DISTRIBUTED SYSTEMS

[R20A0520]

DIGITAL NOTES

B.TECH III YEAR–I SEM(R20)

DEPARTMENT OF INFORMATION TECHNOLOGY

2022-2023

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-YEARI-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, Globalstates.

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

DISTRIBUTED SYSTEMS

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 Centralised 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 doesnot affect others)

Application of DS :

- Tele communication network(telephonen/w,cellularn/w,computern/w)
- Network Applications (WWW, onlineapps, n/wfilesystems, bankingsystems)
- Real-time process control systems(aircraftcontrolsystems)
- Parallel computation(grid computing, clustercomputing)

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:Internetserviceprovider:companiesthatprovidemodemandotherfacilitiestousersandorganizationswhichenable them to access services anywhere in the internet.

Intranet–sub networks operated by companies and other organizations.

Backbone–
linksintranets.Itisan/wlinkwithhightransmissioncapacityandemployssatellitecommunication, 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 beconfigured to enforce local security policies.It is composed of severalLAN's linked by backboneconnections. An Intranet is connected to the Internet via a router, which allows the users inside theintranet to make use of services. It also allows the users in other intranets to access its services. Firewallprotectsan Intranetbypreventingunauthorized messagesleavingorenteringusingfilteringmethod.

3. Mobile&UbiquitousComputing:

Technological advances in device miniaturization and wireless networking have led to the integration ofsmall and portable computing devices into distributed systems (laptops, phones, PDS's, wearable devicesetc)
Mobilecomputingistheperformanceofcomputingtaskswhile theuser isonthe move.

Ubiquitous computing is the harnessing of many small, cheap computational devices present in usersenvironment.Devices become pervasivein everydayobjects.

ResourceSharing:

- Resource sharing is the primary motivation of distributed computing
- Resources types
  - Hardware ,e.g.printer, scanner,camera
  - Datasources, e.g.file, database,webpage
  - Specificresources, e.g. search engine
- Service
  - Managesacollectionofrelatedresourcesandpresentstheirfunctionalitiestousersand applications
- Server
  - aprocessonnetworkedcomputerthatacceptsrequestsfromprocessesonothercomputers to performa serviceandrespondsappropriately
- Client
  - therequestingprocess
- Communicationisthrough messagepassingor Remoteinvocation

Manydistributedsystemscanbeconstructedinthe formofinteracting clientsandservers.Ex:WWW,Email,Networkedprintersetc.

WebBrowser–clientwhichcommunicateswithwebservertorequestwebpages.

World Wide Web:

WWWisanevolvingsystemforpublishingandaccessingresourcesandservicesacrosstheInternetusingwebbrowsers.

Weboriginated at Europeancentrefornuclearresearch,Switzerlandin1989.Documentsexchanged contain hyperlinks.

Web is an open system. Its operation is based on communication standards and document standards.

Initiallywebprovideddataresourcesbutnowincludesservicesalso.Webisbasedonthreemainstandardtechnological components:

1. HTML: hyper text markup language for specifying contents and layouts of pages.
2. URL:uniform resource locator which identifies documents and other resources stored as part of web.
3. A client-server architecture with standard rules for interaction(HTTP)bywhichbrowsersandclients fetch documents and other resources from web servers.

HTML: used to specify the text and images that make up the contents of a web page and to specify how they are laid out and formatted for presentation to the user. 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:Itspurposeistoidentifyaresource.Ithastwotop-level components:

Scheme:Scheme-specific-identifier

(typeof URLieftp,http) (specific info to be retrieved ie www.abc.net/--.html)HTTPURL'sare mostwidelyused.

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 thattheweb server canaccess.

HTTP: defines the ways in which browsers and other types of client interact with web servers.Features:Request-replyinteractions,contenttypes,oneresourceperrequest, simpleaccess control.

DynamicPages:Aprogramthatwebserversruntogeneratecontentfortheirclientsisreferredtoasa Common GatewayInterface(CGI)program.

XML –designedasawayofrepresentingdatainstandard,structured,application-specificforms.Itisusedto describe the capabilities of devices and to describe personal info held about users. The web of linkedmetadataresources 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 betransferredfromonecomputertoanother and run atthedestination–Javaappletsareanexample.

The*virtualmachine*approachprovidesawayofmakingcodeexecutableonavarietyofhostcomputers: the compiler for a particular language generates code for a virtual machine instead ofparticularhardwareordercode.Forexample,theJavacompilerproduces codeforaJavavirtualmachine,whichexecutes itbyinterpretation.

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.Securityforinformationresourceshasthreecomponents: confidentiality (protection against disclosure to unauthorized individuals), integrity(protection againstalteration or corruption), and availability (protection against interference with the means to access theresources).

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 smallintranet to the Internet. A system is described as *scalable* if it will remain effective when there is asignificant increase in the number of resources and the number of users. The design of scalabledistributedsystems presents the followingchallenges:

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 produceincorrect results or may stop before they have completed the intended computation. Failures in adistributedsystemarepartial– thatis,somecomponentsfailwhileotherscontinuetofunction.Thereforethe handlingoffailuresisparticularlydifficult.

Detecting failures: Some failures can be detected. For example, checksums can be used to detectcorrupteddata inamessageorafile.

Masking failures: Some failures that have been detected can be hidden or made less severe. Twoexamplesofhidingfailures:

Messages can be retransmitted when they fail to arrive.

File data can be written to a pair of disks so

that ifoneiscorrupted, the otherwill bethere.

Tolerating failures: For example, when a web browser cannot contact a web server, it does not makethe user wait for ever while it keeps on trying– it informs the user about the problem, leaving themfreeto tryagain later.

Recovery from failures: Recovery involves the design of software so that the state of permanent datacanberecovered or‘rolledback’afteraserver hascrashed.

Redundancy: Services can bemadetotoleratefailures bythe useofredundantcomponents.

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 sametime. Therefore services and applications generally allow multiple client requests tobe processedconcurrently. 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 theseparation of components in a distributed system, so that the system is perceived as a wholerather thanasacollection ofindependentcomponents.Thevarious formsof transparencyare:

Access transparency enables local and remote resources to be accessed using identical operations.*Locationtransparency*enablesresourcestobeaccessedwithoutknowledgeoftheirphysicalornetworklocation (forexample, which buildingorIP address).

Concurrency transparency enablesseveralprocessestooperateconcurrentlyusingsharedresourceswithout interferencebetween them.

Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas byusers orapplicationprogrammers.

*Failuretransparency*enablestheconcealmentoffaults, allowing users and application programs to complete their tasks despite the failure of hardware or softwarecomponents.

Mobilitytransparency allows the movement of resources and clientswithin a system without affecting the operation of users or programs.

Performance transparency allows the system to be reconfigured to improve performance as loadsvary.*Scaling transparency* allows the system and applications to expand in scale without change to thesystemstructureortheapplication 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 relevantaspect of distributed system design:

*Physicalmodels*arethemostexplicitwayinwhichtodescribeasystem; they capture the

hardware composition of a system in terms of the computers (and other devices, such as mobile phones)and their inter connectingnetworks.

Architectural models describe a system in terms of the computational and communication tasksperformedbyitscomputationalelements; the computationalelementsbeingindividualcomputersoraggregatesofthemsupportedbyappropriate network inter connections.

Fundamental models take an abstract perspective in order to examine individual aspects of adistributed system. The fundamental models that examine three important aspects of distributedsystems: *interaction models*, which consider the structure and sequencing of the communicationbetween the elements of the system; *failure models*, which consider the ways in which a systemmay fail to operate correctly and; *security models*, which consider how the system is protected against attempts to interfere with itscorrect operationor to stealits data.

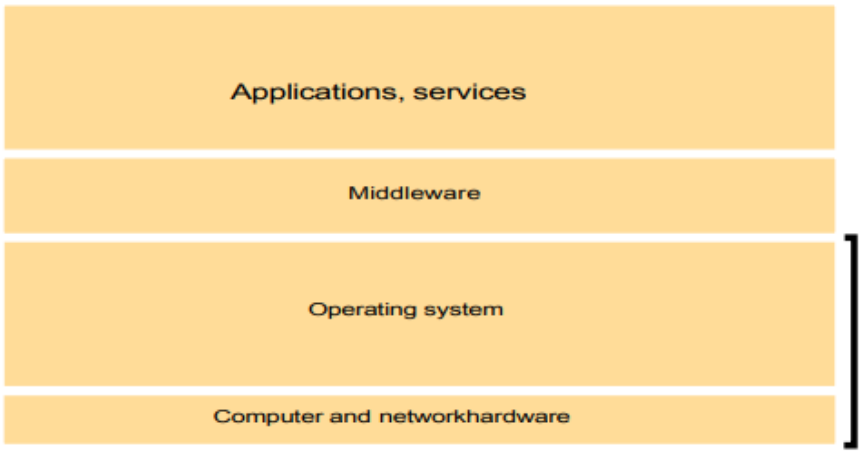
1. Architectural models

Architecture models define the way in which the components of systems interact with one anotherand how they are mapped onto the network. The architecture of a systemis its structure in termsof separately specified components and their interrelationships. The overall goal is to ensure thatthestructurewill meet presentand likelyfuturedemands on it.

Software layers

In alayeredapproach,a complex system is partitioned into a number of layers, withagivenlayermaking use of the services offered by the layer below. In terms of distributed systems, thisequates to a vertical organization of services into service layers. Given the complexity ofdistributed systems, it is often helpful to organize such services into layers. the important terms*platform*and *middleware*, which define asfollows:

A platform for distributed systems and applications consists of the lowest-level hardware andsoftware layers. These low-level layers provide services to the layers above them, which areimplementedindependentlyineachcomputer, bringing the system'sprogramminginterfaceuptoalevelthat facilitates communicationandcoordinationbetween processes.



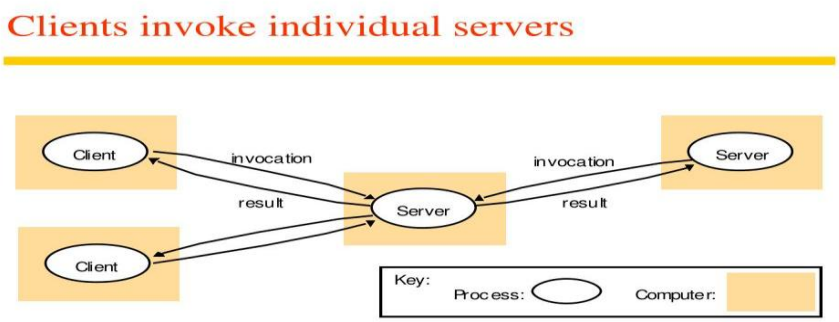
There are two main architectural models:

1. Client-ServerModel
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 servicesfromaserver.

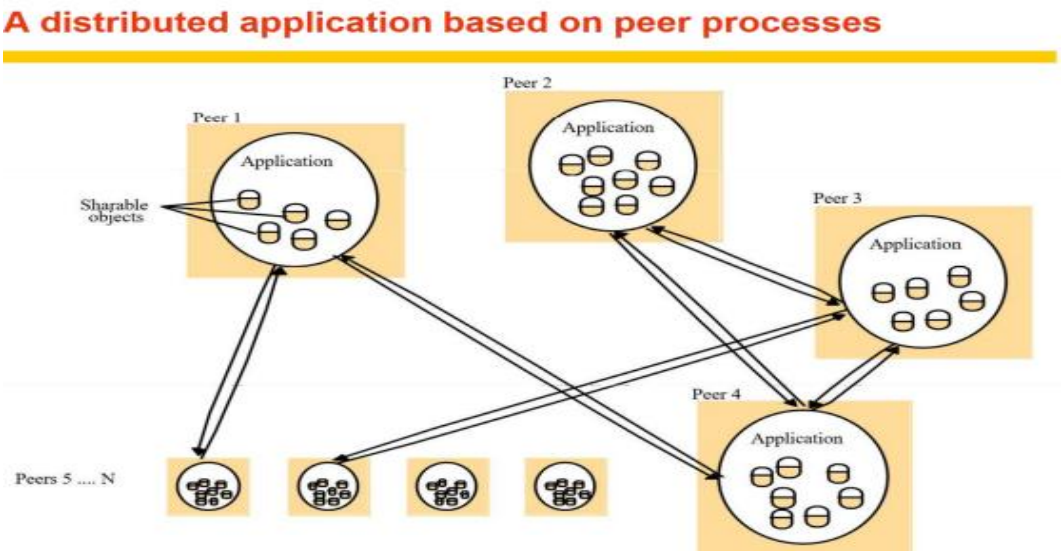
Servers may in turn be clients of other servers, as the figure indicates. For example, a web serveris often a client of a localfile server that manages the files in which the web pagesare stored.ses.

Clients invoke individual servers



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

Peer-to-peer: In this architecture all of the processes involved in a task or activity play similarroles,interacting cooperatively as *peers* without anydistinction between client and server processes or the computers on which they run. In practical terms, all participating processes runthe same program and offer the sameset of interfaces to eachother.While the client-servermodel offers a direct and relatively simple approach to the sharing of data and other resources, itscales poorly. Enables hundreds of computers to provide access to resources they share andmanage. Each object is replicated in several computers. Ex: Napster app for sharing digital musicfiles.



Several variations on the above models can be derived:

1. **Multiple-Servers Model**: In this services are provided by multiple servers. Services can beimplementedas several server processesinseparate host computers.

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

3. MobileCode:

a) Client requests resultsinthedownloadingofappletcode

b) Appletsareawell-knownandwidelyusedexampleofmobilecode.Itisdownloadedfromaweb server andexecutedlocallyresultingin good interactive response.

4. **MobileAgent**:

Amobileagentisarunningprogramthattravelsfromonecomputertoanotherinnetworkcarrying out a task on someones behalf

5. Network Computers:

Networkcomputer

Remotefileserver

Client

network

OS and Files

Networkcomputer:

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 software management and maintenance cost.

6. Thin Client:

Asoftwarelayerthatsupportsawindowbasedinterfaceonacomputerthatislocaltotheuserwhile executingapplication programs onacomputer server

Design requirements for distributed architectures:

1. Performance Issues

2. Quality of Service

3. Use of cache and replication

PerformanceIssues

Responsiveness

Delay,responsetime, slowdown,stretchfactor

Determinedbyloadandperformanceoftheserverandthenetwork,andbydelaysinallsoftwarecompo nentsinvolved

Throughput

Therate atwhichcomputationalworkof theserverordata transferofthenetworkisdone

Load balancing/loadsharing

Enableapplicationsandservice processestoproceedconcurrentlyandexploittheavailableresource

3. Fundamental Models

Modelsofsystemssharesomefundamentalproperties.Inparticular,allof themare composedofprocessesthatcommunicatewithoneanotherbysendingmessages over acomputer 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 those assumptions.The aspects of distributed systems that we wish to capture in our fundamental models are intended to helpustodiscussandreason about:

Interaction: Computation occurs within processes; the processes interact by passing messages,resulting in communication (informationflow) and coordination (synchronization and orderingofactivities)between processes

Failure: The correct operation of a distributed system is threatened whenever a fault occurs inany of the computers on which it runs (including software faults) or in the network that connectsthem.Ourmodel definesand classifies the faults.

Security: The modular nature of distributed systems and their opennessexposes them to attackby both externaland internal agents.Our security model defines andclassifies the formsthatsuch attacks may take, providing a basis for the analysis of threats to a system and for the designofsystems thatareableto resist them.

There are three Fundamental Models:

a) Interactionmodel

Fundamentallydistributedsystemsarecomposedofmanyprocesses, interacting in complex ways.For example:

- Multipleserver processesmaycooperatewithoneanotherto provide aservice;
- A set of peerprocesses may cooperate with one another to achieve a common goal;Two significant factors affecting interacting processes in adistributed system:
- Communication performance is often a limiting characteristic.
- It is impossible to maintain a single global notion of time.

Performanceofcommunicationchannels•Communicationoveracomputernetworkhasthefollowingperformancecharacteristics relatingtolatency, bandwidth and jitter:

The delay between the start of a message's transmission from one process and the beginningofitsreceipt byanother is referred to as*latency*. The latency includes:

- The time taken for the firstof a stringof bits ransmitted throughanetworktor each

its

- Destination. For example, the latency for the transmission of a message through a satellite linkisthe timeforaradio signals to travel tothesatellite and back.

- The*bandwidth*ofacomputernetworkisthetotalamountofinformationthatcanbetrans mitted over it in a given time. When a large number of communication channels are usingthesamenetwork, theyhaveto sharetheavailablebandwidth.

- *Jitter* is the variation in the time taken to deliver a series of messages. Jitter is relevant tomultimedia data. For example,if consecutive samplesof audio data areplayed with differingtimeintervals, thesoundwillbebadlydistorted.

Computer clocks and timing events • Each computer in a distributed systemhas its owninternal 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 theirevents. However, even if the two processes read their clocks at the same time, their local clocksmay 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 therateatwhichacomputer clockdeviatesfromaperfect referenceclock.Eveniftheclocksonall

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 distributed system is one in which the following bound are defined:

- Thetimeto executeeachstepofaprocess hasknown lowerand upperbounds.

- Eachmessagetransmittedoverachannelisreceivedwithin aknownboundedtime.

- Each process has a local clock whose drift rate from real time has a known bound.***Asynchronousdistributedsystems***:makesnoassumptionoftime.Anasynchro nousdistributedsystemisoneinwhich therearenobounds on:

- Process execution speeds–for example, one process step may take only a Pico second and another a century; all that can be said is that each step may take an

arbitrarily long time.

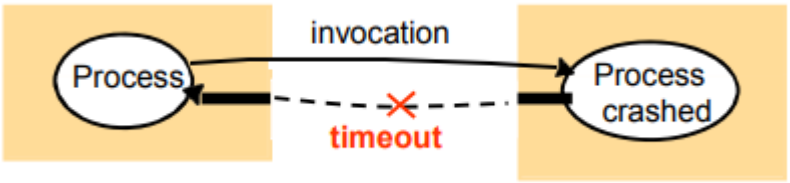
- Message transmission delays – for example, one message from process A to process B may be delivered in negligible time and another may take several years. In other words, a message may be received after an arbitrarily long time.
- Clock drift rates–again; the drif trat of a clock is arbitrary.

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 be slow; 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 have crashed 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 out going 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 dataitems, oritmay return a wrong value in responseto an invocation.

An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takesunintended 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 deliverytime 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 requirer edundant hardware.

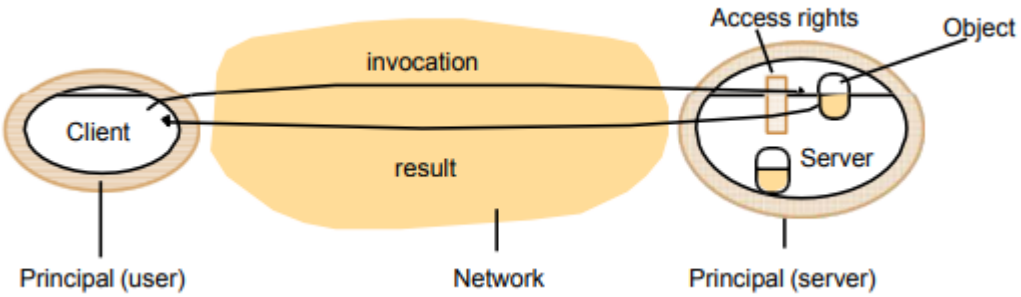
c) Security model

The security of a distributed system can be achieved by securing the processes and the channelsusedfortheirinteractionsandbyprotectingtheobjectsthattheyencapsulateagainstunauthorizedaccess.

