DIGITAL NOTES

ON

DISTRIBUTED SYSTEMS

[R20A0520]

B.TECH III YEAR–I SEM

2023-2024

PREPARED BY
B.RAMYA SRI
V.V.NAGAMANI
T.SRINIDHI

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution–UGC, Govt.of India)

Recognized under2(f)and12(B)ofUGCACT1956
(AffiliatedtoJNTUH,Hyderabad,ApprovedbyAICTE-AccreditedbyNBA&NAAC–'A'Grade-ISO9001:2015Certified)
Maisammaguda,Dhulapally(PostVia.Hakimpet),Secunderabad–500100,TelanganaState,India

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

B.TECH-III -YEAR I-SEM-IT **L/T/P/C**

3/-/-/3

Objectives:

1. To learn the principles, architectures, algorithms and programming models used in distributed systems.
2. To analyze the algorithms of mutual exclusion, election & multicast communication.
3. To evaluate the different mechanisms for Interposes communication and remote invocations.
4. To design and implement sample distributed systems.
5. To apply transactions and concurrency control mechanisms in different distributed environments

UNIT–I:

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

System Models: Introduction, Architectural models, Fundamental models

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.

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.

UNIT–IV:

Distributed File Systems: Introduction, File service Architecture, CaseStudy:1: Sun Network File System, CaseStudy2: The Andrew File System.

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

UNIT-V:

Transactions and Concurrency Control: Introduction, Transactions, Nested Transactions, Locks, Optimistic concurrency control, Time stamp 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.

TEXTBOOKS:

1. Distributed Systems Concepts and Design, G Coulouris, J Dollimore and T Kindberg,Fourth Edition, Pearson Education.2009.

REFERENCEBOOKS

1. Distributed Systems, Principles and paradigms, AndrewS. Tanenbaum, Maarten Vanteen, 2nd Edition, PHI.
2. Distributed Systems, An Algorithm Approach, Sikumar Ghosh, Chapman & Hall/CRC, Taylor & Fransis Group, 2007.

COURSE OUTCOMES:

- Able to compare different types of distributed systems and different models.
- Able to analyze the algorithms of mutual exclusion, election & multi cast communication.
- Able to evaluate the different mechanisms for Interprocess communication and remote invocations.
- Able to design and develop new distributed applications.
- Able to apply transactions and concurrency control mechanisms in different distributed environments.

INDEX

UNIT– I

CHARACTERIZATION OF DISTRIBUTED SYSTEMS:INTRODUCTION

Distributed System–is a system of hardware or software components located at networked computers which communicate and coordinate their actions by passing messages.

- It is a collection of autonomous computers, connected through network and middleware.
- Users perceive the system as a single integrated computed facility.

Features of Centralized System:

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

Features of Distributed System:

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

Characteristics of Distributed System:

1. Concurrency of components (concurrent program execution)
2. Lack of a global clock(no single notion of time for all the systems)
3. Independent failures of components(failure of one component does not affect others)

Application of Distributed Systems:

- Tele communication network(telephone n/w, cellular n/w, computer n/w)
- Network Applications (WWW, online apps, n/w filesystems, banking systems)
- Real-time process control systems(aircraft control systems)
- Parallel computation(grid computing, cluster computing)

Examples of DS:

1. INTERNET: It is a vast interconnected collection of heterogeneous computer networks. It is a very large distributed system which enables users to use services like WWW, email, file transfer etc. Services are open-ended.

ISP: Internet service provider: companies that provide modem and other facilities to users and organizations which enable them to access services anywhere in the internet.

Intranet– sub networks operated by companies and other organizations.

Backbone links intranets. It is a link with high transmission capacity and employs satellite communication, fiber optics and other circuits.

2. INTRANET:

An Intranet is a portion of the Internet that is separately administered and has a boundary that can be configured to enforce local security policies. It is composed of several LAN's linked by backbone connections. An Intranet is connected to the Internet via a router, which allows the users inside the intranet to make use of services. It also allows the users in other intranets to access its services. Firewall protects anIntranet by preventing unauthorized messages leaving or entering using filtering method.

3. Mobile & Ubiquitous Computing:

Technological advances in device miniaturization and wireless networking have led to the integration of small and portable computing devices into distributed systems (laptops, phones, PDS's, wearable devices etc) Mobile computing is the performance of computing tasks while the user is on the move.

Ubiquitous computing is the harnessing of many small, cheap computational devicespresent in users environment. Devices become pervasive in everyday objects.

ResourceSharing:

- Resource sharing is the primary motivation of distributed computing
- Resources types
  - Hardware ,e.g. printer, scanner, camera
  - Data sources, e.g. file, database, webpage
  - Specific resources, e.g. search engine
- Service
  - Manages a collection of related resources and presents the functionalities to users and applications
- Server
  - A process on networked computer that accepts requests from processes on other computers to perform a service and responds appropriately
- Client
  - The requesting process
- Communication is through message passing or Remote invocation

Many distributed systems can be constructed in the form of interacting clients and servers. Ex: WWW, Email, Networked printers etc.

Web Browser– client which communicates with web server to request web pages.

World Wide Web:

WWW is an evolving system for publishing and accessing resources and services across the Internet using web browsers.

Web originated at European Centre for nuclear research, Switzerland in 1989.Documents exchanged contain hyperlinks.

Web is an open system. Its operation is based on communication standards and document standards. Initially web provided data resources but now includes services also. Web is based on three main standard technological components:

1. HTML: hypert ext markup language for specifying contents and layouts ofpages.
2. URL:uniform resource locator which identifies documents and other resourcesstored as part of web.
3. A client-server architecture with standard rules for interaction (HTTP) by which browsers and clients fetch documents and otherresources from web servers.

HTML: used to specify the text and images that make up the contents of a web page and tospecify how they are laid out and formatted for presentation to the user. Web page contains headings, paragraphs, tables and images. HTML is also used to specify links and resources associated with them. HTML text is stored as a file in the web server which is retrieved and interpreted by the webbrowser.HTML directives–tags - <P>

Ex:

<IMG SRC= "http">

< P >WELCOME

<AHREF="http--------"> </A>

< /P>

URL: Its purpose is to identify resource. It has two top-level components:

Scheme: Scheme-specific-identifier

(typeof URL ie ftp, http) (specific info to be retrieved ie www.abc.net/--.html) HTTPURL's are most widely used.

Form ->http://servername[:port] [/path name]Ex:http://www.google.com/search?q=MRCET

The simplest method of publishing a resource on the web is to place the corresponding file in a directory that the web server can access.

HTTP: defines the ways in which browsers and other types of client interact with web servers. Features: Request-reply interactions, content types, one resource per request, simple accesscontrol.

DynamicPages: A program that web servers run to generate content for their clients is referred to as a Common Gateway Interface (CGI) program.

XML –designedasawayofrepresentingdatainstandard,structured,application- specific forms. It is used to describe the capabilities of devices and to describe personal info held about users. The web of linked metadata resources is asemantic web.

CHALLENGES:

The challenges arising from the construction of distributed systems are:

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

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

Different programming languages use different representations for characters and data structures such as arrays and records. Heterogeneity can be handled in three ways:

Middleware •The term *middleware* applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. The Common Object Request Broker (CORBA), is an example.

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

The *virtual machine* approach provides a way of making code executable on a variety of host computers: the compiler for a particular language generates code for a virtual machine instead of particular hardware order code. For example, the Java compiler produces code for a Java virtual machine, which executes it by interpretation.

2. Openness

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

systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

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

3. Security

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

Challenge is not only to conceal the contents of a message but also to establish the identity of senderand receiver. Encryption techniques are used for this purpose. Two challenges not yet fully met are –denialofserviceattacksand securityofmobilecode.

4. Scalability

Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The design of scalable distributed systems presents the following challenges:

Controlling the cost of physical resources. *Controlling the performance lossPreventingsoftwareresourcesrunningoutAvoidingperformancebottlenecks*

5. Failure handling

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

Detecting failures: Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file.

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

Messages can be retransmitted when they fail to arrive.

File data can be written to a pair of disks sothat

if one is corrupted, the other will be there.

Tolerating failures: For example, when a web browser cannot contact a web server, it does not make the user wait forever while it keeps on trying– it informs the user about the problem, leaving them free to try again later.

Recovery from failures: Recovery involves the design of software so that the state of permanent data can be recovered or 'rolled back' after a server has crashed.

Redundancy: Services can be made to tolerate failures by the use of redundant components.

6. 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. Therefore services and applications generally allow multiple client requests to be processed concurrently. In this case processes should ensure correctness and consistency. Operations of objects should be synchronized using semaphores etc.

7. Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. The various forms of transparency are:

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

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

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

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

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

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

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

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

INTRODUCTION TO SYSTEM MODELS

System Models specify the common properties and design issues for a distributed system. They describe the relevant aspects of DS design.

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

Physical models are the most explicit way in which to describe a system; they capture the hardware composition of a system in terms of the computers (and other devices, such as mobile phones) and their inter connecting networks.

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

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

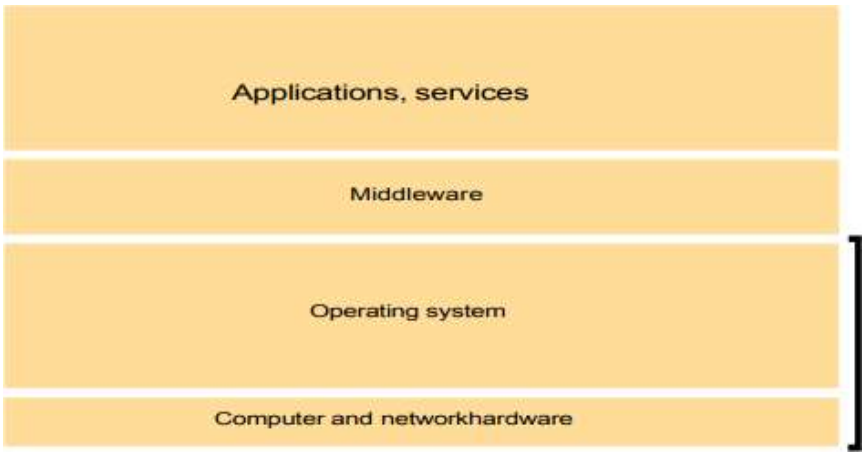
1. Architectural models

Architecture models define the way in which the components of systems interact with one another and how they are mapped onto the network. The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it.

Software layers

In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. In terms of distributed systems, this equates to a vertical organization of services into service layers. Given the complexity of distributed systems, it is often helpful to organize such services into layers. the important terms *platform* and *middleware*, which define as follows:

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



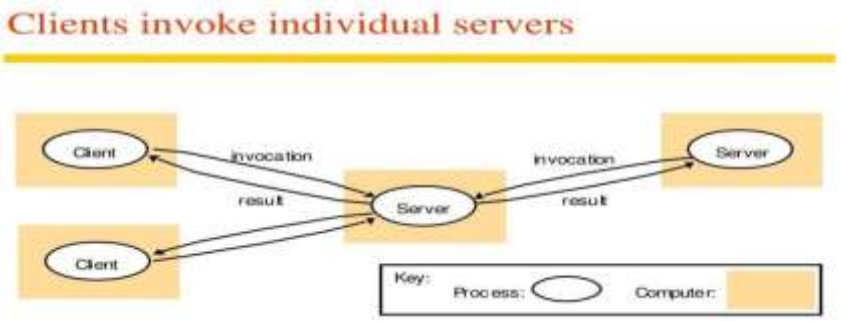
There are two main architectural models:

1. Client-Server Model
2. Peer-to-peer architecture

Client-server: This is the architecture that is most often cited when distributed systems arediscussed.Itishistoricallythemostimportantandremainsthemostwidelyemployed.Serverisa process which accepts requests from other processes and Client is a process requesting services from a server.

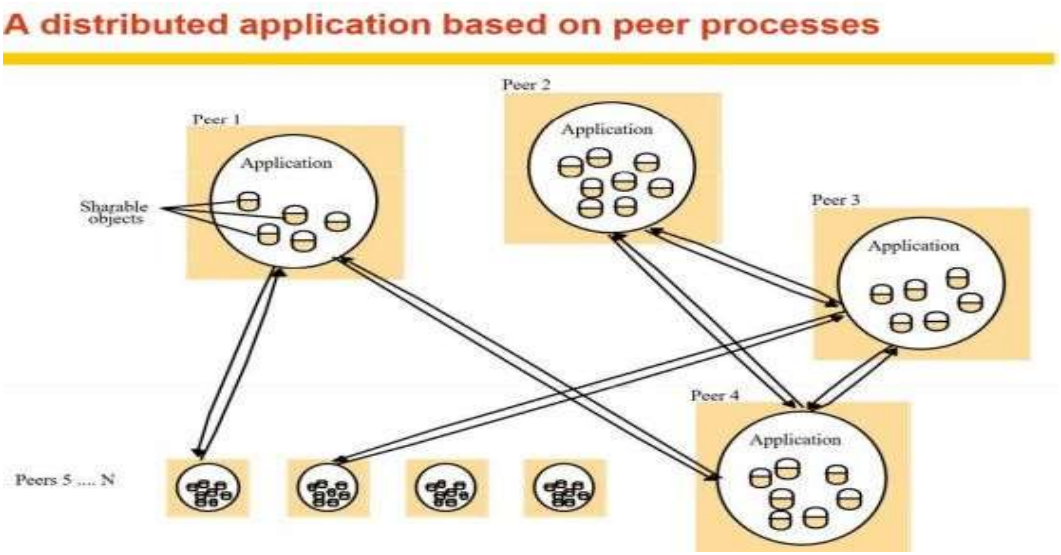
Servers may in turn be clients of other servers, as the figure indicates. For example, a web server is often a client of a local file server that manages the files in which the web pages are stored. ses.

Clients invoke individual servers



Another web-related example concerns *search engines*, which enable users to look up summaries of information available on web pages at sites throughout the Internet. Thus a search engine is both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers.

Peer-to-peer: In this architecture all of the processes involved in a task or activity play similar roles, interacting cooperatively as *peers* without any distinction between client andserver processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each other. While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly. Enables hundreds of computers to provide access to resources they share and manage. Each object is replicated in several computers. Ex: Napster app for sharing digital music files.



Several variations on the above models can be derived:

1. **Multiple-Servers Model**: In this services are provided by multiple servers. Services can be implemented as several server processes in separate host computers.

2. **Web Proxy Server:** It provides a shared cache of recently visited pages and web resources for the client machines at a site or across several sites. Purpose of proxy servers is to increase availability and performance of the service.

3. Mobile Code:

a) Client requests results in the downloading of applet code

b) Applets are a well- known and widely used example of mobile code. It is downloaded from a web server and executed locally resulting in good interactive response.

4. **Mobile Agent**:

A mobile agent is a running program that travels from one computer to another innetwork carrying out a task on someone's behalf

5. **Network Computers:**

Network computer

Remote file server

Client network

OS and Files

Network computer:

Download sits OS and application software needed from a Remotefileserver Applications are run locally but

the files are managed by the remote file server; low softwaremanagement and maintenance cost.

6. Thin Client:

A software layer that supports a window based interface on a computer that is local to the user while executing application programs on a computer server

Design requirements for distributed architectures:

1. Performance Issues

2. Quality of Service

3. Use of cache and replication

Performance Issues

Responsiveness

Delay, response time, slow down, stretch factor

Determined by load and performance of the server and the network, and by delays in all software components involved

Throughput

The rate at which computational work of the server or data transfer of the network is done

Load balancing/ load sharing

Enable applications and service processes to proceed concurrently and exploit the available resource

3. Fundamental Models

Models of systems share some fundamental properties. In particular, all of them are composed of processes that communicate with one another by sending messages over a computer network.

The purpose of such a model is:

- To make explicit all the relevant assumptions about the systems we are modelling.
- To make generalizations concerning what is possible or impossible, given thoseassumptions.

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

Interaction: Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes

Failure: The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs (including software faults) or in the network that connects them. Our model defines and classifies the faults.

Security: The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents. Our security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

There are three Fundamental Models:

a) Interaction model

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

- Multiple server processes may cooperate with one another to provide a service;
- A set of peer processes may cooperate with one another to achieve a commongoal;

Two significant factors affecting interacting processes in a distributed system:

- Communication performance is often a limiting characteristic.
- It is impossible to maintain a single global notion of time.

Performance of communication channels• Communication over a computer network has the following performance characteristics relating to latency, bandwidth and jitter:

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

- The time taken for the first of a string of bits transmitted through a network to reach Its destination. For example, the latency for the transmission of a message through a satellite link is the time for a radio signals to travel to the satellite and back.
- The *bandwidth* of a computer network is the total amount of information that can be transmitted over it in a given time. When a large number of communication channels are using the same network, they have to share the available bandwidth.
- *Jitter* is the variation in the time taken to deliver a series of messages. Jitter is relevant to multimedia data. For example, if consecutive samples of audio data are played with differing time intervals, the sound will be badly distorted.

Computer clocks and timing events • Each computer in a distributed system has its own internal clock, which can be used by local processes to obtain the value of the current time. Therefore two processes running on different computers can each associate timestamps with their events. However, even if the two processes read their clocks at the same time, their local clocks may supply different time values. This is because computer clocks drift from perfect time and, more importantly, their drift

rates differ from one another. The term *clock drift rate* refers to the rate at which a computer clock deviates from a perfect reference clock. Even if the clocks on all the computers in a distributed system are set to the same time initially, their clocks will eventually vary quite significantly unless corrections are applied.

