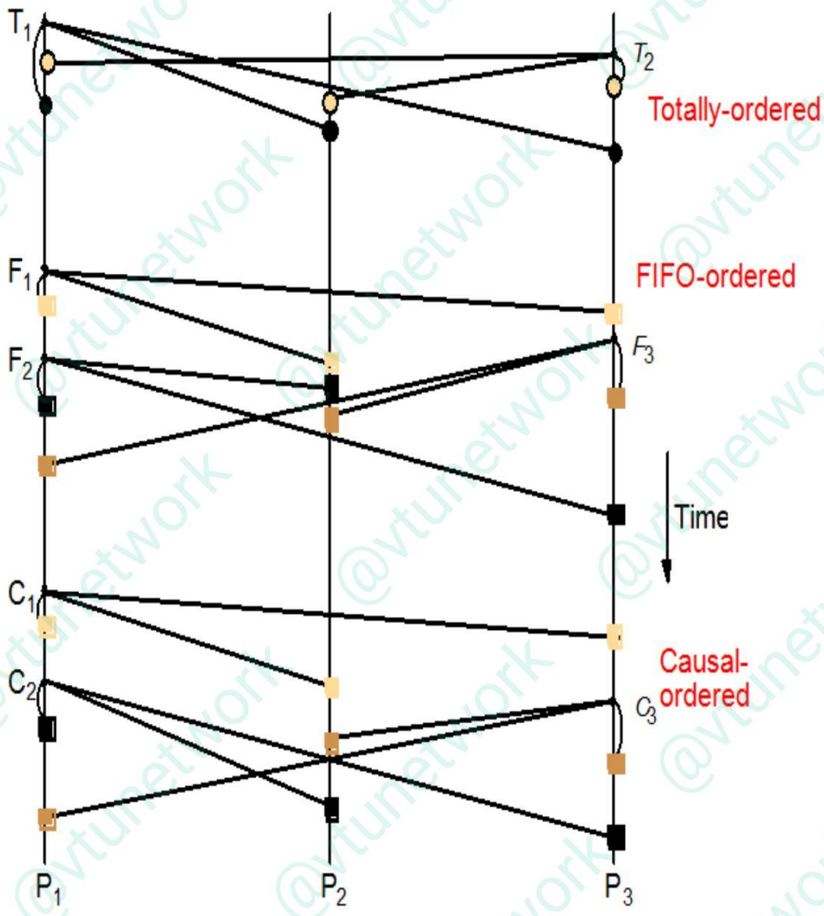
•**Causally-related** messages C_1 and C_3

• Causal ordering implies FIFO ordering

• Total ordering does not imply causal ordering.

• Causal ordering does not imply total ordering.

• Hybrid mode: causal-total ordering, FIFO-total ordering.



Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
  - The set of messages that every process of the group should receive
  - On the delivery ordering across the group members
- Challenges
  - Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
  - Delivery guarantees ensure that operations are completed
- Types of group
  - Static or dynamic: whether joining or leaving is considered
  - Closed or open
    - A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
    - A group is open if processes outside the group can send to it
- Simple basic multicasting **(B-multicast)** is sending a message to every process that is a member of a defined group
  - B-multicast(g, m) for each process p Î group g, send(p, message m)
  - On receive(m) at p: B-deliver(m) at p
- **Reliable multicasting (R-multicast)** requires these properties
  - Integrity: a correct process sends a message to only a member of the group and does it only once
  - Validity: if a correct processsends a message, it will eventually be delivered.
  - Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

On initialization
Received := {};

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

On B-deliver(m) *at process q with g = group*(m)
if ($m \notin$ *Received*)
then
Received := *Received* $\cup$ {m};
if ($q \neq p$) *then B-multicast*(g, m); *end if*
R-deliver m;
end if

- Implementing reliable R-multicast over B-multicast
- When a message is delivered, the receiving process multicasts it
- Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering

Three types of message ordering

- ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message before another, every correct process will deliver the first message before the other
- ***Casual ordering*:** any correct process that delivers the second message will deliver the previous message first
- ***Total ordering*:** if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first

•Note that

- FIFO ordering and casual ordering are only partial orders
- Not all messages are sent by the same sending process
- Some multicasts are concurrent, not able to be ordered by happened- before
- Total order demands consistency, but not a particular order
-

•**Totally ordered** messages T_1 and T_2.

•**FIFO-related** messages F_1 and F_2.

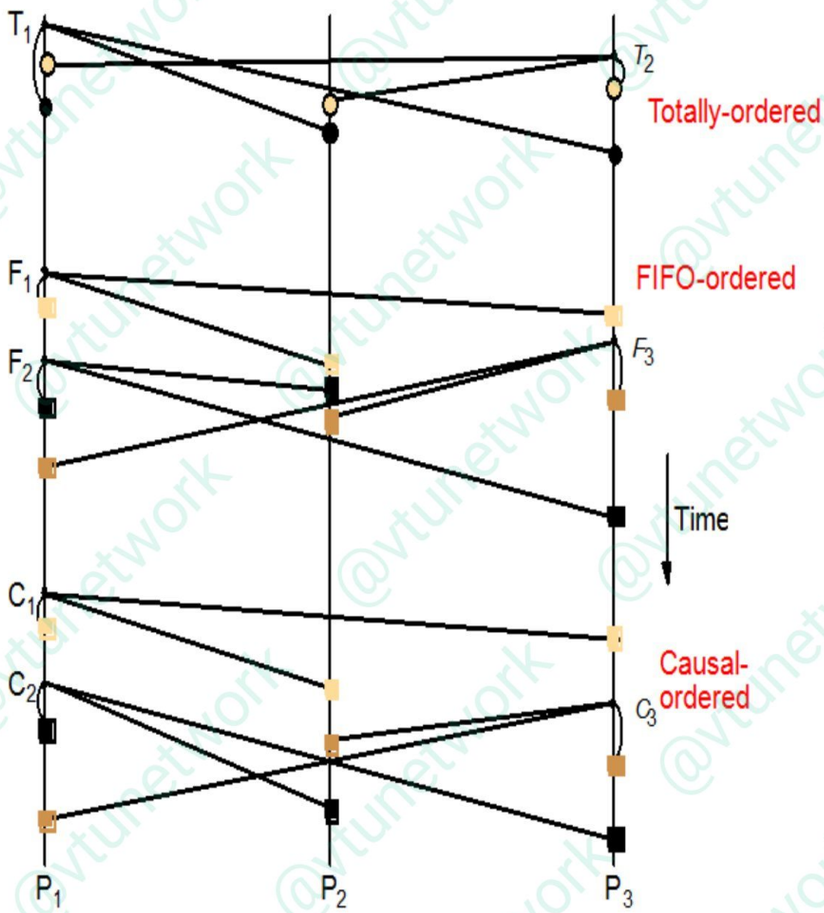
•**Causally-related** messages C_1 and C_3

• Causal ordering implies FIFO ordering

• Total ordering does not imply causal ordering.

• Causal ordering does not imply total ordering.

• Hybrid mode: causal-total ordering, FIFO-total ordering.



UNIT III

Interprocess Communication: Introduction, Characteristics of Interprocess communication, External Data Representation and Marshalling, Client-Server Communication, Group Communication, Case Study: IPC in UNIX.

Distributed Objects and Remote Invocation: Introduction, Communication between Distributed Objects, Remote Procedure Call, Events and Notifications, Case study: Java RMI.

INTRODUCTION:

The java API for inter process communication in the internet provides both datagram and stream communication.

The two communication patterns that are most commonly used in distributed programs.

Client-Server communication

The request and reply messages provide the basis for remote method invocation (RMI) or remote procedure call (RPC).

Group communication

The same message is sent to several processes.

This chapter is concerned with middleware.

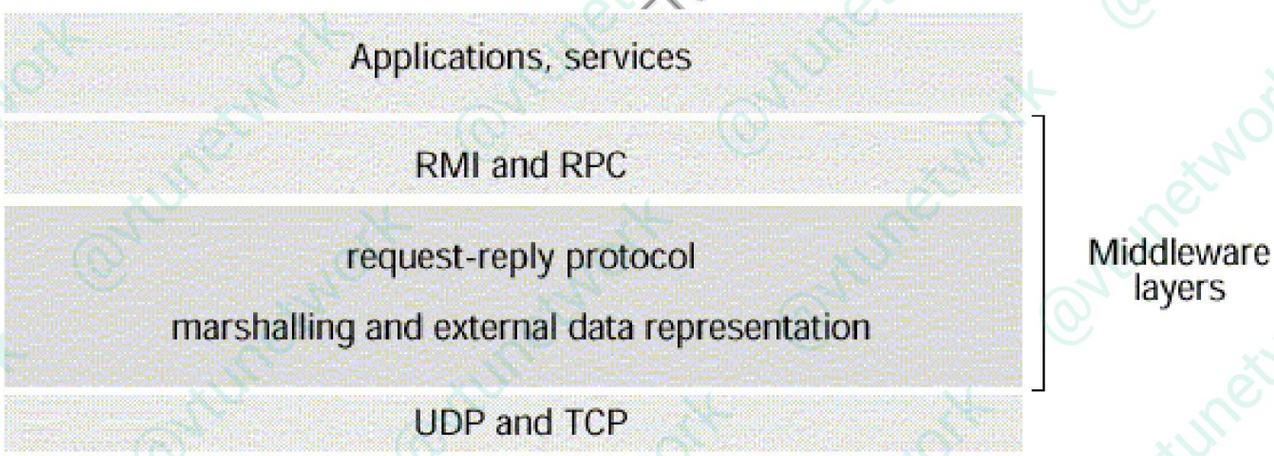


Figure 1. Middleware layers

Remote Method Invocation (RMI)

It allows an object to invoke a method in an object in a remote process.

E.g. CORBA and Java RMI

Remote Procedure Call (RPC)

It allows a client to call a procedure in a remote server.

The CHARACTERISTICS of INTERPROCESS COMMUNICATION

Synchronous and asynchronous communication

- In the synchronous form, both send and receive are blocking operations.
- In the asynchronous form, the use of the send operation is non-blocking and the receive operation can have blocking and non-blocking variants.

Message destinations

- A local port is a message destination within a computer, specified as an integer.
- A port has an exactly one receiver but can have many senders.

Reliability

- A reliable communication is defined in terms of validity and integrity.
- A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost.
- For integrity, messages must arrive uncorrupted and without duplication.

Ordering

- Some applications require that messages be delivered in sender order.

External Data Representation

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

Data representation problems are:

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

Marshalling

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

Unmarshalling

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

Three approaches to external data representation and marshalling are:

- CORBA
- Java's object serialization
- XML

1.CORBA Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0.

It consists 15 primitive types:

- Short (16 bit)
- Long (32 bit)
- Unsigned short
- Unsigned long
- Float(32 bit)
- Double(64 bit)
- Char
- Boolean(TRUE,FALSE)
- Octet(8 bit)
- Any(can represent any basic or constructed type)

- Composite type are shown in Figure 8.

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

Figure 8. CORBA CDR for constructed types

Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure 8.

Figure 9 shows a message in CORBA CDR that contains the three fields of a struct whose respective types are string, string, and unsigned long.

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

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

Figure 9. CORBA CDR message

2. Java object serialization

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

An object is an instance of a Java class.