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

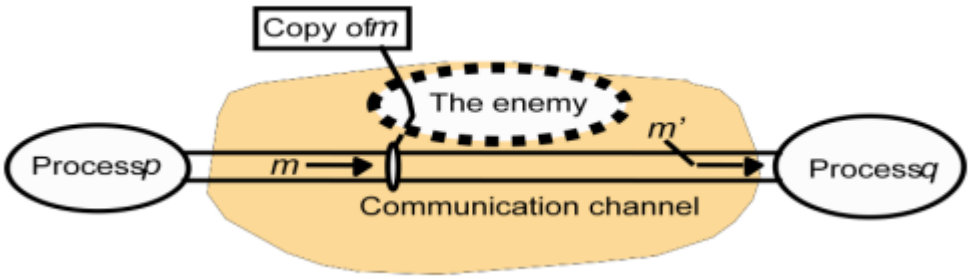
Protecting objects:

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 datasuchaswebpages.Tosupportthis, *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 messagesent between apairofprocesses,asshowninthe followingfigure.Theattackmay

Come from a computer that is legitimatelyconnectedtothenetworkorfromonethatisconnected in an unauthorized manner. The threats from a potential enemy include *threats toprocesses*and *threats tocommunicationchannels*.



Defeating security threats

*Cryptography*isthescienceofkeepingmessagessecure, and *encryption* is the process of scrambling a message in such away as to hide its contents. Modern cryptographyisbased on

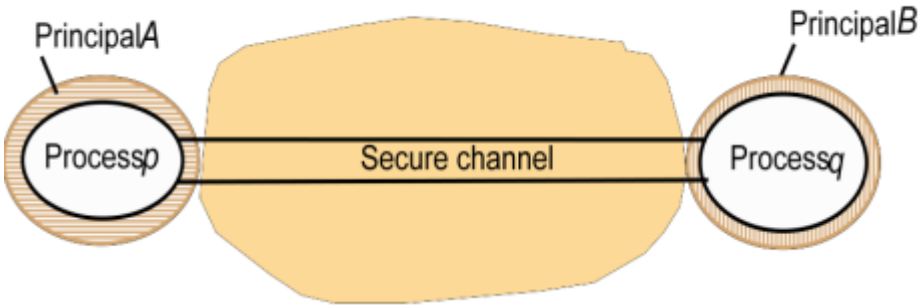
Encryption algorithms that use secret keys–large numbers that are difficulttoguess–totransform data in a manner that can only be reversed with knowledge of the corresponding decryption key.

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:

- Eachoftheprocessesknowsreliablytheidentityoftheprincipalonwhosebehalftheoth erprocessisexecuting.
- 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 messages from being replayed or reordered.



UNIT II

Time and Global States- Introduction-Clocks, events and process states-Synchronizing physical clocks-Logical time and logical clocks-Global states-Distributed debugging.

Coordination and Agreement-Introduction- Distributed mutual exclusion-Elections-Multicast communication-Consensus and related problems.

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 process p_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 clock is 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.

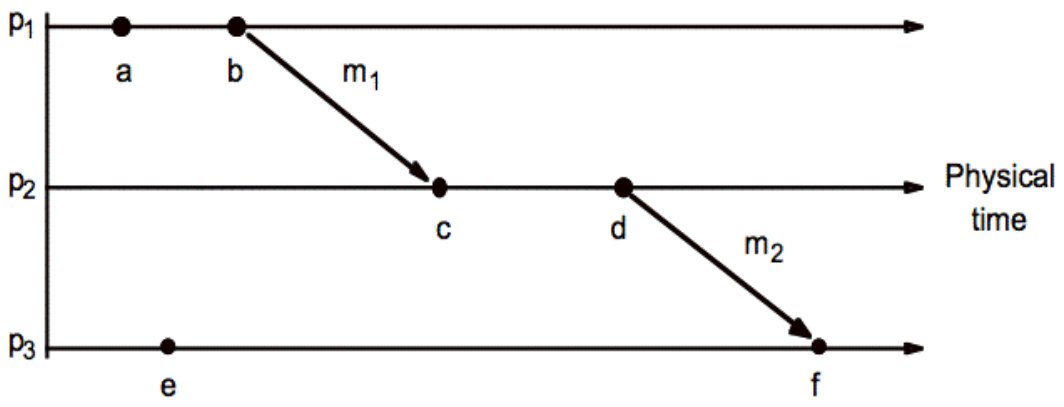
Logical Time

Physical time cannot be perfectly synchronized. Logical time provides a mechanism to define the *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 happened before 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 relationships the *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 need bear no particular relationship to any physical clock. Each process pi keeps its own logical clock, 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.

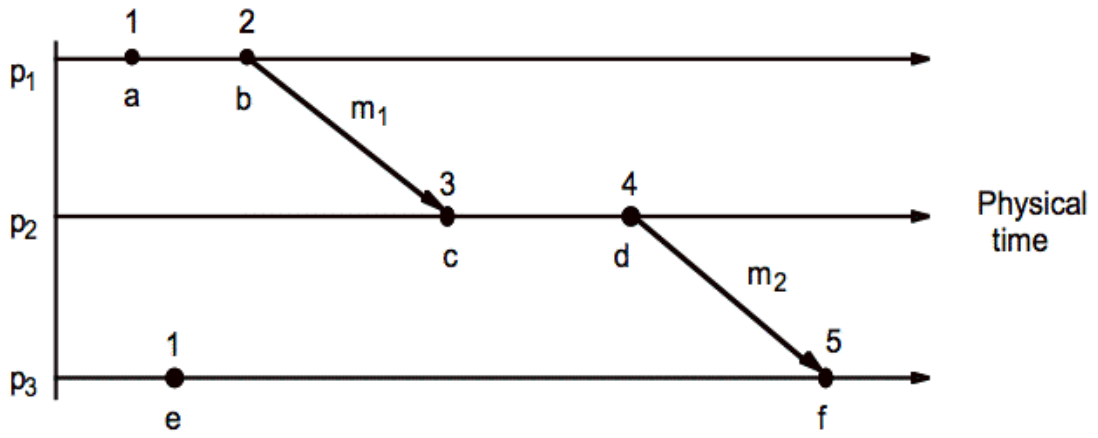
LC2:

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

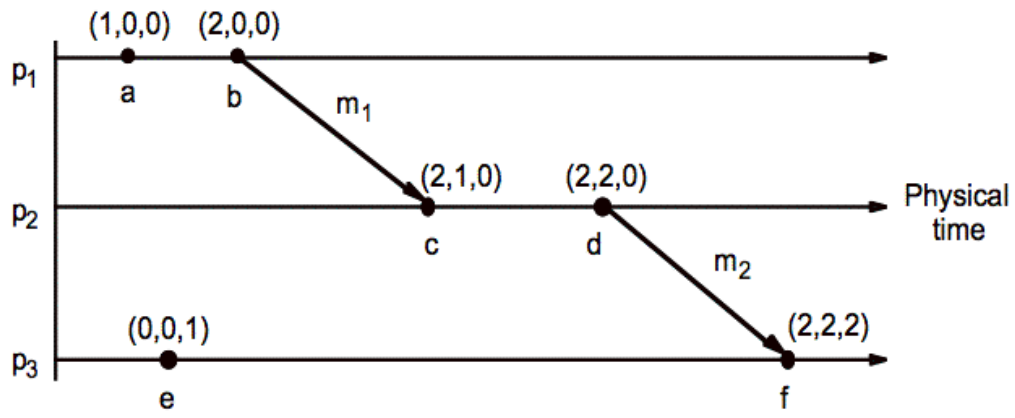
VC2: Just before pi timestamps an event, it sets Vi[i]:=Vi[i]+1.

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 as a 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, ..., N

V < 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 and determine 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 ever true.

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

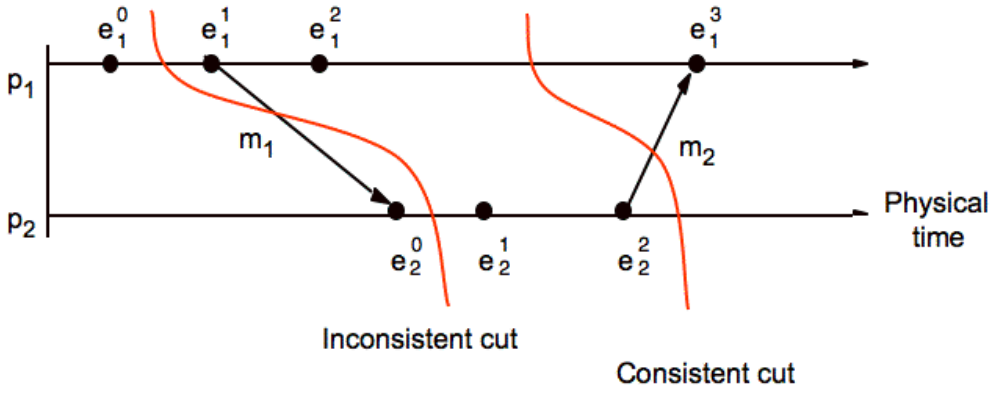
Because physical time cannot be perfectly synchronized in a distributed system it is not possible to gather the global state of the system at a particular time. Cuts provide the ability to "assemble a meaningful global state from local states recorded at different times".

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 records the 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 = $h_0 \cup h_1 \cup h_2 \cup ... \cup h_{N-1}$
- 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)."

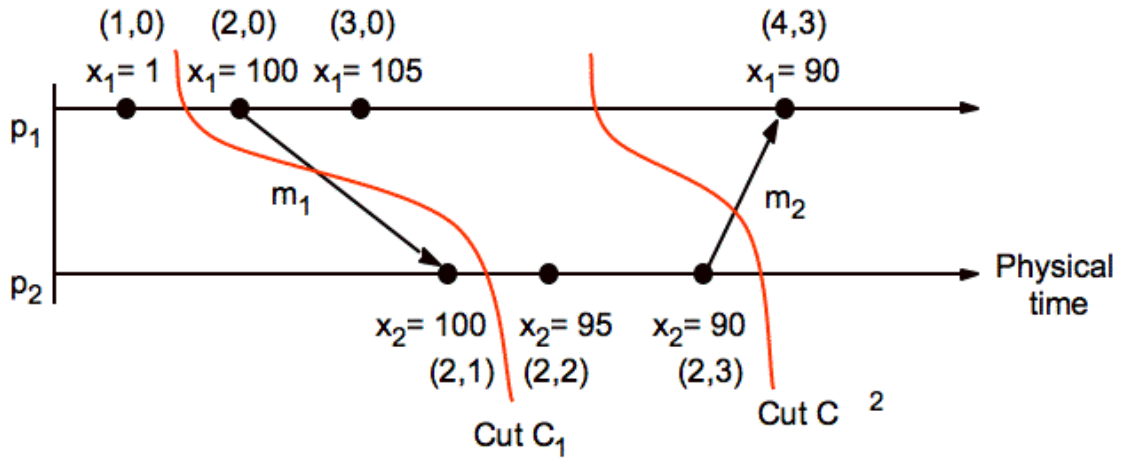
The algorithm we'll discuss is a centralized algorithm that determines post hoc whether the safety condition was ever violated. The processes in the system, p_1, p_2, ..., p_N, send their states to a passive monitoring process, p_0. p_0 is not part of the system. Based on the states collected, p_0 can 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. Then it 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 agree on 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, all must 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 can send 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 agreeing which 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 and updated by one computer at a time.

Failure Assumptions and Detection

Coordination in a synchronous system with no failures is comparatively easy. We'll look at some 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 byzantine failure. 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, it may 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 maximum delay in message delivery) and if it does not hear from *q* then it knows that *q* has failed. In an asynchronous 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, then entry 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 a critical 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 process exits 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

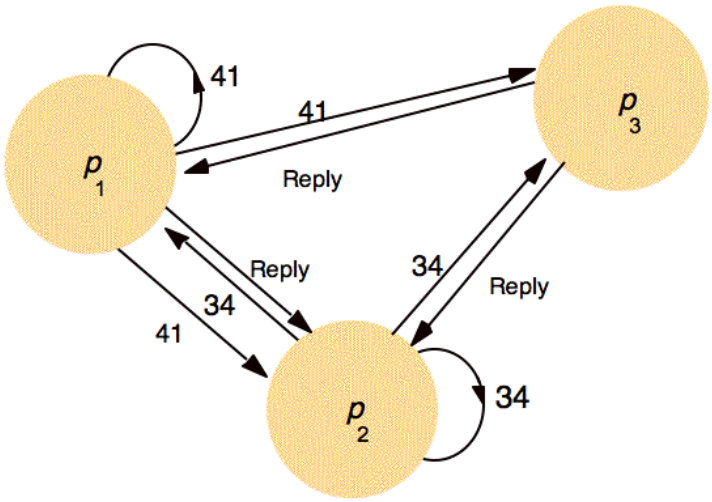
When a message is received from another process, it does the following:

- 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 failure and has problems of scale.

None of the algorithms discussed are appropriate for a system in which failures may occur. In order 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 election algorithm are as follows:

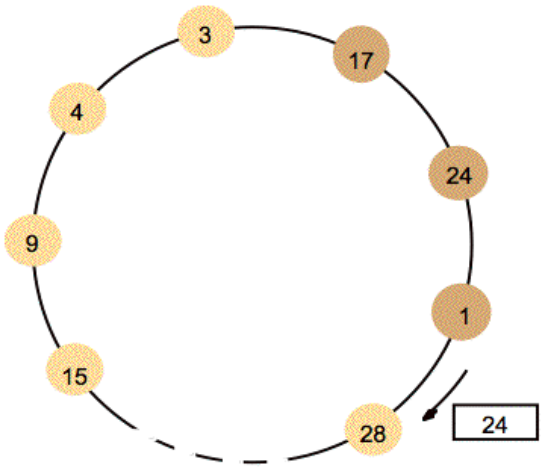
- Safety: Only one process is chosen -- the one with the largest identifying value. The value could be load, uptime, a random number, etc.
- Liveness: All process eventually chooses a winner or crash.

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, a process 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 then sends 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



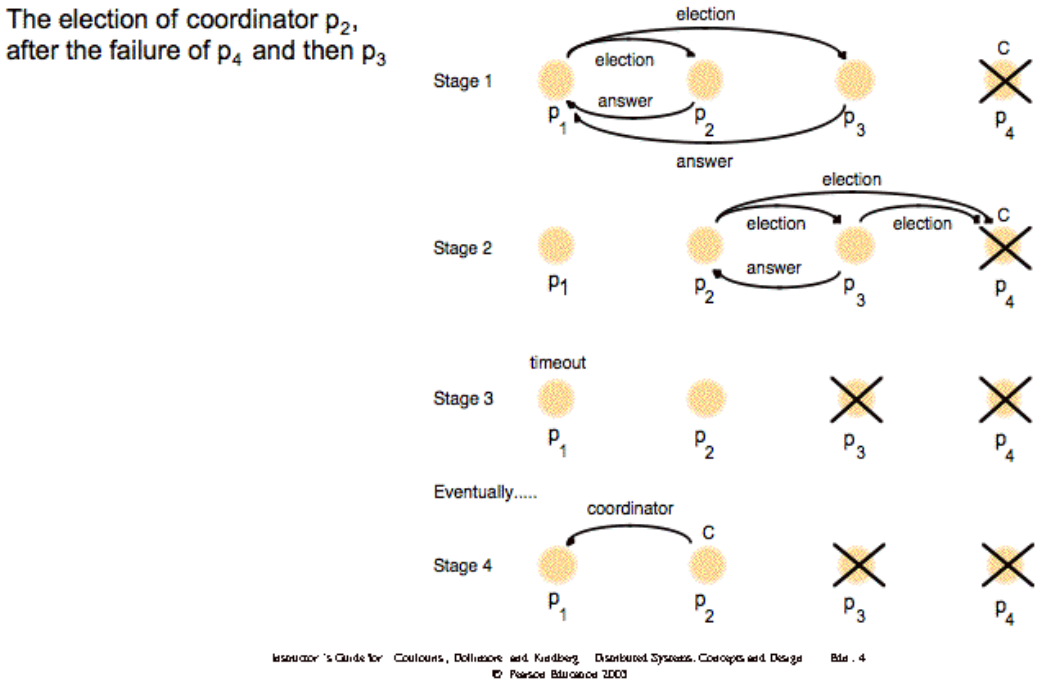
Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

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 there are 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 process receives 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



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

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 is reliable, 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 the presence of 1 crash failure. That means that we can design systems that reach consensus most of 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) as failed.
- Randomization - Use randomized behavior to confuse byzantine processes.

UNIT-III

INTERPROCESSCOMMUNICATION:

Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize the inactions.

The characteristics of inter process communication:

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

Synchronous and asynchronous communication:

A queue is associated with each message destination. Sending processes cause messages to beaddedtoremotequeuesandreceivingprocessesremovemessagesfromlocalqueues.Communication between the sending and receiving processes may be either synchronous or asynchronous.

Synchronous: In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case, both *send* and *receive* are *blocking* operations. Whenever a *send* is issued the sending process (or thread) is blocked until the corresponding *receive* is issued. Whenever a *receive* is issued by a process(or thread), it blocks until a message arrives.

Asynchronous:

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

Message destinations in the Internet protocols, messages are sent to (*Internet address*, *local port*) pairs. A local port is a message destination within a computer, specified as an integer.

Aporthasexactlyonereceiverbutcanhavemanysenders.Processesmayusemultipleportsto

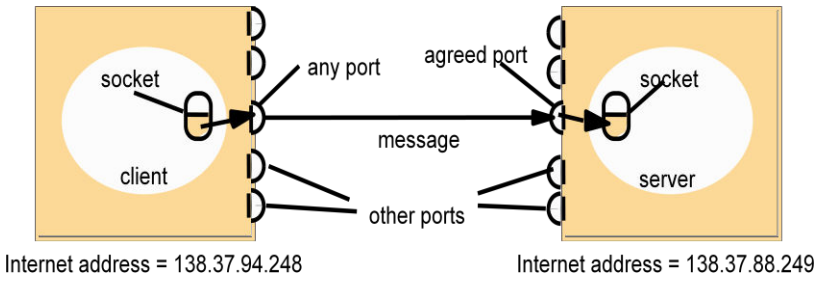
Receive messages. Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients.

Reliability in a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despitea'reasonable'numberofpackets being dropped or lost.

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

Sockets

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides anendpointforommunicationbetweenprocessesInterprocesscommunicationconsistsoftransmitting a message between a socket in one process and a socket in another process, is shown in the following figure.



For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Port. Each socket is associated with a particular protocol – either UDP or TCP.

UDP datagram communication

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

The *receive* method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

The following are some issues relating to datagram communication:

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

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

Timeouts: The *receive* that blocks forever is suitable for use by a server that is waiting to receiver quests from its clients. But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, time outs can be set on sockets.