Clock Drift Rate

Two variants of the interaction model •

Synchronous distributed systems: has a strong assumption of time. Asynchronous distributedsystem is one in which the following bound are defined:

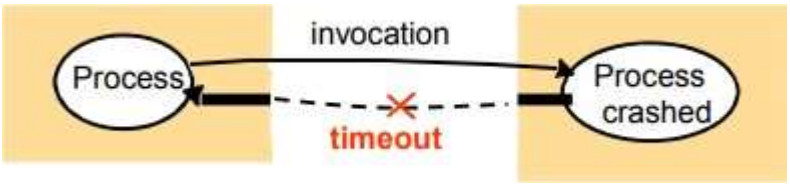
- The time to execute each step of a process has known lower and upper bounds.
- Each message transmitted over a channel is received within a known bounded time.
- Each process has a local clock whose drift rate from real time has a known bound. ***Asynchronous distributed systems***: makes no assumption of time. An asynchronous distributed system is one in which there are no bounds on:
- Process execution speeds–for example, one process step may take only a Picosecond and another a century; all that can be said is that each step may take an arbitrarily long time.
- Message transmission delays – for example, one message from process A to process B may be delivered in negligible time and another may take several years. In other words, a message may be received after an arbitrarily long time.
- Clock drift rates–again; the drif trat of a clock is arbitrary.

b) Failure model

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

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

Process omission failures: The chief omission failure of a process is to crash. When, say that process has crashed we mean that it has halted and will not execute any further steps of its program ever.



In an asynchronous distributed system

- A timeout means that a process is NOT responding; may have crashed or may beslow; or the message may not have arrived

In a synchronous distributed system

- A time out means that a process is crashed, so called fail-stop

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

Communication omission failures: Consider the communication primitives *send* and *receive*.

Process *p* performs a *send* by inserting the message *m* in its outgoing message buffer. The communication channel transports *m* to *q*'s incoming message buffer. Process *q* performs a *receive* by taking *m* from its incoming message buffer and deliver in get. The outgoing and

Incoming message buffer are typically provided by the operating system.

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

An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps.

Communication channels can suffer from arbitrary failures; for example, message contents maybe corrupted, nonexistent messages may be delivered or real messages may be delivered more than once.

Timing failures • Timing failures are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate. Timing

Failures are listed in the following figure. Any one of these failures may result in responses being unavailable to clients within a specified time interval.

Real-time operating systems are designed with a view to providing timing guarantees, but they are more complex to design and may require redundant hardware.

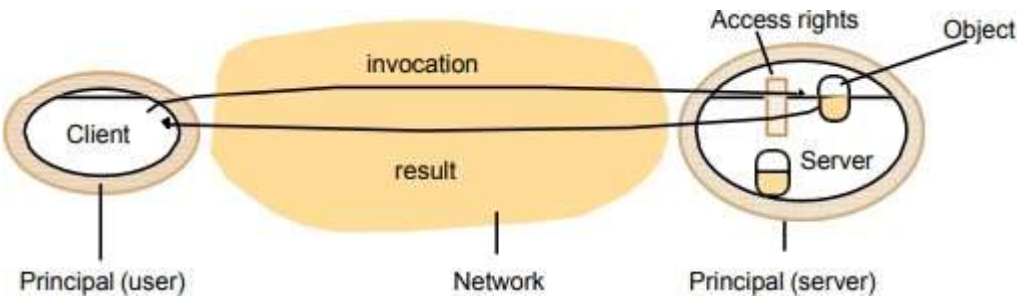
c) Security model

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

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

Protecting objects:

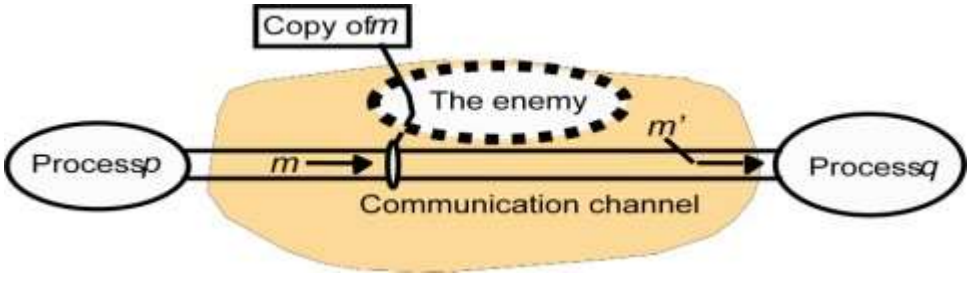
Objects are intended to be used in different ways by different users. For example, some objects may hold a user's private data, such as their mailbox, and other objects may hold shared data such as web pages. To support this, *access rights specify who is allowed to perform*the operations of an object–for example, who is allowed to read or to write its state.



The enemy • To model security threats, we postulate an enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in the following figure. The attack may

Come from a computer that is legitimately connected to the network or from one that is connected

in an unauthorized manner. The threats from a potential enemy include *threats to processes* and *threats to communication channels.*



Defeating security threats

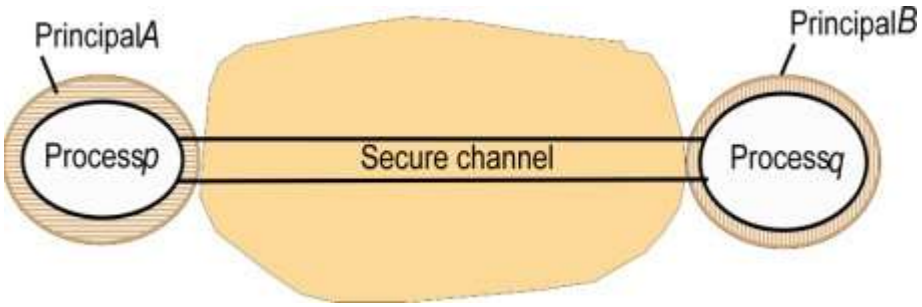
Cryptography is the science of keeping messages secure, and *encryption* is the process of scrambling a message in such away as to hide its contents. Modern cryptography is based on

Encryption algorithms that use secret keys–large numbers that are difficult to guess– to transform data in a manner that can only be reversed with knowledge of the corresponding decryption key.

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

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

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing.
- A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it.
- Each message includes a physical or logical time stamp to prevent messagesfrom being replayed or reordered.



UNIT II

Time and Global States

There are two formal models of distributed systems: synchronous and asynchronous.

Synchronous distributed systems have the following characteristics:

- the time to execute each step of a process has known lower and upper bounds;
- each message transmitted over a channel is received within a known bounded time;
- Each process has a local clock whose drift rate from real time has a known bound.

Asynchronous distributed systems, in contrast, guarantee no bounds on process execution speeds, message transmission delays, or clock drift rates. Most distributed systems we discuss, including the Internet, are asynchronous systems.

Generally, timing is a challenging an important issue in building distributed systems. Consider a couple of examples:

- Suppose we want to build a distributed system to track the battery usage of a bunch of laptop computers and we'd like to record the percentage of the battery each has remaining at exactly 2pm.
- Suppose we want to build a distributed, real time auction and we want to know which of two bidders submitted their bid first.
- Suppose we want to debug a distributed system and we want to know whether variable x_1 in process p_1 ever differs by more than 50 from variable x_2 in processp_2.

In the first example, we would really like to synchronize the clocks of all participating computers and take a measurement of absolute time. In the second and third examples, knowing the absolute time is not as crucial as knowing the order in which events occurred.

Clock Synchronization

Every computer has a physical clock that counts oscillations of a crystal. This hardware clockis used by the computer's software clock to track the current time. However, the hardware clock is subject to *drift* -- the clock's frequency varies and the time becomes inaccurate. As a result, any two clocks are likely to be slightly different at any given time. The difference between two clocks is called their *skew*.

There are several methods for synchronizing physical clocks. *External synchronization* means that all computers in the system are synchronized with an external source of time (e.g., a UTC signal). *Internal synchronization* means that all computers in the system are synchronized with one another, but the time is not necessarily accurate with respect to UTC.

In a synchronous system, synchronization is straightforward since upper and lower bounds on the transmission time for a message are known. One process sends a message to another process indicating its current time, *t*. The second process sets its clock to $t + (max+min)/2$ where max and min are the upper and lower bounds for the message transmission time respectively. This guarantees that the skew is at most $(max-min)/2$.

Cristian's method for synchronization in asynchronous systems is similar, but does not rely on a predetermined max and min transmission time. Instead, a process p_1 requests the current time from another process p_2 and measures the RTT (T_{round}) of the request/reply.
Whenp_1 receives the time *t* from p_2 it sets its time to $t + T_{round}/2$.

The Berkeley algorithm, developed for collections of computers running Berkeley UNIX, is an internal synchronization mechanism that works by electing a master to coordinate the synchronization. The master polls the other computers (called slaves) for their times, computes an average, and tells each computer by how much it should adjust its clock.

The Network Time Protocol (NTP) is yet another method for synchronizing clocks that uses a hierarchical architecture where he top level of the hierarchy (stratum 1) are servers connected to a UTC time source.

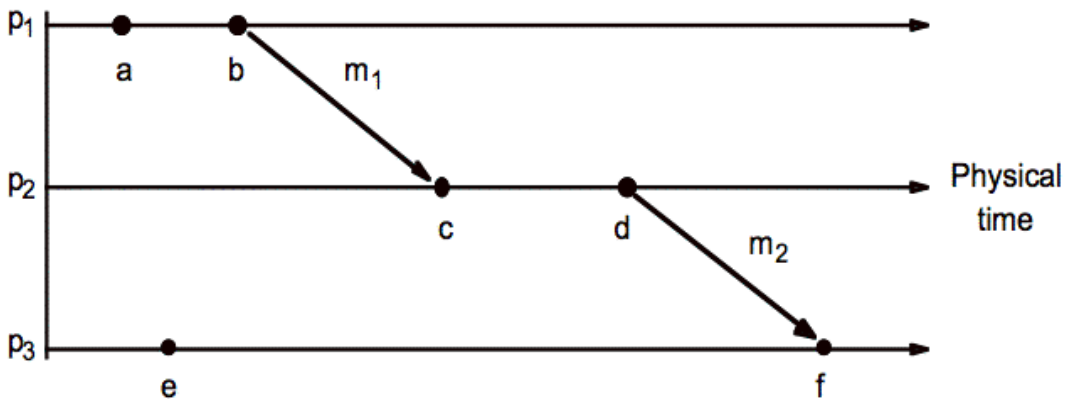
Logical Time

Physical time cannot be perfectly synchronized. Logical time provides a mechanism to definethe *causal order* in which events occur at different processes. The ordering is based on the following:

- Two events occurring at the same process happen in the order in which they are observed by the process.
- If a message is sent from one process to another, the sending of the message happenedbefore the receiving of the message.
- If e occurred before e' and e' occurred before e" then e occurred before e".

"Lamport called the partial ordering obtained by generalizing these two relationshipsthe *happened-before* relation." (→)

Figure 11.5
Events occurring at three processes



In the figure, a → b and c → d . Also, b → c and d → f , which means that a → f . However,we cannot say that a → e or vice versa; we say that they are *concurrent*

(*a* || *e*).

A Lamport logical clock is a monotonically increasing software counter, whose value needbear no particular relationship to any physical clock. Each process pi keeps its own logicalclock, Li, which it uses to apply so-called Lamport timestamps to events.

Lamport clocks work as follows:

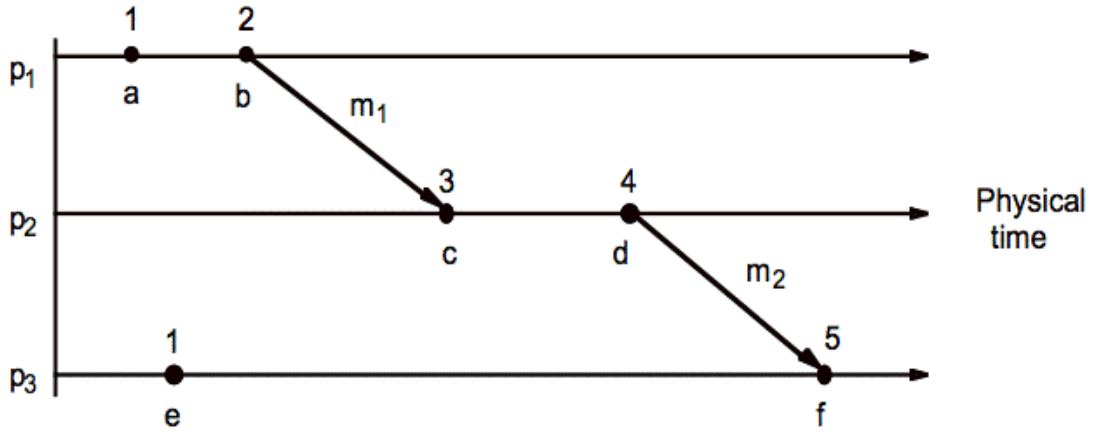
LC1: Li is incremented before each event is issued at pi.LC2:

When a process pi sends a message m, it piggybacks on m the value t = Li.

On receiving (m, t), a process pj computes Lj: = max (Lj, t) and then applies LC1 before time stamping the event receive (m).

An example is shown below:

Figure 11.6
Lamport timestamps for the events shown in Figure 11.5



If e → e ' then L (e) < L (e'), but the converse is not true. Vector clocks address this problem."A vector clock for a system of N processes is an array of N integers." Vector clocks are updated as follows:

VC1: Initially, VI[j] = 0 for I, j = 1, 2, N

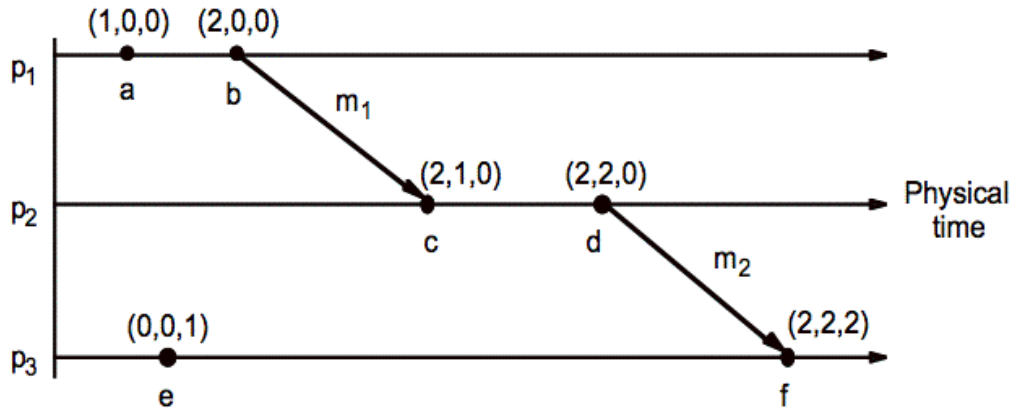
VC2: Just before pi timestamps an event, it sets Vi[i]:=Vi[i]+1.VC3:

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 1, 2, ...N. Taking the component wise maximum of two vector timestamps in this way is known asa merge operation.

An example is shown below:

Figure 11.7
Vector timestamps for the events shown in Figure 11.5



Vector timestamps are compared as follows:

V=V' iff V[j] = V'[j] for j = 1, 2, ..., N

V <= V' iff V[j] <=V'[j] for j = 1, 2, ..., NV <

V' iff V <= V' and V != V'

If e → e ' then V(e) < V(e') and if V(e) < V(e') then e → e ' .

Global States

It is often desirable to determine whether a particular property is true of a distributed system as it executes. We'd like to use logical time to construct a global view of the system state anddetermine whether a particular property is true. A few examples are as follows:

- Distributed garbage collection: Are there references to an object anywhere in the system? References may exist at the local process, at another process, or in the communication channel.
- Distributed deadlock detection: Is there a cycle in the graph of the "waits for" relationship between processes?
- Distributed termination detection: Has a distributed algorithm terminated?

- Distributed debugging: Example: given two processes p_1 and p_2 with variables x_1 and x_2 respectively, can we determine whether the condition $|x_1-x_2| > \delta$ is evertrue.

In general, this problem is referred to as *Global Predicate Evaluation*. "A global state predicate is a function that maps from the set of global state of processes in the system ρ to {True, False}."

- Safety - a predicate always evaluates to false. A given undesirable property(e.g., deadlock) never occurs.
- Liveness - a predicate eventually evaluates to true. A given desirable property(e.g., termination) eventually occurs.

Cuts

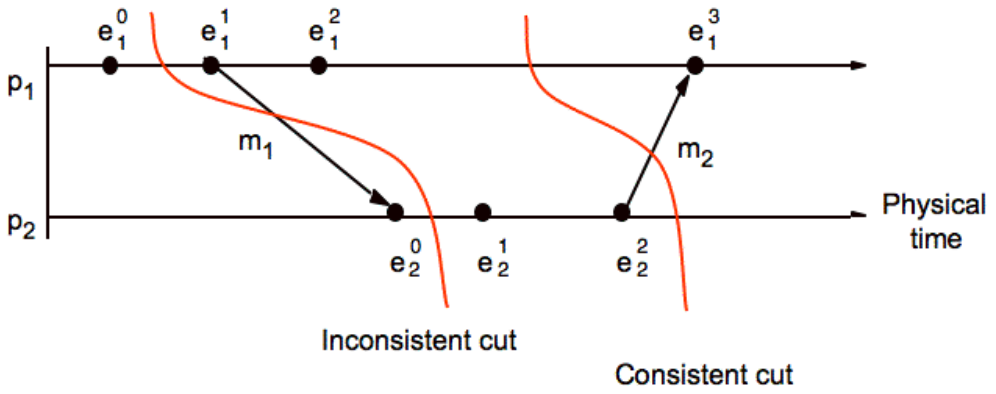
Because physical time cannot be perfectly synchronized in a distributed system it is not possible to gather the global state of the system at a particular time. Cuts provide the abilityto "assemble a meaningful global state from local states recorded at different times".