Example the Java class equivalent to the Person struct

```
Public class Person implements Serializable {
        Private String name;
        Private String place;
        Private int year;
        Public Person(String aName ,String  aPlace, int aYear) {
            name = aName;
            place = aPlace;
            year = aYear;
}
//followed by methods for accessing the instance variables
}
```

The serialized form is illustrated in Figure 10.

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

Figure 10. Indication of Java serialization form

3. Remote Object References

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

generic format for remote object references is shown in Figure 11.

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

Figure 11. Representation of a remote object references

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

Client-Server Communication

- The client-server communication is designed to support the roles and message exchanges in typical client-server interactions.
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.
- Often built over UDP datagrams
- Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary
- Avoidance of connection establishment overhead
- No need for flow control due to small amounts of data are transferred
- The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in Figure 12.

  The request-reply protocol is shown in Figure 12.

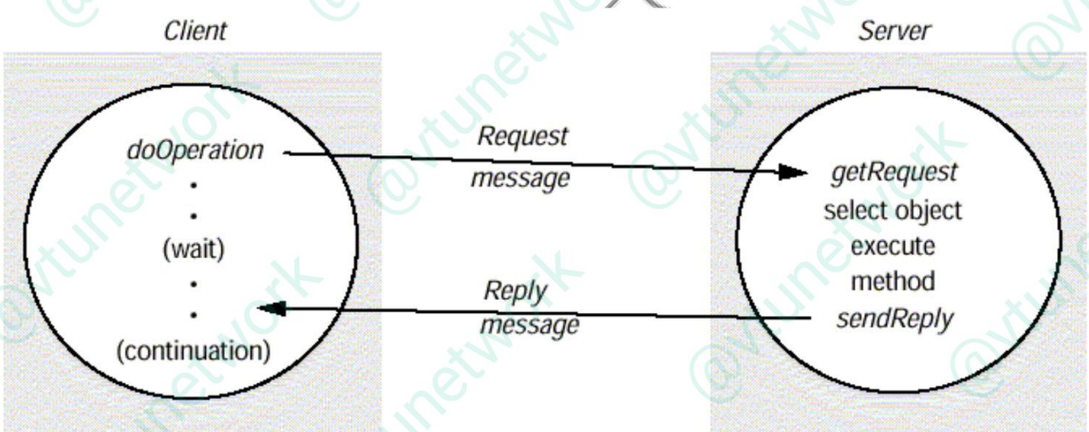


Figure 12. Request-reply communication

- The designed request-reply protocol matches requests to replies.
- If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.
- Figure 13 outlines the three communication primitives.

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

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

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

The information to be transmitted in a request message or a reply message is shown in Figure 14.

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

Figure 14. Request-reply message structure

In a protocol message

- The first field indicates whether the message is a request or a reply message.
- The second field request id contains a message identifier.
- The third field is a remote object reference .
- The forth field is an identifier for the method to be invoked.

Message identifier

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

Failure model of the request-reply protocol

- If the three primitive doOperation, getRequest, and sendReply are implemented over UDP datagram, they have the same communication failures.
  - Omission failure

- Messages are not guaranteed to be delivered in sender order.

RPC exchange protocols

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

The request-reply-acknowledge (RRA) protocol.

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

Figure15.RPC exchange protocols

- In the R protocol, a single request message is sent by the client to the server.
- The R protocol may be used when there is no value to be returned from the remote method.
- The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol.
- RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply.
- HTTP: an example of a request-reply protocol
  - HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.
  - HTTP protocol steps are:
    - Connection establishment between client and server at the default server port or at a port specified in the URL
    - client sends a request
    - server sends a reply
    - connection closure
- HTTP methods
  - GET

- Requests the resource, identified by URL as argument.
- If the URL refers to data, then the web server replies by returning the data
- If the URL refers to a program, then the web server runs the program and returns the output to the client.

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

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

❖ An optional message body

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

Figure 17. HTTP reply message

Group Communication

- The pairwise exchange of messages is not the best model for communication from one process to a group of other processes.
- A multicast operation is more appropriate.
- Multicast operation is an operation that sends a single message from one process to each of the members of a group of processes.
- The simplest way of multicasting, provides no guarantees about message delivery or ordering.
- Multicasting has the following characteristics:
  - Fault tolerance based on replicated services
    - A replicated service consists of a group of servers.
    - Client requests are multicast to all the members of the group, each of which performs an identical operation.
  - Finding the discovery servers in spontaneous networking
- Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
  - Better performance through replicated data
    - Data are replicated to increase the performance of a service.
  - Propagation of event notifications
    - Multicast to a group may be used to notify processes when something happens.

IP multicast

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

The following details are specific to IPv4:

Multicast IP routers

IP packets can be multicast both on local network and on the wider Internet.

Local multicast uses local network such as Ethernet.

To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass- called the time to live, or TTL for short.

Multicast address allocation

Multicast addressing may be permanent or temporary.

Permanent groups exist even when there are no members.

Multicast addressing by temporary groups must be created before use and cease to exit when all members have left.

The session directory (sd) program can be used to start or join a multicast session.

session directory provides a tool with an interactive interface that allows users to browse advertised multicast sessions and to advertise their own session, specifying the time and duration.

Java API to IP multicast

The Java API provides a datagram interface to IP multicast through the class MulticastSocket, which is a subset of DatagramSocket with the additional capability of being able to join multicast groups.

The class MulticastSocket provides two alternative constructors , allowing socket to be creative to use either a specified local port, or any free local port.

A process can join a multicast group with a given multicast address by invoking the joinGroup method of its multicast socket.

Case Study: IPC in UNIX.

A process can be of two type:

Independent process.

Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

Shared Memory

The Figure 1 below shows a basic structure of communication between processes via shared memory method and via message passing.

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.
Let's discuss an example of communication between processes using shared memory method.

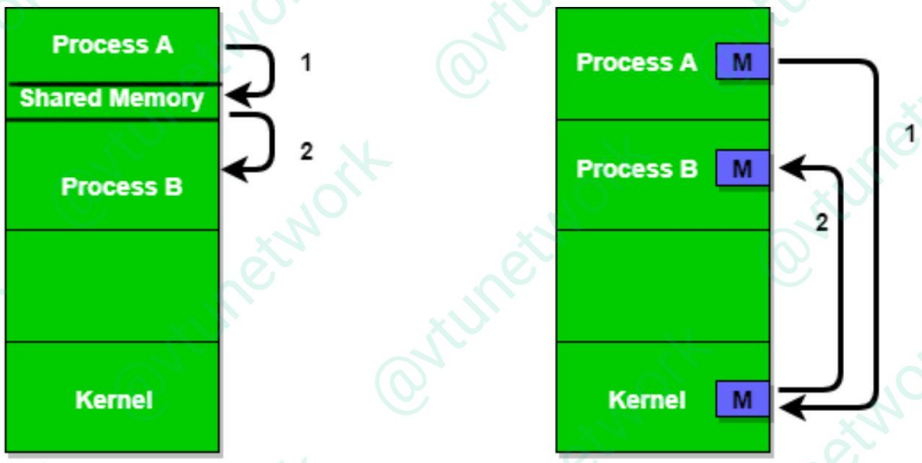


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it.

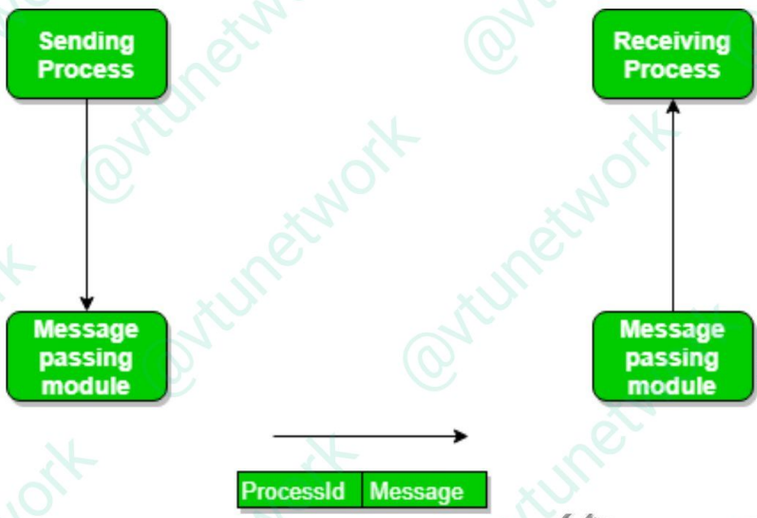
ii) Messaging Passing Method

Now, We will start our discussion for the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

Establish a communication link (if a link already exists, no need to establish it again.)

Start exchanging messages using basic primitives.

We need at least two primitives:
– send(message, destinaion) or send(message)
– receive(message, host) or receive(message)



The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: header and body.
The header part is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Distributed Objects and Remote Invocation

Topics covered in this chapter:

- Communication between distributed objects
- Remote procedure call
- Events and notification
- Java RMI

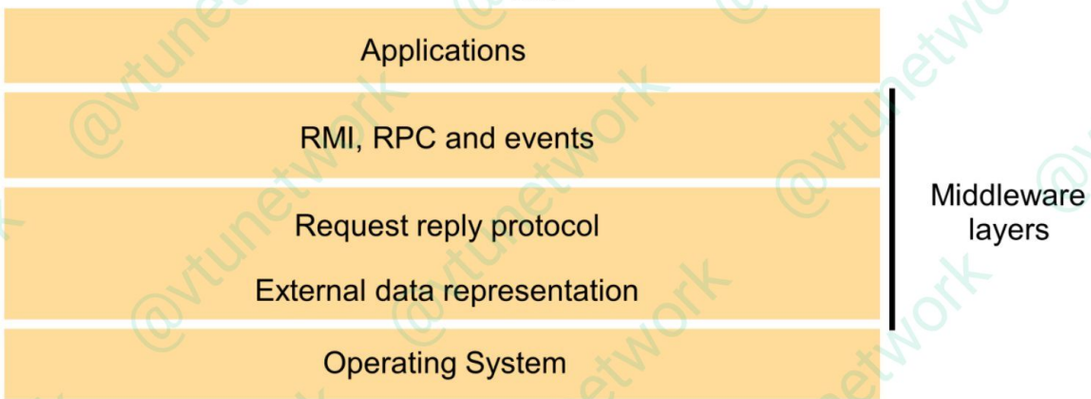
What are issues in distributing objects?

- How can we identify objects?

- What is involved in invoking a method implemented by the class?
    - What methods are available?
    - How can we pass parameters and get results?
- Can we track events in a distributed system?