Receive from any: The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin. The *receive* method returns the Internet address and local port of the sender, allowing the recipient tocheckwherethemessagecame from

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

Omission failures: Messages may be dropped occasionally, either because of checksum error or because no buffer space is available at the source or destination. *Ordering*: Messages can

sometimes be delivered out of sender order.

A reliable delivery service may be constructed from one that suffers from omission failures bytheuseofacknowledgements.

Use of UDP:

1. The Domain Name System, which looks up DNS names in the Internet, is implemented over UDP.
2. Voice over IP(VOIP) also runs over UDP.

TCP stream communication:

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

Lost messages: The TCP protocol uses an acknowledgement scheme. As an example of as impel scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message

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

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

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

JavaAPIforUDPdatagrams:

- The Java API provides datagram communication by means of two classes:DatagramPacket and Datagram Socket

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

Datagram packet

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

An instance of Datagram Packet may be transmitted between processes when one process send sit and another receives it. This class provides another constructor for use when receiving message. Its arguments specify an array of bytes in which to receive the message and the length of the array. A received message is put in the Datagram Packet together with its length and the Internetaddressandportofthesendingsocket.ThemessagecanberetrievedfromtheDatagramPacke t by means of the method get Data. The methods get Port and get Address access the port and Internet address.

Datagram Socket: This class supports sockets for sending and receiving UDP datagram's. Itprovidesaconstructorthattakesaportnumberasitsargument, forusebyprocessesthatneedto use a particular port. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can throw a Socket Exception if the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.

The class Datagram Socket provides methods that include the following:

Send and receive: These methods are for transmitting datagram's between a pair of sockets. The argument of send is an instance of Datagram Packet containing a message and its destination. The argument of receive is an empty Datagram Packet in which to put the message, its length and its origin. The methods send and receive can throw I Exceptions.

Set So Timeout: This method allows a timeout to be set. With a timeout set, the receive method will block for the time specified and then throwanInterrupted I Exception.

Connect: This method is used for connecting to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that

address

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
            // args give message contents and server hostname
            DatagramSocket aSocket = null;
             try {
                        aSocket = new DatagramSocket();
                        byte [] m = args[0].getBytes();
                        InetAddress aHost = InetAddress.getByName(args[1]);
                        int serverPort = 6789;
                        DatagramPacket request = new DatagramPacket(m,  m.length(), aHost, serverPort);
                        aSocket.send(request);
                        byte[] buffer = new byte[1000];
                        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                        aSocket.receive(reply);
                        System.out.println("Reply: " + new String(reply.getData()));
             }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
             }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(aSocket != null) aSocket.close();}
  }
}
```

UDP client sends a message to the server and gets reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
            // args give message contents and server hostname
            DatagramSocket aSocket = null;
             try {
                        aSocket = new DatagramSocket();
                        byte [] m = args[0].getBytes();
                        InetAddress aHost = InetAddress.getByName(args[1]);
                        int serverPort = 6789;
                        DatagramPacket request = new DatagramPacket(m,  m.length(), aHost, serverPort);
                        aSocket.send(request);
                        byte[] buffer = new byte[1000];
                        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                        aSocket.receive(reply);
                        System.out.println("Reply: " + new String(reply.getData()));
             }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
             }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(aSocket != null) aSocket.close();}
  }
}
```

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

JavaAPIforTCPstreams•TheJavainterfacetoTCPstreams is provided in the classes

*ServerSocket*and*Socket*:

ServerSocket:Thisclassisintendedforusebyaservertocreateasocketataserverportforlisteningfor*c*

*onnect*requestsfromclients.Its*accept*methodgetsa*connect*requestfromthe

Queue or, if the queue is empty, blocks until one arrives. The result of executing *accepts* is an

instanceof*Socket* – a socket to use for communicating with the client.

Socket: This class is for use by a pair of processes with a connection. The client uses aconstructortocreateasocket,specifyingtheDNShostnameandportofaserver.Thisconstructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and port number. It can throw an *Unknown Host Exception* if the host name is wrong *IOException* if an IO error occurs.

The *Socket* class provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket. The return types of these methods are *Input Stream* and *Output Stream*, respectively – abstract classes that define methods for reading and writing bytes. The return values can be used as the arguments of constructors for suitable input and outputstreams.Ourexampleuses*DataInputStream*and*DataOutputStream*,whichallowbinaryrepresentationsofprimitivedatatypestobereadandwritteninamachine-independentmanner.