Definitions:

- ρ is a system of N processes p_i (i = 1, 2, ..., N)
- history(p_i) = h_i = < e i 0 , e i 1 ,...>
- h i k =< e i 0 , e i 1 ,..., e i k > - a finite prefix of the process's history
- s i k is the state of the process p_i immediately before the kth event occurs
- All processes record sending and receiving of messages. If a process p_i recordsthe sending of message m to process p_j and p_j has not recorded receipt of the message, then m is part of the state of the channel between p_i and p_j.
- A *global history* of ρ is the union of the individual process histories: H =h_0 U h_1 U h_2 U...Uh_{N-1}
- A *global state* can be formed by taking the set of states of the individual processes: S = (s_1, s_2, ..., s_N)
- A *cut* of the system's execution is a subset of its global history that is a union of prefixes of process histories (see figure below).
- The *frontier* of the cut is the last state in each process.

- A cut is *consistent* if, for all events *e* and *e'*:
  - (e ∈ C and e ' → e) ⇒ e ' ∈ C
- A *consistent global state* is one that corresponds to a consistent cut.

Figure 11.9
Cuts



Distributed Debugging

To further examine how you might produce consistent cuts, we'll use the distributed debugging example. Recall that we have several processes, each with a variable x_i. "The safety condition required in this example is $|x_i-x_j| <= \delta$ (i, j = 1, 2, ..., N)."

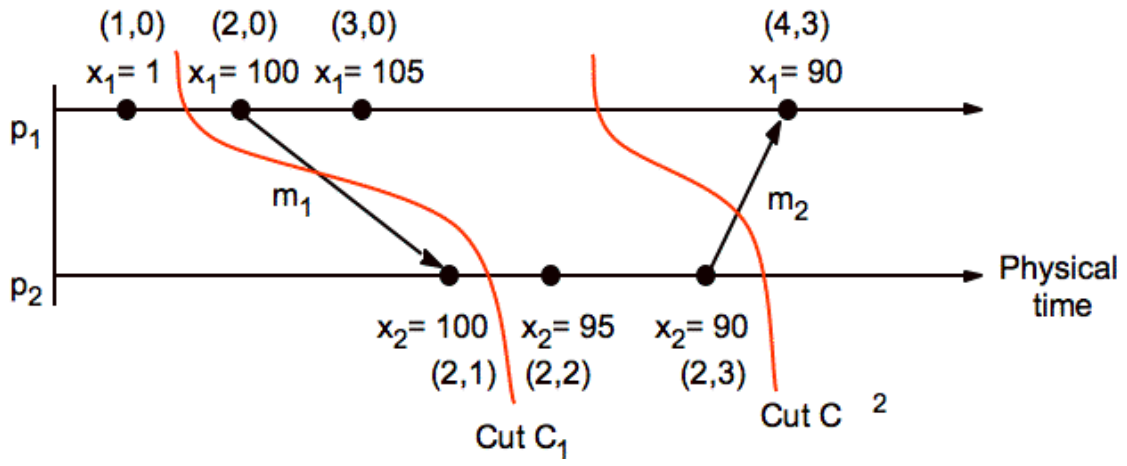
The algorithm we'll discuss is a centralized algorithm that determines post hoc whether the safety condition was ever violated. The processes in the system, p_1, p_2, ..., p_N, send their statesto a passive monitoring process, p_0. p_0 is not part of the system. Based on the states collected, p_0 can evaluate the safety condition.

Collecting the state: The processes send their initial state to a monitoring process and send updates whenever relevant state changes, in this case the variable x_i. In addition, the processes need only send the value of x_i and a vector timestamp. The monitoring process maintains an ordered queue (by the vector timestamps) for each process where it stores the state messages. It can then create consistent global states which it uses to evaluate the safety condition.

Let S = (s1, s2, ..., SN) be a global state drawn from the state messages that the monitor process has received. Let V(si) be the vector timestamp of the state si received from pi. Thenit can be shown that S is a consistent global state if and only if:

V(si)[i] >= V(sj)[i] for i, j = 1, 2, ..., N

Figure 11.14
Vector timestamps and variable values for the execution of Figure 11.9



Coordination and Agreement

Overview

We start by addressing the question of why process need to coordinate their actions and agreeon values in various scenarios.

1. Consider a mission critical application that requires several computers to communicate and decide whether to proceed with or abort a mission. Clearly, allmust come to agreement about the fate of the mission.
2. Consider the Berkeley algorithm for time synchronization. One of the participate computers serves as the coordinator. Suppose that coordinator fails. The remaining computers must elect a new coordinator.
3. Broadcast networks like Ethernet and wireless must agree on which nodes cansend at any given time. If they do not agree, the result is a collision and no message is transmitted successfully.

4. Like other broadcast networks, sensor networks face the challenging of agreeingwhich nodes will send at any given time. In addition, many sensor network algorithms require that nodes elect coordinators that take on a server-like responsibility. Choosing these nodes is particularly challenging in sensor networks because of the battery constraints of the nodes.

5. Many applications, such as banking, require that nodes coordinate their access of a shared resource. For example, a bank balance should only be accessed andupdated by one computer at a time.

Failure Assumptions and Detection

Coordination in a synchronous system with no failures is comparatively easy. We'll look atsome algorithms targeted toward this environment. However, if a system is asynchronous, meaning that messages may be delayed an indefinite amount of time, or failures may occur,then coordination and agreement become much more challenging.

A *correct process* "is one that exhibits no failures at any point in the execution under consideration." If a process fails, it can fail in one of two ways: a crash failure or a byzantinefailure. A crash failure implies that a node stops working and does not respond to any messages. A byzantine failure implies that a node exhibits arbitrary behavior. For example, itmay continue to function but send incorrect values.

Failure Detection

One possible algorithm for detecting failures is as follows:

- Every *t* seconds, each process sends an "I am alive" message to all other processes.
- Process *p* knows that process *q* is either *unsuspected*, *suspected*, or *failed*.
- If *p* sees *q*'s message, it sets *q*'s status to unsuspected.

This seems ok if there are no failures. What happens if a failure occurs? In this case, *q* will not send a message. In a synchronous system, *p* waits for *d* seconds (where *d* is the maximumdelay in message delivery) and if it does not hear from *q* then it knows that *q* has failed. In anasynchronous system, *q* can be suspected of failure after a timeout, but there is no guarantee that a failure has occurred.

Mutual Exclusion

The first set of coordination algorithms we'll consider deal with mutual exclusion. How can we ensure that two (or more) processes do not access a shared resource simultaneously? This problem comes up in the OS domain and is addressed by negotiating with shared objects (locks). In a distributed system, nodes must negotiate via message passing.

Each of the following algorithms attempts to ensure the following:

- Safety: At most one process may execute in the critical section (CS) at a time.
- Liveness: Requests to enter and exit the critical section eventually succeed.
- Causal ordering: If one request to enter the CS happened-before another, thenentry to the CS is granted in that order.

Central Server

The first algorithm uses a central server to manage access to the shared resource. To enter acritical section, a process sends a request to the server. The server behaves as follows:

- If no one is in a critical section, the server returns a token. When the processexits the critical section, the token is returned to the server.
- If someone already has the token, the request is queued.

Requests are serviced in FIFO order.

If no failures occur, this algorithm ensures safety and liveness. However, ordering is not preserved (**why?**). The central server is also a bottleneck and a single point of failure.

Token Ring

The token ring algorithm arranges processes in a logical ring. A token is passed clockwise around the ring. When a process receives the token it can enter its critical section. If it does not need to enter a critical section, it immediately passes the token to the next process.

This algorithm also achieves safety and liveness, but not ordering, in the case when no failures occur. However, a significant amount of bandwidth is used because the token is passed continuously even when no process needs to enter a CS.

Multicast and Logical Clocks

Each process has a unique identifier and maintains a logical clock. A process can be in one of three states: released, waiting, or held. When a process wants to enter a CS it does the following:

- sets its state to waiting
- sends a message to all other processes containing its ID and timestamp
- once all other processes respond, it can enter the CS

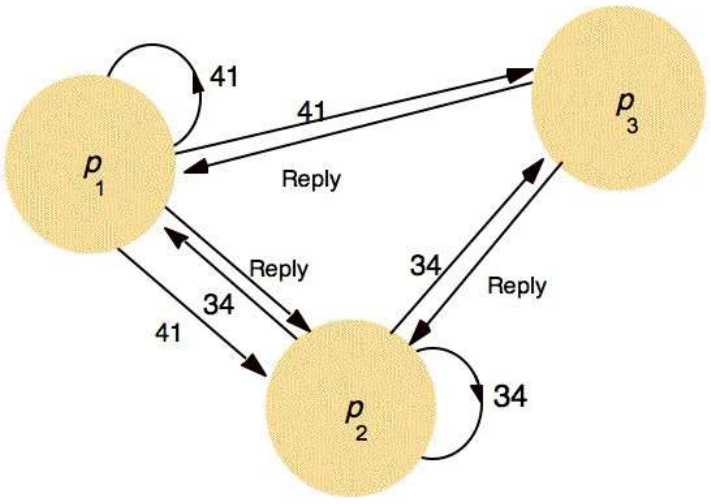
When a message is received from another process, it does the following:

- if the receiver process state is held, the message is queued
- if the receiver process state is waiting and the timestamp of the message is after the local timestamp, the message is queued (if the timestamps are the same, the process ID is used to order messages)
- else - reply immediately

When a process exits a CS, it does the following:

- sets its state to released
- replies to queued requests

Figure 12.5
Multicast synchronization



This algorithm provides safety, liveness, and ordering. However, it cannot deal with failureand has problems of scale.

None of the algorithms discussed are appropriate for a system in which failures may occur. Inorder to handle this situation, we would need to first detect that a failure has occurred and then reorganize the processes (e.g., form a new token ring) and reinitialize appropriate state (e.g., create a new token).

Election

An election algorithm determines which process will play the role of coordinator or server. All processes need to agree on the selected process. Any process can start an election, for example if it notices that the previous coordinator has failed. The requirements of an electionalgorithm are as follows:

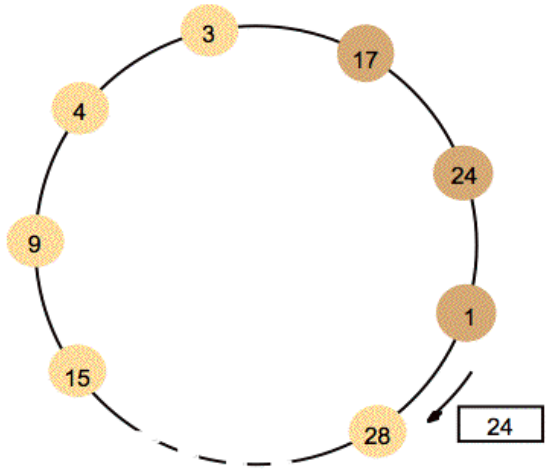
- Safety: Only one process is chosen -- the one with the largest identifying value.The value could be load, uptime, a random number, etc.
- Liveness: All process eventually chooses a winner or crash.

Ring-based

Processes are arranged in a logical ring. A process starts an election by placing its ID and value in a message and sending the message to its neighbor. When a message is received, aprocess does the following:

- If the value is greater that its own, it saves the ID and forwards the value to its neighbor.
- Else if its own value is greater and then it has not yet participated in the election,it replaces the ID with its own, the value with its own, and forwards the message.
- Else if it has already participated it discards the message.
- If a process receives its own ID and value, it knows it has been elected. It thensends an elected message to its neighbor.
- When an elected message is received, it is forwarded to the next neighbor.

Figure 12.7
A ring-based election in progress



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

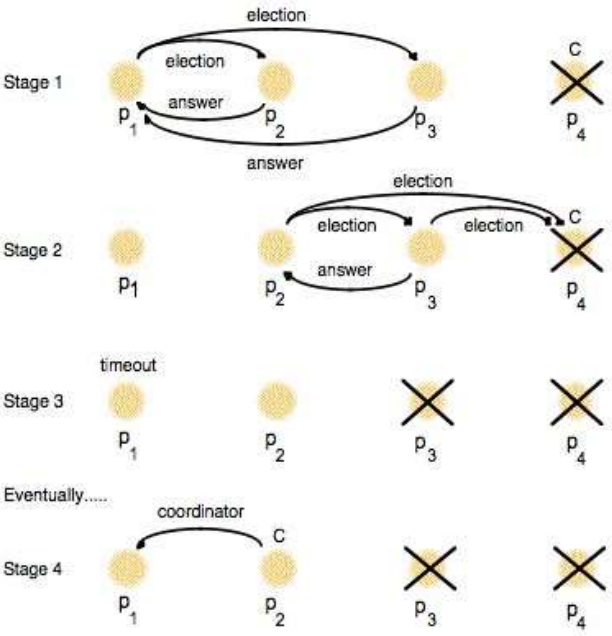
Safety is guaranteed - only one value can be largest and make it all the way through the ring.Liveness is guaranteed if there are no failures. However, the algorithm does not work if thereare failures.

Bully

The bully algorithm can deal with crash failures, but not communication failures. When a process notices that the coordinator has failed, it sends an election message to all higher- numbered processes. If no one replies, it declares itself the coordinator and sends a new coordinator message to all processes. If someone replies, it does nothing else. When a processreceives an election message from a lower-numbered process it returns a reply and starts an election. This algorithm guarantees safety and liveness and can deal with crash failures.

Figure 12.8
The bully algorithm

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



Consensus

All of the previous algorithms are examples of the consensus problem: how can we get all processes to agree on a state? Here, we look at when the consensus problem is solvable.

The system model considers a collection of processes p_i (i = 1, 2, ..., N). Communication isreliable, but processes may fail. Failures may be crash failures or byzantine failures.

The goals of consensus are as follows:

- Termination: Every correct process eventually decides on a value.
- Agreement: All processes agree on a value.
- Integrity: If all correct processes propose the same value, that value is the one selected.

We consider the Byzantine Generals problem. A set of generals must agree on whether to attack or retreat. Commanders can be treacherous (faulty). This is similar to consensus, but differs in that a single process proposes a value that the others must agree on. The requirements are:

- Termination: All correct processes eventually decide on a value.
- Agreement: All correct processes agree on a value.
- Integrity: If the commander is correct, all correct processes agree on what the commander proposed.

If communication is unreliable, consensus is impossible. Remember the blue army discussion from the second lecture period. With reliable communication, we can solve consensus in a synchronous system with crash failures.

We can solve Byzantine Generals in a synchronous system as long as less than 1/3 of the processes fail. The commander sends the command to all of the generals and each general sends the command to all other generals. If each correct process chooses the majority of all commands, the requirements are met. Note that the requirements do not specify that the processes must detect that the commander is fault.

It is impossible to guarantee consensus in an asynchronous system, even in thepresence of 1 crash failure. That means that we can design systems that reach consensus mostof the time, but cannot guarantee that they will reach consensus every time. Techniques for reaching consensus in an asynchronous system include the following:

- Masking faults - Hide failures by using persistent storage to store state and restarting processes when they crash.
- Failure detectors - Treat an unresponsive process (that may still be alive) asfailed.
- Randomization - Use randomized behavior to confuse byzantine processes.

UNIT-III

INTER PROCESS COMMUNICATION:

Introduction:

Inter process Communication is a process of exchanging the data between two or more independent process in a distributed environment is called as Inter process communication. Inter process communication on the internet provides both Datagram and stream communication.

Characteristics of Inter Process Communication:

Synchronous System Calls:
In the synchronous system calls both sender and receiver use blocking system calls to transmit the data which means the sender will wait until the acknowledgment is received from the receiver and receiver waits until the message arrives.

Asynchronous System Calls:
In the asynchronous system calls, both sender and receiver use non-blocking system calls to transmit the data which means the sender doesn't wait from the receiver acknowledgment.

Message Destination:
A local port is a message destination within a computer, specified as an integer. Aport has exactly one receiver but many senders. Processes may use multiple ports from which to receive messages. Any process that knows the number of a port can send the message to it.

Reliability:
It is defined as validity and integrity.

Integrity:
Messages must arrive without corruption and duplication to the destination.

Validity:
Point to point message services are defined as reliable, If the messages are guaranteed to be delivered without being lost is called validity.

Ordering:
It is the process of delivering messages to the receiver in a particular order. Some applications require messages to be delivered in the sender order i.e the order in which they were transmitted by the sender.
The information consists of a sequence of bytes in messages that are moving between components in a distributed system. So, conversion is required from the data structure to a sequence of bytes before the transmission of data. On the arrival of the message, data should also be able to be converted back into its original data structure.

Different types of data are handled in computers, and these types are not the same in every position where data must be transmitted. Individual primitive data items can have a variety of data values, and not all computers store primitive values like integers in the same order. Different architectures also represent floating-point numbers differently. Integers are ordered in two ways, big-endian order, in which the Most Significant Byte (MSB) is placed first, and little-endian order, in which the Most Significant Byte (MSB) is placed last or the Least Significant Byte (LSB) is placed first. Furthermore, one more issue is the set of codes used to represent characters. Most applications on UNIX systems use ASCII character coding, which uses one byte per character, whereas the Unicode standard uses two bytes per character and allows for the representation of texts in many different languages.

Marshalling: Marshalling is the process of transferring and formatting a collection of data structures into an external data representation type appropriate for transmission in a message.
Unmarshalling: The converse of this process is unmarshalling, which involves reformatting the transferred data upon arrival to recreate the original data structures at the destination.

Approaches:

There are three ways to successfully communicate between various sorts of data between computers.