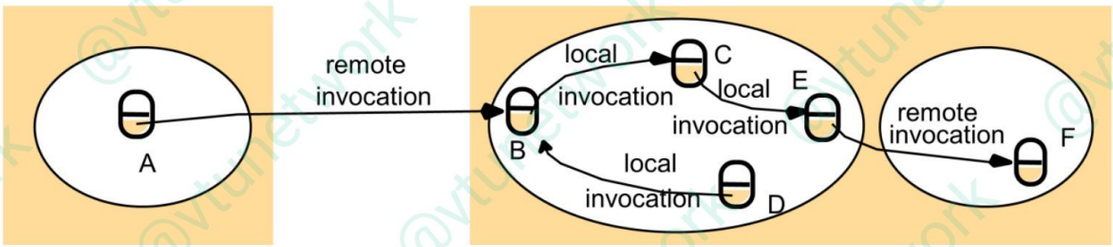
Distributed Objects

- 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
- This mechanism must be location-transparent.
- Middleware Roles
    - provide high-level abstractions such as RMI
    - enable location transparency
    - free from specifics of
        - communication protocols
        - operating systems and communication hardware
    - interoperability



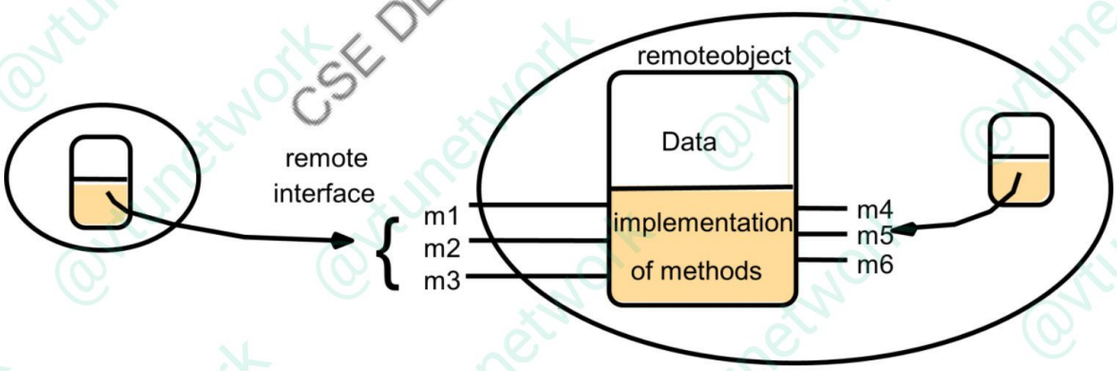
The Distributed Objects Model

- Remote method invocation – Method invocations between objects in different processes, whether in the same computer of not.
- Local method invocation – Method invocations between objects in the same process.
- Remote object – Objects that can receive remote invocations.
- Remote and local method invocations are shown in Figure 5.3.



- each process contains objects, some of which can receive remote invocations, others only local invocations
- those that can receive remote invocations are called *remote objects*
- objects need to know the *remote object reference* of an object in another process in order to invoke its methods. How do they get it?
- the *remote interface* specifies which methods can be invoked remotely
- Remote object reference
    - An object must have the remote object reference of an object in order to do remote invocation of an object
    - Remote object references may be passed as input arguments or returned as output arguments
- Remote interface
    - Objects in other processes can invoke only the methods that belong to its remote interface (Figure 5.4).
    - CORBA – uses IDL to specify remote interface

JAVA – extends interface by the **Remote** keyword.



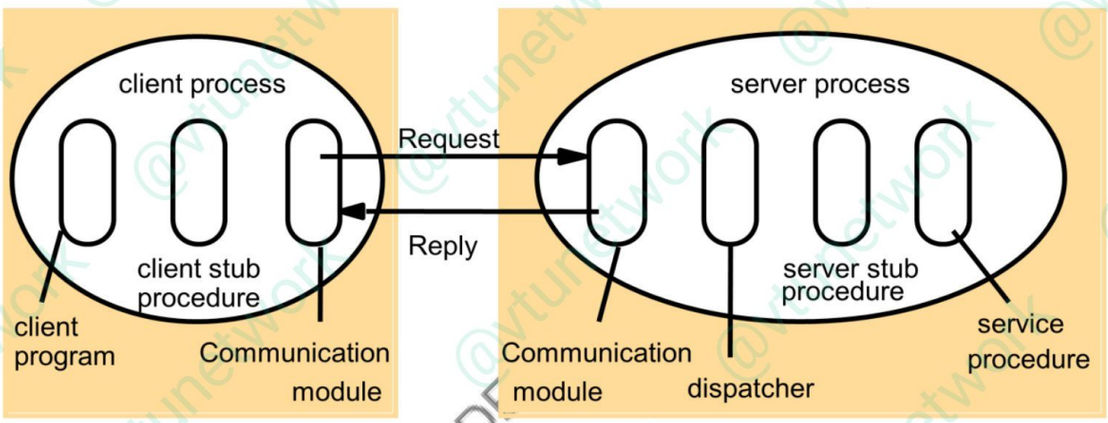
two important issues in making RMI natural extension of local method: (These problems won't occur in the local invocation.)

- **Number of times of invocations** are invoked in response to a single remote invocation
- **Level of location transparency**

- **Exactly once invocation semantics** - Every method is executed exactly once. (Ideal situation)

Remote Procedure Call Basics

- Problems with sockets
  - The read/write (input/output) mechanism is used in socket programming.
  - Socket programming is different from procedure calls which we usually use.
  - To make distributed computing transparent from locations, input/output is not the best way.
- A procedure call is a standard abstraction in local computation.
- Procedure calls are extended to distributed computation in Remote Procedure Call (RPC) as shown in Figure 5.7.
  - A caller invokes execution of procedure in the called via the local stub procedure.
  - The implicit network programming hides all network I/O code from the programmer.
  - Objectives are simplicity and ease of use.

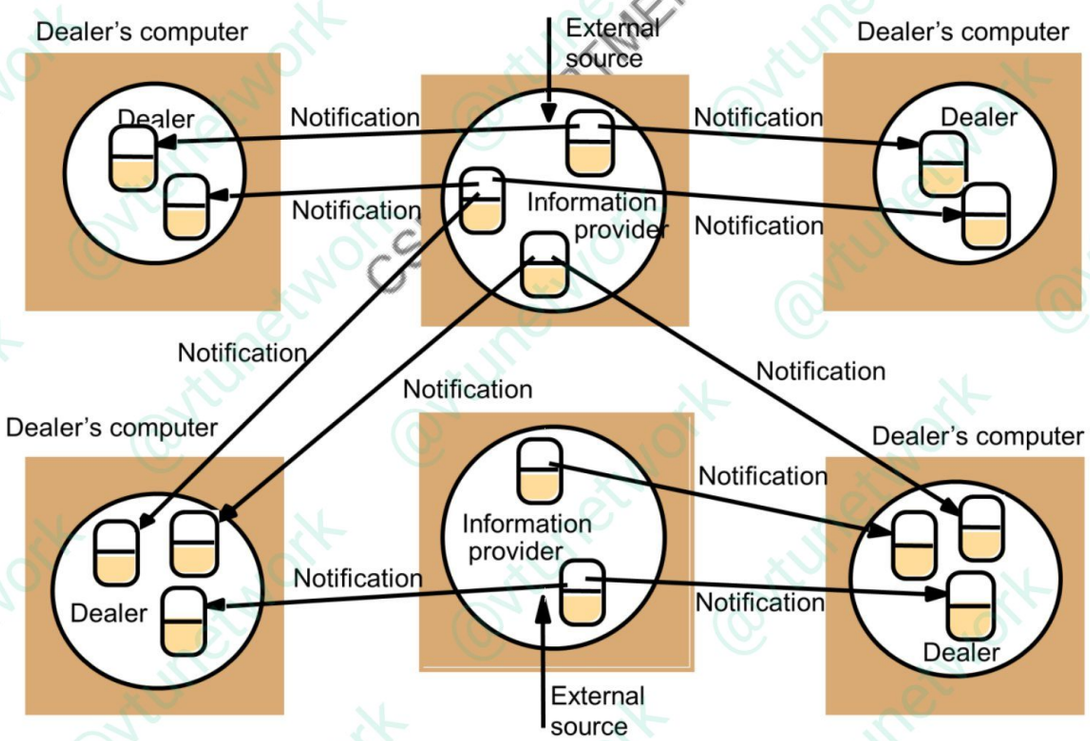


- The concept is to provide a transparent mechanism that enables the user to utilize remote services through standard procedure calls.
- Client sends request, then blocks until a remote server sends a response (reply).
- **Advantages**: user may be unaware of remote implementation (handled in a stub in library); uses standard mechanism.
- **Disadvantages**: prone to failure of components and network; different address spaces; separate process lifetimes.
- Differences with respect to message passing:
  - Message passing systems are peer-to-peer while RPC is more master/slave.
  - In message passing the calling process creates the message while in RPC the system create the message.
- Semantics of RPC:
  - Caller blocks.
  - Caller may send arguments to remote procedure.

- Callee may return results.
- Caller and callee access different address spaces.

Events and Notifications

- The idea behind the use of events is that one object can react to a change occurring in another object.
- The actions done by the user are seen as events that cause state changes in objects.
- The objects are notified whenever the state changes.
- Local event model can be extended to distributed event-based systems by using the publish-subscribe paradigm.
- In **publish-subscribe** paradigm
  - An object that has event publishes.
  - Those that have interest subscribe.
- Objects that represent events are called **notifications**.
- Distributed event-based systems have two main characteristics:
  - **Heterogeneous** – Event-based systems can be used to connect heterogeneous components in the Internet.
  - **Asynchronous** – Notification are sent asynchronously by event-generating objects to those subscribers.

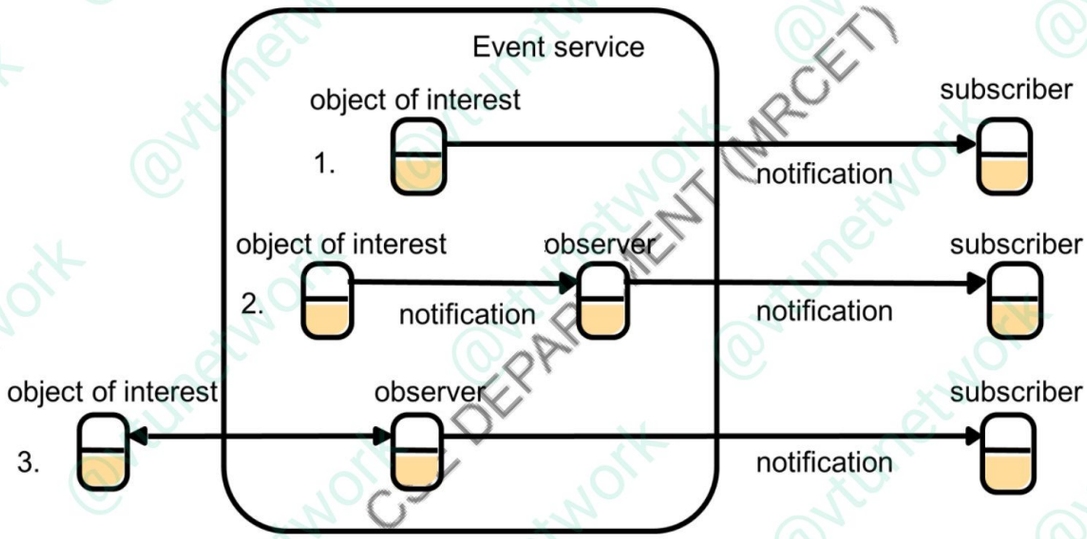


- The architecture of distributed event notification specifies the roles of participants as in Fig. 5.10:
  - It is designed in a way that publishers work independently from subscribers.

- Event service maintains a database of published events and of subscribers' interests.

- The **roles of the participants** are:
  - **Object of Interest** – This is an object experiences changes of state, as a result of its operations being invoked.