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

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
            // args give message contents and server hostname
            DatagramSocket aSocket = null;
             try {
                        aSocket = new DatagramSocket();
                        byte [] m = args[0].getBytes();
                        InetAddress aHost = InetAddress.getByName(args[1]);
                        int serverPort = 6789;
                        DatagramPacket request = new DatagramPacket(m,  m.length(), aHost, serverPort);
                        aSocket.send(request);
                        byte[] buffer = new byte[1000];
                        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                        aSocket.receive(reply);
                        System.out.println("Reply: " + new String(reply.getData()));
             }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
             }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(aSocket != null) aSocket.close();}
  }
}
```

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

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

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

External data representation and marshalling:

To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened in an agreed format. An agreed standard for the representation of data structures and primitive values is called an *external data representation*.

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. ***Unmarshallin****g* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination

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

1.**CORBA'sCommonData Representation(CDR):**

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

These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*,*unsignedlong*, *float* (32-bit),*double*(64-bit),*char*, *Boolean*(TRUE, FALSE)-

Primitive types: CDR defines a representation for both big-endian and little-endian orderings. Values are transmitted in the sender's ordering, which is specified in each message. The recipient translates if it requires a different ordering

Constructedtypes:Theprimitivevaluesthatcompriseeachconstructedtypeareaddedtoasequenceo fbytes in a particular order, asshown in Figure4.7.

Figure4.7.

CORBACDR for constructed types

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

Figure 4.8 shows a message in CORBA CDR that contains the three fields of a *struct* whose respective types are *string*, *string* and *unsigned long*. The figure shows the sequence of bytes with four bytes in each row

Figure4.8 CORBACDRmessage

indexinnotes

Sequence of bytes 4bytes *on representation*

0–3	5	*lengthofstring*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*lengthofstring*
16–19	"Lond"	*'London'*
20–23	"on"	
24–27	1984	*unsignedlong*

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

Marshalling in CORBA • Marshalling operations can be generated automatically from the specification of the types of dataitems to be transmitted in a message

TheCORBAinterfacecompilergeneratesappropriatemarshallingandunmarshallingoperations for the arguments and results of remote methods from the definitions of the types of their parameters and results.

2. Java objects serialization:

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations

For example,

:*publicclassPersonimplements Serializable*

{

private String name;private String place;privateint year;

public Person(String aName, String aPlace, int aYear) {name=aName;

place = aPlace;year=aYear;

}

//followed by methods for accessing theinstance variables

}

In Java, the term *serialization* refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message Deserialization consists of restoring the state of an object or a set of objects from their serialized form

The information about a class consists of the name of the class and a version number. The version number is intended to change when major changes are made to the class. It can be set by the programmer or calculated automatically as a hash of the name of the class and itsinstancevariables,methodsandinterfaces.Theprocessthatdeserializesanobjectcancheckthatit has the correct version of the class.

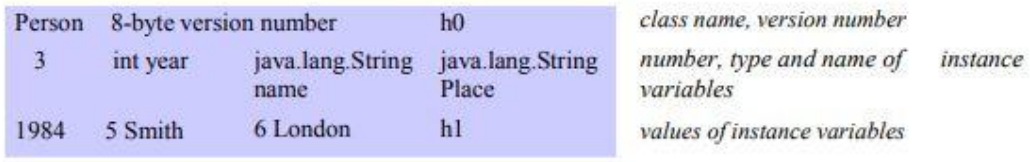
Java objects can contain references to other objects. When an object is serialized, all the objects that it references are serialized together with it to ensure that when the object is reconstructed, all of its references can be fulfilled at the destination. References are serialized as *handles.*

To serialize an object,itsclassinformationiswrittenout,followedby thetypesandnamesofitsinstancevariables.Iftheinstancevariablesbelongtonewclasses,thentheircl assinformationmustalsobewrittenout,followedby the types and names of their instancevariables. This recursive procedure continues until the class information and types and names of the instance variables of all of the necessaryclasses have been written out

As an example,consider the serialization of the following object:

Personp=newPerson("Smith","London",1984);

The serialized for mis illustrated in the following Figure



To make use of Java serialization, for example to serialize the *Person* object, create an instance of the class *Object Output Stream* and invoke its *write Object* method, passing the *Person* objects its argument. To desterilize an object from a stream of data, open an *ObjectInputStream* on the stream and use its *readObject* method to reconstruct the original object. The use of this pair of classes is similar to the use of *DataOutputStream*and *DataInputStream*

3. Extensible Markup Language(XML)

XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services. XML is *extensible* in the sense that users can define their own tags, in contrast to HTML,which uses a fixed set of tags

Figure4.10XMLdefinition of the *Person* structure

<personid="123456789">

<name>Smith</name>

<place>London</place>

<year>1984</year>

<!--acomment-->

</person>

XML elements and attributes•

Elements: An element in XML consists of a portion of character data surrounded by matching start and end tags. For example, one of the elements in Figure 4.10 consists of the data *Smith* contained within the *<name>* ... *</name>* tag pair. Note that the elementwiththe*<name>*tagisenclosedintheelementwiththe*<personid="123456789">*...

*</person>*tagpair.Theabilityofanelementtoencloseanotherelementallowshierarchic data to be

represented– a very important aspect of XML. An empty tag hasnocontentandisterminatedwith/>insteadof>.Forexample,theemptytag

<*european*/> could be included within the<*person*>...</*person*>tag

XML elements can have attributes. By the use of attributes we can add the information about the element.

<**book** publisher="Tata McGraw Hill"></**book**>Herepublisheris aattribute

In our above example **id** is an attribute

It is a matter of choice as to which items are represented as elements and which ones as attributes

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

Parsing and well-formed documents • An XML document must be well formed – that is, it must conform to rules about its structure. A basic rule is that every start tag has a matching end tag. Another basic rule is that all tags are correctly nested– for example, <*x*>..<*y*>..</*y*>..</*x*>is correct, whereas <*x*>..<*y*>..</*x*>..</*y*> is not. Finally, every XML document must have single root element that encloses all the other elements. These rules make it very simple to implement parsers for XML documents. When a parser reads an XML document that is not well formed, it will report an error.

XML prolog:Every XML document must have a *prolog*as its first line. The prolog must at least specify the version of XML in use(which iscurrently1.0).For example:

<?XML version="1.0" encoding="UTF-8"standalone="yes"?>

The prolog may specify the encoding(UTF-8)which is default

XML Namespaces

XML**Name space** isused *to avoid element name conflict* in XMLdocument.

XML Namespace Declaration

An XML namespace is declared using the reserved XML attribute. This attribute name must be started with"xmlns".

Let's see the XML namespace syntax:

<element xmlns:name="URL">

Here, namespace starts with keyword **"xmlns"**. The word **name** is a namespace prefix. The **URL** is a namespace identifier.

Let's take an example with two tables:

Table1:

<table>

<tr>

<td>Aries</td>

<td>Bingo</td>

</tr>

</table>

Table2: This table carries information about a computer table.

<table>

<name>Computer table</name>

<width>80</width>

<length>120</length>

</table>

If you add these both XML fragments together, there would be a name conflict because both have <table> element. Although they have different name and meaning.

We can get rid of this name conflict by using namespaces By Using xmlns Attribute

You can use xmlns attribute to define namespace with the following syntax:

<element xmlns:name="URL">

Foreg:

<root>

<h:tablexmlns:h="http://www.abc.com/TR/html4/">

<h:tr>

<h:td>Aries</h:td>

<h:td>Bingo</h:td>

</h:tr>

</h:table>

<f:tablexmlns:f="http://www.xyz.com/furniture">

<f:name>Computertable</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

</root>

XMLschemas: An XML schema is used to define the structure of anXMLdocument.It is like DTD but provides more control on XML structure.

An XML schema forthe *Person* structure

*<xsd:schemaxmlns:xsd=*URL of XMLschema definitions>

<xsd:elementname="person"type="personType"/>

<xsd:complexTypename="personType">

<xsd:sequence>

<xsd:elementname="name" type="xs:string"/>

<xsd:elementname="place" type="xs:string"/>

<xsd:elementname="year"type="xs:positiveInteger"/>

</xsd:sequence>

<xsd:attributename="id"type="xs:positiveInteger"/>

</xsd:complexType>

</xsd:schema>

GroupCommunication

A *multicast operation* is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that themembershipofthegroupis transparent to the sender.

IPmulticast–An implementation of multicast communication

A *multicast group* is specified by a Class D Internet address (see Figure 3.15) – that is, an address whose first 4 bits are 1110 inIPv4

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

AnapplicationprogramperformsmulticastsbysendingUDPdatagramswithmulticastaddresses and ordinary port numbers. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group.

The following details are specific toIPv4

*Multicastrouters*Internetmulticastsmakeuseofmulticastrouters,whichforwardsingledatagrams to routers on other networks, where they are again multicast to local members. To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass – called the *time to live*, or TTL for short.

Multicast address allocation Class D addresses (that is, addresses in the range 224.0.0.0 to239.255.255.255)arereservedformulticasttrafficandmanagedgloballybytheInternetAssigned Numbers Authority(IANA).

Multicast addresses may be permanent or temporary. Permanent groups exist even when the reareno members and therange224.0.6.000 to 224.0.6.127

The remainder of the multicast addresses is available for use by temporary groups,which must be created before use and cease to exist when all the members have left

Java API to IP multicast • The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*,which is a subclass of *DatagramSocket* with the additional capability

of being able to join multicast groups. The class *Multicast Socket* provides two alternative

constructors, allowing sockets to be created to use either a specified local port (6789, in Figure4.14) or any free local port. A process can join a multicast group with a given multicast address by invoking the *join Group* method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will receive datagram's sent by processes on other computers to that group at that port. A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket.

In the example in Figure 4.14, the arguments to the *main* method specify a message to be multicast and the multicast address of a group (for example, *"228.5.6.7").* After joining that multicast group, the process makes an instance of *DatagramPacket* containing the message and sends it through its multicast socket to the multicast group address at port 6789. After that, it attempts to receive three multicast messages from its peers via its socket, which also belongs to the group on the same port. When several instances of this program are run simultaneously on different computers, all of them join the same group, and each of them should receive its own message and the messages from those that joined after it.

TheJavaAPIallowstheTTLtobesetforamulticastsocketbymeansofthe*setTimeToLive*

method.Thedefaultis1,allowingthemulticasttopropagateonlyon the local network.

Figure4.14 Multicastpeerjoinsagroupandsendsandreceivesdatagrams

import java.net.;importjava.io.*;*

publicclassMulticastPeer{

publicstatic voidmain(Stringargs[]){

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

try{

InetAddressgroup=InetAddress.getByName(args[1]);s=newMulticastSocket(6789);s.joinGroup(group);

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

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

byte[]buffer=newbyte[1000];

for(inti=0;i<3;i++){//getmessagesfromothersin groupDatagramPacketmessageIn=

new DatagramPacket(buffer, buffer.length);s.receive(messageIn);

System.out.println("Received:"+newString(messageIn.getData()));

}

s.leaveGroup(group);

}catch(SocketExceptione){System.out.println("Socket:"+e.getMessage());

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

}

}

Distributed Objects and Remote Invocation

Programming Models for Distributed Application:

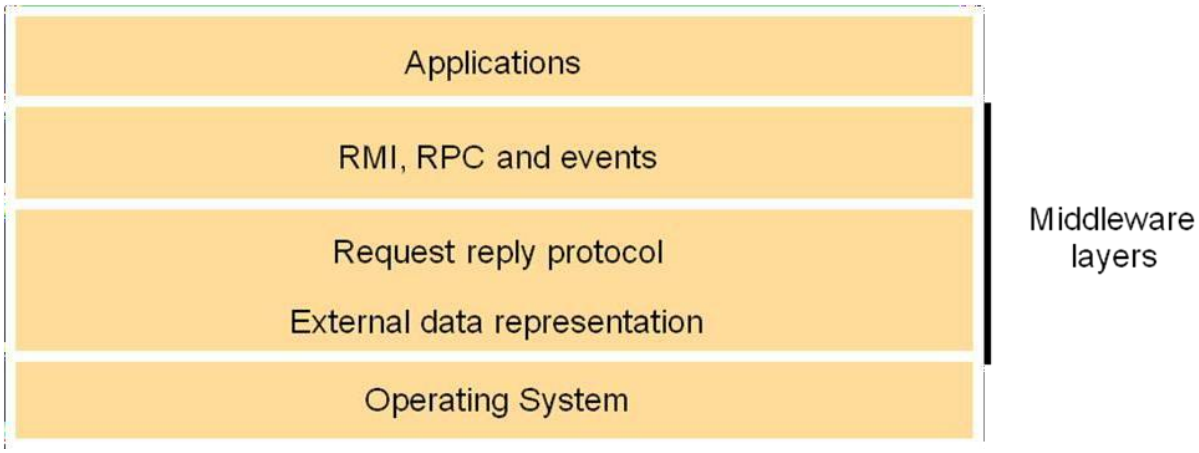
- Remote procedure call – client calls the procedures in a server program that is running in a different process.
- Remote method invocation (RMI) – an object in one process can invoke methods of objects in another process
- Event notification – objects receive notification of events at other objects for which they have registered.

Middleware:

The important aspects of middleware are:

- **Location transparency**: In RPC ,the client that calls a procedure cannot tell whether the procedure runs in the same process or in different process, possibly on a different computer.

Similarly in RMI the object making the invocation cannot tell whether the object it invokes is local or not and does not need to know the location.



- ✓ Itisalsofreefromthespecificsofcommunicationprotocols,operatingsystem and communication hardware

Interfaces:

In most of the programming languages program is divided into set of modules and these modules communicate with each other. In distributed systems these modules are present indifferent processes. The interface of a module specifies the procedures and the variables that can be accessed from other modules.

There are two types of interfaces:

- **Service interface**: In client server model ,The server specifies set of procedures and input-output parameters available to the client.
- **Remote interface:** In Distributed object model ,a remote interface specifies the methods of an object that are available for invocation by other objects and also the input output arguments**.**

Interface Definition Language: It provides a notation for defining interfaces. It can also specify type of arguments.

Examples:CORBAIDL for RMI,SunXDR for RPCCORBA IDLExample:

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

In the above example, add person and get person are methods that are available for RPC

Communication between Distributed Objects:

1. The Object Model:

- ✓ An object encapsulates both data and methods. Objects can be accessed via **object references.**
- ✓ An **interface** provides a definition of the signatures of a set of methods
- ✓ Actions are performed by **method invocations:**

The invocation of a method has three effects:

i. The state of the receiver may be changed

ii. A new object maybe instantiated

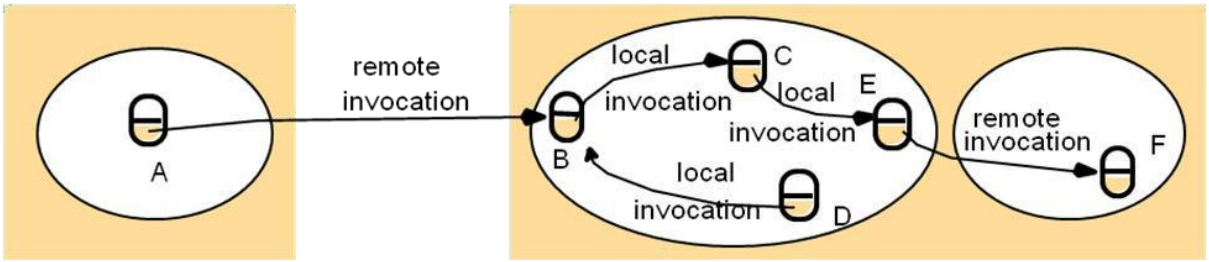
iii. Further invocations on methods in other objects may take place.

- ✓ Exceptions may be thrown to caller when an error occurs.
- ✓ **Garbage collection** frees the space occupied by objects when they are no longer needed.

The Distributed Objects Model:

Here we discuss the object model that is applicable to distributed objects.

- Remote method invocation – Method invocations between objects in different processes,whether in the same computer of not.
- Localmethodinvocation–Methodinvocationsbetweenobjectsinthesameprocess.

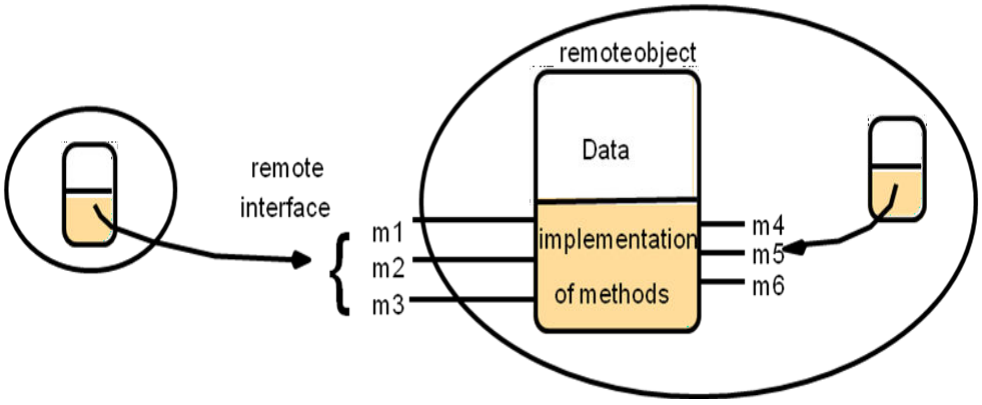


- Remote object – Objects that can receive remote invocations Remote and local method invocations areshowninFigure5.3.
- Each process contains objects, some of which can receive remote invocations, others only local invocations
- Those that can receive remote in vocations are called *remoteobjects*
- Objects need to know the *remote object reference* of an object in another process in order to invoke its methods
- the *remote interface* specifies which methods can be invoked remotely. The two fundamental concepts that are heart of distributed object model are:
    1. Remote object reference: An object must have the remote object reference of an object in order to do remote invocation of an object. Remote object references may be passed as input arguments or returned as output arguments
    2. Remote interface: Objects in other processes can invoke only the methods that belong to its remote interface (Figure5.4).

CORBA– uses IDL to specify remote interface

JAVA – extends interface by the **Remote** keywordFigure5.4

A remote object and its remote interface



Here the methods m1,m2,m3 are provided in the remote interface .so, client can access only these methods.

2.**DesignIssuesforRMI:**

Two important design issues in making RMI a natural extension of local method are:1.Invocationsemantics and ii. Transparency

1. Remote Invocation Semantics:

To provide a more reliable request-reply protocol, these fault-tolerant measures can be employed:

Retry request message: whether to transmit the request message until either a reply is received or server is assumed to be failed.

Duplicate Filtering: when transmissions are used , whether to filter out duplicate requests at theserver.

Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at theserver.

Combinations of these measures lead to a variety of possible semantics for the reliability of remote invocations.

The choices of RMI invocation semantics are defines as follows:

i. **Maybe invocation semantics**:Remote method may be executed once or not at all. If then result message is not received after a timeout there will be no retries,it is uncertain whether the method has been executed. . On the other hand, the procedure may have been executed and the result message has been lost. Maybe semantics isusefulonlyforapplicationsin whichoccasional failedcalls areacceptable

It suffers the following Failures:

1. Omission failures if the invocation or result message is lost.
2. Crash failures when the server containing the remote object fails

ii . **At-least-once semantics**: With at-least-once semantics, the invoker receives either a result, in which case the invoker knows that the procedure was executed at least once, or an exception in forming it that no result was received.

At-least-once semantics can suffer from the following types of failure:

- Crash failures when the server containing the remote procedure fails;
- arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values to be stored or returned

iii. **At-most-once semantics**: With at-most-once semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all.

Invocation Semantics:

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

2. Transparency Issues:

Goal is to make a remote invocation as similar as possible a local invocation The Issues (Differences from the local invocation)faced here are:

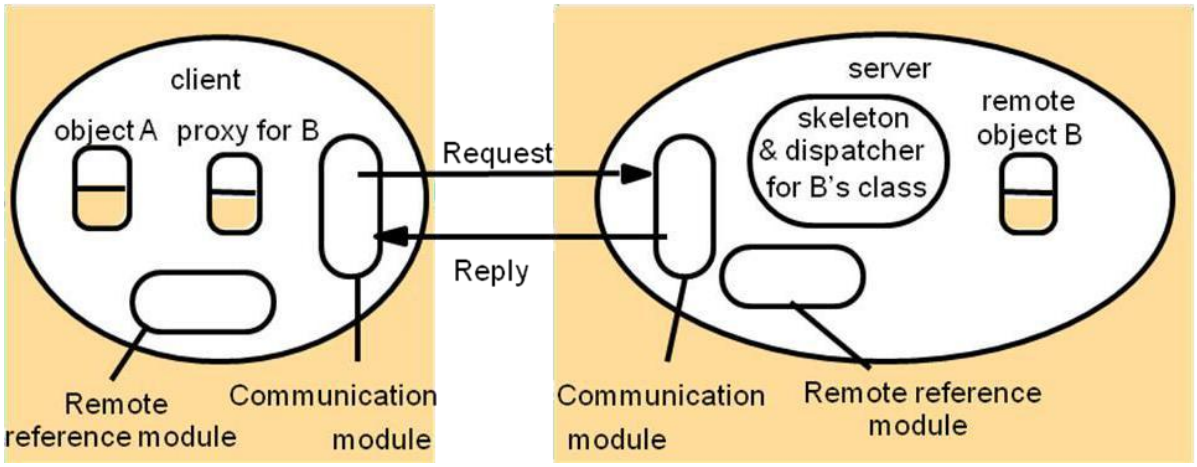
1. Syntax maybemadeidenticalbutbehavioraldifferencesexists.Thecausecouldbe

   Failure and **latency.**

2. Exceptions and exception handling are needed

3. Implementation of RMI:

Figure5.6 shows an object A invokes a method in a remote object B



Remote Reference Module: Responsibilities:

i. Translation between local and remote object references

ii. theremotereferencemoduleineachprocesshasaremoteobjecttablethatincludes:

- An entry for all the remote objects held by the process. For example in the above fig the remote object B will be recorded in the table at theserver
- An entry for each local proxy. for example in the above fig the proxy for B will be recorded in the table at the client.

RMI Software:

Proxy—provides remote invocation transparency

- marshal arguments, unmarshal results, send and receive messages Dispatcher–
- handles transfer of requests to correct method
- receive requests, select correct method, and pass on request message Skeleton –
- implements methods of remote interface
- unmarshal arguments from request, invoke the method of the remote object, and marshal the results

RMI Server and Client Programs:

Server: contains

- classesfordispatchers,skeletons and remote objects
- initializationsectionforcreatingsomeremoteobjects
- registration of remote objects with the binderClientcontains:

- classes for proxies of all remote objects
- binder to look up remote object references

RMI Binder and Server Threads:

A binder in a distributed system is a separate service that maintains a table containing mappings from textual names to remote object references

Server threads:

- sometimes implemented so that remote invocation causes a new thread to be created to handle the call
- server with several remote objects might also allocate separate threads to handle each object Activation of remote objects:
- A remote object is described as **active** when it is a running process.
- A remote object is described as **passive** when it can be made active if requested.
- An object that can live between activations of processes is called a **persistent object**

A**llocation service**helpsclients to locate remote objects from their remote references

RMI Distributed Garbage Collection:

Aim-recover memory if no reference t o an object exists. If there is a reference object should still exists.

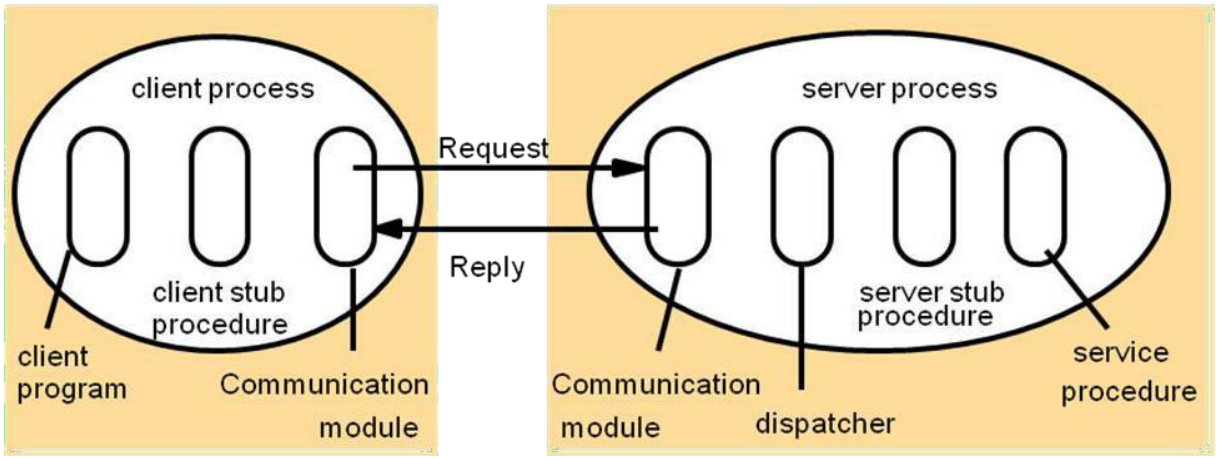
The distributed garbage collector works in cooperation with the local garbage collector.

- Each server has table(Beholders)that maintains list of references to an object.
- When the client C first receives a reference to an object B, it invokes add Ref(B)and then creates a proxy. The server adds C to the remote object holder Beholders.
- When remote object B is no longer reachable, it deletes the proxy and invokes recovered(B).
- When **Beholders** is empty, the server reclaims the space occupied by B.

Remote Procedure Call:

A RPC call is very similar to RMI ,in which a client program calls a procedure in another program running in server process

Figure5.7Roleofclientandserverstub procedures in RPC



The software components required to implement RPC are shown in the above Figure.

The stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server. When the reply message arrives, it unmarshals the results.

The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface

The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message

The server stub procedure then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals there turn values for the reply message

The service procedures implement the procedures in the service interface

Client and servers tub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition no f the service.

Case study :Sun RPC:

- It is designed for client-server communication over Sun NFS network file system.
- UDP or TCP can be used. If UDP is used, the message length is restrictedto64 KB

Interface Definition Language:

The notation is rather primitive compared to CORBA IDL or JAVA as shown in Figure5.8.

- Instead of no interface definition, a program number and a version number are supplied.
- The procedure number is used as a procedure definition.
- Single input parameter and output result are being passed.

Figure5.8

Files interface in SunXDR

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

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

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

For example, see the XDR definition in Figure 5.8 of an interface with a pair of procedures for writing and reading files. The program number is 9999 and the version number is 2. The *READ* procedure (line 2) takes as its input parameter a structure with three components specifying a file identifier, a position in the file and the number of bytes required. Its result is a structure containing the number of bytes returned and the file data. The *WRITE* procedure (line 1) has no result. The *WRITE* and *READ* procedures are given

numbers 1 and 2. The number 0 is reserved for a null procedure, which is generated automatically and is intended to be used to test whether a server is available.

The interface compiler *rpcgen* can be used to generate the following from an interface definition:

Client stub procedures;

Server *main* procedure, dispatcher and server stub procedures;

XDR marshalling and unmarshalling procedures for use by the dispatcher and client and server stub procedures

Binding • Sun RPC runs a local binding service called the *port mapper* at a well-known port number on each computer. Each instance of a port mapper records the programnumber,version number and port number in use by each service running locally. When a server starts up it registers its program number, version number and port number with the local port mapper.When a client starts up, it finds out the server's port by making a remote request to the port mapper at the server's host, specifying the program number and version number.

Events and Notifications

- The idea behind the use of events is that one object can react to a change occurring in another object.
- The actions done by the user are seen as **events** that cause state changes in objects.
- The objects are **notified** whenever the state changes.
- Local event model can be extended to distributed event-based systems by using the **publish-subscribe** paradigm.

In **publish-subscribe** paradigm:

- An object that has event publishes.
- Those that have interest subscribe.
- Objects that represent events are called **notifications**.
- Distributed event-based systems have two main characteristics:
- **Heterogeneous** – Event-based systems can be used to connect heterogeneous

components in the Internet.

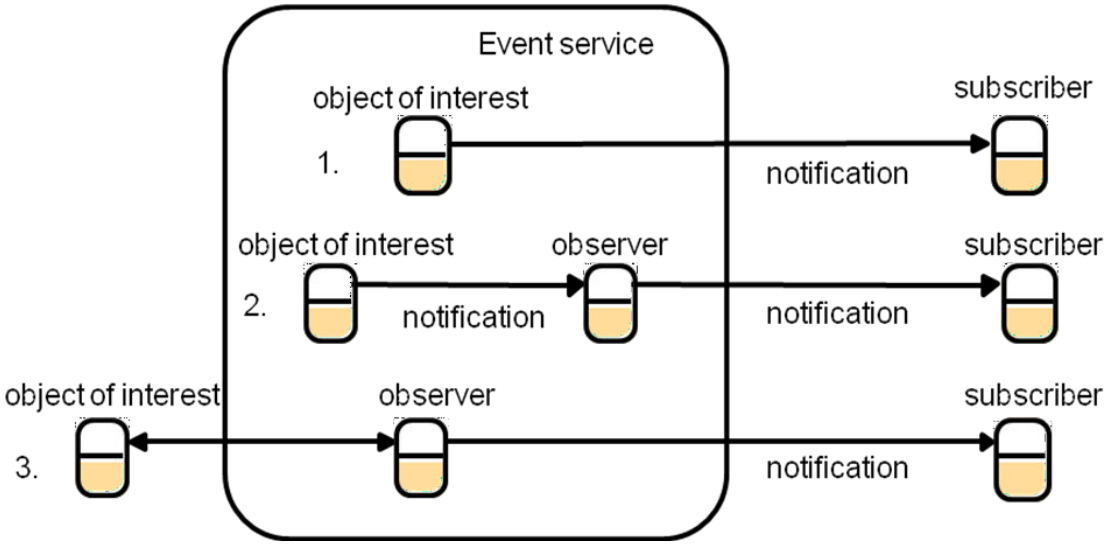
- **Asynchronous** – Notification are sent asynchronously by event-generating objects to those subscribers

The architecture of distributed event notification specifies the roles of participants as in Fig.5.10:

It is designed in a way that publishers work independently from subscribers.

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

Fig.5.10: Architecturefordistributedeventnotification



- The **roles of the participants** are:
- **Object of Interest** – This is an object experiences change of state, as a result of its operations being invoked.
- **Event** – An event occurs at an object of interest as the result of the completion of a method invocation.
- **Notification** – A notification is an object that contains information about an event.
- **Subscriber** – A subscriber is an object that has subscribed to some type of events in another object.

- **Observer objects** – The main purpose of an observer is to separate an object of interest from its subscribers.
- **Publisher** – This is an object that declares that it will generate notifications of particular types of event.

Figure5.10showsthreecases:

- An object of interest inside the event service sends notification directly to the subscribers.
- An object of interest inside the event service sends notification via the observer to the subscribers.
- The observer queries the object of interest outside the event service and sends notifications to the subscribers.

Roles for observers–the task of processing notifications can be divided among observers:

- **Forwarding** – Observers simply forward notifications to subscribers.
- **Filtering of notifications** – Observers address notifications to those subscribers who find these notifications are useful.
- **Patterns of events**–Subscribers can specify patterns of events of interest.
- **Notification mailboxes** – A subscriber can set up a notification mailbox which receives the notification on behalf of the subscriber.

JAVARMI (RemoteMethodInvocation)

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java.TheRMIallowsanobject toinvokemethodsonanobjectrunningin another JVM.

The RMI provides remote communication between the applications using two objects *proxy(stub)*and *skeleton*.

RMI uses proxy and skeleton object for communication with the remote object.

proxy

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. Itinitiates a connection with remoteVirtual Machine(JVM),
2. It writes and transmits(marshals)the parameters to the remote Virtual Machine(JVM),
3. It waits forthe result
4. It reads(unmarshals)the return value or exception, and
5. It finally,returns the value to the caller.skeleton

The skeleton is an object,acts as a gateway forthe server side object. All the in coming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter forthe remote method
2. It invokes the method on the actual remote object,and
3. It writes and transmits (marshals)the result to the caller.

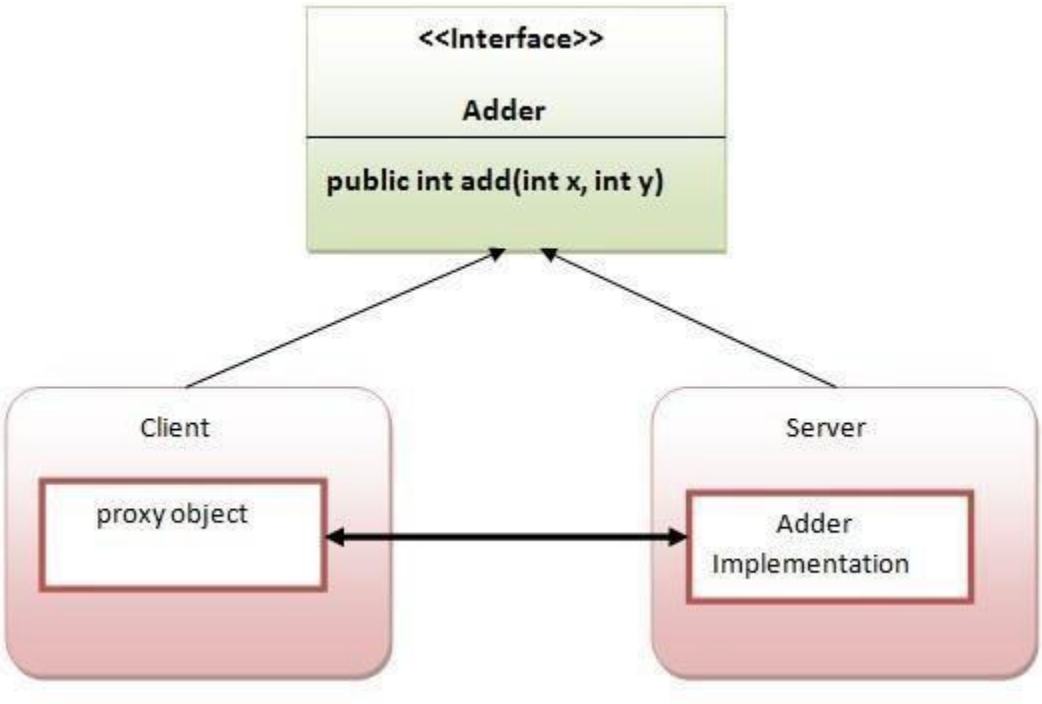
JavaRMI Example

Thereare6 steps to write the Microgram.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using thermictool
4. Start the registry service by rmiregistrytool
5. Create and start the remote application
6. Create and start the client application

RMI Example

In this example, we have followed all the 6 steps to create and run the rmi application. The client application needs only two files, remote interface and client application. In the rmi application, both client and server interact with the remote interface. The client application invokes methods on the proxy object, RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.



1) create the remote interface

For creating the remote interface, extend the Remote interface and declare theRemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and itdeclaresRemoteException.