1. Common Object Request Broker Architecture (CORBA):

CORBA is a specification defined by the Object Management Group (OMG) that is currently the most widely used middleware in most distributed systems. It allows systems with diverse architectures, operating systems, programming languages, and computer hardware to work together. It allows software applications and their objects to communicate with one another. It is a standard for creating and using distributed objects.
Data Representation in CORBA:
Common Data Representation (CDR) is used to describe structured or primitive data types that are supplied as arguments or results during remote invocations on CORBA distributed objects. It allows clients and servers' built-in computer languages to communicate with one another. To exemplify, it converts little-endian to big-endian.
There are 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), and any (which can represent any basic or constructed type), as well as a variety of composite types.
CORBA CDR Constructed Types:
Let's have a look at Types with their representation:
sequence: It refers to length (unsigned long) to be followed by elements in order
string: It refers to length (unsigned long) followed by characters in order (can also have wide characters)
array: The elements of the array follow order and length is fixed so not specified.
struct: in the order of declaration of components
enumerated: It is unsigned long and here, the values are specified by the order declared.
union: type tag followed by the selected member

CORBA CDR message

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

index in sequence of bytes	◄—4 bytes —►	*notes on representatio*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1984	*unsigned long*

Marshalling CORBA:
From the specification of the categories of data items to be transmitted in a message, Marshalling CORBA operations can be produced automatically. CORBA IDL describes the types of data structures and fundamental data items and provides a language/notation for specifying the types of arguments and results of RMI methods.

2. Java's Object Serialization:

Java Remote Method Invocation (RMI) allows you to pass both objects and primitive data values as arguments and method calls. In Java, the term serialization refers to the activity of putting an object (an instance of a class) or a set of related objects into a serial format suitable for saving to disk or sending in a message.

Java provides a mechanism called object serialization. This allows an object to be represented as a sequence of bytes containing information about the object's data and the type of object and the type of data stored in the object. After the serialized object is written to the file, it can be read from the file and deserialized. You can recreate an object in memory with type information and bytes that represent the object and its data.

Moreover, objects can be serialized on one platform and deserialized on completely different platforms as the whole process is JVM independent.

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

Serialized values				Explanation
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name	java.lang.String place	*number, type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

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

3. Extensible Markup Language (XML):

Clients communicate with web services using XML, which is also used to define the interfaces and other aspects of web services. However, XML is utilized in a variety of different applications, including archiving and retrieval systems; while an XML archive is larger than a binary archive, it has the advantage of being readable on any machine. Other XML applications include the design of user interfaces and the encoding of operating system configuration files.

In contrast to HTML, which employs a fixed set of tags, XML is extensible in the sense that users can construct their tags. If an XML document is meant to be utilized by several applications, the tag names must be unique.

Clients, for example, typically interface with web servers via SOAP messages. SOAP is an XML standard with tags that web services and their customers can utilize. Because it is expected that the client and server sharing a message have prior knowledge of the order and types of information it contains, some external data representations (such as CORBA CDR) do not need to be self-describing. On the other hand, XML was

designed to be utilized by a variety of applications for a variety of reasons. This has been made possible by the inclusion of tags and the usage of namespaces to specify the meaning of the tags. Furthermore, the usage of tags allows applications to pick only the portions of a document that they need to process.

Usage:

Marshalling is used to create various remote procedure call (RPC) protocols, where separate processes and threads often have distinct data formats, necessitating the need for marshalling between them.

To transmit data across COM object boundaries, the Microsoft Component Object Model (COM) interface pointers employ marshalling. When a common-language-runtime-based type has to connect with other unmanaged types via marshalling, the same thing happens in the.NET framework. DCOM stands for Distributed Component Object Model.

Scripts and applications based on the Cross-Platform Component Object Model (XPCOM) technology are two further examples where marshalling is crucial. The Mozilla Application Framework makes heavy use of XPCOM, which makes considerable use of marshalling.

So, XML (Extensible Markup Language) is a text-based format for expressing structured data. It was designed to represent data sent in messages exchanged by clients and servers in web services

The primitive data types are marshalled into a binary form in the first two ways- CORBA and Java's object serialization. The primitive data types are expressed textually in the third technique (XML). A data value's textual representation will typically be longer than its binary representation. The HTTP protocol is another example of the textual approach.

On the other hand, type information is included in both Java serialization and XML, but in distinct ways. Although Java serializes all of the essential type information, XML documents can refer to namespaces, which are externally specified groups of names (with types).

XML definition of the *Person* structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

An XML schema for the *Person* structure

```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
    <xsd:element name= "person" type ="personType" />
        <xsd:complexType name="personType">
            <xsd:sequence>
                <xsd:element name = "name" type="xs:string"/>
                <xsd:element name = "place" type="xs:string"/>
                <xsd:element name = "year" type="xs:positiveInteger"/>
            </xsd:sequence>
            <xsd:attribute name= "id" type = "xs:positiveInteger"/>
        </xsd:complexType>
</xsd:schema>
```

CASE STUDY : IPC IN UNIX

Pipes are a simple synchronized way of passing information between two processes. A pipe can be viewed as a special file that can store only a limited amount of data and uses a FIFO access scheme to retrieve data. In a logical view of a pipe, data is written to one end and read from the other. The processes on the ends of a pipe have no easy way to identify what process is on the other end of the pipe. The system provides synchronization between the reading and writing process. It also solves the producer/consumer problem: writing to a full pipe automatically blocks, as does reading from an empty pipe. The system also assures that there are processes on both ends of the pipe at all time. The programmer is still responsible, however, for preventing deadlock between processes.

Pipes come in two varieties: · Unnamed. Unnamed pipes can only be used by related processes (i.e. a process and one of its child processes, or two of its children). Unnamed pipes cease to exist after the processes are done using them. · Named. Named pipes exist as directory entries, complete with permissions. This means that they are persistent and that unrelated processes can use them. 2.1 UNIX Most UNIX systems limit pipes to 5120K (typically ten 512K chunks). The unbuffered system call write() is used to add data to a pipe. Write() takes a file descriptor (which can refer to the pipe), a buffer containing the data to be written, and the size of the buffer as parameters. The system assures that no interleaving will occur between writes, even if the pipeline fills temporarily. To get data from a pipe, the read() system call is used. Read() functions on pipes much the same as it functions on files.

However, seeking is not supported and it will block until there is data to be read. The pipe() system call is used to create unnamed pipes in UNIX. This call returns two pipes. Both support bidirectional communication (two pipes are returned for historical reasons: at one time pipes were unidirectional so two pipes were needed for bidirectional communication). In a full duplex environment (i.e. one that supports bidirectional pipes) each process reads from one pipe and writes to the other; in a half-duplex (i.e. unidirectional) setting, the first file descriptor is always used for reading and the second for writing. Pipes are commonly used on the UNIX command line to send the output of one process to another process as input. When a pipe is used both processes run concurrently and there is no guarantee as to the sequence in which each process will be allowed to run. However, since the system manages the producer/consumer issue, both proceed per usual, and the system provides automatic blocking as required.

Using unnamed pipes in a UNIX environment normally involves several steps: · Create the pipe(s) needed · Generate the child processes · Close/duplicate the file descriptors to associate the ends of the pipe · Close the unused end(s) of the pipe(s) · Perform the communication · Close the remaining file descriptors · Wait for the child process to terminate To simplify this process, UNIX provides two system calls that handle this procedure. The call popen() returns a pointer to a file after accepting a shell command to be executed as input. Also given as input is a type flag that determines how the returned file descriptor will be used. The popen() call automatically generates a child process, which exec()s a shell and runs the indicated command. Depending on the flag passed in, this command could have either read or write access to the file. The pclose() call is used to close the data stream opened with popen(). It takes the file descriptor returned by popen() as its only parameter.

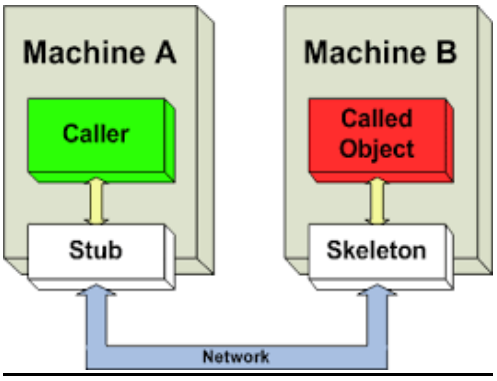
Named pipes can be created on the UNIX command line using mknod, but it is more interesting to look at how they can be used programmatically. The mknod() system call, usable only by the superuser, takes a path, access permissions, and a device (typically unused) as parameters and creates a pipe referred to by the user-specified path. Often, mkfifo() will be provided as an additional call that can be used by all users but is only capable of making FIFO pipes.

Distributed Objects & Remote Invocation :

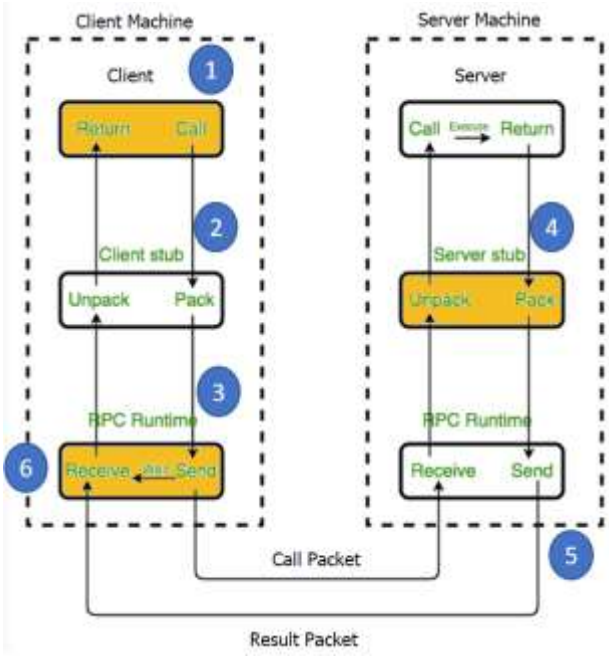
A distributed object is an object that can be accessed remotely. This means that a distributed object can be used like a regular object, but from anywhere on the network.

Communication between Distributed Objects:

- Stub and skeleton objects works as communication objects in distributed system.
- The stub acts as a gateway for client side objects and all outgoing requests from client side to the server-side objects.
- The skeleton acts as the gateway for server side objects & for all incoming clients requests.
- The skeleton wraps or binds server/called object functionality & exposes it to the clients; moreover by adding the network logic ensures the reliable communication channel between clients & server.



RPC (Remote Procedure Call)



Step 1) The client, the client stub, and one instance of RPC run time execute on the client machine.

Step 2) A client starts a client stub process by passing parameters in the usual way. The client stub stores within the client's own address space. It also asks the local RPC Runtime to send back to the server stub.

Step 3) In this stage, RPC accessed by the user by making regular Local Procedural Cal. RPC Runtime manages the transmission of messages between the network across client and server. It also performs the job of retransmission, acknowledgment, routing, and encryption.

Step 4) After completing the server procedure, it returns to the server stub, which packs (marshalls) the return values into a message. The server stub then sends a message back to the transport layer.

Step 5) In this step, the transport layer sends back the result message to the client transport layer, which returns back a message to the client stub.

Step 6) In this stage, the client stub demarshalls (unpack) the return parameters, in the resulting packet, and the execution process returns to the caller.

Advantages of RPC:

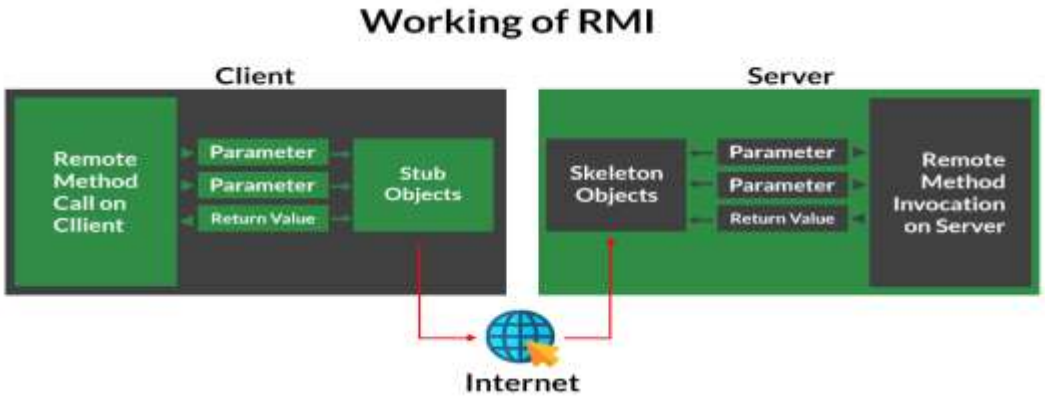
- RPC method helps clients to communicate with servers by the conventional use of procedure calls in high-level languages.
- RPC method is modeled on the local procedure call, but the called procedure is most likely to be executed in a different process and usually a different computer.
- RPC supports process and thread-oriented models.
- RPC makes the internal message passing mechanism hidden from the user.
- The effort needs to re-write and re-develop the code is minimum.
- Remote procedure calls can be used for the purpose of distributed and the local environment.
- It commits many of the protocol layers to improve performance.
- RPC provides abstraction. For example, the message-passing nature of network communication remains hidden from the user.
- RPC allows the usage of the applications in a distributed environment that is not only in the local environment.
- With RPC code, re-writing and re-developing effort is minimized.
- Process-oriented and thread-oriented models support by RPC.

Disadvantages of RPC

- Remote Procedure Call Passes Parameters by values only and pointer values, which is not allowed.
- Remote procedure calling (and return) time (i.e., overheads) can be significantly lower than that for a local procedure.
- This mechanism is highly vulnerable to failure as it involves a communication system, another machine, and another process.
- RPC concept can be implemented in different ways, which is can't standard.
- Not offers any flexibility in RPC for hardware architecture as It is mostly interaction-based.
- The cost of the process is increased because of a remote procedure call.

RMI (REMOTE METHOD INVOCATION)

Remote Method Invocation (RMI) is an API that allows an object to invoke a method on an object that exists in another address space, which could be on the same machine or on a remote machine. Through RMI, an object running in a JVM present on a computer (Client-side) can invoke methods on an object present in another JVM (Server-side). RMI creates a public remote server object that enables client and server-side communications through simple method calls on the server object.



Stub Object: The stub object on the client machine builds an information block and sends this information to the server.
The block consists of
An identifier of the remote object to be used
Method name which is to be invoked
Parameters to the remote JVM
Skeleton Object: The skeleton object passes the request from the stub object to the remote object. It performs the following tasks
It calls the desired method on the real object present on the server.
It forwards the parameters received from the stub object to the method.

Working of an RMI Application

The following are the points on how an RMI application works −

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remote Ref**. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

Advantages of RMI:

Object Oriented: RMI can pass full objects as arguments and return values, not just predefined data types. This means that you can pass complex types, such as a standard Java hashtable object, as a single argument. In existing RPC systems you would have to have the client decompose such an object into primitive data types, ship those data types, and the recreate a hashtable on the server. RMI lets you ship objects directly across the wire with no extra client code.

Mobile Behavior: RMI can move behavior (class implementations) from client to server and server to client. For example, you can define an interface for examining employee expense reports to see whether they conform to current company policy. When an expense report is created, an object that implements that interface can be fetched by the client from the server. When the policies change, the server will start returning a different implementation of that interface that uses the new policies. The constraints will therefore be checked on the client side-providing faster feedback to the user and less load on the server-without installing any new software on user's system. This gives you maximal flexibility, since changing policies requires you to write only one new Java class and install it once on the server host.

Design Patterns: Passing objects lets you use the full power of object oriented technology in distributed computing, such as two- and three-tier systems. When you can pass behavior, you can use object oriented design patterns in your solutions. All object oriented design patterns rely upon different behaviors for their power; without passing complete objects-both implementations and type-the benefits provided by the design patterns movement are lost.

Safe and Secure: RMI uses built-in Java security mechanisms that allow your system to be safe when users downloading implementations. RMI uses the security manager defined to protect systems from hostile applets to protect your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.

Easy to Write/Easy to Use: RMI makes it simple to write remote Java servers and Java clients that access those servers. A remote interface is an actual Java interface. A server has roughly three lines of code to declare itself a server, and otherwise is like any other Java object. This simplicity makes it easy to write servers for full-scale distributed object systems quickly, and to rapidly bring up prototypes and early versions of software for testing and evaluation. And because RMI programs are easy to write they are also easy to maintain.

Connects to Existing/Legacy Systems: RMI interacts with existing systems through Java's native method interface JNI. Using RMI and JNI you can write your client in Java and use your existing server implementation. When you use RMI/JNI to connect to existing servers you can rewrite any parts of you server in Java when you choose to, and get the full benefits of Java in the new code. Similarly, RMI interacts with existing relational databases using JDBC without modifying existing non-Java source that uses the databases. Write Once, Run Anywhere: RMI is part of Java's "Write Once, Run Anywhere" approach. Any RMI based system is 100% portable to any Java Virtual Machine *, as is an RMI/JDBC system. If you use RMI/JNI to interact with an existing system, the code written using JNI will compile and run with any Java virtual machine.

Distributed Garbage Collection: RMI uses its distributed garbage collection feature to collect remote server objects that are no longer referenced by any clients in the network. Analogous to garbage collection inside a Java Virtual Machine, distributed garbage collection lets you define server objects as needed, knowing that they will be removed when they no longer need to be accessible by clients.
Parallel Computing: RMI is multi-threaded, allowing your servers to exploit Java threads for better concurrent processing of client requests.
The Java Distributed Computing Solution: RMI is part of the core Java platform starting with JDK?? 1.1, so it exists on every 1.1 Java Virtual Machine. All RMI systems talk the same public protocol, so all Java systems can talk to each other directly, without any protocol translation overhead.