- The **roles of the participants** are (continued):
  - **Event** – An event occurs at an object of interest as the result of the completion of a method invocation.
  - **Notification** – A notification is an object that contains information about an event.
  - **Subscriber** – A subscriber is an object that has subscribed to some type of events in another object.
  - **Observer objects** – The main purpose of an observer is to separate an object of interest from its subscribers.
  - **Publisher** – This is an object that declares that it will generate notifications of particular types of event.



- A variety of **delivery semantics** can be employed:
  - **IP multicast protocol** – information delivery on the latest state of a player in an Internet game
  - **Reliable multicast protocol** – information provider / dealer
  - **Totally ordered multicast** - Computer Supported Cooperative Working (CSCW) environment
  - **Real-time** – nuclear power station / hospital patient monitor
- **Roles for observers** – the task of processing notifications can be divided among observers:
  - **Forwarding** – Observers simply forward notifications to subscribers.
  - **Filtering of notifications** – Observers address notifications to those subscribers who find these notifications are useful.
  - **Patterns of events** – Subscribers can specify patterns of events of interest.

- **Notification mailboxes** – A subscriber can set up a notification mailbox which receives the notification on behalf of the subscriber.

Java RMI

- **Start the server** in one window or in the background with the security policy

java -Djava.security.policy=policy HelloServer

or without the security policy

java HelloServer

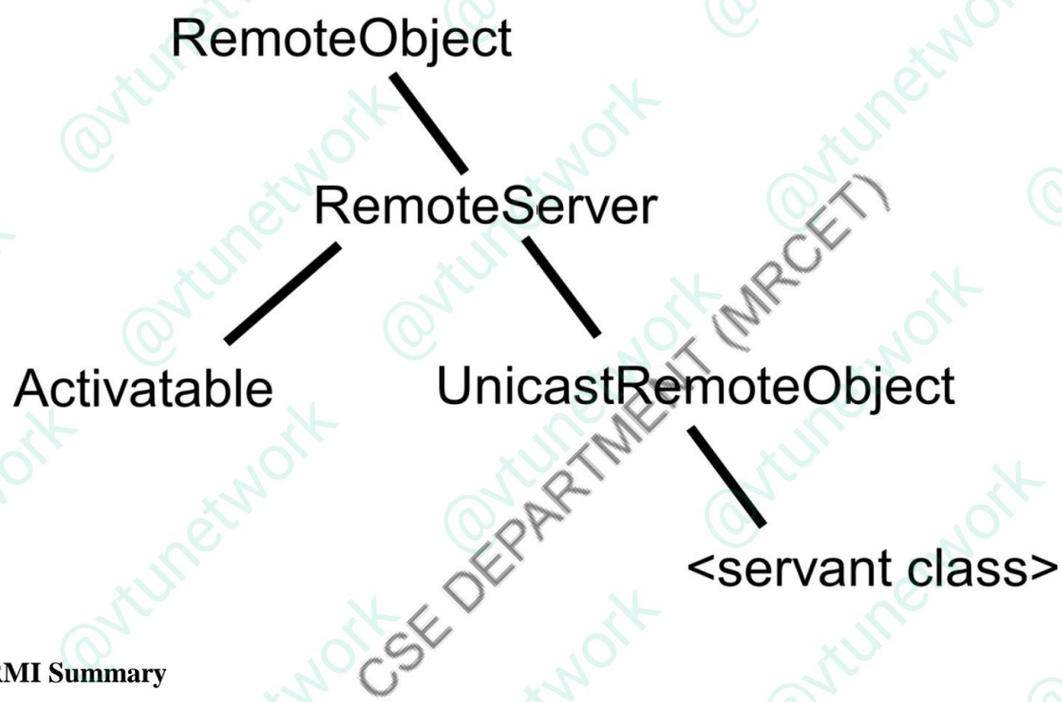
- **Run the client** in another window

java HelloClient testing

- An object must have the remote object reference of other object in order to do remote invocation of that object.
- Parameter and result passing
  - Remote object references may be passed as input arguments or returned as output arguments.
  - Parameters of a method in Java are input parameters.
  - Returned result of a method in Java is the single output parameter.
  - Objects are serialized to be passed as parameters.
  - When a remote object reference is returned, it can be used to invoke remote methods.
  - Local serializable objects are copied by value.
- Downloading of classes
  - Java is designed to allow classes to be downloaded from one virtual machine to another.
  - If the recipient of a remote object reference does not posses the proxy class, its code is downloaded automatically.
- RMIregistry
  - The RMIregistry is designed to allow is the binder for Java RMI.
  - It maintains a table mapping textual, URL-style names to references to remote objects.
- Server Program
  - The server consists of a main method and a servant class to implement each of its remote interface.
  - The main method of a server needs to create a security manager to enable Java security to apply the protection for an RMI server.
- Client Program
  - Any client program needs to get started by using a binder to look up a remote reference.

- A client can set a security manager and then looks up a remote object reference.

- **Callback** refers to server's action in notifying the client.
- **Callback Facility -** Instead of client polling the server, the server calls a method in the client when it is updated.
- Details
  - Client creates a remote object that implements an interface for the server to call.
  - The server provides an operation for clients to **register** their callbacks.
  - When an event occurs, the server calls the interested clients.



RMI Summary

- 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

Distributed File Systems: Introduction, File service Architecture, Case Study: 1: Sun Network File System , Case Study 2: The Andrew File System.

Distributed Shared Memory: Introduction, Design and Implementation issues, Consistency Models.

Distributed File Systems: Introduction

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

	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 (Ch. 18)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	✓	OceanStore(Ch. 10)

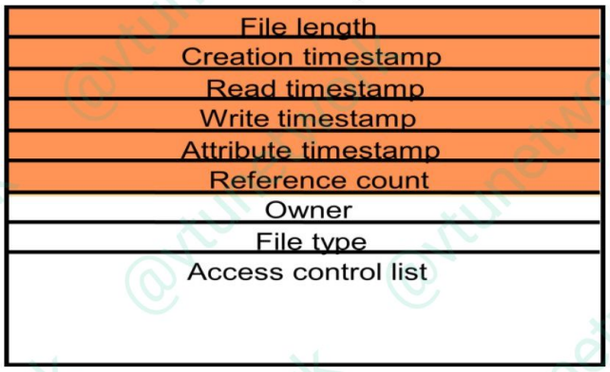
Figure 1. Storage systems and their properties

Types of consistency between copies: 1 - strict one-copy consistency
√ - approximate consistency
X - no automatic consistency

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

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

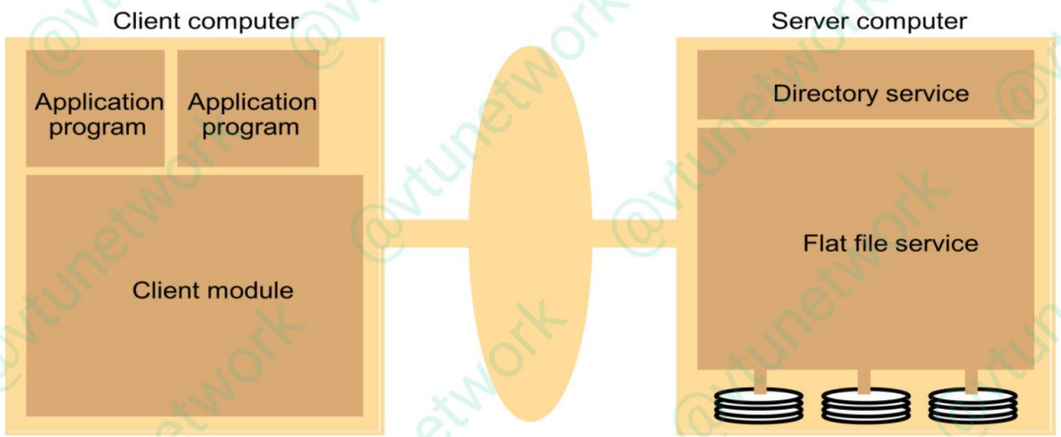
- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- Files contain both data and attributes.
- A typical attribute record structure is illustrated in Figure 3.



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

File Service Architecture

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



- The Client module implements exported interfaces by flat file and directory services on server side.
- Responsibilities of various modules can be defined as follows:
  - Flat file service:
    - Concerned with the implementation of operations on the contents of file. Unique File Identifiers (UFIDs) are used to refer to files in all requests for flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.
  - Directory service:
    - Provides mapping between text names for the files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.
  - Client module:
    - It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.
    - It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.
  - 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.

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

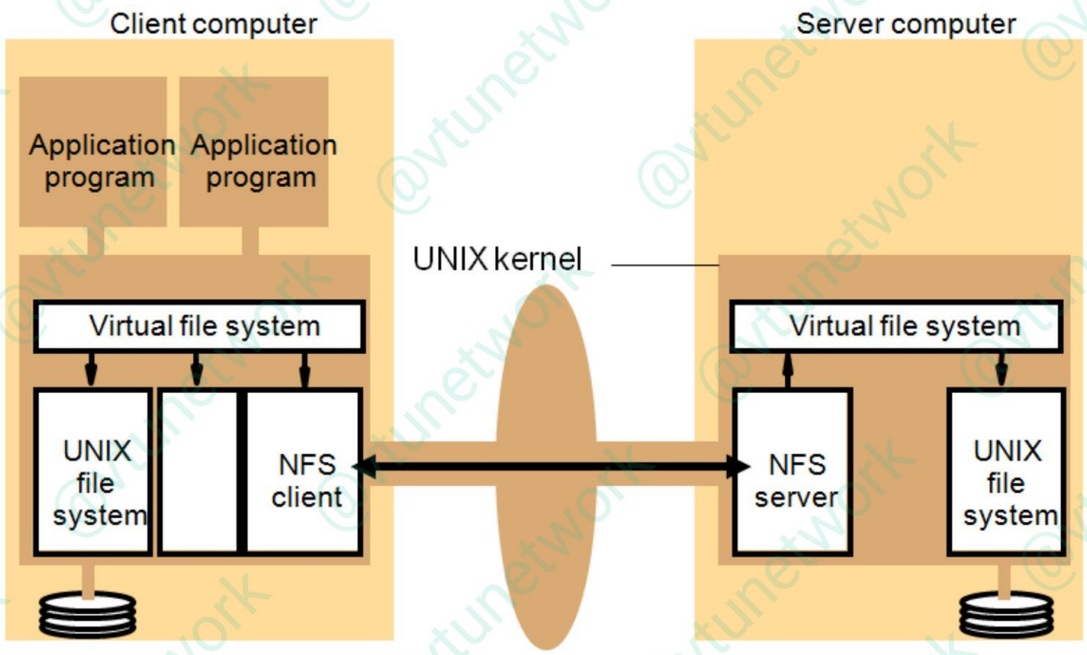
- Hierarchic file system
  - A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.
- File Group
  - A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.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

Case Study: 1: Sun Network File System

NFS (Network File System)

- Developed by Sun Microsystems (in 1985)
- Most popular, open, and widely used.
- NFS protocol standardized through IETF (RFC 1813)

NFS architecture



fh = file handle:

Filesystem identifier	i-node number	i-node generation

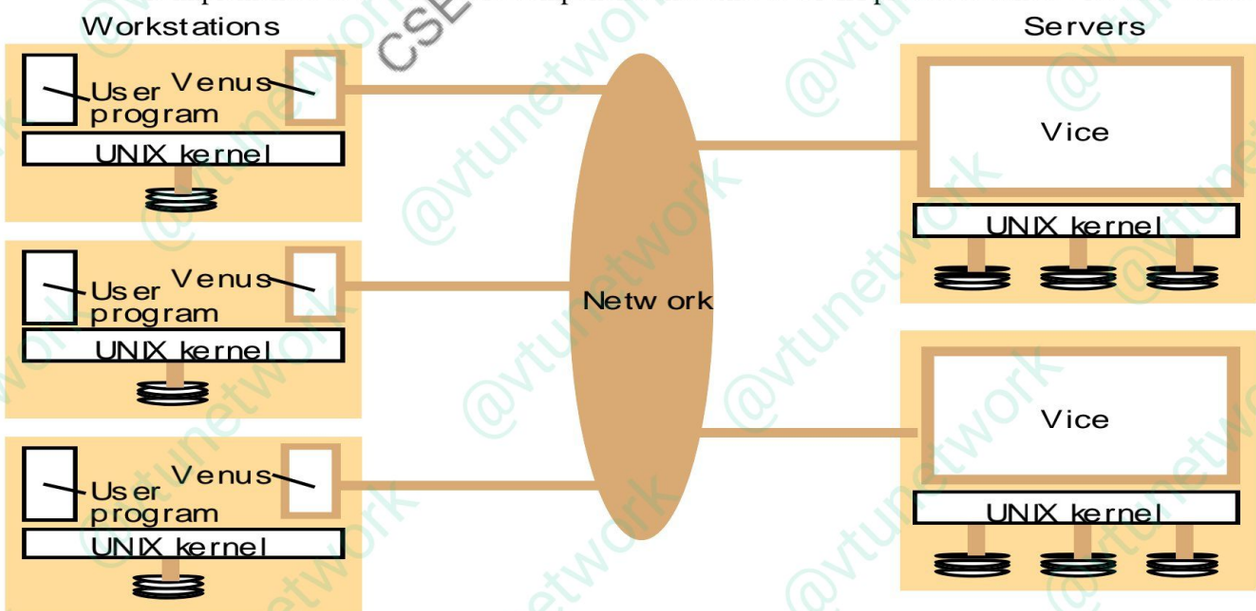
A simplified representation of the RPC interface provided by NFS version 3 servers is shown in Figure

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

case study 2: AFS (Andrew File System)

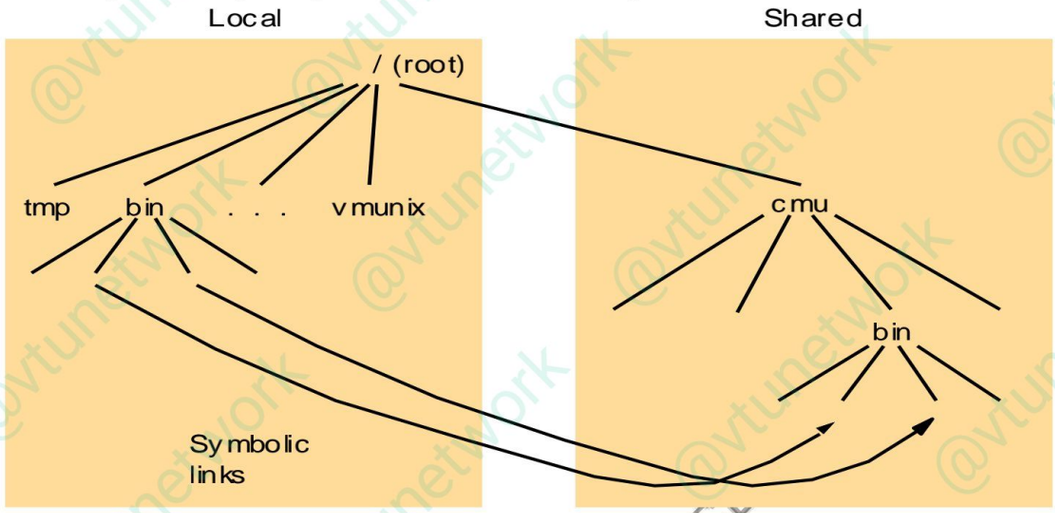
- Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
- A research project to create campus wide file system.
- Public domain implementation is available on Linux (LinuxAFS)
- It was adopted as a basis for the DCE/DFS file system in the Open Software Foundation (OSF, www.opengroup.org) DEC (Distributed Computing Environment

- 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.

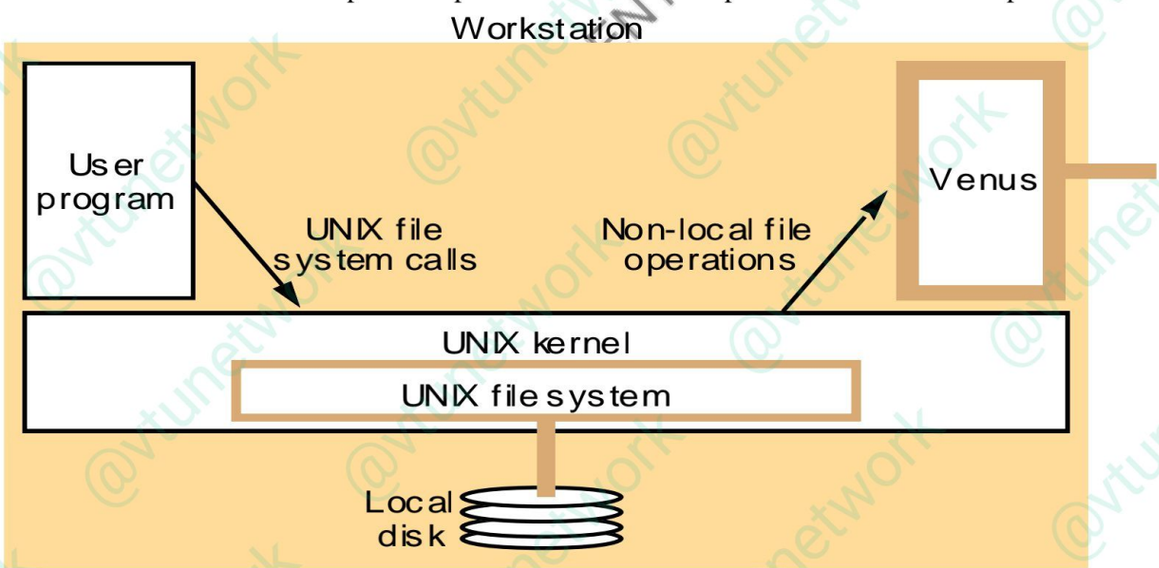


- The files available to user processes running on workstations are either local or shared.

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



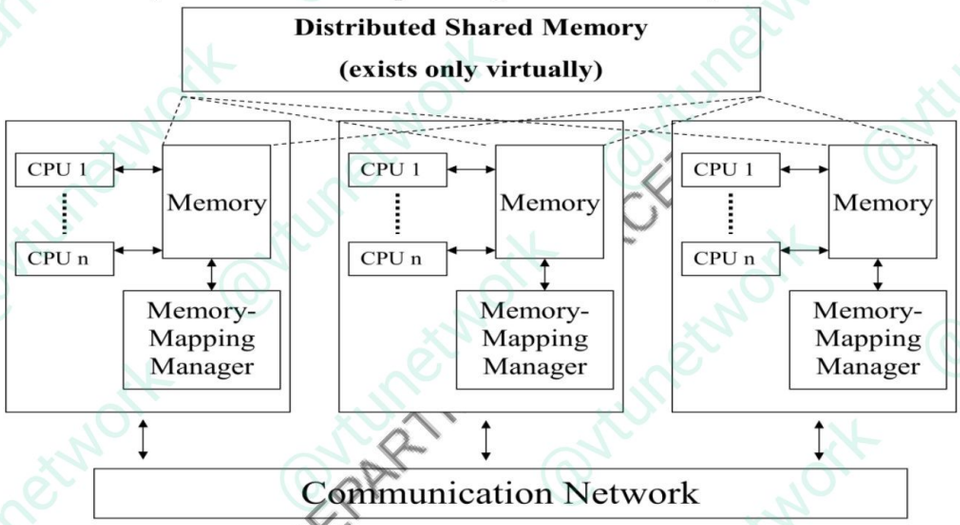
- The UNIX kernel in each workstation and server is a modified version of BSD UNIX.
- The modifications are designed to intercept open, close and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer.

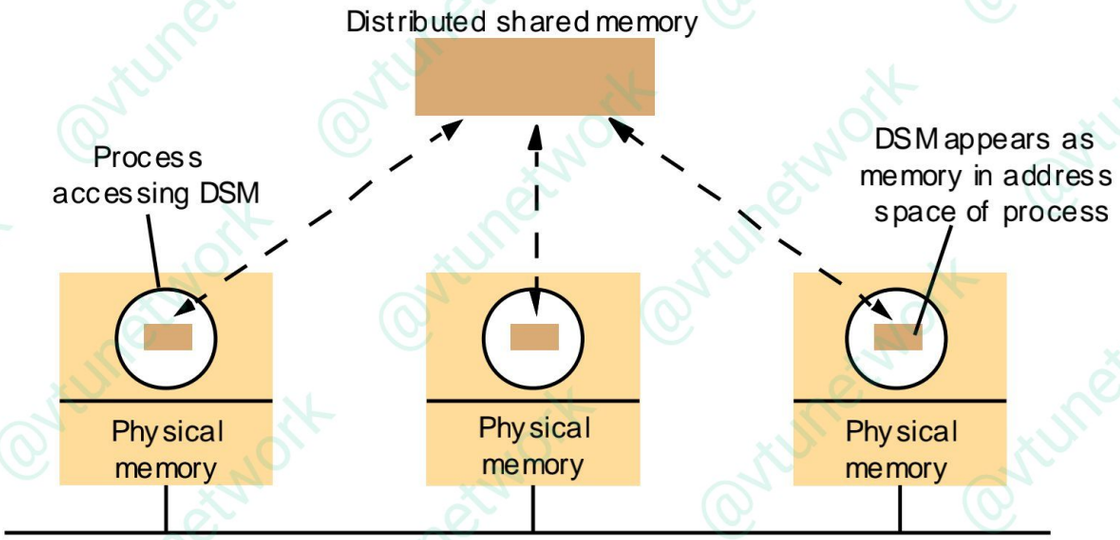


System call interception in AFS

Distributed Shared Memory: Introduction

- Distributed Shared Memory (DSM) allows programs running on separate computers to share data without the programmer having to deal with sending messages
- Instead underlying technology will send the messages to keep the DSM consistent (or relatively consistent) between computers
- DSM allows programs that used to operate on the same computer to be easily adapted to operate on separate computers
- Programs access what appears to them to be normal memory
- Hence, programs that use DSM are usually shorter and easier to understand than programs that use message passing
- However, DSM is not suitable for all situations. Client-server systems are generally less suited for DSM, but a server may be used to assist in providing DSM functionality for data shared between clients





DSM vs Message passing

DSM	Message passing
Variables are shared directly	Variables have to be marshalled yourself
Processes can cause error to one another by altering data	Processes are protected from one another by having private address spaces
Processes may execute with non-overlapping lifetimes	Processes must execute at the same time
Invisibility of communication's cost	Cost of communication is obvious

DSM implementations

- Hardware: Mainly used by shared-memory multiprocessors. The hardware resolves LOAD and STORE commands by communicating with remote memory as well as local memory
- Paged virtual memory: Pages of virtual memory get the same set of addresses for each program in the DSM system. This only works for computers with common data and paging formats. This implementation does not put extra structure requirements on the program since it is just a series of bytes.
- Middleware: DSM is provided by some languages and middleware without hardware or paging support. For this implementation, the programming language, underlying system libraries, or middleware send the messages to keep the data synchronized between programs so that the programmer does not have to.