```
importjava.rmi.*;
publicinterfaceAdderextends Remote
{
publicintadd(intx,inty)throwsRemoteException;
}
```

2) Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

- EitherextendtheUnicastRemoteObject class,
- or use the export Object() method of the Unicast Remote Object class

In case, you extendtheUnicastRemoteObjectclass,youmustdefineaconstructorthatdeclaresRemoteException.

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemoteObject implements Adder
{
AdderRemote() throws RemoteException
{
super();
}
public int add(int x,int y)
{return x+y;}
}
```

3) create the stub and skeleton objects using the rmic compilerrmicAdderRemote

4) RMIREGISTRY:

Rmi registry is the binder for java RMI.this is maintained in every server hosting remoteobjects.ItmaintainsatablemappingtextualURLstylenamestoreferencesto remoteobjects .hosted on that computer.It is accessed by the mehods of naming class.which takes the url formatted string of the following form:

//computername:port/object name

Where computername:port refers to the location of the RMI registry;TheNamingclass provides 5 methods.

1. Remotelookup(stringname):methodisusedbyclientstolookuparemoteobjectbyname.A remote object reference is returned.
2. void rebind(string name, Remote obi) : this method is used by a server to register are mote object by name
3. void bind(string name, Remote obi) : this method is used by a server to register are mote object by name but if the name is already bound to a remote objectreferenceanexception is thrown
4. void unbind(stringname,Remoteobj):this method removes a binding
5. string[] list():It returns an array of the names of the remote objects bound in the registry

5) Create and run the server application

Now rmi services need to be hosted in a server process.

In this example,we are binding the remoteobject by the names.

```
importjava.rmi.*;
importjava.rmi.registry.*;
publicclassMyServer{
publicstaticvoidmain(Stringargs[]){
try{
Adder s=newAdderRemote();Naming.rebind("rmi://localhost:5000/sonoo",s);
}catch(Exceptione){System.out.println(e);}
}
}
```

6) Createandruntheclientapplication

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server andclientapplications,inthesamemachinesoweareusinglocalhost.Ifyouwanttoaccessthe remote object from another machine, change the local host to the host name (or IPaddress)where the remote object is located.

```
importjava.rmi.*;
publicclassMyClient{
publicstaticvoidmain(Stringargs[]){
try{
Adder
stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");System.out.println(stub.add(34,4
));
}catch(Exceptione){}
}
}
```

UNIT-IV

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

A distributed file system is to present certain degrees of transparency to the user and the system: **Access transparency:** Clients are unaware that files are distributed and can access them in the same way as local files are accessed.

Location transparency: A consistent name space exists encompassing local as well as remote files.The name of a file does not give it location.

Concurrency transparency: All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.

Failure transparency: The client and client programs should operate correctly after a server failure. **Heterogeneity:** File service should be provided across different hardware and operating system platforms.

Scalability: The file system should work well in small environments (1 machine, a dozen machines)and also scalegracefully to huge ones(hundredsthroughtensofthousandsof systems).

Replication transparency: To support scalability, we may wish to replicate files across multiple servers.Clients should be unaware of this.

Migrationtransparency: Filesshouldbeabletomovearoundwithouttheclient'sknowledge.

Supportfine-
graineddistributionofdata:Tooptimizeperformance,wemaywishtolocateindividualobjectsnear the processes that use them.

Tolerancefornetworkpartitioning: Theentirenetworkorcertainsegmentsofitmaybeunavailable to a client during certain periods (e.g. disconnected operation of laptop). The file system should be tolerant of this.

File service types

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

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

File system were originally developed for centralized computer systems and desktop computers .to disk storage.

- File system was as an operating system facility providing a convenient programming interface
- Distributed file systems support the sharing of information in the form of files and hardware resources.
- With the advent of distributed ob

become more complex.

jectsystems(CORBA,Java)andtheweb,thepicturehas

- Figure 1 provides an overview of types of storage system.

Storage systems and their properties

	Sharing	Persistent	Distributed cache/replicas	Consistency maintenance	Example
Main memory	No	No	No	1	RAM
File systemn	No	Yes	No	1	UNIX file system
Distributed file system	Yes	Yes	Yes	Yes	Sun NFS
web	Yes	Yes	Yes	No	Web server
Distributed shared memory	Yes	No	Yes	Yes	Ivy(DSM)
Remote objects(RMI/ORB)	Yes	No	No	1	CORBA
Persistent object store	Yes	Yes	No	1	CORBA persistent state service
Peer to peer storage system	Yes	Yes	Yes	2	Ocean Store

file system modules:

Directorymodule:	RelatesfilenamestofileIDs
Filemodule:	Relatesfile IDstoparticularfiles
Access controlmodule:	Checks permission for operation requested
File access module:	Read or writes file data or attributes
Block module:	Accesses and allocates disk blocks
Device module:	Disk I/O and buffering

The below table Summarizes the main operations on files that are available to applications in UNIX systems.

Figure 4. UNIX file system operations

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

List out the UNIXfilesystem Operations:

fieldes=open(name,mode)fieldes=create(name,mode)status=close(fieldes)count=read(fieldes, buffer,n)count=write(fieldes,buufer,n)pos=Iseek(filedes,offset,whence)status=unlink(nmae)st atus=link(name1,nmae2)status=stat(name,buffer)

List out the transparencies in file system.

- Access transparency
    - Location transparency
    - Mobility transparency
    - Performance transparency
    - Scaling transparency

What is meant by concurrency control:

Changestoafilebyoneclientshouldnotinterferewiththeoperationofotherclientssimultane ously accessing or changing the same file. This is well-known issue of concurrencycontrol .The need for concurrency control for access to shared data in many applicationsIswidely accepted and techniques are known for its implementation ,but they are costly .Mostcurrent file services follow morden UNIX standards in providing advisery or mandatory file orrecord-levellocking.

What is file replication:

In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits-its enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed. Few file services support replication fully, but most support the catching of files or portions of files locally,alimited form of replication.

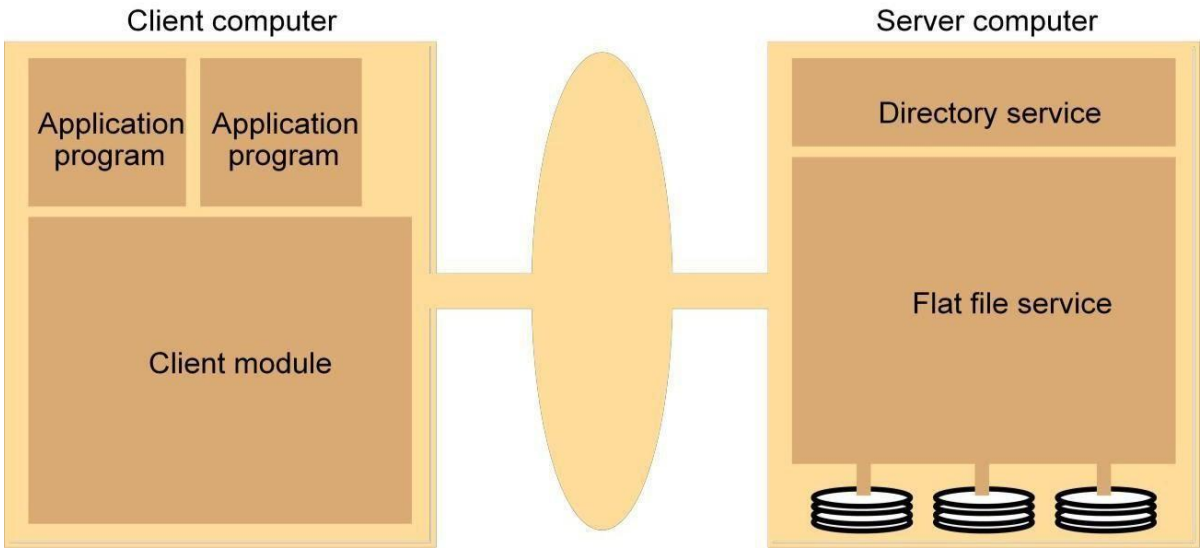
What is meant by directory services:

The directory services provide a mapping between text names for files and their UFIDs. Client may obtain the UFIDs of a file by quoting its text name to the directory services. The directory services provide the function needed to generate directories, to add new file name to directories and to obtain UFIDs from directories. It is client of the flat file services; its directory is stored infilesoftheflatservices.Whenahierarchicfile-namingschemeisadoptedasinUNIX,directorieshold references tootherdirectories.

Case studies :

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

Sketch the file service architecture:



List the flat file service operation.

Read(file/d,I,N)>data-throws bad position -if1≤ 1≤length(file):reads a sequence of up to NitemsFrom a file starting at item/andreturnsit in data

Write(File/D,I,Ddata)-throws bad position -if1≤1≤length(file)+1: writes a sequence of datatoaFile,starting at item1,extendingthefileif necessary

Create()->Field -createsanewfileoflength0 and delivers aUFID for it

Delete(Field) -removes the file from the file store

Get Attributes(Field)->> -returns the file attributes forth file

Set Attributes(FileID) -setsthefileattributes(only those attributes that not Shaded in)

List the directory service operation.

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

Sun'sNetwork File System:

The earliest successful distributed system could be attributed to Sun Microsystems, which developed the Network File System (NFS). NFSv2 was the standard protocol followed for many years, designed with the goal of simple and fast server crash recovery. This goal is of

utmost importance in multi-client and single-server based network architectures because a single instant of server crash means that all clients are unserviced. The entire system goes down.

Stateful protocols make things complicated when itcomes to crashes. Consider a client A trying to access some data from the server. However, just after the first read, the server crashed. Now, when the server is up and running, client A issues the second read request. However, the server does not know which file the client is referring to, since all that information was temporary and lost during the crash.

Stateless protocols come to our rescue. Such protocols are designed so as to not store any stateinformationintheserver.Theserverisunawareofwhattheclientsaredoing—whatblocksthey are caching, which files are opened by them and where their current file pointers are. Theserver simply delivers all the information that is required to service a client request. If a server crash happens, the client would simply have to retry the request. Because of their simplicity,NFS implements a stateless protocol.

File Handles:

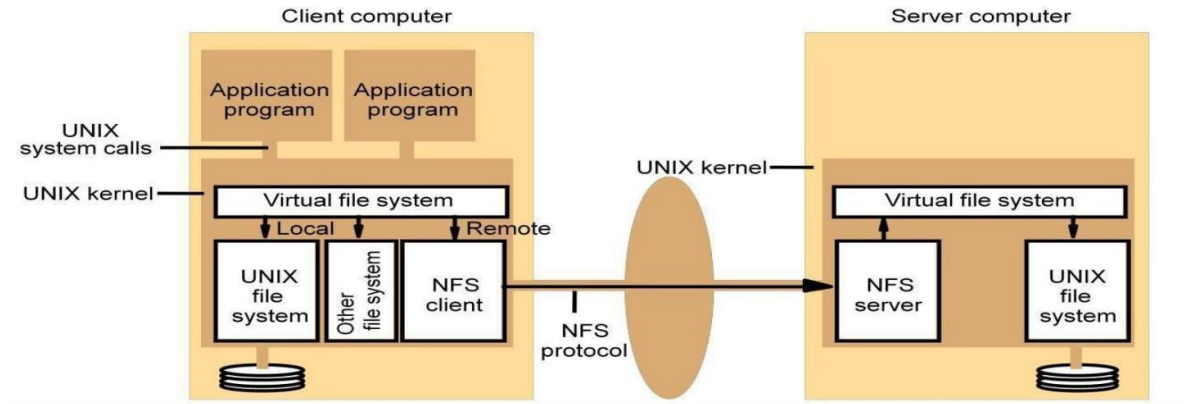
NFS uses file handles to uniquely identify a file or a directory that the current operation is being performed upon. This consists of the following components:

- **Volume Identifier –** An NFS server may have multiple file systems or partitions. The volume identifier tells the server which file system is being referred to.
- **InodeNumber–** This number identifies the file within the partition.
- GenerationNumber–This number isused while reusing an inode number.

File Attributes:

"File attributes" is a term commonly used in NFS terminology. This is a collective term for the trackedmetadataofafile,includingfilecreationtime,lastmodified,size,ownershippermissionsetc. This can be accessed by calling stat()on the file.

NFS architecture.



Distributed Systems Page139

NFS access control and authentication:

 The NFS server is stateless server, so the user's identity and access right

server on each request.

 It is not shown in the Figure 8.9 because they
 In the local file system they are checked only on the file
smust be checked by the's access permission attribute.
 Every client request is accompanied by the userID and groupID

are inserted by the RPC system.

 Kerberos has been integrated with NFS to provide a stronger and more comprehensive security

solution.

Mount service:

 Mount operation:

mount(remotehost,remotedirectory,localdirectory)

Servermaintainsatableofclientswhohavemountedfilesystemsatthatserver.

 Each client maintains a table of mounted file systems holding:

<IPaddress,port number,file handle>

-mountedorsoft-mountedinaclientcomputer. Figure 10 illustrates a Client with two remotely
 Remote file systems may be hard
mounted filestores.

Server caching:

Similar to UNIX file caching for local files:

 pages (blocks) from disk are held in a main memory buffer cache until the space is required for newer pages.Read-ahead and delayed-write optimizations.

 For local files, writes are deferred to next sync event (30 second intervals).
 Works well in local context, where files are always accessed through the local cache, but in the remotecaseitdoesn'toffernecessarysynchronizationguaranteestoclients.

Achievement of transparencies is other goals of NFS:

 NFS is an excellent example of a simple, robust, high-performance distributed service.

Access transparency: The API is the UNIX system call interface for both local and remote files.

Locationtransparency: Namingoffilesystemsiscontrolledbyclientmountoperations,but transparencycan be ensured by an appropriate system configuration.

Mobility transparency: Hardly achieved; relocation of files is not possible, relocation of file systems is possible, but requires updates to client configurations.

Scalabilitytransparency: Filesystems(filegroups)maybesubdividedand allocated to separate servers.

Replication transparency:

–Limitedtoread-onlyfilesystems;forwritablefiles,theSUNNetworkInformationService(NIS)runsover NFSand is used to replicate essential system files.

CaseStudy:The Andrew File System(AFS):

AFS differs markedly from NFS in its design and implementation. The differences are primarily attribute able 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 capability 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(inAFS-3,fileslargerthan64kbytesare transferredin64-kbytechunks).

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 usedtosatisfyclients' openrequestsinpreferencetoremotecopieswheneverpossible.

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

OPERATIONOFAFS:

i) 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.

ii) ThecopyisstoredinthelocalUNIXfilesysteminthe clientcomputer.Thecopyisthenopenedand the resulting UNIX file descriptor is returned to the client.

iii) Subsequentread,writeandotheroperationsonthefilebyprocessesintheclientcomputerareappliedtothelocalcopy.

iv) 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 thefile. Thecopyontheclient'slocaldiskisretainedincaseitisneededagainbyauser-levelprocessonthe same workstation.

AFS is a distributed file system, with scalability as a major goal. Its efficiency can be at tribute to the following practical assumptions(as also seen in UNIX file system):

- Files are small(i.e.entire file can be cached)
- Frequency of reads much more than those of writes
- Sequential access common
- Files are not shared(i.e. read and written by only one user)

- Shared files are usually not written
- Diskspace is plentiful

AFSdistinguishesbetweenclientmachines(workstations)anddedicatedservermachines. Caching files in the client side cache reduces computation at the server side, thus enhancing performance. However, the problem of sharing files arises. Tosolvethis,allclientswithcopiesofafilebeingmodifiedbyanotherclientarenot informed the moment the client makes changes. That client thus updates its copy, and the changes are reflected in the distributed file system only after the client closes the file.

The key software components in AFS are:

- **Vice**: The server side process that resides on top of the Unix kernel, providing shared file services to each client
- **Venus**: The client side cache manager which acts as an interface between the application program and the Vice

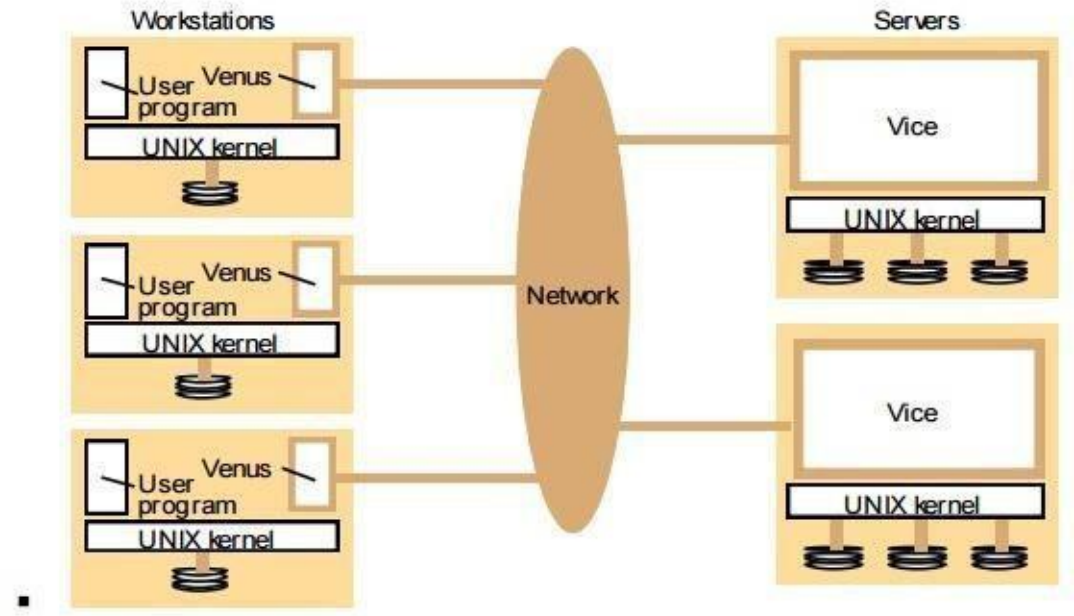
All the files in AFS are distributed among the servers. The set of files in one server is referred to as a **volume**. In case a request cannot be satisfied from this set of files, the vice server informs the client where it can find the required file.