CASE STUDY JAVA RMI:-

These are the steps to follow −

- Define the remote interface
- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program
- Compile the application
- Execute the application

Remote interfaces in Java RMI • Remote interfaces are defined by extending an interface called *Remote* provided in the *java.rmi* package. The methods must throw *RemoteException*, but application-specific exceptions may also be thrown. Figure 5.16 shows an example of two remote interfaces called *Shape* and *ShapeList*. In this example, *GraphicalObject* is a class that holds the state of a graphical object – for example, its type, its position, enclosing rectangle, line colour and fill colour – and provides operations for accessing and updating its state. *GraphicalObject* must implement the *Serializable* interface. Consider the interface *Shape* first: the *getVersion* method returns an integer, whereas the *getAllState* method returns an instance of the class *GraphicalObject*. Now consider the interface *ShapeList*: its *newShape* method passes an instance of *GraphicalObject* as its argument but returns an object with a remote.

Figure 5.16 Java Remote interfaces *Shape* and *ShapeList*

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

Figure 5.17 The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)
This method is used by a server to register the identifier of a remote object by name, as shown in Figure 5.18, line 3.

void bind (String name, Remote obj)
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)
This method removes a binding.

Remote lookup(String name)
This method is used by clients to look up a remote object by name, as shown in Figure 5.20, line 1. A remote object reference is returned.

String [] list()
This method returns an array of *Strings* containing the names bound in the registry.

Downloading of classes • Java is designed to allow classes to be downloaded from one virtual machine to another. This is particularly relevant to distributed objects that communicate by means of remote invocation. We have seen that non-remote objects are passed by value and remote objects are passed by reference as arguments and results of RMIs. If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically. Similarly, if the recipient of a remote object reference does not already possess the class for a proxy, its code is downloaded automatically. This has two advantages:
1. There is no need for every user to keep the same set of classes in their working environment.
2. Both client and server programs can make transparent use of instances of new classes whenever they are added.
As an example, consider the whiteboard program and suppose that its initial implementation of *GraphicalObject* does not allow for text. A client with a textual object can implement a subclass of *GraphicalObject* that deals with text and pass an instance to the server as an argument of the *newShape* method. After that, other clients may retrieve the instance using the *getAllState* method. The code of the new class will be downloaded automatically from the first client to the server and then to other clients as needed.

Figure 5.18 Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();                    1
            ShapeList stub =                                                  2
                (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);3
            Naming.rebind("//bruno.ShapeList", stub );                        4
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Figure 5.19 Java class *ShapeListServant* implements interface *ShapeList*

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

Figure 5.20 Java client of *ShapeList*

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

UNIT-IV

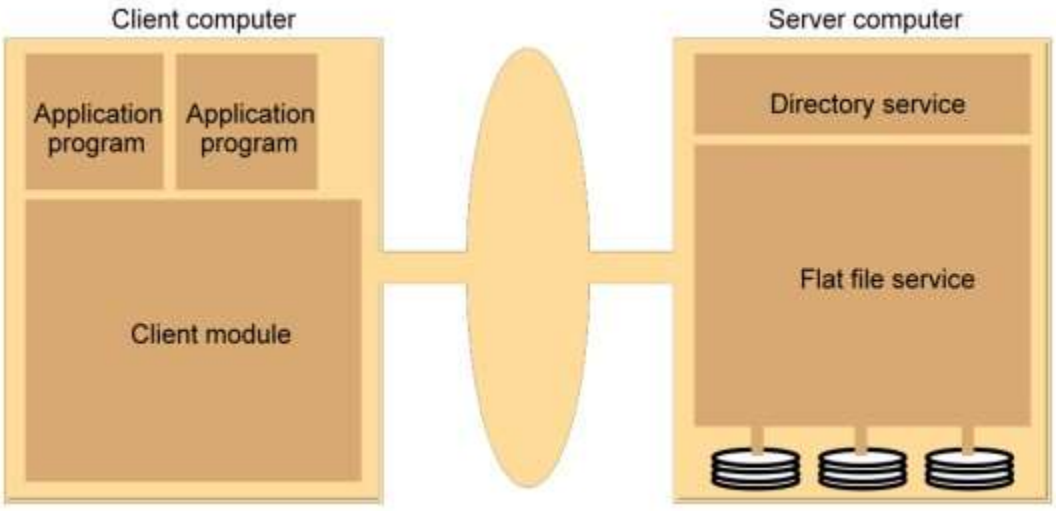
Distributed file system

Introduction:

It allows programs to access or store isolated files as they do with the local ones, allowing programmers to access files from any network or computer.

The DFS makes it convenient to share information and files among users on a network in a controlled and authorized way. The server allows the client users to share files and store data just as if they are storing the information locally. However, the servers have full control over the data, and give access control to the clients.

FILE SERVICE ARCHITECTURE:-



1. Flat file service: A flat file service is used to perform operations on the contents of a file. The Unique File Identifiers (UFIDs) are associated with each file in this service. For that long sequence of bits is used to uniquely identify each file among all of the available files in the distributed system. When a request is received by the Flat file service for the creation of a new file then it generates a new UFID and returns it to the requester.

Flat File Service Model Operations:
Read(FileId, i, n) -> Data: Reads up to n items from a file starting at item 'i' and returns it in Data.
Write(FileId, i, Data): Write a sequence of Data to a file, starting at item I and extending the file if necessary.
Create() -> FileId: Creates a new file with length 0 and assigns it a UFID.
Delete(FileId): The file is removed from the file store.

GetAttributes(FileId) -> Attr: Returns the file's file characteristics.
SetAttributes(FileId, Attr): Sets the attributes of the file.

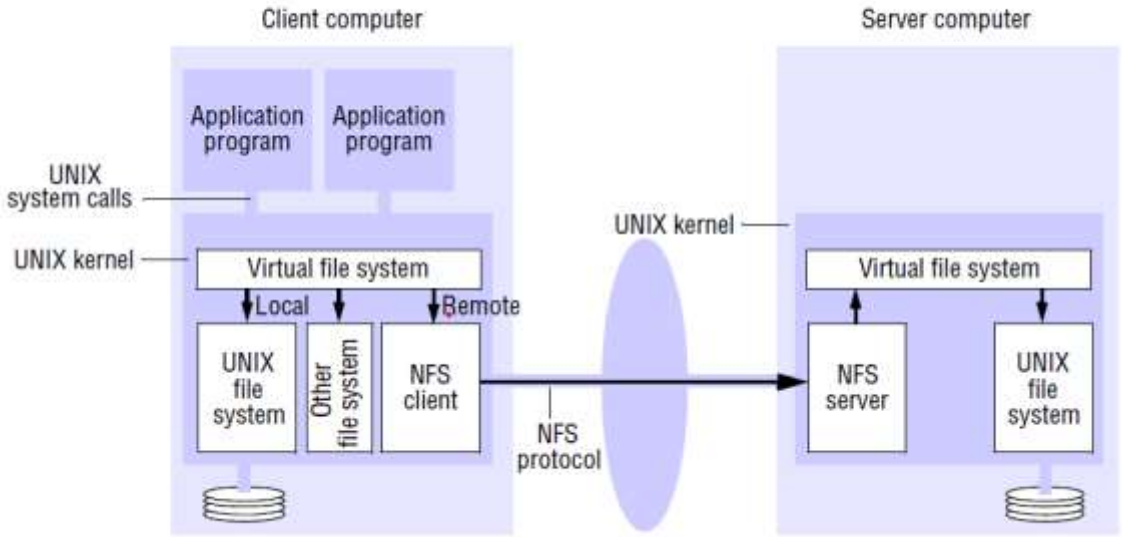
2. Directory Service: The directory service serves the purpose of relating file text names with their UFIDs (Unique File Identifiers). The fetching of UFID can be made by providing the text name of the file to the directory service by the client. The directory service provides operations for creating directories and adding new files to existing directories.
Directory Service Model Operations:
Lookup(Dir, Name) -> FileId : Returns the relevant UFID after finding the text name in the directory. Throws

an exception if Name is not found in the directory.
AddName(Dir, Name, File): Adds(Name, File) to the directory and modifies the file's attribute record if Name is not in the directory. If a name already exists in the directory, an exception is thrown.
UnName(Dir, Name): If Name is in the directory, the directory entry containing Name is removed. An exception is thrown if the Name is not found in the directory.
GetNames(Dir, Pattern) -> NameSeq: Returns all the text names that match the regular expression Pattern in the directory.

3. Client Module: The client module executes on each computer and delivers an integrated service (flat file and directory services) to application programs with the help of a single API. It stores information about the network locations of flat files and directory server processes. Here, recently used file blocks hold in a cache at the client-side, thus, resulting in improved performance.

CASE STUDY 1:- SUN NETWORK FILE SYSTEM

Figure 12.8 NFS architecture



All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems, and we shall describe the UNIX implementation the NFS protocol (version 3).The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.
The NFS client and server modules communicate using remote procedure calls. Sun's RPC system, described in Section 5.3.3, was developed for use in NFS. It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be
acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.

Virtual file system • Figure 12.8 makes it clear that NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way.

The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the filesystems that are currently available both locally and remotely, and it passes each request to the appropriate local system module (the UNIX file system, the NFS client module or the service module for another file system).The file identifiers used in NFS are called *file handles*. A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file. In UNIX implementations of NFS, the file handle is derived from the file's *i-node number* by adding two extra fields as follows (the i-node number of a UNIX file is a number that serves to identify and locate the file within the file system in which the file is stored):

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

NFS adopts the UNIX mountable filesystem as the unit of file grouping defined in the preceding section. The *filesystem identifier* field is a unique number that is allocated to each filesystem when it is created (and in the UNIX implementation is stored in the superblock of the file system). The *i-node generation number* is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed. In the VFS extensions to the UNIX file system, a generation number is stored with each file and is incremented each time the i-node number is reused (for example, in a UNIX *create* system call). The client obtains the first file handle for a remote file system when it mounts it. File handles are passed from server to client in the results of *lookup*, *create* and *mkdir* operations (see Figure 12.9) and from client to server in the argument lists of all server operations.
The virtual file system layer has one VFS structure for each mounted file system and one *v-node* per open file. A VFS structure relates a remote file system to the local directory on which it is mounted. The v-node contains an indicator to show whether a file is local or remote. If the file is local, the v-node contains a reference to the index of the local file (an i-node in a UNIX implementation). If the file is remote, it contains the file handle of the remote file.

Client integration • The NFS client module plays the role described for the client module in our architectural model, supplying an interface suitable for use by conventional application programs. But unlike our model client module, it emulates the semantics of the standard UNIX file system primitives precisely and is integrated with the UNIX kernel. It is integrated with the kernel and not supplied as a library for loading into client processes so that:
• user programs can access files via UNIX system calls without recompilation or reloading;
• a single client module serves all of the user-level processes, with a shared cache of recently used blocks (described below);

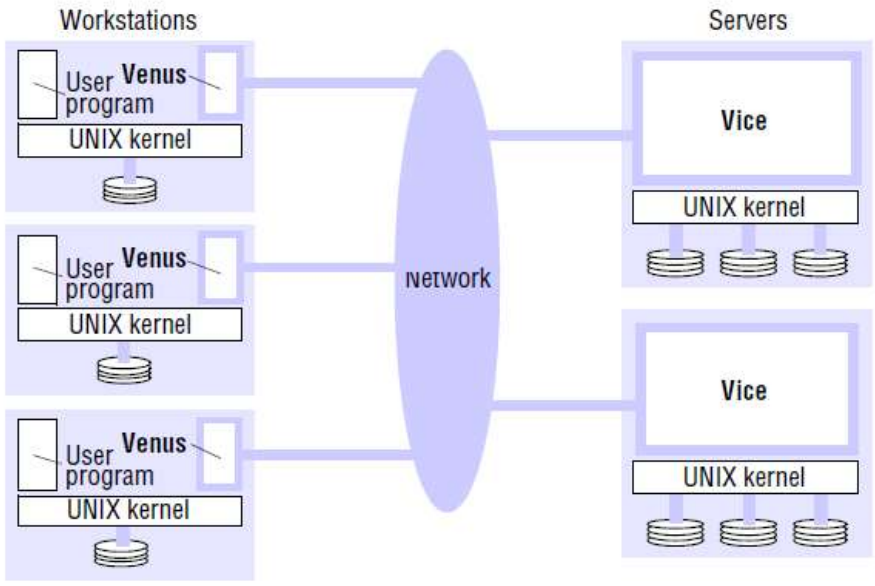
Access control and authentication • Unlike the conventional UNIX file system, the NFS server is stateless and does not keep files open on behalf of its clients. So the server must check the user's identity against the file's access permission attributes afresh on each request, to see whether the user is permitted to access the file in the manner requested. The Sun RPC protocol requires clients to send user authentication information (for example, the conventional UNIX 16-bit user ID and group ID) with each request and this is checked against the access permission in the file attributes. These additional parameters are not shown in our overview of the NFS protocol in Figure 12.9; they are supplied automatically by the RPC system. In its simplest form, there is a security loophole in this access-control mechanism. An NFS server provides a conventional RPC interface

at a well-known port on each host and any process can behave as a client, sending requests to the server to access or update a file. The client can modify the RPC calls to include the user ID of any user, impersonating the user without their knowledge or permission. This security loophole has been closed by the use of an option in the RPC protocol for the DES encryption of the user's authentication information. More recently, Kerberos has been integrated with Sun NFS to provide a stronger and more comprehensive solution to the problems of user authentication and security; we describe this below.

- Cache:
    - Client side:
        - cache file data and metadata by block that is read from server in local memory
        - Cache serves as a temporary buffer for writes (allow asyncronous write)
        - Advantage: reduce network usage, improve performance
        - Disadvantage: write lost in memory after crash (safety vs. performance tradeoff)
    - Server side:
        - server can buffer the write in memory and write to disk asychronously
        - Problem: write in memory can lost
        - Sol:
            - battery-backed memory
            - commit each WRITE to stable storage before ack WRITE success to clients

CASE STUDY -2 ANDREW FILE SYSTEM

Figure 12.11 Distribution of processes in the Andrew File System



AFS provides transparent access to remote shared files for UNIX programs running on workstations. Access to AFS files is via the normal UNIX file primitives, enabling existing UNIX programs to access AFS files without modification or recompilation. AFS is compatible with NFS. AFS servers hold 'local' UNIX files, but the filing system in the servers is NFS-based, so files are referenced by NFS-style file handles rather than i-node numbers, and the files may be remotely accessed via NFS. AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems.

The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics: *Whole-file serving*: The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks). *Whole-file caching*: Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' *open* requests in preference to remote copies whenever possible

Operation of AFS:

1. When a user process in a client computer issues an *open* system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.
2. The copy is stored in the local UNIX file system in the client computer. The copy is then *open*ed and the resulting UNIX file descriptor is returned to the client.
3. Subsequent *read, write* and other operations on the file by processes in the client computer are applied to the local copy.
4. When the process in the client issues a *close* system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client's local disk is retained in case it is needed again

by a user-level process on the same workstation For shared files that are infrequently updated (such as those

containing the code of UNIX commands and libraries) and for files that are normally accessed by only a single user (such as most of the files in a user's home directory and its subtree), locally cached copies are likely to remain valid for long periods The local cache can be allocated a substantial proportion of the disk space on each workstation – say, 100 megabytes. This is normally sufficient for the establishment of a working set of the files used by one user. The provision of sufficient cache storage for the establishment of a working set ensures that files in regular use on agiven workstation are normally retained in the cache until they are needed again.

• The design strategy is based on some assumptions about average and maximum file size and locality of reference to files in UNIX systems. These assumptions are derived from observations of typical UNIX workloads in academic and other environments [Satyanarayanan 1981, Ousterhout *et al.* 1985, Floyd 1986]. The most important observations are:

– Files are small; most are less than 10 kilobytes in size.

– Read operations on files are much more common than writes (about six times more common).

– Sequential access is common, and random access is rare.

– Most files are read and written by only one user. When a file is shared, it is usually only one user who modifies it.

– Files are referenced in bursts. If a file has been referenced recently, there is a high probability that it will be referenced again in the near future.

AFS is implemented as two software components that exist as UNIX processes called *Vice* and *Venus*. Figure 12.11 shows the distribution of Vice and Venus processes. Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.

The files available to user processes running on workstations are either *local* or *shared*. Local files are handled as normal UNIX files. They are stored on a 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.

One of the file partitions on the local disk of each workstation is used as a cache, holding the cached copies of files from the shared space. Venus manages the cache, removing the least recently used files when a new file is acquired from a server to make the required space if the partition is full. The workstation cache is usually large enough to accommodate several hundred average-sized files, rendering the workstation largely independent of the Vice servers once a working set of the current user's files and frequently used system files has been cached.

AFS resembles the abstract file service model described in Section 12.2 in these respects:

• A flat file service is implemented by the Vice servers, and the hierarchic directory structure required by UNIX user programs is implemented by the set of Venus processes in the workstations.

• Each file and directory in the shared file space is identified by a unique, 96-bit file identifier (*fid*) similar to a UFID. The Venus processes translate the pathnames issued by clients to *fid*s.

Files are grouped into *volumes* for ease of location and movement. Volumes are generally smaller than the UNIX filesystems, which are the unit of file grouping in NFS.

For example, each user's personal files are generally located in a separate volume. Other volumes are allocated for system binaries, documentation and library code. The representation

of *fid*s includes the volume number for the volume containing the file (*cf.* the *file group identifier* in UFIDs), an NFS file handle identifying the file within the volume (*cf.* the *file number* in UFIDs) and a *uniquifier* to ensure that file identifiers are not reused: User programs use conventional UNIX pathnames to refer to files, but AFS uses *fid*s in the communication between the Venus and Vice processes. The Vice servers accept requests only in terms of *fid*s. Venus translates the pathnames supplied by clients into *fid*s using a step-by-step lookup to obtain the information from the file directories held in the Vice servers.