Efficiency

- DSM systems can perform almost as well as equivalent message-passing programs for systems that run on about 10 or less computers.
- There are many factors that affect the efficiency of DSM, including the implementation, design approach, and memory consistency model chosen.

Design and implementation issues

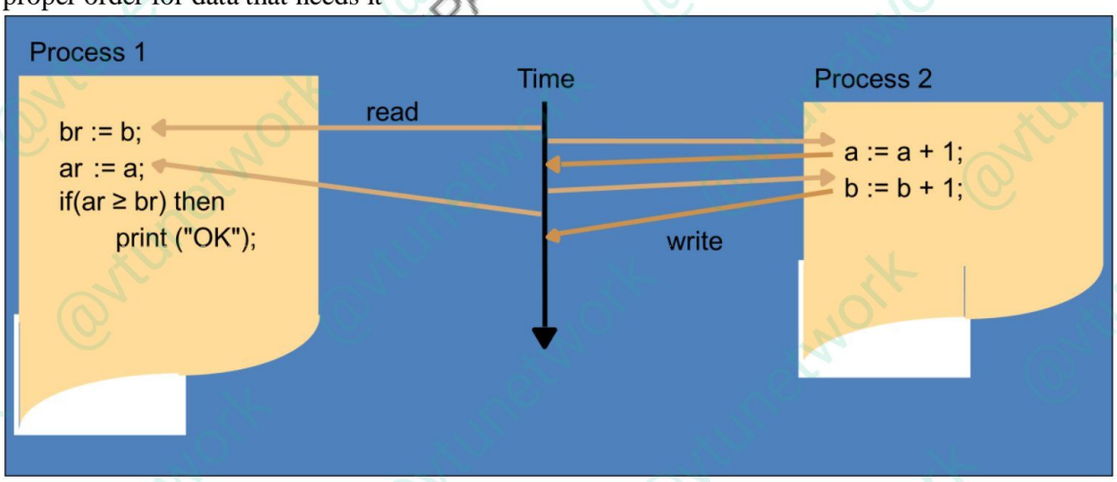
- Byte-oriented: This is implemented as a contiguous series of bytes. The language and programs determine the data structures
- Object-oriented: Language-level objects are used in this implementation. The memory is only accessed through class routines and therefore, OO semantics can be used when implementing this system
- Immutable data: Data is represented as a group of many tuples. Data can only be accessed through read, take, and write routines

Memory consistency

- To use DSM, one must also implement a distributed synchronization service. This includes the use of locks, semaphores, and message passing
- Most implementations, data is read from local copies of the data but updates to data must be propagated to other copies of the data
- Memory consistency models determine when data updates are propagated and what level of inconsistency is acceptable

Memory consistency models

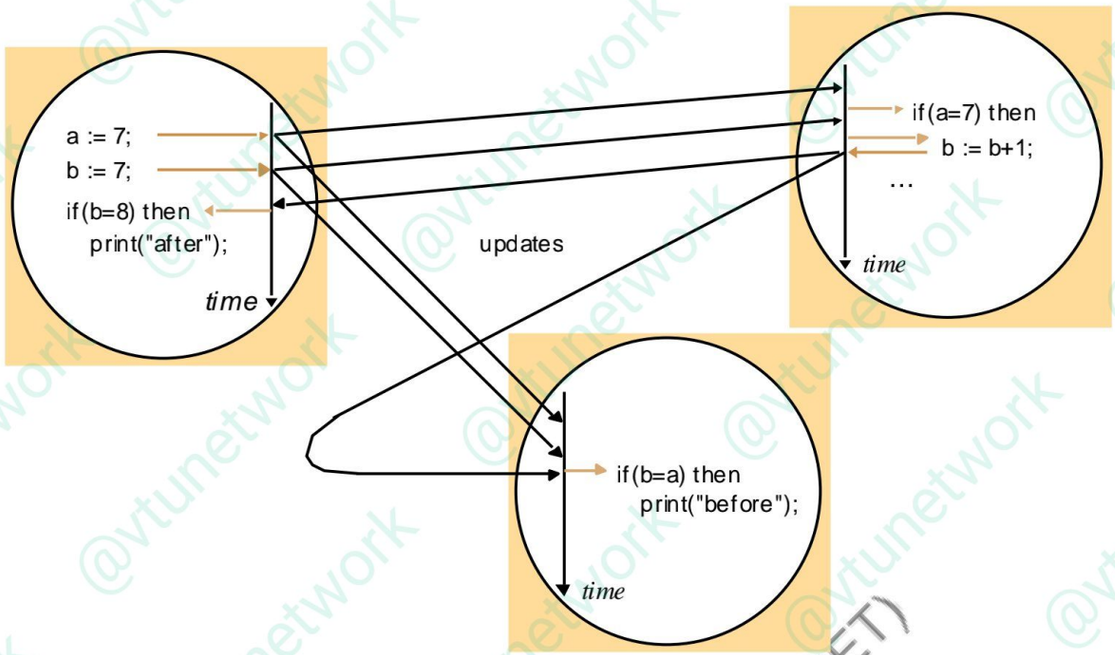
- Linearizability or atomic consistency is the strongest model. It ensures that reads and writes are made in the proper order. This results in a lot of underlying messaged being passed.
  - Variables can only be changed by a write operation
  - The order of operations is consistent with the real times at which the operations occurred in the actual execution
- Sequential consistency is strong, but not as strict. Reads and writes are done in the proper order in the context of individual programs.
  - The order of operations is consistent with the program order in which each individual client executed them
- Coherence has significantly weaker consistency. It ensures writes to individual memory locations are done in the proper order, but writes to separate locations can be done in improper order
- Weak consistency requires the programmer to use locks to ensure reads and writes are done in the proper order for data that needs it



Interleaving under sequential consistency

Update options

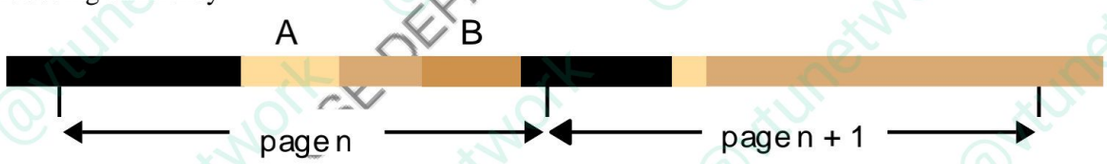
- Write-update: Each update is multicast to all programs. Reads are performed on local copies of the data
- Write-invalidate: A message is multicast to each program invalidating their copy of the data before the data is updated. Other programs can request the updated data



DSM using write-update

Granularity

- Granularity is the amount of data sent with each update
- If granularity is too small and a large amount of contiguous data is updated, the overhead of sending many small messages leads to less efficiency
- If granularity is too large, a whole page (or more) would be sent for an update to a single byte, thus reducing efficiency



Trashing

- Thrashing occurs when network resources are exhausted, and more time is spent invalidating data and sending updates than is used doing actual work
- Based on system specifics, one should choose write-update or write-invalidate to avoid thrashing

Consistency models:

- **Strict consistency**
- **Sequential consistency**
- **Release Consistency**
- **Causal consistency**
- **Processor consistency**

UNIT V

Transactions and Concurrency Control: Introduction, Transactions, Nested Transactions, Locks, Optimistic concurrency control, Timestamp ordering, Comparison of methods for concurrency control.

Distributed Transactions: Introduction, Flat and Nested Distributed Transactions, Atomic commit protocols, Concurrency control in distributed transactions, Distributed deadlocks, Transaction recovery.

Transactions and Concurrency Control: Introduction

Banking transaction for a customer (e.g., at ATM or browser)

Transfer $100 from saving to checking account;

Transfer $200 from money-market to checking account;

Withdraw $400 from checking account.

Transaction (invoked at client): /* Every step is an RPC */

1. savings.withdraw(100) /* includes verification */
2. checking.deposit(100) /* depends on success of 1 */
3. mnymkt.withdraw(200) /* includes verification */
4. checking. deposit(200) /* depends on success of 3 */
5. checking.withdraw(400) /* includes verification */
6. dispense(400)
7. commit

- All the following are RPCs from a client to the server
- Transaction calls that can be made at a client, and return values from the server:

Transactions

openTransaction() -> trans;

starts a new transaction and delivers a unique transaction identifier (TID) trans. This TID will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);

ends a transaction: a commit return value indicates that the transaction has committed; an abort return value indicates that it has aborted.

abortTransaction(trans);

aborts the transaction.

- TID can be passed implicitly (for other operations between open and close) with CORBA
- deposit(amount)

- deposit amount in the account
- withdraw(amount)
- withdraw amount from the account
- getBalance() -> amount
- return the balance of the account
- setBalance(amount)
- set the balance of the account to amount
- Sequence of operations that forms a single step, transforming the server data from one consistent state to another.
  - All or nothing principle: a transaction either completes successfully, and the effects are recorded in the objects, or it has no effect at all. (even with multiple clients, or crashes)
- A transactions is indivisible (atomic) from the point of view of other transactions
  - No access to intermediate results/states of other transactions
  - Free from interference by operations of other transactions

But…

- Transactions could run concurrently, i.e., with multiple clients
- Transactions may be distributed, i.e., across multiple servers
- Atomicity: All or nothing
- Consistency: if the server starts in a consistent state, the transaction ends the server in a consistent state.
- Isolation: Each transaction must be performed without interference from other transactions, i.e., the non-final effects of a transaction must not be visible to other transactions.
- Durability: After a transaction has completed successfully, all its effects are saved in permanent storage.
- Atomicity: store tentative object updates (for later undo/redo) – many different ways of doing this
- Durability: store entire results of transactions (all updated objects) to recover from permanent server crashes.
- The effect of an operation refers to
- The value of an object set by a write operation
- The result returned by a read operation.
- Two operations are said to be conflicting operations, if their combined effect depends on the order they are executed, e.g., read-write, write-read, write-write (all on same variables). NOT read-read, NOT on different variables.
- Two transactions are serially equivalent if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.

- ❖ Why? Can start from original operation sequence and swap the order of non-conflicting operations to obtain a series of operations where one transaction finishes completely before the second transaction starts
- ❖ Why is the above result important? Because: Serial equivalence is the basis for concurrency control protocols for transactions.

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Concurrency control

- Lost update
  - 3 accounts (A, B, C)
    - » with balances 100, 200, 300
  - T1 transfers from A to B, for 10% increase
  - T2 transfers from C to B, for 10% increase
  - Both T1, T2 read balance of B (200)
  - T1 overwrites the update by T2
    - » Without seeing it
- Inconsistent retrievals
  - T1: transfers 10% of account A to account B
  - T2: computes sum of account balances
  - T2 computes sum before T1 updates B

Recoverability from aborts

- Servers must prevent a aborting Tx from affecting other concurrent Tx's.
  - Dirty reads:
    - » T2 sees result update by T1 on account A
    - » T2 performs its own update on A & then commits.