The basic file operations can be described more completely as:

- Open a file: Venus traps application generated file open system calls, and checks whether it can be serviced locally (i.e. a copy of the file already exists in the cache) before requesting Vice for it. It then returns a file descriptor to the calling application. Vice, along with a copy of the file, transfers a callback promise,whenVenus requests for a file.
- Read and Write: Reads/Writes are done from/to the cached copy.
- Close a file: Venus traps file close system calls and closes the cached copy of the file. If the file had been updated, it informs the Vice server which then replaces its copy with the updated one, as well as issues callbacks to all clients holding call back promises on this file. On receiving a call back, the client discards its copy,and works on this fresh copy.

The server wishes to maintain its states at all times, so that no information is lost due

to crashes. This is ensured by the Vice which writes the states to the disk. When theservercomesupagain,italsoinformsalltheserversaboutitscrash,sothatinformation about updates may bypassed toot.



Distributed shared memory(DSM)

Shared memory is the memory block that can be accessed by more than one program. A shared memory concept is used to provide a way of communication and provide less redundant memory management.

Distributed Shared Memory abbreviated as **DSM** is the implementation of shared memory concept in distributed systems. The DSM system implements the shared memory models in loosely coupled systems that are deprived of a local physical shared memory in the system. In this type of system distributed shared memory provides a virtual memory space that is accessible by all the system (also known as **nodes**)of the distributed hierarchy.

Message passing versus DSM

The message passing and DSM can be compared based on services they offer and in terms of their efficiency

Message Passing	Distributed Shared Memory
Services Offered: Variables have to be marshalledfromoneprocess,transmittedandunmarshalledintoothervariablesatthereceivingprocess.	The processes share variables directly, sonomarshallingandunmarshalling.Sharedvariablescanbenamed,storedandaccessedin DSM.
Processes can communicate with other processes. They can be protected fromone anotherbyhavingprivateaddress spaces.	Here,aprocess does not have private Address space So on eprocesscanalterthe executionofother.
This technique can be used in heterogeneouscomputers.	Thiscannotbeusedtoheterogeneous computers.
Synchronization between processes is throughmessagepassingprimitives.	Synchronization is through locks and semaphores.
Processes communicating via message passingmustexecuteatthesametime.	Processes communicating through DSM may execute with non-overlapping lifetimes.
Efficiency: Allremotedataaccessesareexplicitandtherefore the programmer is always awareofwhetheraparticularoperationisin-processorinvolvestheexpenseofcommunication.	Any particular read or update may or maynotinvolvecommunicationbytheunderlying runtime support.

Synchronizationmodel:

- ✓ Manyapplicationsapplyconstraintsconcerningthevaluesstoredinsharedmemory.). For example, if *a* and *b* are two variables stored in DSM, then aconstraint might be that *a* = *b* always. If two or more processes execute thefollowingcode:

  a :=*a*+1;

  b :=*b*+1;

thenaninconsistencymayarise.Suppose*a*and*b*areinitiallyzeroandthatprocess1

gets as far as setting *a* to 1. Before it can increment *b*, process 2 sets *a* to 2 and *b* to

1.Theconstraint hasbeenbroken.Thesolutionistomakethiscodefragmentintoacriticalsection: tosynchronize processes to ensurethat onlyonemayexecute it atatime.

- ✓ Inorder touseDSM,then,adistributedsynchronization serviceneedstobeprovided,whichincludesfamiliarconstructssuch as locks andsemaphores

Consistencymodel

TheissueofconsistencyarisesforasystemsuchasDSM, which replicates thecontents ofsharedmemorybycachingit atseparate computers.

- eachprocesshas alocalreplicamanager,whichholds cachedreplicasofobjects.Inmostimplementations, data is read from local replicas for efficiency, but updates have to bepropagatedto theother replica managers
- Consideranapplicationinwhichtwoprocessesaccesstwovariables,*a*and*b*(Figure18. 3),which areinitialized to zero.
- Process 2 increments *a* and *b*, in that order. Process 1 reads the values of *b* and *a* intolocalvariables*br*and*ar*,inthatorder.Notethatthereisnoapplication-levelsynchronization.
- Intuitively, process 1 should expect to see one of the following combinations ofvalues, depending upon the points at which the read operations applied to *a* and *b*(implied in the statements *br* := *b* and *ar* := *a*) occur with respect to process 2'sexecution: *ar* = 0, *br* = 0; *ar* = 1, *br* = 0; *ar* = 1, *br* = 1. In other words, the condition*ar br* should always be satisfied and process 1 should print 'OK'. However, a DSMimplementation might deliver the updates to*a* and *b* out of order to the replicamanagerforprocess1,inwhichcasethecombination*ar* =0,*br* = 1could occur.

Figure18.3 Two processes accessing shared variables

The main consistency models that can be practically realized in DSM implementationsaresequential consistencyand modelsthat arebased on weakconsistency.

Thecentralquestiontobeaskedinordertocharacterizeaparticularmemoryconsistencymodel is this:when areadaccessismadeto amemorylocation,whichwriteaccesses to the location are candidates whose values could be supplied to the read? Attheweakestextreme,theansweris:anywritethatwasissuedbeforetheread.

Atthestrongestextreme,allwrittenvaluesareinstantaneouslyavailabletoallprocesses: a read returns the most recent write at the time that the read takes place. Thisdefinition is problematic in two respects. First, neither writes nor reads take place at asingle point in time, so the meaning of 'most recent' is not always clear. Each type ofaccesshas awell-definedpoint of issue,but theycomplete atsome later time

Linearizability is more usually called atomic consistency in the DSM literature. We now restatethedefinition oflinearizability

A replicated shared object service is said to be linearizable if for any execution there issome interleaving of the series of operations issued by all the clients that satisfies thefollowingtwo criteria:

L1: The interleaved sequence of operations meets the specification of a (single) correctcopyof theobjects.

L2: The order of operations in the interleaving is consistent with the real times at whichtheoperations occurred in theactual execution.

Consider the simple case where the shared memory is structured as a setof variablesthatmaybereadorwritten.Theoperationsareallreadsandwrites,whichweintroduced a

notation for in Section 18.2.1: a read of value *a* from variable *x* is denoted*R*(*x*)*a*; a write of value *b* to variable *x* is denoted *W* (*x)b*. We can now express the firstcriterionL1 in terms ofvariables(theshared objects) as follows:

L1': The interleaved sequence of operations is such that if *R(x)a* occurs in the sequence, theneitherthelastwriteoperation thatoccurs beforeitintheinterleavedsequenceis*W(x)a*, orno writeoperationoccursbefore it and *a* isthe initialvalue of*x*.

Thiscriterionstatesourintuitionthatavariablecanonlybechangedbyawriteoperation.Thes econdcriterionforlinearizability,L2,remainsthesame

Sequential consistency Linearizability is too strict for most practical purposes. The strongestmemorymodelforDSMthat is usedin practiceis*sequential consistency*

A DSM system is said to be sequentially consistent if*for any execution* there is someinterleaving of the series of operations issued by all the processes that satisfies the followingtwocriteria:

SC1:Theinterleavedsequenceofoperationsis suchthatif *R(x)a*occursinthe

sequence,theneitherthelastwriteoperationthatoccursbeforeitinthe interleavedsequenceis*W(x)a* , ornowriteoperation occurs beforeit and*a* is theinitial valueof*x*.

SC2:Theorderofoperationsintheinterleavingisconsistentwiththeprogramorder in whicheachindividual client executed them.

CriterionSC1isthesameasL1'.CriterionSC2referstoprogramorderratherthantemporalorder,whic h is whatmakesit possibleto implement sequentialconsistency

The combination *ar* = 0, *br* = 1 in the above example could not occur under sequential consistency,because process 1 would be reading values that conflict with process 2's program order. An exampleinterleavingoftheprocesses'memoryaccessesinasequentiallyconsistentexecutionissho wninFigure18.4

Coherence:

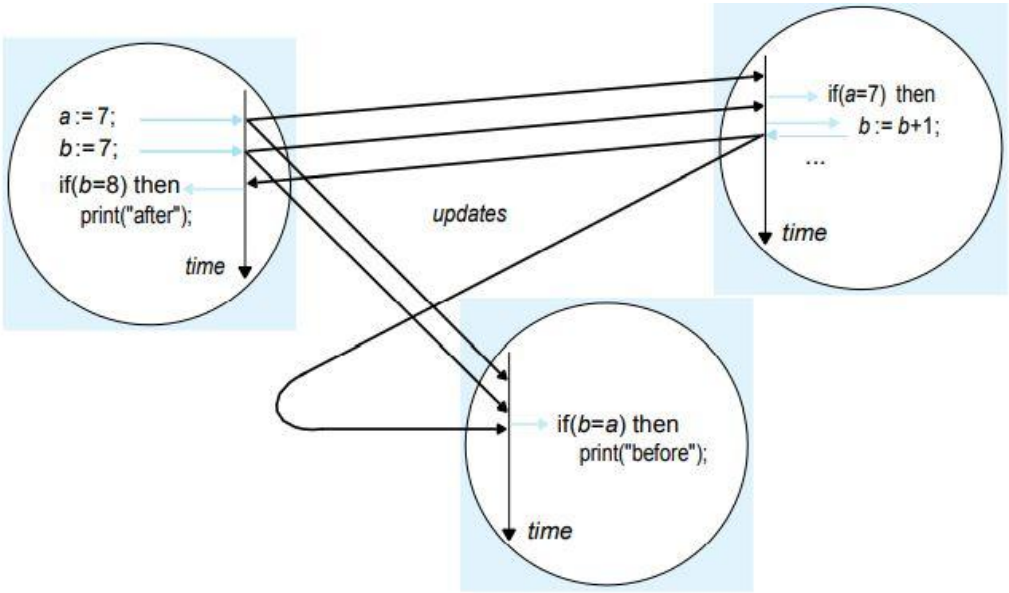
*Coherence*is anexample ofa weaker formof consistency.
Undercoherence,everyprocessagreesontheorder of *write* operations to the same location, but they do not necessarily agree on the ordering of *write*operationstodifferentlocations

Updateoptions

Two main implementation choices have been devised for propagating updates made byone process to the others: write-update and write-invalidate. These are applicable to avariety of DSM consistency models, including sequential consistency. In outline, theoptionsareas follows:

Write-update: The updates made by a process are made locally and multicast to all otherreplica managers possessing a copy of the data item, which immediately modify the dataread by local processes (Figure 18.5). Processes read the local copies of data items,without the need for communication. In addition to allowing multiple readers,severalprocesses may write the same data item at the same time; this is known as *multiple-reader/multiple-writersharing*

Figure18.5 DSMusingwrite-update



Write-invalidate:Thisiscommonlyimplementedintheformofmultiple-reader/single-writersharing.Atany time,adataitemmayeitherbeaccessedinread-onlymodebyoneormoreprocesses,oritmaybereadandwrittenbyasingleprocess.Anitemthatiscurrentlyaccessedinread-onlymodecanbecopiedindefinitelytootherprocesses.Whenaprocessattemptstowritetoit,amulticastmessageisfirstsent toallothercopiestoinvalidatethemandthisisacknowledgedbeforethewritecantakeplace;theotherprocessesaretherebypreventedfromreadingstaledata(thatis,datathatarenotuptodate).Anyprocesse sattempting toaccessthedataitemareblockedif a writer exists. Eventually, control

istransferredfromthewritingprocess,andotheraccessesmay takeplaceoncetheupdatehasbeen sent. The effect is to process all accesses to the item on a first-come, first-served basis

Thrashing

A potential problem with write-invalidate protocols is thrashing. Thrashing is said to occurwhere the DSM runtime spends an inordinate amount of time invalidating and transferringshared data compared with the time spent by application processes doing useful work. Itoccurs when several processes compete for the same data item, or for falsely shared dataitems. If, for example, one process repeatedly reads a data item that another is regularlyupdating, then this item will be constantly transferred from the writer and invalidated at thereader. This is an example of a sharing pattern for which write-invalidate is inappropriate andwrite-update would bebetter

SequentialconsistencyandIvycasestudy:

Paging is transparent to the application components within processes; they can logically bothreadandwriteanydatainDSM.However,theDSMruntimerestrictspageaccesspermissions in order to maintain sequential consistency when processing reads and writes.Paged memory management units allow the access permissions to a data page to be set to*none*, *read-only* or *read-write*. If a process attempts to exceed the current access permissions,then it takes a read or write page fault, according to the type of access. The kernel redirectsthepagefaulttoahandlerspecifiedbytheDSMruntime layer in each process.

Theproblemofwrite-update

Suppose that every update has to be multicast to the remaining replicas. Suppose that apagehasbeenwrite-protected.Whenaprocessattemptstowrite uponthe page,ittakesapagefaultandahandlerroutineiscalled.Thishandlercould,inprinciple,examine faulting instruction to determine the value and address being written and multicast theupdatebeforerestoringwriteaccessandreturningtocompletethefaultinginstruction.

But now that write access has been restored, subsequent updates to the page will not cause apage fault. To make every write access produce a page fault, it would be necessary for thepage fault handler to set the process into TRACE mode, whereby the processor generates aTRACE exception after each instruction. The TRACE exception handler would turn off

writepermissionsto thepageand turn off TRACE modeoncemore

Writeinvalidation

A process with the most up-to-date version of a page *p* is designated as its *owner* – referredto as *owner*(*p*). This is either the single writer, or one of the readers. The set of processes thathaveacopyofapage*p* is called its *copyset* – referred toas*copyset*(*p*).

Thepossiblestatetransitions areshowninFigure 18.8.Whenaprocess$P_W$attemptstowriteapage *p* to which it has no access or read-only access, a page fault takes place. The page-faulthandlingprocedureis asfollows:

- ✓ The page is transferred to P_W , if it does not already have an up-to-date read-onlycopy.
- ✓ Allothercopies areinvalidated: thepagepermissions aresetto noaccessatallmembersof*copyset*(*p*).
- ✓ *copyset*(*p*) := $\{P_W\}$.
- ✓ *owner*(*p*):=P_W.
- ✓ The DSM runtime layer in P_W places the page with read-write permissions at theappropriatelocation initsaddress spaceandrestarts thefaultinginstruction.

Statetransitionsunderwrite-invalidation

Notethattwoormoreprocesseswithread-onlycopiesmay takewritefaultsatmoreor

less the same time. A read-only copy of a page may be out-of-date when ownership iseventually granted. To detect whether a current read-only copy of a page is out-of-date,each page can be associated with a sequence number, which is incremented wheneverownership is transferred. A process requiring write access encloses the sequence numberof its read-only copy, if it possesses one. The current owner can then tell whether thepage has been modified and therefore needs to be sent. This scheme is described byKesslerandLivny[1989] as the‘shrewdalgorithm’.

When a process P_R attempts to read a page *p* for which it has no access permissions, a read pagefault takes place. Thepage-fault handlingprocedureis as follows:

- ✓ Thepageis copiedfrom*owner*(*p*)toP_R.

- ✓ If the current owner is a single writer, then it remains as p's owner and its accesspermission for p is set to read-only access. Retaining read access is desirable in case theprocess attempts to read the page subsequently – it will have retained an up-to-dateversion of the page. However, as the owner it will have to process subsequent requestsfor the page even if it does not access the page again. So it might turn out to have beenmoreappropriateto reducepermissiontonoaccessandtransferownershipto P_R.
- ✓ *copyset*(p):=*copyset*(p) $\{P_R\}$.
- ✓ TheDSMruntimelayerin$P_R$placesthepagewithread-onlypermissionsattheappropriatelocation initsaddress spaceandrestartsthe faultinginstruction.

It is possible for a second page fault to occur during the transition algorithms just described. Inorder that transitions take place consistently, any new request for the page is not processed untilafterthe current transition has completed . Thedescriptionjustgivenhasonlyexplained*what*mustbedone.Theproblemof*how*toimplementpagefault handlingefficientlyis nowaddressed

Invalidationprotocols

Twoimportantproblemsremaintobeaddressedinaprotocoltoimplementtheinvalidationscheme:

- ✓ Howtolocate*owner*(p)for agiven page$p$.
- ✓ Wheretostore*copyset*(p).

For Ivy, Li and Hudak [1989] describe several architectures and protocols that take varyingapproaches to these problems. The simplest we shall describe is their improved centralizedmanageralgorithm.

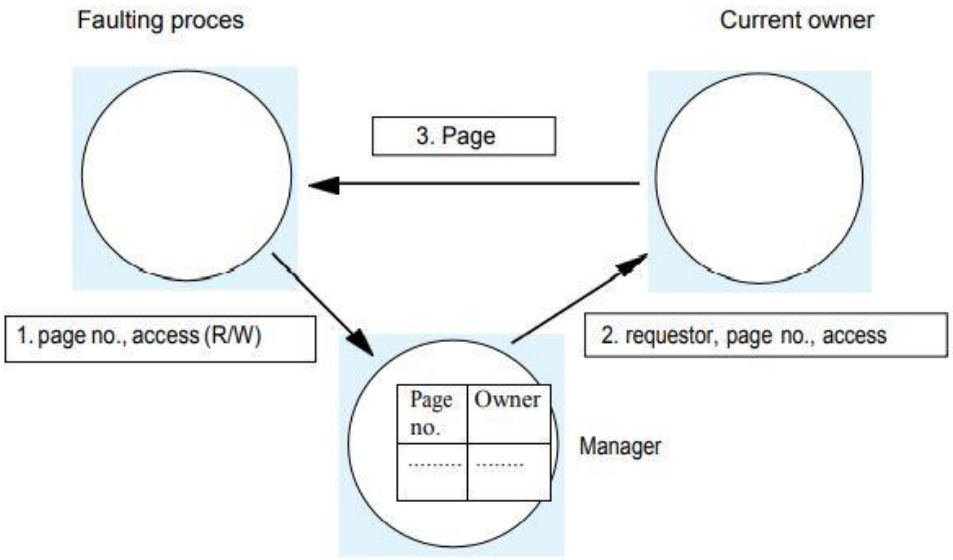
In it, a single server called a manager is used to store the location (transport address) of *owner*(p)foreverypage$p$.Themanagercouldbeoneoftheprocessesrunningtheapplication,oritcouldbeany other process. In this algorithm, the set *copyset*(p) is stored at *owner*(p). That is, the identifiersandtransport addresses of themembers of*copyset*(p) arestored.

As shown in Figure 18.9, when a page fault occurs the local process (which we shall refer to as the*client*) sends a message to the manager containing the page number and the type of

access required(read or read-write). The client awaits a reply. The manager handles the request by looking up theaddress of *owner*(*p*) and forwarding the request to the owner. In the case of a write fault, themanager sets the new owner to be the client. Subsequent requests are thus queued at the client untilithas completed the transfer ofownership to itself.

The previous owner sends the page to the client. In the case of a write fault, it also sends the page'scopy set. The client performs the invalidation when it receives the copy set. It sends a multicastrequesttothemembersofthecopyset,awaitingacknowledgementfromalltheprocessesconcerned that invalidation has taken place. The multicast need not be ordered. The former ownerneed not be included in the list of destinations, since it invalidates itself. The details of copy setmanagement are left to the reader, who should consult the general invalidation algorithms givenabove.