Figure 12.14 describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues each of the system calls mentioned in our outline scenario above. The *callback promise* mentioned here is a mechanism for ensuring that cached copies of files are updated when another client closes the same file after

updating it

Figure 12.14 Implementation of file system calls in AFS

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

Cache consistency

When Vice supplies a copy of a file to a Venus process it also provides a *callback promise* – a token issued by the Vice server that is the custodian of the file, guaranteeing that it will notify the Venus process when any other client modifies the file. Callback promises are stored with the cached files on the workstation disks and have two states: *valid* or *cancelled*. When a server performs a request to update a file it notifies all of the Venus processes to which it has issued callback promises by sending a *callback* to each – a callback is a remote procedure call from a server to a Venus process.

When the Venus process receives a callback, it sets the *callback promise* token for the relevant file to *cancelled*.

Whenever Venus handles an *open* on behalf of a client, it checks the cache. If the required file is found in the cache, then its token is checked. If its value is *cancelled*, then a fresh copy of the file must be fetched from the Vice server, but if the token is *valid*, then the cached copy can be opened and used without reference to Vice.

When a workstation is restarted after a failure or a shutdown, Venus aims to retain as many as possible of the cached files on the local disk, but it cannot assume that the callback promise tokens are correct, since some callbacks may have been missed. Before the first use of each cached file or directory after a restart, Venus therefore generates a cache validation request containing the file modification timestamp to the server that is the custodian of the file. If the timestamp is current, the server responds with *valid* and the token is reinstated. If the timestamp shows that the file is out of date, then the server responds with *cancelled* and the token is set to *cancelled*. Callbacks must be renewed before an *open* if a time *T* (typically on the order of a few minutes) has elapsed since the file was cached without communication from the server. This is to deal with possible

communication failures, which can result in the loss of callback messages.

This callback-based mechanism for maintaining cache consistency was adopted as offering the most scalable approach, following the evaluation in the prototype (AFS-1) of a timestamp-based mechanism similar to that used in NFS. In AFS-1, a Venus process holding a cached copy of a file interrogates the Vice process on each *open* to determine whether the timestamp on the local copy agrees with that on the server. The callback based approach is more scalable because it results in communication between client and server and activity in the server only when the file has been updated, whereas the timestamp approach results in a client-server interaction on each *open*, even when there is a valid local copy. Since the majority of files are not accessed concurrently, and *read* operations predominate over *writes* in most applications, the *callback* mechanism results in a dramatic reduction in the number of client-server interactions.

The callback mechanism used in AFS-2 and later versions of AFS requires Vice servers to maintain some state on behalf of their Venus clients, unlike AFS-1, NFS and our file service model. The client-dependent state required consists of a list of the Venus processes to which callback promises have been issued for each file. These callback lists must be retained over server failures – they are held on the server disks and are updated using atomic operations.

Figure 12.15 The main components of the Vice service interface

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

Distributed Shared Memory

DSM is a mechanism that manages memory across multiple nodes and makes inter-process communications transparent to end-users. The applications will think that they are running on shared memory. DSM is a mechanism of allowing user processes to access shared data without using inter-process communications. In DSM every node has its own memory and provides memory read and write services and it provides consistency protocols. The distributed shared memory (DSM) implements the shared memory model in distributed systems but it doesn't have physical shared memory. All the nodes share the virtual address space provided by the shared memory model.

Design and implantation issues :-

1. Granularity: Granularity refers to the block size of a DSM system. Granularity refers to the unit of sharing and the unit of data moving across the network when a network block shortcoming then we can utilize the estimation of the block size as words/phrases. The block size might be different for the various networks.

2.Structure of shared memory space: Structure refers to the design of the shared data in the memory. The structure of the shared memory space of a DSM system is regularly dependent on the sort of applications that the DSM system is intended to support.

3. Memory coherence and access synchronization: In the DSM system the shared data things ought to be accessible by different nodes simultaneously in the network. The fundamental issue in this system is data irregularity. The data irregularity might be raised by the synchronous access. To solve this problem in the DSM system we need to utilize some synchronization primitives, semaphores, event count, and so on.

4. Data location and access: To share the data in the DSM system it ought to be possible to locate and retrieve the data as accessed by clients or processors. Therefore the DSM system must implement some form of data block finding system to serve network data to meet the
requirement of the memory coherence semantics being utilized.

5. Replacement strategy: In the local memory of the node is full, a cache miss at the node implies not just a get of the gotten to information block from a remote node but also a replacement. A data block of the local memory should be replaced by the new data block. Accordingly, a position substitution methodology is additionally vital in the design of a DSM system.

6. Thrashing: In a DSM system data blocks move between nodes on demand. In this way on the off chance that 2 nodes complete for write access to the single data item. The data relating data block might be moved to back and forth at such a high rate that no genuine work can get gone. The DSM system should utilize an approach to keep away from a situation generally known as thrashing.

7. Heterogeneity: The DSM system worked for homogeneous systems and need not address the heterogeneity issue. In any case, assuming the underlined system environment is heterogeneous, the DSM system should be designed to deal with heterogeneous, so it works appropriately with machines having different architectures.

Consistency Model:-

Strict Consistency Model: "The strict consistency model is the strongest form of memory coherence, having the most stringent consistency requirements. A shared-memory system is said to support the strict consistency model if the value returned by a read operation on a memory address is always the same as the value written by the most recent write operation to that address, irrespective of the locations of the processes performing the read and write operations. That is, all writes instantaneously become visible to all processes."

Sequential Consistency Model: "The sequential consistency model was proposed by Lamport ... . A shared-memory system is said to support the sequential consistency model if all processes see the same order of all memory access operations on the shared memory. The exact order in which the memory access operations are interleaved does not matter. ... If one process sees one of the orderings of ... three operations and another process sees a different one, the memory is not a sequentially consistent memory."

Casual Consistency Model: "The causal consistency model ... relaxes the requirement of the sequential model for better concurrency. Unlike the sequential consistency model, in the causal consistency model, all processes see only those memory reference operations in the same (correct) order that are potentially causally related. Memory reference operations that are not potentially causally related may be seen by different processes in different orders.

FIFO Consistency Model: For FIFO consistency, "Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

"FIFO consistency is called PRAM consistency in the case of distributed shared memory systems."
Pipelined Random-Access Memory (PRAM) Consistency Model: "The pipelined random-access memory (PRAM) consistency model ... provides a weaker consistency semantics than the (first three) consistency models described so far. It only ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed as if all the write operations performed by a single

process are in a pipeline. Write operations performed by different processes may be seen by different processes in different orders."

Weak Consistency Model: "Synchronization accesses (accesses required to perform synchronization operations) are sequentially consistent. Before a synchronization access can be performed, all previous regular data accesses must be completed. Before a regular data access can be performed, all previous synchronization accesses must be completed. This essentially leaves the problem of consistency up to the programmer. The memory will only be consistent immediately after a synchronization operation.

Release Consistency Model: "Release consistency is essentially the same as weak consistency, but synchronization accesses must only be processor consistent with respect to each other. Synchronization operations are broken down into acquire and release operations. All pending acquires (e.g., a lock operation) must be done before a release (e.g., an unlock operation) is done. Local dependencies within the same processor must still be respected.
"Release consistency is a further relaxation of weak consistency without a significant loss of coherence.

Entry Consistency Model: "Like ... variants of release consistency, it requires the programmer (or compiler) to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, entry consistency requires each ordinary shared data item to be associated with some synchronization variable, such as a lock or barrier. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. When an acquire is done on a synchronization variable, only those data guarded by that synchronization variable are made consistent.

Processor Consistency Model: "Writes issued by a processor are observed in the same order in which they were issued. However, the order in which writes from two processors occur, as observed by themselves or a third processor, need not be identical. That is, two simultaneous reads of the same location from different processors may yield different results.

General Consistency Model: "A system supports general consistency if all the copies of a memory location eventually contain the same data when all the writes issued by every processorhavecompleted."

UNIT-V

TRANSACTIONS & Concurrency Control Introduction:

A transaction defines a sequence of server operations that is guaranteed by the server to be atomic in the presence of multiple clients and server crashes and must be done in an ACID-compliant manner.
Atomicity –
The transaction is completed entirely or not at all.
Consistency –
It is a term that refers to the transition from one consistent state to another.
Durability: After a transaction has completed successfully, all its effects are saved in permanent storage. We use the term 'permanent storage' to refer to files held on disk or another permanent medium. Data saved in a file will survive if the server process crashes.

Isolation: Each transaction must be performed without interference from other transactions; in other words, the intermediate effects of a transaction must not be visible to other transactions. The box below introduces a mnemonic, ACID, for remembering the properties of atomic transactions.

The goal of transactions is to ensure that all of the objects managed by a server remain in a consistent state when they are accessed by multiple transactions and in the presence of server crashes.

Transaction capabilities can be added to servers of recoverable objects. Each transaction is created and managed by a coordinator, which implements the *Coordinator* interface

Operations in the *Coordinator* interface

openTransaction() → *trans;*
Starts a new transaction and delivers a unique TID *trans*. This identifier 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.

The coordinator gives each transaction an identifier, or TID. The client invokes the *openTransaction* method of the coordinator to introduce a new transaction – a transaction identifier or TID is allocated and returned. At the end of a transaction, the client invokes the *closeTransaction* method to indicate its end – all of the recoverable objects accessed by the transaction should be saved. If, for some reason, the client wants to abort a transaction, it invokes the *abortTransaction* method – all of its effects should be removed from sight.

A transaction is achieved by cooperation between a client program, some recoverable objects and a coordinator. The client specifies the sequence of invocations on recoverable objects that are to comprise a transaction. To achieve this, the client sends with each invocation the transaction identifier returned by *openTransaction*. One way to make this possible is to include an extra argument in each operation of a recoverable object to carry the TID.

Normally, a transaction completes when the client makes a *closeTransaction* request. If the transaction has progressed normally, the reply states that the transaction is *committed* – this constitutes a promise to the client that all of the changes requested in the transaction are permanently recorded and that any future transactions that access the same data will see the results of all of the changes made during the transaction. Alternatively, the transaction may have to *abort* for one of several reasons related to the nature of the transaction itself, to conflicts with another transaction or to the crashing of a process or computer. When a transaction is aborted the parties involved (the recoverable objects and the coordinator) must ensure that none of its effects are visible to future transactions, either in the objects or in their copies in permanent storage. A transaction is either successful or is aborted in one of two ways – the client aborts it (using an *abortTransaction* call to the server) or the server aborts it shows these three alternative life histories for transactions.

We refer to a transaction as *failing* in both of the latter cases.

Service actions related to process crashes • If a server process crashes unexpectedly, it is eventually replaced. The new server process aborts any uncommitted transactions and uses a recovery procedure to restore the values of the objects to the values produced by the most recently committed transaction. To deal with a client that crashes unexpectedly during a transaction, servers can give each transaction an expiry time and abort any transaction that has not completed before its expiry time.

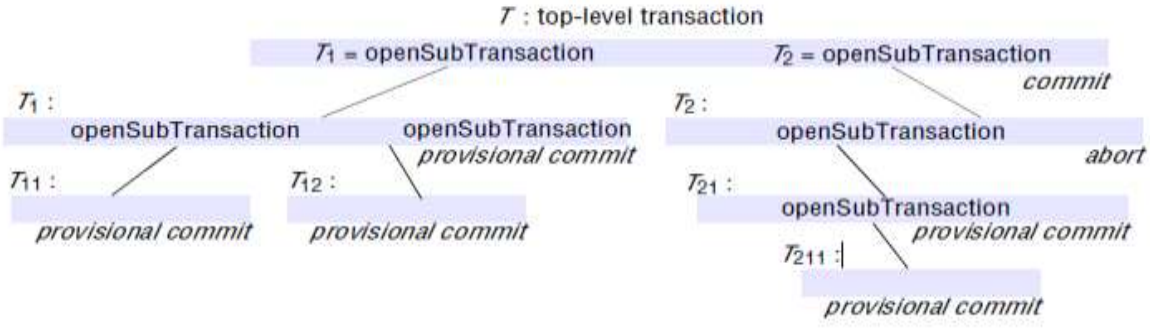
Client actions related to server process crashes • If a server crashes while a transaction is in progress, the client will become aware of this when one of the operations returns an exception after a timeout. If a server crashes and is then replaced during the progress of a transaction, the transaction will no longer be valid and the client must be informed via an exception to the next operation. In either case, the client must then formulate a plan, possibly in consultation with the human user, for the completion or abandonment of the task of which the transaction was a part.

Nested Transactions :-

A transaction that includes other transactions within its initiating point and a end point are known as nested transactions. So the nesting of the transactions is done in a transaction. The nested transactions here are called sub-transactions.

The top-level transaction in a nested transaction can open sub-transactions, and each sub-transaction can open more sub-transactions down to any depth of nesting. When a subtransaction aborts, the parent transaction can sometimes choose an alternative subtransaction to complete its task

Figure 16.13 Nested transactions



The rules for committing of nested transactions are rather subtle:

• A transaction may commit or abort only after its child transactions have completed.

• When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. Its decision to abort is final.

• When a parent aborts, all of its subtransactions are aborted. For example, if *T*2 aborts then *T*21 and *T*211 must also abort, even though they may have provisionally committed.

• When a subtransaction aborts, the parent can decide whether to abort or not. In our example, *T* decides to

commit although *T*2 has aborted
• If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted. In our example, *T*'s commitment allows *T*1, *T*11 and *T*12 to commit, but not *T*21 and *T*211 since their parent, *T*2, aborted. Note that the effects of a subtransaction are not permanent until the top-level transaction commits.

Nested transactions have the following main advantages:

1. Subtransactions at one level (and their descendants) may run concurrently with other subtransactions at the same level in the hierarchy. This can allow additional concurrency in a transaction. When subtransactions run in different servers, they can work in parallel.

2. Subtransactions can commit or abort independently. In comparison with a single transaction, a set of nested subtransactions is potentially more robust.

All of the concurrency control protocols are based on the criterion of serial equivalence and are derived from rules for conflicts between operations. Three methods are described:

• Locks are used to order transactions that access the same objects according to the order of arrival of their operations at the objects.
• Optimistic concurrency control allows transactions to proceed until they are ready to commit, whereupon a check is made to see whether they have performed conflicting operations on objects.
• Timestamp ordering uses timestamps to order transactions that access the same objects according to their starting times.

LOCKS:-

In this locking scheme, the server attempts to lock any object that is about to be used by any operation of a client's transaction. If a client requests access to an object that is already locked due to another client's transaction, the request is suspended and the client must wait until the object is unlocked.

Figure 16.14 Transactions *T* and *U* with exclusive locks

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(bal*1.1)*		*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)		*c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

In this example, it is assumed that when transactions *T* and *U* start, the balances of the accounts *A*, *B* and *C* are not yet locked. When transaction *T* is about to use account *B*, it is locked for *T*. When transaction *U* is about to use *B* it is still
locked for *T*, so transaction *U* waits. When transaction *T* is committed, *B* is unlocked, whereupon transaction *U* is resumed. The use of the lock on *B* effectively serializes the access to *B*. Note that if, for example, *T*

released the lock on *B* between its *getBalance* and *setBalance* operations, transaction *U*'s *getBalance* operation on *B* could be
interleaved between them.

Serial equivalence requires that all of a transaction's accesses to a particular object be serialized with respect to accesses by other transactions. All pairs of conflicting operations of two transactions should be executed in the same order. To ensure this, a transaction is not allowed any new locks after it has released a lock. The first phase of each transaction is a 'growing phase', during which new locks are acquired. In the
second phase, the locks are released (a 'shrinking phase'). This is called *two-phase locking*.

It is preferable to adopt a locking scheme that controls the access to each object so that there can be several concurrent transactions reading an object, or a single transaction writing an object, but not both. This is commonly referred to as a 'many readers/single writer' scheme. Two types of locks are used: *read locks* and *write locks*. Before a transaction's *read* operation is performed, a read lock should be set on the object. Before a transaction's *write* operation is performed, a write lock should be set on the object. Whenever it is impossible to set a lock immediately, the transaction (and the client) must wait until it is possible to do so – a client's request is never rejected.

As pairs of *read* operations from different transactions do not conflict, an attempt to set a read lock on an object with a read lock is always successful. All the transactions reading the same object share its read lock – for this reason, read locks are sometimes called *shared locks*.
The operation conflict rules tell us that:
1. If a transaction *T* has already performed a *read* operation on a particular object, then a concurrent transaction *U* must not *write* that object until *T* commits or aborts.
2. If a transaction *T* has already performed a *write* operation on a particular object, then a concurrent transaction *U* must not *read* or *write* that object until *T* commits or aborts.

To enforce condition 1, a request for a write lock on an object is delayed by the presence of a read lock belonging to another transaction. To enforce condition 2, a request for either a read lock or a write lock on an object is delayed by the presence of a write lock belonging to another transaction

Figure 16.16 Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule b is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

The granting of locks will be implemented by a separate object in the server that we call the *lock manager*. The lock manager holds a set of locks, for example in a hash table. Each lock is an instance of the class *Lock* and is associated with a particular object. The class *Lock* is shown in Figure 16.17. Each instance of *Lock*

maintains the following information in its instance variables:
• the identifier of the locked object;
• the transaction identifiers of the transactions that currently hold the lock (shared locks can have several holders);
• a lock type.

if no other transaction holds the lock, just add the given transaction to the holders and set the type;
• else if another transaction holds the lock, share it by adding the given transaction to the holders (unless it is already a holder);
• else if this transaction is a holder but is requesting a more exclusive lock, promote the lock.