» T1 aborts -> T2 has seen a “transient” value

• T2 is not recoverable

» If T2 delays its commit until T1’s outcome is resolved:

• Abort(T1) -> Abort(T2)

• However, if T3 has seen results of T2:

– Abort(T2) -> Abort(T3) !

- Premature writes:
  - Assume server implements abort by maintaining the “before” image of all update operations
    - » T1 & T2 both updates account A
    - » T1 completes its work before T2
    - » If T1 commits & T2 aborts, the balance of A is correct
    - » If T1 aborts & T2 commits, the “before” image that is restored corresponds to the balance of A before T2
    - » If both T1 & T2 abort, the “before” image that is restored corresponds to the balance of A as set by T1
- Tx’s should delay both their reads & updates in order to avoid interference
  - Strict execution -> enforce isolation
- Servers should maintain tentative versions of objects in volatile memory

- Tx’s should delay both their reads & updates in order to avoid interference
  - Strict execution -> enforce isolation
- Servers should maintain tentative versions of objects in volatile memory

Concurrency Control: Locks

- Transactions:
  - Must be scheduled so that their effect on shared data is serially equivalent
  - Two types of approach
    - » Pessimistic → If something can go wrong, it will

Operations are synchronized before they are carried out

» Optimistic → In general, nothing will go wrong

Operations are carried out, synchronization at the end of the transaction

– Locks (pessimistic)

» can be used to ensuring serializability

» lock(x), unlock(x)

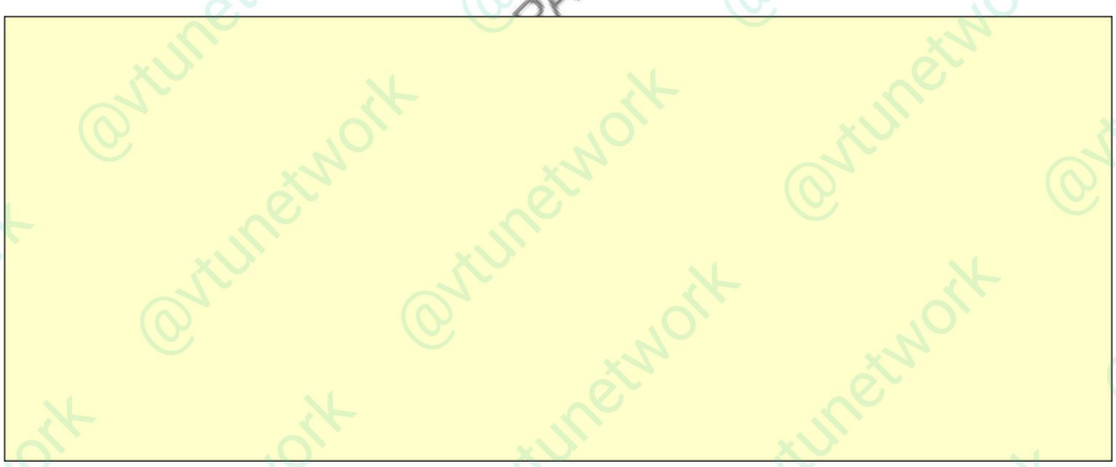
- Oldest and most widely used CC algorithm
- A process before read/write ➔ requests the scheduler to grant a lock
- Upon finishing read/write ➔ the lock is released
- In order to ensure serialized transaction Two Phase Locking (2PL) is used
- How Locks prevent consistency problems
  - Lost update and inconsistent retrieval:
  - Causes:
    - » are caused by the conflict between $r_i(x)$ and $w_j(x)$
    - » two transactions read a value and use it to compute new value
  - Prevention:
    - » delay the reads of later transactions until the earlier ones have completed
    - » Disadvantage of Locking
  - Deadlocks
- Strict 2PL avoids Cascading Aborts
  - A situation where a committed transaction has to be undone because it saw a file it shouldn't have seen.
- Problems of Locking
  - Deadlocks
  - Livelocks
    - » A transaction can't proceed for an indefinite amount of time while other transactions continue normally. It happens due to unfair locking.
  - Lock overhead
    - » If the system doesn't allow shared access--wastage of resources
  - Avoidance of Cascading Aborts may be costly
    - » Strict 2PL in fact, reduces the effect of concurrency
- Transaction managers (on server side) set locks on objects they need. A concurrent trans. cannot access locked objects.
- Two phase locking:
  - In the first (growing) phase of the transaction, new locks are only acquired, and in the second (shrinking) phase, locks are only released.

– A transaction is not allowed acquire *any* new locks, once it has released any one lock.

Deadlock with write locks

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s	*a.withdraw(200);*	waits for *T*'s
	lock on *B*	•••	lock on *A*
•••		•••	
•••		•••	

T locks A and waits for U to release the lock on B, U on the other hand locks B and waits for T to release the lock on A
➔ Circular hold and wait ➔ Deadlock



❖ Necessary conditions for deadlocks

- ❑ Non-shareable resources (exclusive lock modes)
- ❑ No preemption on locks
- ❑ Hold & Wait or Circular Wait

Naïve Deadlock Resolution Using Timeout

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	•••	
	T's lock on *A* becomes vulnerable, unlock *A*, abort T	•••	
		a.withdraw(200);	write lock *A*
			unlock *A*, *B*

Disadvantages?

Strategies to Fight Deadlock

- Lock timeout (costly and open to false positives)
- Deadlock Prevention: violate one of the necessary conditions for deadlock (from 2 slides ago), e.g., lock all objects before transaction starts, aborting entire transaction if any fails
- Deadlock Avoidance: Have transactions declare max resources they will request, but allow them to lock at any time (Banker's algorithm)
- Deadlock Detection: detect cycles in the wait-for graph, and then abort one or more of the transactions in cycle
- We have seen locking has some problems
- OCC based on the following simple idea:

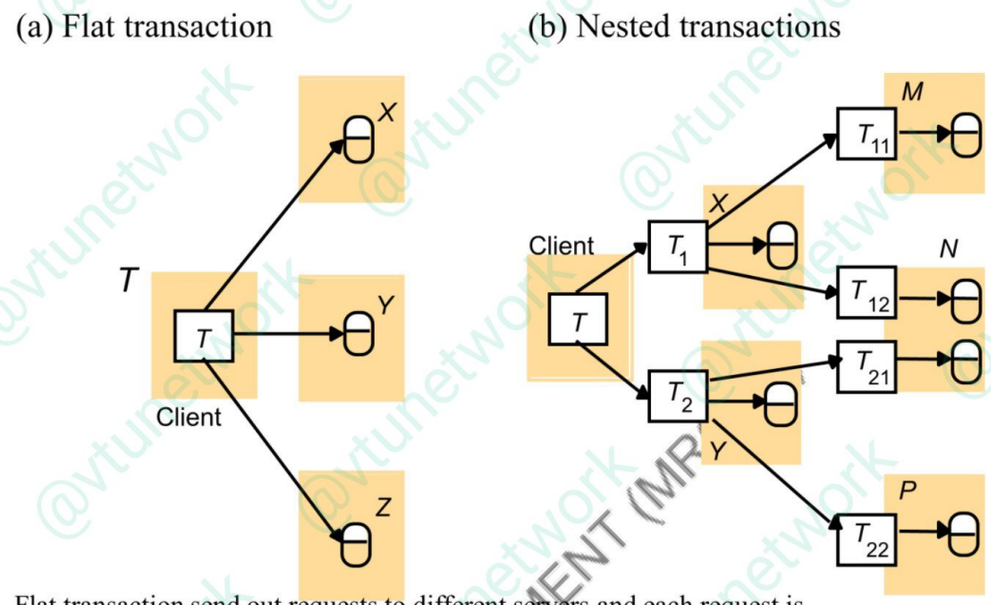
Distributed Transactions

In previous chapter, we discussed transactions accessed objects at a single server. In the general case, a transaction will access objects located in different computers. Distributed transaction accesses objects managed by multiple servers.

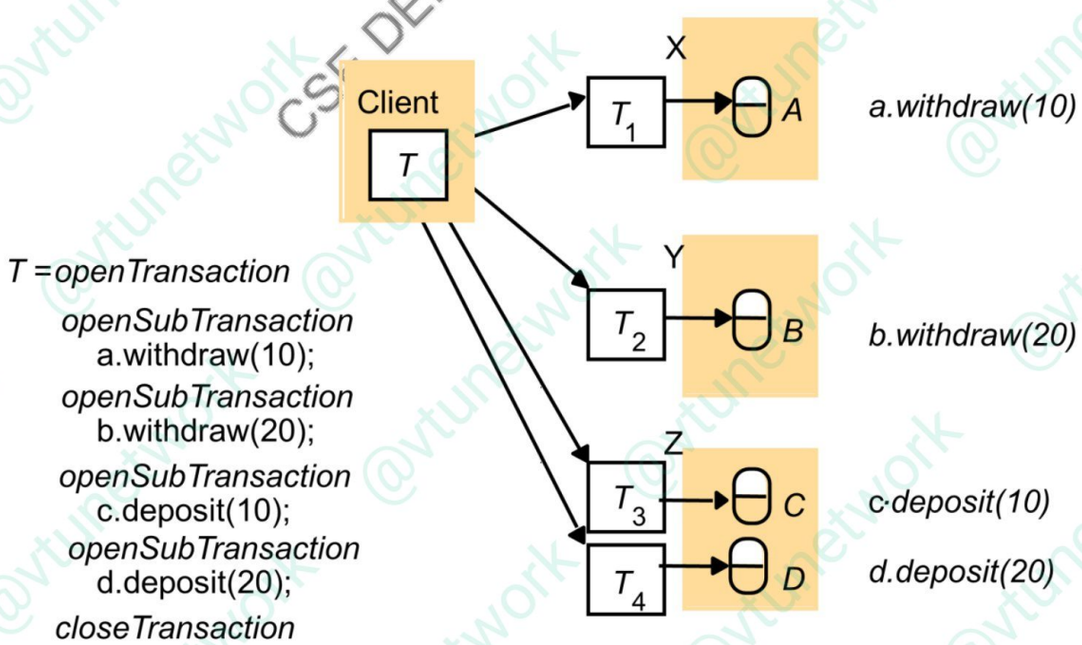
The atomicity property requires that either all of the servers involved in the same transaction commit the transaction or all of them abort. Agreement among servers are necessary.

Transaction recovery is to ensure that all objects are recoverable. The values of the objects reflect all changes made by committed transactions and none of those made by aborted ones.

Figure 14.1
Distributed transactions



Flat transaction send out requests to different servers and each request is completed before client goes to the next one. Nested transaction allows sub-transactions at the same level to execute concurrently.



A transaction comes to an end when the client requests that a transaction be committed or aborted.

Simple way is: coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they had carried it out.

Inadequate because when the client requests a commit, it does not allow a server to make a unilateral decision to abort a transaction. E.g. deadlock avoidance may force a transaction to abort at a server when locking is used. So any server may fail or abort and client is not aware.