Figure18.9 Central managerandassociatedmessages



A dynamic distributed manager algorithm

A dynamic distributed manager algorithm, allows page ownership to be transferred betweenprocesses but which uses an alternative to multicast as its method of locating a page's owner. Theidea is to divide the overheads of locating pages between those computers that access them. Everyprocesskeeps,foreverypage*p*,ahint as tothepage's currentowner–the probableownerof*p*, or*probOwner*(*p*). Initially, every process is supplied with accurate page locations. In general,however, these values are *hints*, because pages can be transferred elsewhere at any time. As inpreviousalgorithms, ownership istransferredonlywhen

awritefaultoccurs.

Theownerofapageis located byfollowingchains ofhints that areset up as ownershipofthepage is transferred from computer to computer. The length of the chainthat is, the number offorwarding messages necessary to locate the owner – threatens to increase indefinitely. Thealgorithm overcomes this by updating the hints as more up- to-date values become available.Hintsareupdated andrequestsareforwarded asfollows:

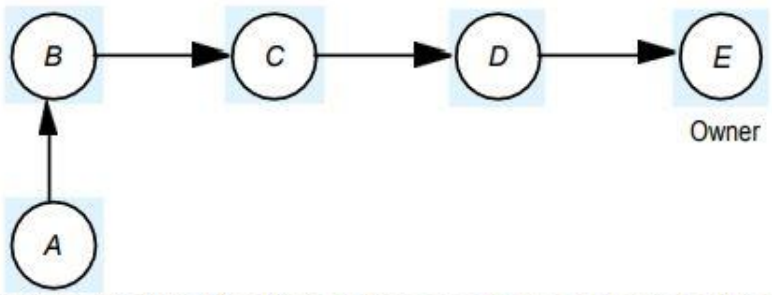
- Whenaprocesstransfersownershipofpage*p*toanotherprocess,itupdates *probOwner*(*p*)tobetherecipient.
- Whenaprocesshandlesaninvalidationrequestforapage*p*,itupdates *probOwner*(*p*)tobetherequester.
- Whenaprocessthathasrequestedreadaccesstoapage*p*receivesit,itupdates *probOwner*(*p*)tobetheprovider.
- When a process receives a request for a page *p* that it does not own, it forwards therequestto *probOwner*(*p*)andresets *probOwner*(*p*)tobe therequester.

The first three updates follow simply from the protocol for transferring page ownership andproviding read-only copies. The rationale for the update when forwarding requests is that, forwriterequests,the requesterwillsoonbetheowner, eventhoughitis notcurrently.

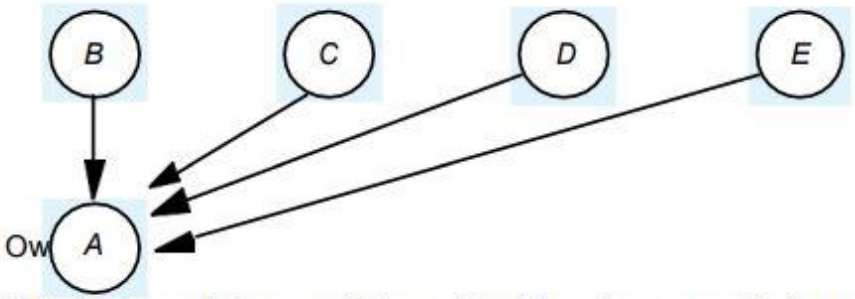
Figure 18.10 ((a) and (b)) illustrates *probOwner* pointers before and after process *A* takes awrite page fault. *A*'s *probOwner* pointer for the page initially points to *B*. Processes *B*, *C* and*D* forward the request to *E* by following their own *probOwner* pointers; thereafter, all are setto point to *A* as a result of the update rules just described. The arrangement after faulthandlingisclearlybetterthan that whichprecededit:thechain ofpointershascollapsed.

If,however,*A*takesareadfault,thenprocess *B*isbetteroff(twostepsinsteadofthreeto*E*),*C*'ssituationisthesameasitwasbefore(twosteps),but*D*is worseoff,withtwo stepsinstead ofone(Figure18.10(c)).

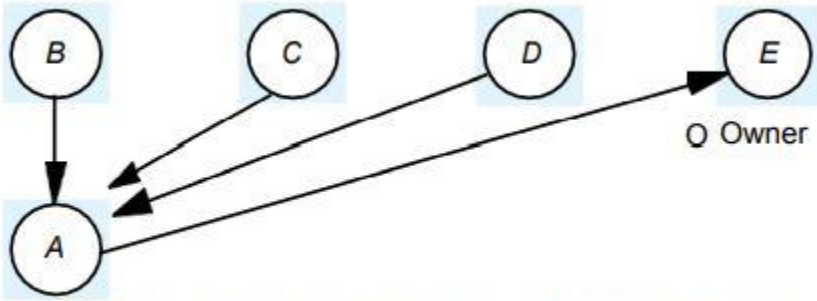
Updating *probOwner* pointers



(a) *probOwner* pointers just before process *A* takes a page fault for a page owned by *E*



(b) Write fault: *probOwner* pointers after *A*'s write request is forwarded



(c) Read fault: *probOwner* pointers after *A*'s read request is forwarded

Releaseconsistencymodel:

Theideaofreleaseconsistencyistoreduce DSMoverheadsbyexploitingthefactthatprogrammers use synchronization objects such as semaphores, locks and barriers

Memoryaccesses

In order to understand release consistency – or any other memory model that takessynchronization into account – we begin by categorizing memory accesses according

totheirrole,ifany,insynchronization.Furthermore,weshalldiscusshowmemoryaccesses may be performed asynchronously to gain performance and give a simpleoperationalmodel ofhow memoryaccesses takeeffect.

As we said above, DSM implementations on general-purpose distributed systems may usemessage passing rather than shared variables to implement synchronization, for reasons ofefficiency. But it may help to bear shared-variable-based synchronization in mind in thefollowingdiscussion.Thefollowingpseudocodeimplementslocksusingthe*testAndSet*operation variables. The function *testAndSet* sets the lock to 1 and returns 0 if it finds it zero; otherwiseitreturns 1.It does this atomically

acquireLock(varintlock)://*lockispassedby-*

referencewhile(*testAndSet*(*lock*)=1)

skip;

releaseLock(varintlock)://*lockispassed by-referencelock*:=0

Typesofmemoryaccess

Themaindistinctionisbetween*competing*accesses and *non- competing* (*ordinary*) accesses. Two accesses arecompetingif:

- theymayoccurconcurrently(thereis no enforcedorderingbetween them)andatleast one is a*write*.
- So two *read* operations can never be competing; a *read* and a *write* to the samelocation made by two processes that synchronize between the operations (and soorderthem)arenon-competing.
- Wefurtherdividecompetingaccessesinto*synchronization*and*non-synchronization*accesses:

- synchronization accessesare *read* or *write* operations that contributetosynchronization;
- non-synchronization accesses are *read* or *write* operations that areconcurrentbutthatdonotcontribute to synchronization.

- The*write*operationimpliedby‘*lock*:=0’in*releaseLock*(above)isasynchronization access.So isthe*read*operation implicitin *testAndSet*

Releaseconsistency

Therequirements thatwewishtomeetare:

- topreservethesynchronizationsemantics ofobjectssuchaslocksand barriers;
- togainperformance,weallowadegreeofasynchronicityformemoryoperations;
- toconstraintheoverlapbetweenmemoryaccesses inordertoguaranteeexecutionsthatprovide theequivalent ofsequential consistency.
- Release-consistent memoryisdesignedto satisfytheserequirements

RC1: before an ordinary *read* or *write* operation is allowed to perform withrespectto anyother process, all previous*acquire*accessesmust beperformed.

RC2: before a *release* operation is allowed to perform with respect to any otherprocess,allpreviousordinary*read*and *write*operationsmust beperformed.

RC3: *acquire*and *release*operationsaresequentiallyconsistentwith respecttooneanother.

- RC1andRC2guaranteethat,whenareleasehastakenplace,nootherprocessacquiringa lockcanreadstale versions ofdata modifiedbythe processthatperforms the release
- Consider the processes in Figure 18.12, which acquire and release a lock in order to access a pairofvariables*a*and*b*(*a*and*b*areinitializedtozero).Process1updates*a*and*b*underconditionsofmut ualexclusion,sothatprocess2cannotread*a*and*b*atthe sametimeandso willfind*a*

= *b* = 0 or *a* = *b* = 1. The critical sections enforce consistency – equality of *a* and *b* – *atthe application level*. It is redundant to propagate updates to the variables affected duringthe critical section. If process 2 tried to access *a*, say, outside a critical section, then itmightfind astale value.

Figure18.12 Processesexecutingonarelease-consistentDSM

```
Process1:
acquireLock();a := a+1;
b := b+1;
releaseLock();
Process2:
acquireLock();
//entercriticalsection
```

//leavecriticalsection

//entercriticalsection

print("The valuesofaandb are:", *a*,*b*);

releaseLock(); //leavecriticalsection

Under release consistency, process 1 will not block when it accesses *a* and *b*. The DSM runtimesystemnoteswhichdata havebeenupdatedbutneedtakenofurtheractionatthattime.Itisonlywhen process 1 has released the lock that communication is required. Under a write-updateprotocol, the updates to *a* and *b* will be propagated; under a write-invalidation protocol, theinvalidationsshould besent.

UNIT-V

TransactionsandConcurrencyControl

A Transaction defines a sequence of server operations that is guaranteed by the server to beatomic in the presence of multiple clients and server crashes. Nested transactions are structuredfrom sets of other transactions. They are particularly useful in distributed systems because theyallowadditional concurrency.

All of the concurrency control protocols are based on the criterion of serial equivalence and arederivedfromrulesfor conflictsbetweenoperations.Three methods aredescribed:

- Locksareusedtoordertransactions that accessthesameobjectsaccordingtotheorderofarrival oftheiroperations at the objects.
- Optimisticconcurrencycontrolallowstransactions to proceed untiltheyarereadyto commit, whereupon a check is made to see whether they have performedconflictingoperations onobjects.
- Timestamporderingusestimestampstoordertransactionsthataccessthesameobjects accordingto theirstartingtimes.

The goal of transactions is to ensure that all of the objects managed by a server remain in aconsistent state when they are accessed by multiple transactions and in the presence of servercrashes.Transactionsdealwithcrashfailuresofprocessesandomissionfailuresincommunication,but not anytypeofarbitrary(or Byzantine) behaviour.

Toexplain withabankingexample,eachaccountisrepresentedbyaremoteobjectwhoseinterface, Account, provides operations for making deposits and withdrawals and for enquiringaboutandsettingthebalance.Eachbranchofthebankisrepresentedbyaremoteobjectwhoseinterface,Branch,providesoperationsforcreatinganewaccount,forlookingupanaccountbyname and forenquiringabout the total funds atthat branch.

OperationsoftheAccountinterfacedeposit(amount)

deposit amount in the accountwithdraw(amount)

withdrawamountfromtheaccountgetBalance()□amount

returnthebalanceoftheaccountsetBalance(amount)

setthebalanceoftheaccounttoamountOperationsoftheBranchinterfacecreate(name)□account

createanewaccount with agivenname

lookUp(name)□account

returnareferencetotheaccountwiththegivennamebranchTotal()□amount

returnthetotalofallthebalancesatthebranch

Failure model for transactions

Lampson proposed a fault model for distributed transactions that accounts forfailures of disks,servers and communication. In this model, the claim is that thealgorithms work correctly in thepresence of predictable faults, but no claims are madeabout their behaviour when a disasteroccurs.Althougherrorsmayoccur, theycan be

detectedanddealtwithbefore anyincorrectbehaviourresults.Themodelstatesthefollowing:

- Writes to permanent storage may fail, either by writing nothing or by writing awrong value –for example, writing to the wrong block is a disaster. File storagemay also decay. Reads frompermanentstoragecan detect (byachecksum)whena blockofdata is bad.
- Servers may crash occasionally. When a crashed server is replaced by a newprocess, its volatilememory is first set to a state in which it knows none of thevalues (for example, of objects) frombefore the crash. After that it carries out arecovery procedure using information in permanentstorageandobtainedfromotherprocessestosetthevaluesofobjectsincludingthoserelatedtothe two-phasecommit protocol (see Section 17.6). When a processor is faulty, it is made tocrashsothat itis prevented from sendingerroneousmessagesand from writingwrong values to permanent storage – that is, so it cannot produce arbitrary failures.Crashes can occur atanytime; in particular, theymayoccurduringrecovery.
- There may be an arbitrary delay before a message arrives. A message may be lost,duplicated orcorrupted. The recipient can detect corrupted messages using achecksum. Both forged messagesandundetectedcorrupt messages areregarded asdisasters.

TRANSACATIONS:

Insomesituations,clientsrequireasequenceofseparaterequests toaservertobeatomicin thesensethat:

1. Theyarefreefrom interferencebyoperationsbeingperformedon behalf of otherconcurrentclients.
2. Eitherall of theoperations mustbe completed successfullyortheymusthaveno effect atallinthepresenceofservercrashes.

Aclient'sbankingtransaction

TransactionT:

a.withdraw(100);b.deposit(100);c.withdraw(200);b.deposit(200);

Banking example to illustrate transactions. A client that performs a sequence of operations on aparticular bank account on behalf of a user will first lookup the account by name and then applythedeposit,withdrawandgetBalanceoperationsdirectlytotherelevantaccount.Inourexample s,weuseaccountswithnamesA,BandC.Theclientlooksthemupandstoresreferencesto them in variablesa, bandcoftypeAccount.

AsimpleclienttransactionspecifyingaseriesofrelatedactionsinvolvingthebankaccountsA ,B and C. The first two actions transfer $100 from A to B and the second two transfer $200 fromCto B. Aclient achievesatransferoperation bydoingawithdrawal followed byadeposit.

Transactions can be provided as a part of middleware.For example, CORBA provides thespecification for an Object Transaction Service with IDL interfaces allowing clients' transactionstoinclude multipleobjects at multipleservers.

The client is provided with operations to specify the beginning and end of a transaction. Theclient maintains a context for each transaction, which it propagates with each operation in thattransaction. In CORBA, transactional objects are invoked within the scope of a transaction andgenerallyhavesomepersistent storeassociated with them.

ACID properties:

a transaction applies to recoverable objects and is intended to be atomic. It is often called an atomic transaction.There are twoaspectstoatomicity:

All or nothing: A transaction either completes successfully, in which case the effects of all of its operations arerecorded in the objects, or (if it fails or is deliberately aborted) has no effect at all. This all-or-nothing effect has twofurther aspects ofits own:

Failure atomicity: Theeffectsareatomicevenwhentheservercrashes.

Consistency:atransactiontakesthesystemfromoneconsistentstatetoanotherconsistent state;

Isolation: Each transaction must be performed without interference from other transactions; in other words, theintermediate effectsofa transactionmustnot bevisibletoothertransactions.

Durability: After a transaction has completed successfully, all its effects are saved in permanent storage. We use theterm 'permanent storage' torefer to files held on disk or another permanentmedium.Data saved in a filewillsurviveiftheserverprocess crashes.

Transaction life histories

Successful	*Aborted by client*	*Aborted by server*	
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•	server aborts	•
•	•	*transaction* →	•
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

TransactionPrimitives:

Primitive	**Description**
BEGIN_TRANSACTION	Makethestartofatransaction
END_TRANSACTION	Terminatethetransactionandtrytocommit
ABORT_TRANSACTION	Endthetransactionand***restore*** theoldvalues
READ	Readdatafromafile,atable,orotherwise
WRITE	Writedatatoafile,atable,orotherwise

Nestedtransactionsextendtheabovetransactionmodelbyallowingtransactionstobecomposedof othertransactions.Thusseveraltransactionsmaybestartedfromwithinatransaction, allowing transactions to be regarded as modules that can be composed asrequired.The outermost transaction in a set of nested transactions is called thetop-level transaction.Transactionsotherthan thetop-leveltransaction arecalled subtransactions.

Asubtransactionappearsatomictoitsparentwithrespecttotransactionfailuresandtoconcurrent access. Subtransactions at the same level, such asT1 and T2, can run concurrently,but their access to common objects is serialized. Each subtransaction can fail independently ofitsparent and of the other subtransactions. When a subtransaction aborts, the parenttransaction cansometimeschoosean alternative subtransactiontocompleteits task.

Forexample, a transaction to deliver a mail message to a list of recipients could be structuredas aset of subtransactions, each of which delivers the message to one of the recipients.If one or moreof the subtransactions fails, the parent transaction could record the factand then commit, with theresult that all the successful child transactions commit.

LOCKS:

Transactions must be scheduled so that their effect on shared data is serially equivalent. A servercan achieve serial equivalence of transactions by serializing access to the objects. Transactions TandUbothaccess accountB,but Tcompletes itsaccess beforeUstartsaccessingit.

A simple example of a serializing mechanism is the use of exclusive locks. In this lockingscheme, the server attempts to lock any object that is about to be used by any operation of aclient's transaction. If a client requests access to an object that is already locked due to anotherclient'stransaction,therequestissuspendedandtheclientmustwaituntiltheobjectisunlocke d.

As pairs of read operations from different transactions do not conflict, an attempt to set a readlock on an object with a read lock is always successful. All the transactions reading the sameobjectshareits readlock–forthisreason,read locksaresometimescalled sharedlocks.

Theoperationconflictrules tell usthat:

1. IfatransactionThasalreadyperformedareadoperationonaparticularobject,thenacon currenttransactionU must notwritethat objectuntil Tcommitsoraborts.
2. IfatransactionThasalreadyperformedawriteoperationonaparticularobject,thenaco

ncurrenttransactionU must notreadorwritethatobject untilT commitsoraborts.

Useoflocksin stricttwo-phaselocking:

1. Whenanoperationaccessesanobjectwithinatransaction:

   (a) Iftheobjectisnot alreadylocked,it islockedandtheoperationproceeds.

   (b) If the object has a conflicting lock set by another transaction, the transactionmust wait until itisunlocked.

   (c) If the object has a non-conflicting lock set by another transaction, the lock isshared and theoperationproceeds.

   (d) If the object has already been locked in the same transaction, the lock will bepromoted ifnecessary and the operation proceeds. (Where promotion isprevented by a conflicting lock, rule bis used.)

2. When a transaction is committed or aborted, the server unlocks all objects itlocked for thetransaction.

Lock compatibility table for hierarchic locks

For one object		*Lock to be set*			
		read	*write*	*I-read*	*I-write*
Lock already set	*none*	OK	OK	OK	OK
	read	OK	wait	OK	wait
	write	wait	wait	wait	wait
	I-read	OK	wait	OK	OK
	I-write	wait	wait	OK	OK

Optimistic concurrency control

Optimistic concurrency control is a concurrency control method applied to transactional systemssuch as relational database management systems and software transactional memory. It assumesthat multiple transactions can frequently complete without interfering with each other. Whilerunning, transactions use data resources without acquiring locks on those resources.Beforecommitting,eachtransactionverifiesthatnoothertransactionhasmodifiedthedata ithasread.If the check reveals conflicting modifications, the committing transaction rolls back and can berestarted and it is generally used in environments with low data contention. When

conflicts arerare,transactions cancomplete without the expense of managing locks and without havingtransactions wait for other transactions' locks to clear, leading to higher throughput than otherconcurrencycontrol methods.