All requests to set locks and to release them on behalf of transactions are sent to an instance of *LockManager*:
• The *setLock* method's arguments specify the object that the given transaction wants to lock and the type of lock. It finds a lock for that object in its hashtable or, if necessary, creates one. It then invokes the *acquire* method of that lock.
• The *unLock* method's argument specifies the transaction that is releasing its locks.
It finds all of the locks in the hashtable that have the given transaction as a holder.
For each one, it calls the *release* method

Locking rules for nested transactions

• The aim of a locking scheme for nested transactions is to serialize access to objects so that:
1. Each set of nested transactions is a single entity that must be prevented from observing the partial effects of any other set of nested transactions.
2. Each transaction within a set of nested transactions must be prevented from observing the partial effects of the other transactions in the set.
The second rule is enforced as follows:
• Parent transactions are not allowed to run concurrently with their child transactions. If a parent transaction has a lock on an object, it *retains* the lock during the time that its child transaction is executing. This means that the child
transaction temporarily acquires the lock from its parent for its duration.
• Subtransactions at the same level are allowed to run concurrently, so when they access the same objects, the locking scheme must serialize their access.

The use of locks can lead to deadlock. Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock

Consider the use of locks shown in Figure 16.19.

Figure 16.19 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 lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
•••		•••	
•••		•••	

Since the *deposit* and *withdraw* methods are atomic, we show them acquiring write locks – although in practice they read the balance and then write it. Each of them acquires a lock on one account and then gets blocked when it tries to access the account that the other one has locked. This is a deadlock situation – two transactions are waiting, and
each is dependent on the other to release a lock so it can resume

A *wait-for graph* can be used to represent the waiting relationships between current transactions. In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions – there is an edge from node *T* to node *U* when transaction *T* is waiting for transaction *U* to release a lock.

Figure 16.20 The wait-for graph for Figure 16.19

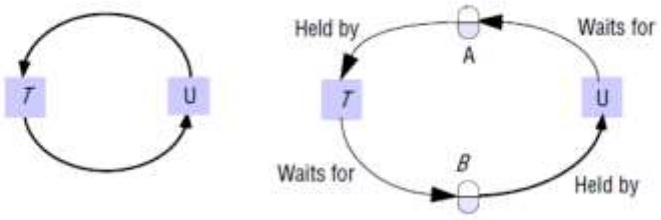
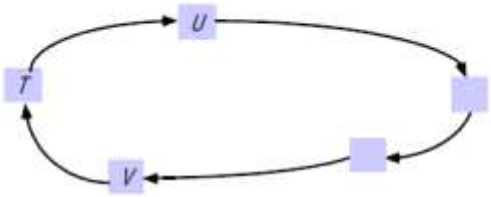


Figure 16.21 A cycle in a wait-for graph



Each transaction is waiting for the next transaction in the cycle. All of these transactions are blocked waiting for locks. None of the locks can ever be released, and the transactions are deadlocked. If one of the transactions in a cycle is aborted, then its locks are released and that cycle is broken. For example, if transaction *T* in Figure 16.21 is aborted, it will release a lock on an object that *V* is waiting for – and *V* will no longer be waiting for *T*.

Deadlock prevention • One solution is to prevent deadlock. An apparently simple but not very good way to overcome the deadlock problem is to lock all of the objects used by a transaction when it starts. This would need to be done as a single atomic step so as to avoid deadlock at this stage. Such a transaction cannot run into deadlocks with other transactions, but this approach unnecessarily restricts access to shared resources. In addition, it is sometimes impossible to predict at the start of a transaction which objects will be used. This is generally the case in interactive applications, for the user would have to say in advance exactly which objects they were planning to use – this is inconceivable in browsing-style applications, which allow users to find objects they do not know about in advance. Deadlocks can also be prevented by requesting locks on
objects in a predefined order, but this can result in premature locking and a reduction in concurrency.

Upgrade locks • CORBA's Concurrency Control Service introduces a third type of lock, called *upgrade*, the use of which is intended to avoid deadlocks. Deadlocks are often caused by two conflicting transactions first taking read locks and then attempting to promote them to write locks. A transaction with an upgrade lock on a data item is permitted to read that data item, but this lock conflicts with any upgrade locks set by
other transactions on the same data item. This type of lock cannot be set implicitly by the use of a *read* operation, but must be requested by the client.

Deadlock detection • Deadlocks may be detected by finding cycles in the wait-for graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle. The software responsible for deadlock detection can be part of the lock manager. It must hold a representation of the wait-for graph so that it can check it for cycles from time to time. Edges are added to the graph and removed from the graph by the lock manager's *setLock* and *unLock* operations.

Figure 16.23 Resolution of the deadlock in Figure 16.19

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*

Timeouts • Lock timeouts are a method for resolution of deadlocks that is commonly used. Each lock is given a limited period in which it is invulnerable. After this time, a lock becomes vulnerable. Provided that no other transaction is competing for the object that is locked, an object with a vulnerable lock remains locked. However, if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken (that is, the object is unlocked) and the waiting transaction resumes. The transaction whose lock has been broken is normally aborted.

There are many problems with the use of timeouts as a remedy for deadlocks: the worst problem is that transactions are sometimes aborted due to their locks becoming vulnerable when other transactions are waiting for them, but there is actually no deadlock.

OPTIMISTIC CONCURRENCY CONTROL:-

Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce significantly the potential for concurrency. The use of locks can result in deadlock The alternative approach proposed by Kung and Robinson is 'optimistic' because it is based on the observation that, in most applications, the likelihood of two clients' transactions accessing the same object is low. Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a *closeTransaction* request. When a conflict arises, some transaction is generally aborted and will need to be restarted by the client.

Each transaction has the following phases:

Working phase: During the working phase, each transaction has a tentative version of each of the objects that it updates. This is a copy of the most recently committed version of the object. The use of tentative versions allows the transaction to abort (with no effect on the objects), either during the working phase or if it fails validation due to other conflicting transactions. *read* operations are performed immediately – if a tentative version for that transaction already exists, a *read* operation accesses it; otherwise, it accesses the most recently committed value of the object. *write*

operations record the new values of the objects as tentative values (which are invisible to other transactions).

When there are several concurrent transactions, several different tentative values of the same object may coexist.

Validation phase: When the *closeTransaction* request is received, the transaction is validated to establish whether or not its operations on objects conflict with operations of other transactions on the same objects. If the validation is successful, then the transaction can commit. If the validation fails, then some form of conflict resolution must be used and either the current transaction or, in some cases, those with which it conflicts will need to be aborted.

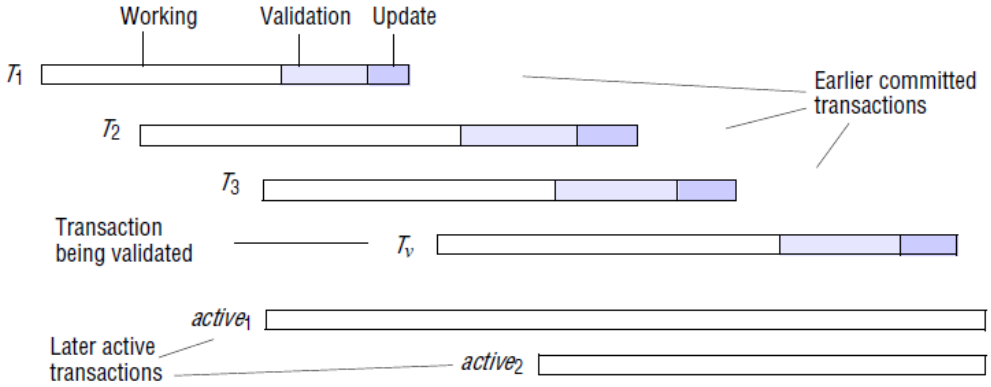
Update phase: If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transactions can commit immediately after passing validation. Write transactions are ready to commit once the tentative versions of the objects have been recorded in permanent storage.

The validation test on transaction *Tv* is based on conflicts between operations in pairs of transactions *Ti* and *Tv*. For a transaction *Tv* to be serializable with respect to an overlapping transaction *Ti*, their operations must conform to the following rules:

T_v	T_i	*Rule*	
write	*read*	1.	T_i must not read objects written by T_v.
read	*write*	2.	T_v must not read objects written by T_i.
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i.

As the validation and update phases of a transaction are generally short in duration compared with the working phase, a simplification can be achieved by making the rule that only one transaction may be in the validation and update phase at one time. When no two transactions may overlap in the update phase, rule 3 is satisfied.

Figure 16.28 Validation of transactions



Backward validation • As all the *read* operations of earlier overlapping transactions were performed before the validation of *Tv* started, they cannot be affected by the *writes* of the current transaction (and rule 1 is satisfied). The validation of transaction *Tv* checks whether its read set (the objects affected by the *read* operations of *Tv*) overlaps with any of the write sets of earlier overlapping transactions, *Ti* (rule 2). If there is any overlap, the validation fails.

Let *startTn* be the biggest transaction number assigned (to some other committed transaction) at the time when transaction *Tv* started its working phase and *finishTn* be the biggest transaction number assigned at the time when *Tv* entered the validation phase.

The following program describes the algorithm for the validation of *Tv*:

boolean valid = true;
for (int *Ti* = *startTn+1*; *Ti* <= *finishTn*; *Ti*++){
if (read set of *Tv* intersects write set of *Ti*) valid = false;
}

In backward validation, the read set of the transaction being validated is compared with the write sets of other transactions that have already committed. Therefore, the only way to resolve any conflicts is to abort the transaction that is undergoing validation. In backward validation, transactions that have no *read* operations (only *write*
operations) need not be checked
.

Forward validation • In forward validation of the transaction *Tv*, the write set of *Tv* is compared with the read sets of all overlapping active transactions – those that are still in their working phase (rule 1). Rule 2 is automatically fulfilled because the active transactions do not write until after *Tv* has completed. Let the active transactions have (consecutive) transaction identifiers *active*1 to *activeN*. The following program describes the algorithm for the forward validation of *Tv*:

boolean valid = true;
for (int *Tid* = *active1*; *Tid* <= *activeN*; *Tid*++){
if (write set of *Tv* intersects read set of *Tid*) valid = false;
}

As the read sets of the transaction being validated are not included in the check, read-only transactions always pass the validation check. As the transactions being compared with the validating transaction are still active, we have a choice of whether to abort the validating transaction or to pursue some alternative way of resolving the conflict

In general, the read sets of transactions are much larger than the write sets. Therefore, backward validation compares a possibly large read set against the old write sets, whereas forward validation checks a small write set against the read sets of active transactions. We see that backward validation has the overhead of storing old write sets until they are no longer needed. On the other hand, forward validation has to allow for new transactions starting during the validation process.

Starvation • When a transaction is aborted, it will normally be restarted by the client program. But in schemes that rely on aborting and restarting transactions, there is no guarantee that a particular transaction will ever pass the validation checks, for it may come into conflict with other transactions for the use of objects each time it is restarted. The prevention of a transaction ever being able to commit is called starvation. Occurrences of starvation are likely to be rare, but a server that uses optimistic concurrency control must ensure that a client does not have its transaction aborted repeatedly.

TIME STAMP ORDERING :-

In concurrency control schemes based on timestamp ordering, each operation in a transaction is validated when it is carried out. If the operation cannot be validated, the transaction is aborted immediately and can then be restarted by the client. Each transaction is assigned a unique timestamp value when it starts. The timestamp defines its position in the time sequence of transactions. Requests from transactions can be
totally ordered according to their timestamps. The basic timestamp ordering rule is based on operation conflicts and is very simple:

A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction.

Figure 16.29 Operation conflicts for timestamp ordering

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$. This requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$. This requires that $T_c >$ the write timestamp of the committed object.

The *write* operations may be performed after the *closeTransaction* operation has returned, without making the client wait. But the client must wait when *read* operations need to wait for earlier transactions to finish. This cannot lead to deadlock, since transactions only wait for earlier ones (and no cycle could occur in the wait-for graph).
Timestamps may be assigned from the server's clock or, as in the previous section, a 'pseudo-time' may be based on a counter that is incremented whenever a timestamp value is issued.

Every object has a write timestamp and a set of tentative versions, each of which has a write timestamp associated with it; each object also has a set of read timestamps. The write timestamp of the (committed) object is earlier than that of any of its tentative versions, and the set of read timestamps can be represented by its maximum member. Whenever a transaction's *write* operation on an object is accepted, the server creates a new tentative version of the object with its write timestamp set to the transaction timestamp. A transaction's *read* operation is directed to the version with the maximum write timestamp less than the transaction timestamp. Whenever a transaction's *read* operation on an object is accepted, the timestamp of the transaction is added to its set of read timestamps. When a transaction is committed, the values of the
tentative versions become the values of the objects, and the timestamps of the tentative versions become the timestamps of the corresponding objects.

In timestamp ordering, each request by a transaction for a *read* or *write* operation on an object is checked to see whether it conforms to the operation conflict rules. A request by the current transaction *Tc* can conflict with previous operations done by other transactions, *Ti*, whose timestamps indicate that they should be later than *Tc*.

Timestamp ordering write rule:
If a tentative version with write timestamp *Tc* already exists, the *write* operation is addressed to it; otherwise, a new tentative version is created and given write timestamp *Tc*. Note that any *write* that 'arrives too late' is aborted – it is too late in the sense that a transaction with a later timestamp has already read or written the object.

Timestamp ordering write rule:
If transaction *Tc* has already written its own version of the object, this will be used.

• A *read* operation that arrives too early waits for the earlier transaction to complete. If the earlier transaction commits, then *Tc* will read from its committed version. If it aborts, then *Tc* will repeat the read rule (and select the previous version). This rule prevents dirty reads.
• A *read* operation that 'arrives too late' is aborted – it is too late in the sense that a transaction with a later timestamp has already written the object

When a coordinator receives a request to commit a transaction, it will always be able to do so because all the operations of transactions are checked for consistency with those of earlier transactions before being carried out. The committed versions of each object must be created in timestamp order. Therefore, a coordinator sometimes needs to wait for earlier transactions to complete before writing all the committed versions of the objects accessed by a particular transaction, but there is no need for the client to wait.

Timestamp ordering algorithm is a strict one – it ensures strict executions of transactions. The timestamp ordering read rule delays a transaction's *read* operation on any object until all transactions that had previously written that object have committed or aborted. The arrangement to commit versions in order ensures that the execution of a transaction's *write* operation on any object is delayed until all transactions that had previously written that object have committed or aborted.

Comparison of methods for concurrency control

The timestamp ordering method is similar to two-phase locking in that both usepessimistic approaches in which conflicts between transactions are detected as each object is accessed. On the one hand, timestamp ordering decides the serialization order statically – when a transaction starts. On the other hand, two-phase locking decides the serialization order dynamically – according to the order in which objects are accessed.

Timestamp ordering, and in particular multiversion timestamp ordering, is better than strict two-phase locking for read-only transactions. Two-phase locking is better when the operations in transactions are predominantly updates. Some work uses the observation that timestamp ordering is beneficial for transactions with predominantly *read* operations and that locking is beneficial for transactions with more *writes* than *reads* as an argument for allowing hybrid schemes in which some transactions use timestamp ordering and others use locking for concurrency control

When optimistic concurrency control is used, all transactions are allowed to proceed, but some are aborted when they attempt to commit, or in forward validation transactions are aborted earlier. This results in relatively efficient operation when there are few conflicts, but a substantial amount of work may have to be repeated when a transaction is aborted.

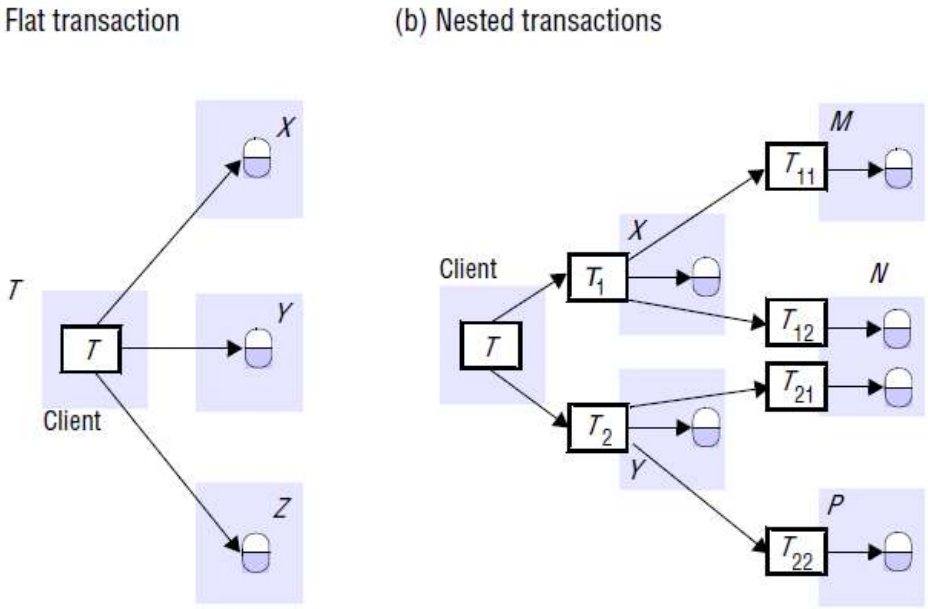
For Example Dropbox
uses an optimistic form of concurrency control, keeping track of consistency and preventing clashes between users' updates – which are at the granularity of whole files. Thus if two users make concurrent updates to the same file, the first write will be accepted and the second rejected. However, Dropbox provides a version history to enable users to merge their updates manually or restore previous versions.