Allow any participant to abort its part of a transaction. Due to atomicity, the whole transaction must also be aborted.

In the first phase, each participant votes for the transaction to be committed or aborted. Once voted to commit, not allowed to abort it. So before votes to commit, it must ensure that it will eventually be able to carry out its part, even if it fails and is replaced.

A participant is said to be in a **prepared** state if it will eventually be able to commit it. So each participant needs to save the altered objects in the permanent storage device together with its status-prepared.

Phase 1 (voting phase):

1. The coordinator sends a canCommit? request to each of the participants in the transaction.

2. When a participant receives a canCommit? request it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.

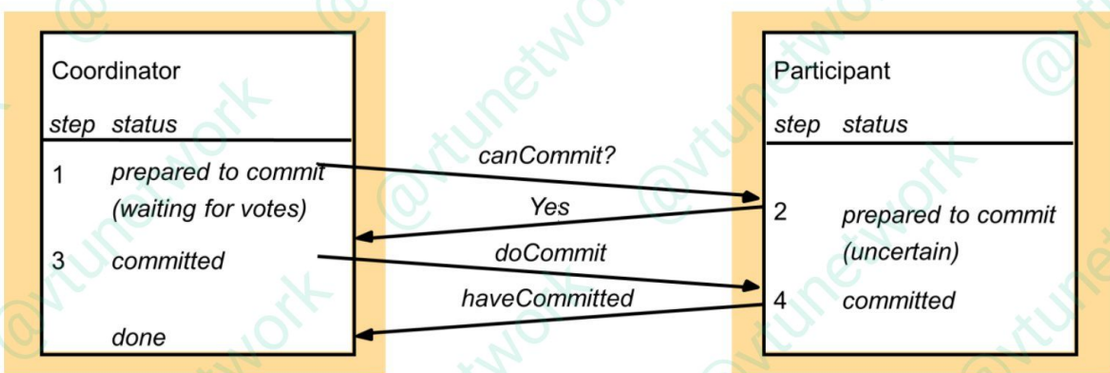
Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).

(a) If there are no failures and all the votes are Yes the coordinator decides to commit the transaction and sends a doCommit request to each of the participants.

(b) Otherwise the coordinator decides to abort the transaction and sends doAbort requests to all participants that voted Yes.

4. Participants that voted Yes are waiting for a doCommit or doAbort request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a haveCommitted call as confirmation to the coordinator.



Consider when a participant has voted Yes and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort.

Such a participant is uncertain and cannot proceed any further. The objects used by its transaction cannot be released for use by other transactions.

Participant makes a getDecision request to the coordinator to determine the outcome. If the coordinator has failed, the participant will not get the decision until the coordinator is replaced resulting in extensive delay for participant in uncertain state.

Timeout are used since exchange of information can fail when one of the servers crashes, or when messages are lost So process will not block forever.

Provided that all servers and communication channels do not fail, with N participants

N number of canCommit? Messages and replies

Followed by N doCommit messages

The cost in messages is proportional to 3N

The cost in time is three rounds of message.

The cost of haveCommitted messages are not counted, which can function correctly without them- their role is to enable server to delete stale coordinator information.

Performance of two-phase commit protocol

Provided that all servers and communication channels do not fail, with N participants

N number of *canCommit?* Messages and replies

Followed by N *doCommit* messages

The cost in messages is proportional to 3N

The cost in time is three rounds of message.

The cost of *haveCommitted* messages are not counted, which can function correctly without them- their role is to enable server to delete stale coordinator information.

When a participant has voted *Yes* and is waiting for the coordinator to report on the outcome of the vote, such participant is in uncertain stage. If the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the uncertain state.

Concurrency Control in Distributed Transactions

Concurrency control for distributed transactions: each server applies local concurrency control to its own objects, which ensure transactions serializability locally.

However, the members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. Thus global serializability is required.

Locks

Lock manager at each server decide whether to grant a lock or make the requesting transaction wait.

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

A lock managers in different servers set their locks independently of one another. It is possible that different servers may impose different orderings on transactions.

Timestamp ordering concurrency control

In a single server transaction, the coordinator issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them.

In distributed transactions, we require that each coordinator issue globally unique time stamps. The coordinators must agree as to the ordering of their timestamps. <local timestamp, server-id>, the agreed ordering of pairs of timestamps is based on a comparison in which the server-id is less significant.

The timestamp is passed to each server whose objects perform an operation in the transaction.

To achieve the same ordering at all the servers, The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. E.g. If T commits after U at server X, T must commits after U at server Y.

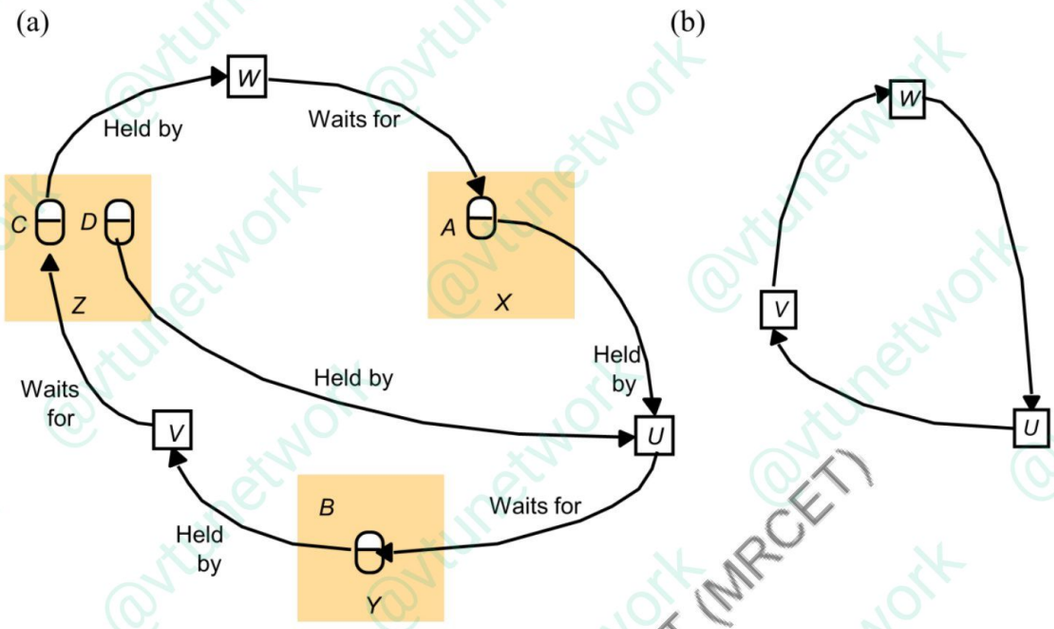
Conflicts are resolved as each operation is performed. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the participants.

Distributed Deadlock

Deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks.

Using timeout to resolve deadlock is a clumsy approach. Why? Another way is to detect deadlock by detecting cycles in a wait for graph

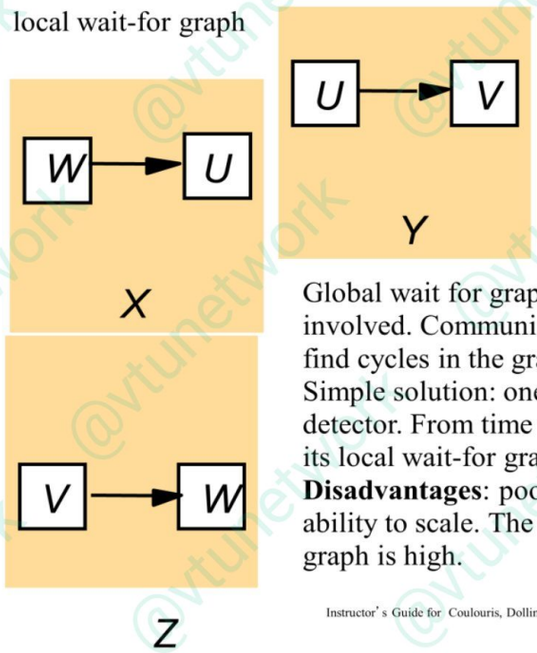
Figure 14.14
Distributed deadlock



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005

- A deadlock that is detected but is not really a deadlock is called a phantom deadlock.
- As the procedure of sending local wait-for graph to one place will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

Figure 14.14
Local and global wait-for graphs



Global wait for graph is held in part by each of the several servers involved. Communication between these servers is required to find cycles in the graph.
Simple solution: one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for graph.
Disadvantages: poor availability, lack of fault tolerance and no ability to scale. The cost of frequent transmission of local wait-for graph is high.

At about the same time, T waits for U (T->U) and W waits for V (W->V). Two probes occur, two deadlocks detected by different servers.

We want to ensure that only one transaction is aborted in the same deadlock since different servers may choose different transaction to abort leading to unnecessary abort of transactions.

So using priorities to determine which transaction to abort will result in the same transaction to abort even if the cycles are detected by different servers.

Using priority can also reduce the number of probes. For example, we only initiate probe when higher priority transaction starts to wait for lower priority transaction.

If we say the priority order from high to low is: T, U, V and W. Then only the probe of T->U will be sent and not the probe of W->V.

Transaction recovery

Atomic property of transactions can be described in two aspects:

Durability: objects are saved in permanent storage and will be available indefinitely thereafter. Acknowledgement of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's volatile object.

Failure atomicity: the effects of transactions are atomic even when the server crashes.

Both can be realized by recovery manager.

Recovery manager

Tasks of a recovery manager:

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.

Figure 14.18
Types of entry in a recovery file

Type of entry	*Description of contents of entry*
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared*, *committed* *aborted*) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <identifier of object>, < Position of value of object>.

Intention list records all of its currently active transactions. A list of a particular transaction contains a list of the references and the values of all the objects that are altered. When committed, the committed version of each object is replaced by the tentative version made by that transaction. When a transaction aborts, the server uses the intention list to delete all the tentative versions of objects.

When a participant says it is prepared to commit, its recovery manager must have saved both its intention list for that transaction and the objects in that intention list in its recovery file, so it will be able to carry out the commitment later on, even if it crashes in the interim.

Log with entries relating to two-phase commit protocol

In phase 1, when the coordinator is prepared to commit and has already added a prepared status entry, its recovery manager adds a coordinator entry. Before a participant can vote Yes, it must have already prepared to commit and must have already added a prepared status entry. When it votes Yes, its recovery manager records a participant entry and adds an uncertain status. When a participant votes No, it adds an abort status to recovery file.

In phase 2, the recovery manager of the coordinator adds either a committed or an aborted, according to the decision. Recovery manager of participants add a commit or abort status to their recovery files according to

message received from coordinator. When a coordinator has received a confirmation from all its participants, its recovery manager adds a done status.

When a server is replaced after a crash, the recovery manager has to deal with the two-phase commit protocol in addition to restore the objects.

For any transaction where the server has played the coordinator role, it should find a coordinator entry and a set of transaction status entries. For any transaction where the server has played the participant role, it should find a participant entry and a set of set of transaction status entries. In both cases, the most recent transaction status entry, that is the one nearest the end of log determine the status at the time of failure.

The action of the recovery manage with respect to the two-phase commit protocol for any transaction depends on whether the server was the coordinator or a participant and on its status at the time of failure as shown in the following table.

Figure 14.22
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.