Distributed Transactions:-

Distributed transactions – those that involve more than one server. Distributed transactions may be either flat or nested.
A client transaction becomes distributed if it invokes operations in several different servers. There are two different ways that distributed transactions can be structured: as flat transactions and as nested transactions
.
In a flat transaction, a client makes requests to more than one server. For example, in Figure 17.1(a), transaction *T* is a flat transaction that invokes operations on objects in servers *X*, *Y* and *Z*. A flat client transaction completes each of its requests before going on to the next one. Therefore, each transaction accesses servers' objects sequentially. When servers use locking, a transaction can only be waiting for one object at a time.

In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting.

Figure 17.1 Distributed transactions



Atomic commit protocols

In the case of a distributed transaction, the client has requested operations at more than one server. A transaction comes to an end when the client requests that it be committed or aborted. A simple way to complete the transaction in an atomic manner is for the 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 have carried it out. This is an example of a *onephase atomic commit protocol*. This simple one-phase atomic commit protocol is inadequate, though, because it does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit. Reasons that prevent a server from being able to commit its part of a transaction generally relate to issues of concurrency control

The *two-phase commit protocol* is designed to allow any participant to abort its part of a transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must be aborted.

Two – Phase Commit Protocol

In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit; in the second, it tells them to commit (or abort) the transaction. If a participant can commit its part of a transaction, it will agree as soon as it has recorded the changes it has made (to the objects) and its status in permanent storage and is therefore prepared to commit. The coordinator in a distributed transaction communicates with the participants to carry out the two-phase commit protocol by means of the operations

Figure 17.4 Operations for two-phase commit protocol

canCommit?(trans)→ Yes / No
Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)
Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)
Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)
Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) → Yes / No
Call from participant to coordinator to ask for the decision on a transaction when it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

The two-phase commit protocol consists of a voting phase and a completion phase, as shown in Figure 17.5. By the end of step 2, the coordinator and all the participants that voted *Yes* are prepared to commit. By the end of step 3, the transaction is effectively completed. At step 3a the coordinator and the participants are committed, so the coordinator can report a decision to commit to the client. At 3b the coordinator reports a decision to abort to the client. At step 4 participants confirm that they have committed so that the coordinator knows when the information it has recorded about the transaction is no longer needed

Figure 17.5 The two-phase commit protocol

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.

Performance of the two-phase commit protocol • Provided that all goes well – that is, that the coordinator, the participants and the communications between them do not fail – the two-phase commit protocol involving *N* participants can be completed with *N canCommit?* messages and replies, followed by *N doCommit* messages. That is, the cost in messages is proportional to 3*N*, and the cost in time is three rounds of messages. The *haveCommitted* messages are not counted in the estimated cost of the protocol, which can function correctly without them – their role is to enable servers to delete stale coordinator information.

In the worst case, there may be arbitrarily many server and communication failures during the two-phase commit protocol. However, the protocol is designed to tolerate a succession of failures (server crashes or lost messages) and is guaranteed to complete eventually, although it is not possible to specify a time limit within which it will be completed.

Figure 17.7 Operations in coordinator for nested transactions

openSubTransaction(trans) → *subTrans*
Opens a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.

getStatus(trans)→ *committed, aborted, provisional*
Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed*, *aborted* or *provisional*.

Two-phase commit protocol for nested transactions

When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. A provisional commit is different from being prepared to commit: nothing is backed up in permanent storage. If the server crashes subsequently, its replacement will not be able to commit. After all subtransactions have completed, the provisionally committed ones participate in a two-phase commit protocol, in which servers of provisionally committed subtransactions express their intention to commit and those with an aborted ancestor will abort. Being prepared to commit guarantees a subtransaction will be able to commit, whereas a provisional commit only means it has finished correctly – and will probably agree to commit when it is subsequently asked to

.

A coordinator for a subtransaction will provide an operation to open a subtransaction, together with an operation enabling that coordinator to enquire whether its parent has yet committed or aborted A client starts a set of nested transactions by opening a top-level transaction with an *openTransaction* operation, which returns a transaction identifier for the top-level transaction. The client starts a subtransaction by invoking the *openSubTransaction* operation, whose argument specifies its parent transaction. The new subtransaction automatically *joins* the parent transaction, and a transaction identifier for a subtransaction is returned.

An identifier for a subtransaction must be an extension of its parent's TID, constructed in such a way that the identifier of the parent or top-level transaction of a subtransaction can be determined from its own transaction identifier. In addition, all subtransaction identifiers should be globally unique. The client makes a set of nested transactions come to completion by invoking *closeTransaction* or *abortTransaction* on the coordinator of the top-level transaction.

Meanwhile, each of the nested transactions carries out its operations. When they are finished, the server managing a subtransaction records information as to whether the subtransaction committed provisionally or aborted. Note that if its parent aborts, then the subtransaction will be forced to abort too.

Figure 17.9 Information held by coordinators of nested transactions

Coordinator of transaction	*Child transactions*	*Participant*	*Provisional commit list*	*Abort list*
T	T_1, T_2	yes	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	yes	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	no (aborted)		T_2
T_{11}		no (aborted)		T_{11}
T_{12}, T_{21}		T_{12} but not T_{21}*	T_{21}, T_{12}	
T_{22}		no (parent aborted)	T_{22}	

* T_{21}'s parent has aborted

Concurrency control in distributed transactions

Each server manages a set of objects and is responsible for ensuring that they remain consistent when accessed by concurrent transactions. Therefore, each server is responsible for applying concurrency control to its own objects

Locking

In a distributed transaction, the locks on an object are held locally (in the same server). The local lock manager can 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. When locking is used for concurrency control, the objects remain locked and are unavailable for other transactions during the atomic commit protocol, although an aborted transaction releases its locks after phase 1 of the protocol.
As lock managers in different servers set their locks independently of one another, it is possible that different servers may impose different orderings on transactions. When a deadlock is detected, a transaction is aborted to resolve the deadlock. In this case, the coordinator will be informed and will abort the transaction at the participants involved in the transaction.

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 timestamps. A globally unique transaction timestamp is issued to the client by the first coordinator accessed by a transaction. The transaction timestamp is passed to the coordinator at each server whose objects perform an operation in the transaction.

The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. The same ordering of transactions can be achieved at all the servers even if their local clocks are not synchronized. However, for reasons of efficiency it is required that the timestamps issued by one coordinator be roughly synchronized with those issued by the other coordinators. When this is the case, the ordering of transactions generally corresponds to the order in which they are started in real time.

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. Therefore any transaction that reaches the client request to commit should always be able to commit, and participants in the two-phase commit protocol will normally agree to commit. The only situation in which a participant will not agree to commit is if it has crashed during the transaction.

Optimistic concurrency control

In optimistic concurrency control, each transaction is validated before it is allowed to commit. Transaction numbers are assigned at the start of validation and transactions are serialized according to the order of the transaction numbers. A distributed transaction is validated by a collection of independent servers, each of which validates transactions that access its own objects. This validation takes place during the first phase of the two-phase commit protocol.

If parallel validation is used, transactions will not suffer from commitment deadlock. However, if servers simply perform independent validations, it is possible that different servers in a distributed transaction may serialize the same set of transactions in different orders – for example, with *T* before *U* at server *X* and *U* before *T* at server *Y*, in our example.

The servers of distributed transactions must prevent this happening. One approach is that after a local validation by each server, a global validation is carried out . The global validation checks that the combination of the orderings at the individual servers is serializable; that is, that the transaction being validated is not involved in a cycle.

Another approach is that all of the servers of a particular transaction use the same globally unique transaction number at the start of the validation The coordinator of the two-phase commit protocol is responsible for generating the globally unique transaction number and passes it to the participants in the *canCommit?* messages. As different servers may coordinate different transactions, the servers must (as in the distributed timestamp ordering protocol) have an agreed order for the transaction numbers they generate.

Distributed deadlocks

In a distributed system involving multiple servers being accessed by multiple transactions, a global wait-for graph can in theory be constructed from the local ones. There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a *distributed deadlock*. Recall that the wait-for graph is a directed graph in which nodes represent transactions and objects, and edges represent either an object held by a transaction or a transaction waiting for an object. There is a deadlock if and only if there is a cycle in the wait-for graph.
Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions. Local wait-for graphs can be built by the lock manager at each server
As the 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. A simple solution is to use centralized deadlock detection, in which 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 to the global deadlock detector, which amalgamates the information in the local graphs in order to construct a global wait-for graph. The global deadlock detector checks for cycles in the global wait-for graph.
When it finds a cycle, it makes a decision on how to resolve the deadlock and tells the servers which transaction to abort.

Centralized deadlock detection is not a good idea, because it depends on a single server to carry it out. It suffers from the usual problems associated with centralized solutions in distributed systems – poor availability, lack of fault tolerance and no ability to scale. In addition, the cost of the frequent transmission of local wait-for graphs is high. If the global graph is collected less frequently, deadlocks may take longer to be detected.

Phantom deadlocks • A deadlock that is 'detected' but is not really a deadlock is called a *phantom deadlock*. In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected.

Edge chasing • A distributed approach to deadlock detection uses a technique called *edge chasing* or *path pushing*. In this approach, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges.
The servers attempt to find cycles by forwarding messages called *probes*, which follow the edges of the graph throughout the distributed system. A probe message consists of transaction wait-for relationships representing a path in the global wait-for graph

Edge-chasing algorithms have three steps:

Initiation: When a server notes that a transaction *T* starts waiting for another transaction *U*, where *U* is waiting to access an object at another server, it initiates detection by sending a probe containing the edge < *T* *U* > to the server of the
object at which transaction *U* is blocked. If *U* is sharing a lock, probes are sent to all the holders of the lock. Sometimes further transactions may start sharing the lock later on, in which case probes can be sent to them too.

Detection: Detection consists of receiving probes and deciding whether a deadlock has occurred and whether to forward the probes

Resolution: When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

Transaction recovery

When a server is running it keeps all of its objects in its volatile memory and records its committed objects in a *recovery file* or files. Therefore recovery consists of restoring the server with the latest committed versions of its objects from permanent storage. Databases need to deal with large volumes of data. They generally hold the objects in stable storage on disk with a cache in volatile memory.
The requirements for durability and failure atomicity are not really independent of one another and can be dealt with by a single mechanism – the *recovery manager*.
The tasks of a recovery manager are:
• to save objects in permanent storage (in a recovery file) for committed transactions;
• to restore the server's objects after a crash;
• to reorganize the recovery file to improve the performance of recovery;
• to reclaim storage space (in the recovery file).
In some cases, we require the recovery manager to be resilient to media failures. Corruption during a crash, random decay or a permanent failure can lead to failures of the recovery file, which can result in some of the data on the disk being lost. In such cases we need another copy of the recovery file. Stable storage, which is implemented so as to be very unlikely to fail by using mirrored disks or copies at a different location may be used for this purpose.

Intentions list • Any server that provides transactions needs to keep track of the objects accessed by clients' transactions when a client opens a transaction, the server first contacted provides a new transaction identifier and
returns it to the client. Each subsequent client request within a transaction up to and including the *commit* or *abort* request includes the transaction identifier as an argument. During the progress of a transaction, the update operations are applied to a private set of tentative versions of the objects belonging to the transaction

At each server, an *intentions list* is recorded for all of its currently active transactions – an intentions list of a particular transaction contains a list of the references and the values of all the objects that are altered by that transaction. When a transaction is committed, that transaction's intentions list is used to identify the objects it affected.

The committed version of each object is replaced by the tentative version made by that transaction, and the new value is written to the server's recovery file. When a transaction aborts, the server uses the intentions list to delete all the tentative versions of objects made by that transaction. At the point when a participant says it is prepared to commit a transaction, its recovery manager must have saved both its intentions list for that transaction and the objects in that intentions list in its recovery file, so that it will be able to carry out the commitment later, even if it crashes in the interim.

When all the participants involved in a transaction agree to commit it, the coordinator informs the client and then sends messages to the participants to commit their part of the transaction. Once the client has been informed that a transaction has committed, the recovery files of the participating servers must contain sufficient information to ensure that the transaction is committed by all of the servers, even if some of them crash between preparing to commit and committing.

Entries in recovery file • To deal with recovery of a server that can be involved in distributed transactions, further information in addition to the values of the objects is stored in the recovery file. This information concerns the *status* of each transaction –whether it is *committed*, *aborted* or *prepared* to commit. In addition, each object in the recovery file is associated with a particular transaction by saving the intentions list in the recovery file

Logging

In the logging technique, the recovery file represents a log containing the history of all the transactions performed by a server. The history consists of values of objects, transaction status entries and transaction intentions lists. The order of the entries in the log reflects the order in which transactions have prepared, committed and aborted at that server. In practice, the recovery file will contain a recent snapshot of the values of all the objects in the server followed by a history of transactions postdating the snapshot.
During the normal operation of a server, its recovery manager is called whenever a transaction prepares to commit, commits or aborts a transaction. When the server is prepared to commit a transaction, the recovery manager appends all the objects in its intentions list to the recovery file, followed by the current status of that transaction (*prepared*) together with its intentions list. When a transaction is eventually committed or aborted, the recovery manager appends the corresponding status of the transaction to its recovery file. If the server fails, only the last write can be incomplete.

To make efficient use of the disk, several subsequent writes can be buffered and then written to disk as a single write. An additional advantage of the logging technique is that sequential writes to disk are faster than writes to random locations.

After a crash, any transaction that does not have a *committed* status in the log is aborted. Therefore when a transaction commits, its *committed* status entry must be *forced* to the log – that is, written to the log together with any other buffered entries. The recovery manager associates a unique identifier with each object so that the successive versions of an object in the recovery file may be associated with the server's objects.

Recovery of objects • When a server is replaced after a crash, it first sets default initial values for its objects and then hands over to its recovery manager. The recovery manager is responsible for restoring the server's objects so that they include all the effects of the committed transactions performed in the correct order and none of the effects of incomplete or aborted transactions.
The most recent information about transactions is at the end of the log. There are two approaches to restoring the data from the recovery file. In the first, the recovery manager starts at the beginning and restores the values of all of the objects from the most recent checkpoint (discussed in the next section). It then reads in the values of each of the objects, associates them with their transaction's intentions lists and for committed transactions replaces the values of the objects. In this approach, the transactions are replayed in the order in which they were executed and there could be a large number of them. In the second approach, the recovery manager will restore a server's objects by'reading the recovery file backwards'.

Reorganizing the recovery file • A recovery manager is responsible for reorganizing its recovery file so as to make the process of recovery faster and to reduce its use of space. If the recovery file is never reorganized, then the recovery process must search backwards through the recovery file until it has found a value for each of its objects.

Conceptually, the only information required for recovery is a copy of the committed version of each object in the server. This would be the most compact form for the recovery file. The name *checkpointing* is used to refer to the process of writing the current committed values of a server's objects to a new recovery file, together with transaction status entries and intentions lists of transactions that have not yet been fully resolved (including information related to the two-phase commit protocol). The term *checkpoint* is used to refer to the information stored by the checkpointing process. The
purpose of making checkpoints is to reduce the number of transactions to be dealt with during recovery and to reclaim file space. Checkpointing can be done immediately after recovery but before any new transactions are started. However, recovery may not occur very often. Therefore, checkpointing may need to be done from time to time during the normal activity of a server. The checkpoint is written to a future recovery file, and the current recovery file remains in use until the checkpoint is complete. Checkpointing consists of 'adding a mark' to the recovery file when the checkpointing starts, writing the server's objects to the future recovery file and then copying to that file (1) all entries before the mark that
relate to as-yet-unresolved transactions and (2) all entries after the mark in the recoveryfile. When the checkpoint is complete, the future recovery file becomes the recovery file. The recovery system can reduce its use of space by discarding the old recovery file.

Shadow versions

The logging technique records transaction status entries, intentions lists and objects all in the same file – the log. The *shadow versions* technique is an alternative way to organize a recovery file. It uses a *map* to locate versions of the server's objects in a file called a *version store*. The map associates the identifiers of the server's objects with the positions of their current versions in the version store. The versions written by each transaction are 'shadows' of the previous committed versions. As we shall see, the transaction status entries and intentions lists are stored separately. Shadow versions are described first.

When a transaction is prepared to commit, any of the objects changed by the transaction are appended to the version store, leaving the corresponding committed versions unchanged. These new as-yet-tentative versions are called *shadow* versions.

When a transaction commits, a new map is made by copying the old map and entering the positions of the shadow versions.

To complete the commit process, the new map replaces the old map. To restore the objects when a server is replaced after a crash, its recovery manager reads the map and uses the information in the map to locate the objects in the version store.

The switch from the old map to the new map must be performed in a single atomic step. To achieve this it is essential that stable storage is used for the map, so that there is guaranteed to be a valid map even when a file write operation fails. The shadow versions method provides faster recovery than logging because the positions of the current committed objects are recorded in the map, whereas recovery from a log requires searching throughout the log for objects. Logging should be faster than shadow versions during the normal activity of the system, though. This is because logging requires only a sequence of append operations to the same file, whereas shadow versions require an additional stable storage write (involving two unrelated disk blocks).

Shadow versions on their own are not sufficient for a server that handles distributed transactions. Transaction status entries and intentions lists are saved in a file called the *transaction status file*. Each intentions list represents the part of the map that will be altered by a transaction when it commits. The transaction status file may, for example, be organized as a log.

There is a chance that a server may crash between the time when a *committed* status is written to the transaction status file and the time when the map is updated – in which case the client will not have been acknowledged. The recovery manager must allow for this possibility when the server is replaced after a crash, for example by checking whether the map includes the effects of the last committed transaction in the transaction status file. If it does not, then the latter should be marked as aborted.