Optimisticconcurrencycontroltransactionsinvolvethesephases:

- **Begin**:Recordatimestampmarkingthetransaction's beginning.
- **Modify**:Readdatabasevalues, andtentativelywritechanges.
- **Validate**: Check whether other transactions have modified data that this transaction hasused (read or written). This includes transactions that completed after this transaction's starttime,and optionally, transactionsthatarestill activeat validation time.
- **Commit/Rollback**: If there is no conflict, make all changes take effect. If there is aconflict, resolve it, typically by aborting the transaction, although other resolution schemesarepossible.

The stateless nature of HTTP makes locking infeasible for web user interfaces. It's common for auser to start editing a record, then leave without following a "cancel" or "logout" link. If lockingis used, other users who attempt to edit the same record must wait until the first user's lock timesout.

Some database management systems offer Optimistic concurrency control natively - withoutrequiring special application code. For others, the application can implement an OCC layeroutside of thedatabase,andavoidwaitingorsilentlyoverwritingrecords.Insuchcases,the form includes a hidden field with the record's original content, a timestamp, a sequencenumber, or an opaque token. On submit, this is compared against the database. If it differs, theconflictresolution algorithm is invoked.

Timestamp based Concurrency Control

Concurrency Controlcan be implemented in differentways. One way to implementitisbyusingLocks. Now, lets discuss aboutTimeStamp OrderingProtocol.

As earlier introduced, **Timestamp** is a unique identifier created by the DBMS to identify atransaction. They are usually assigned in the order in which they are submitted to the system.RefertothetimestampofatransactionTas **TS(T)**.ForbasicsofTimestampyoumayreferhere.

Timestamp Ordering Protocol–

The main idea for this protocol is to order the transactions based on their Timestamps.Aschedule in which the transactions participate is then serializable and the only equivalent serialschedule permitted has the transactions in the order of their Timestamp Values. Stating simply,thescheduleisequivalenttotheparticularSerialOrdercorrespondingtotheorderoftheTransaction timestamps. Algorithm must ensure that, for each items accessed by ConflictingOperations in the schedule, the order in which the item is accessed does not violate the ordering.Toensurethis, use twoTimestampValues relating to eachdatabaseitem**X**.

- **W_TS(X)** is the largest timestamp of any transaction thatexecuted**write(X)**successfully.
- **R_TS(X)**isthelargesttimestampofanytransactionthatexecuted**read(X)**successfully.

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.

BasicTimestampOrdering–

Every transaction is issued a timestamp based on when it enters the system. Suppose, if an oldtransaction T_i hastimestampTS(T_i), a new transaction T_j isassignedtimestampTS(T_j) suchthat**TS(T_i)<TS(T_j)**.Theprotocolmanagesconcurrentexecutionsuchthatthetimestampsdetermine the serializability order. The timestamp ordering protocol ensures that any conflictingreadand write operationsare executed in timestamporder. Whenever some Transaction T triestoissue aR_item(X)oraW_item(X),theBasicTOalgorithm compares thetimestampof T with **R_TS(X) & W_TS(X)** to ensure that the Timestamp order is not violated.ThisdescribetheBasic TO protocol in followingtwo cases.

1. WheneveraTransactionTissuesa**W_item(X)**operation,checkthefollowingconditions:

   - If **R_TS(X)>TS(T)**orif**W_TS(X)>TS(T)**,thenabortandrollbackTandrejecttheopera tion. else,
   - ExecuteW_item(X)operationofTandsetW_TS(X)toTS(T).

2. Whenevera TransactionTissuesa**R_item(X)**operation,checkthefollowing conditions:

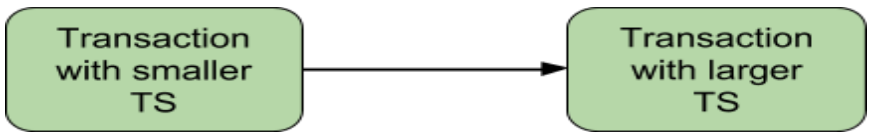
   - If**W_TS(X)>TS(T)**,thenabortandreject Tandrejecttheoperation,else
   - IfW_TS(X)<=TS(T),thenexecutetheR_item(X)operationofTandsetR_TS(X)to thelarger ofTS(T)andcurrent R_TS(X).

Whenever the Basic TO algorithm detects twp conflicting operation that occur in incorrect order,it rejects the later of the two operation by aborting the Transaction that issued it. Schedulesproduced by Basic TO are guaranteed to be conflict serializable. Already discussed that usingTimestamp,can ensurethat our schedulewill bedeadlockfree.

One drawback of Basic TO protocol is that it **Cascading Rollback** is still possible. Suppose wehave a Transaction $T_1$and T_2has used a value written by T_1. If $T_1$is aborted and resubmitted tothe system then, T must also be aborted and rolled back. So the problem of Cascading aborts stillprevails.

Let'sgisttheAdvantagesandDisadvantagesofBasicTOprotocol:

- TimestampOrderingprotocolensuresserializablitysincetheprecedencegrap hwillbeoftheform:



Image–PrecedenceGraph forTSordering

- Timestampprotocolensuresfreedomfromdeadlockasnotransactioneverwaits.
- Buttheschedulemaynotbecascadefree,andmaynoteven berecoverable.

StrictTimestampOrdering–

A variation of Basic TO is called**Strict TO** ensures that the schedules are both Strict andConflict Serializable. In this variation, a Transaction T that issues a R_item(X) or

W_item(X)such that TS(T) > W_TS(X) has its read or write operation delayed until the Transaction **T'** thatwrotethevalues ofXhascommitted oraborted.

Multiversion timestamp ordering write rule: As any potentially conflicting read operation willhave been directed to the most recent versionof an object, the server inspects the versionDmaxEarlier with the maximum write timestamp less than or equal to Tc. We have the followingruleforperformingawrite operation requestedbytransaction Tcon object

D:

if(readtimestampofDmaxEarlierTc)performwriteoperationonatentativeversionofDwithwriteti

mestamp Tc

elseaborttransactionTc

Theobjectalreadyhascommittedversions withwritetimestamps T1andT2.
Theobjectreceivesthefollowing

sequenceofrequests foroperations ontheobject:

T3read;T3write;T5read;T4 write.

1. T3requests areadoperation,whichputsareadtimestampT3onT2'sversion.

2. T3 requests a write operation, which makes a new tentative version with writetimestamp T3.

3. T5requestsareadoperation,whichusestheversionwithwritetimestampT3(thehighesttime stampthat is less than T5).

4. T4 requests a write operation, which is rejected because the read timestamp T5 of the versionwith write timestamp T3 is bigger than T4. (If it were permitted, the write timestamp of the newversionwouldbeT4.Ifsuchaversionwereallowed,thenitwouldinvalidateT5'sreadoperation, which shouldhaveusedtheversion with timestampT4.)

DistributedTwo-phaseLockingAlgorithm

The basic principle of distributed two-phase locking is same as the basic two-phase lockingprotocol. However, in a distributed system there are sites designated as lock managers. A lockmanager controls lock acquisition requests from transaction monitors. In order to enforce co-ordination between the lock managers in various sites, at least one site is

given the authority toseeall transactions and detect lock conflicts.

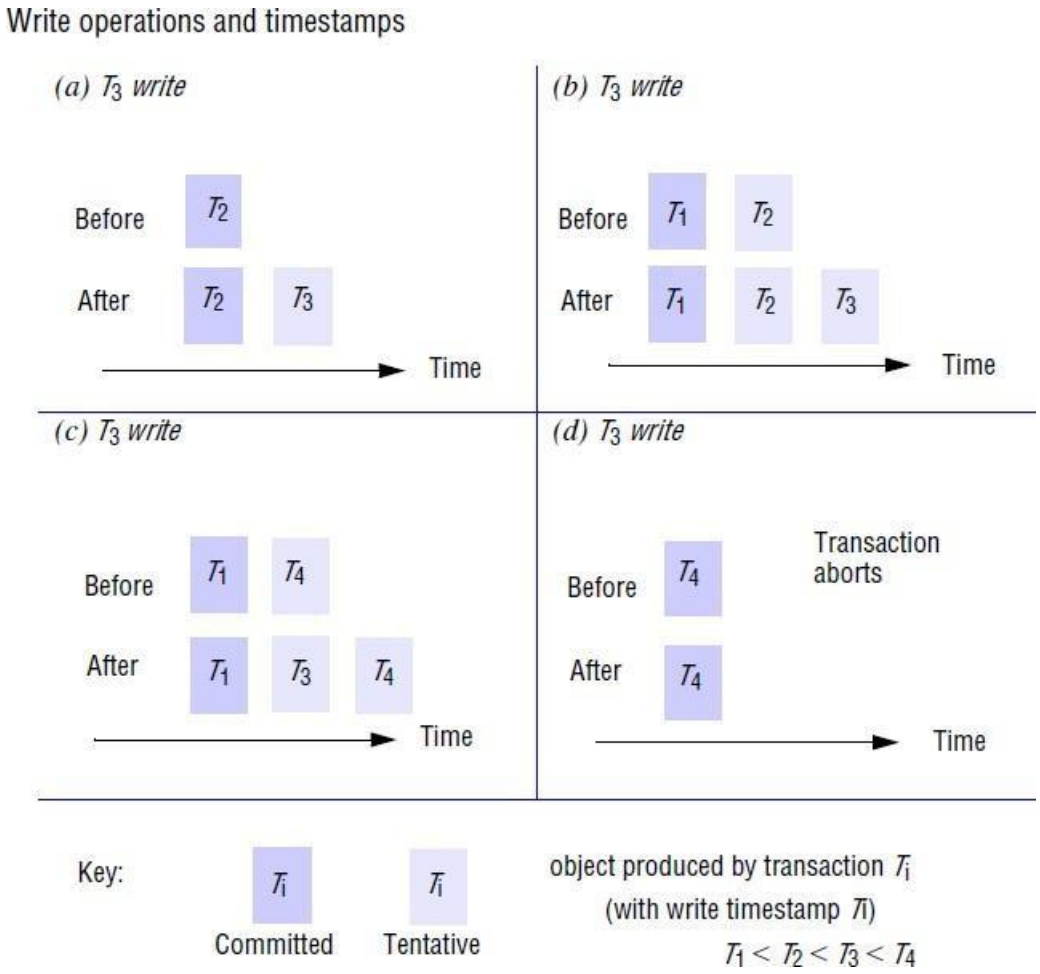
Depending uponthe numberof siteswhocandetectlockconflicts,distributedtwo-phaselockingapproaches can beofthreetypes −

- **Centralized two-phase locking** − In this approach, one site is designated as the centrallock manager. All the sites in the environment know the location of the central lockmanager and obtain lockfrom it duringtransactions.
- **Primary copy two-phase locking** − In this approach, a number of sites are designated aslock control centers. Each of these sites has the responsibility of managing a defined setoflocks.Allthesitesknowwhichlockcontrolcenterisresponsibleformanaginglock ofwhich data table/fragment item.
- **Distributed two-phase locking** − In this approach, there are a number of lock managers,whereeachlockmanagercontrolslocksofdata itemsstoredatitslocalsite.Thelocationofthelock manageris based upondata distribution and replication.

DistributedTimestampConcurrencyControl

In a centralized system,timestamp of any transaction is determined by the physical clockreading. But, in a distributed system, any site's local physical/logical clock readings cannot beused as global timestamps, since they are not globally unique. So, a timestamp comprises of acombinationofsiteIDand thatsite'sclock reading.

For implementing timestamp ordering algorithms, each site has a scheduler that maintains aseparate queue for each transaction manager. During transaction, a transaction manager sends alockrequesttothesite'sscheduler.Thescheduler putstherequesttothecorrespondingqueueinincreasing timestamp order. Requests are processed from the front of the queues in the order oftheirtimestamps, i.e. theoldest first.



Validation of transactions • Validation uses the read-write conflict rules to ensure that thescheduling of a particular transaction is serially equivalent with respect to all other overlappingtransactions–

thatis,anytransactionsthathadnotyetcommittedatthetimethistransaction

started. To assist in performing validation, each transaction is assigned a transaction numberwhen it enters the validation phase (that is, when the client issuesa closeTransaction). If thetransaction is validated and completes successfully, it retains this number; if it fails the validationchecksandisaborted,or ifthetransactionis readonly,thenumberisreleasedforreassignment.

Transaction numbers are integers assigned in ascending sequence; the number of a transactiontherefore defines itsposition in time – a transaction always finishes its working phase after alltransactions with lower numbers. That is, a transaction with the number Ti always precedes atransaction with the number Tj if i <j. (If the transaction number were to be assigned at thebeginning of the working phase, then a transaction that reached the end of the working phasebefore one with a lower number would have to wait until the earlier one had

completed before itcouldbevalidated.)ThevalidationtestontransactionTvisbasedonconflictsbetweenoperationsin pairs oftransactionsTi and Tv.

Comparisonofmethods forconcurrencycontrol:

We have described three separate methods for controlling concurrent access to shared data: stricttwo-phase locking, optimistic methods and timestamp ordering. All of the methods carry someoverheads in the time and space they require, and they all limit to some extent the potential forconcurrentoperation.

The timestamp ordering method is similar to two-phase locking in that both use pessimisticapproaches in which conflicts between transactions are detected as each object is accessed. Onthe one hand, timestamp ordering decides the serialization order statically – when a transactionstarts.On theotherhand,two-phaselockingdecides the

serializationorderdynamically–accordingtotheorderinwhichobjectsareaccessed.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 intransactions are predominantly updates. Some work uses the observation that timestamp orderingisbeneficialfortransactionswithpredominantlyreadoperationsandthatlockingisbenefici alfor transactions with more writes than reads as an argument for allowing hybrid schemes inwhichsometransactions use timestamp orderingand others uselockingforconcurrencycontrol.

The pessimistic methods differ in the strategy used when a conflicting access to an object isdetected. Timestamp ordering aborts the transaction immediately, whereas locking makes thetransaction wait–but with a possible laterpenaltyofabortingto avoid deadlock.

Distributedtransactionsmaybeeitherflatornested:

An atomic commit protocol is a cooperative procedure used by a set of serversinvolved in adistributed transaction. It enables the servers to reach a joint decision as towhether a transactioncanbecommitted oraborted.

Servers that provide transactions include a recovery manager whose concern is toensure that theeffects of transactions on the objects managed by a server can berecovered when it is replacedafterafailure.Therecoverymanagersavestheobjectsinpermanentstoragetogetherwithint entionslists and information about thestatus ofeachtransaction.

In the general case, a transaction, whether flat or nested, will access objectslocated in severaldifferentcomputers.Weusetheterm*distributedtransaction*torefertoaflatornestedtransacti on that accesses objects managed by multiple servers.When a distributed transactioncomes to an end, the atomicity property oftransactions requires that either all of the serversinvolved commit the transaction or allof them abort the transaction. To achieve this, one of theservers takes on a *coordinator*role, which involves ensuring the same outcome at all of theservers.Themanner in

which the coordinator achieves this depends on the protocol chosen. A protocol knownas the‘two-phase commit protocol’ is the most commonly used. This protocol allowsthe servers tocommunicatewith oneanotherto reachajointdecision as towhethertocommitor abort.

Flatandnesteddistributedtransactions

Client transaction becomes distributed if it invokes operations in several differentservers. Thereare two different ways that distributed transactions can be structured: asflat transactions and asnestedtransactions.

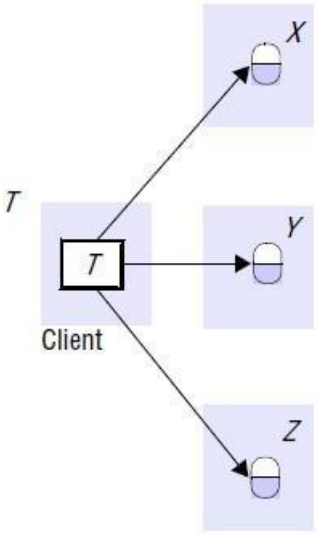
In a flat transaction, a client makes requests to more than one server. For example, transaction *T*isaflattransactionthatinvokesoperationsonobjectsinservers*X*,*Y*and*Z*.Aflatclienttransactionco mpleteseachof itsrequestsbeforegoingontothe nextone.Therefore,eachtransaction accesses servers’ objects sequentially.When servers use locking, a transaction canonlybewaitingforoneobject at a time.

Inanestedtransaction,thetop-leveltransactioncanopensubtransactions,andeachsubtransactioncanopenfurthersubtransactions downtoanydepthofnesting,aclienttransaction *T* that opens two subtransactions, *T*1 and *T*2,which access objects at servers *X* and *Y*.The subtransactions*T*1 and*T*2 open furthersubtransactions*T*11, *T*12,*T*21, and*T*22, whichaccessobjects at servers*M*, *N* and *P*.

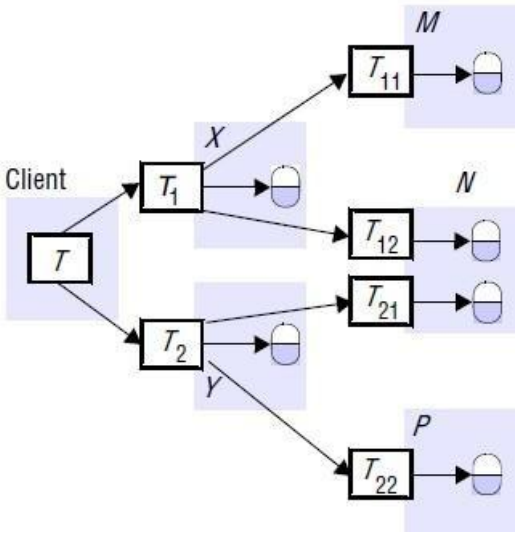
In thenested case, subtransactions at the same level can run concurrently, so *T1* and *T2* areconcurrent,andastheyinvokeobjectsindifferentservers,theycanruninparallel.Thefoursubtran sactions*T*11, *T*12, *T*21 and*T*22 also run concurrently.

Distributed transactions

(a) Flat transaction



(b) Nested transactions




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

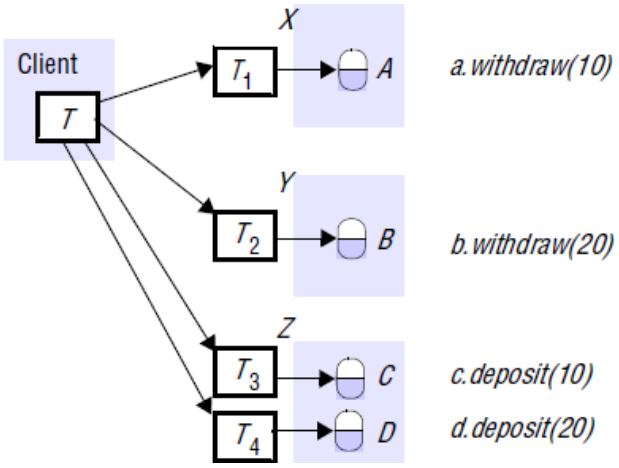
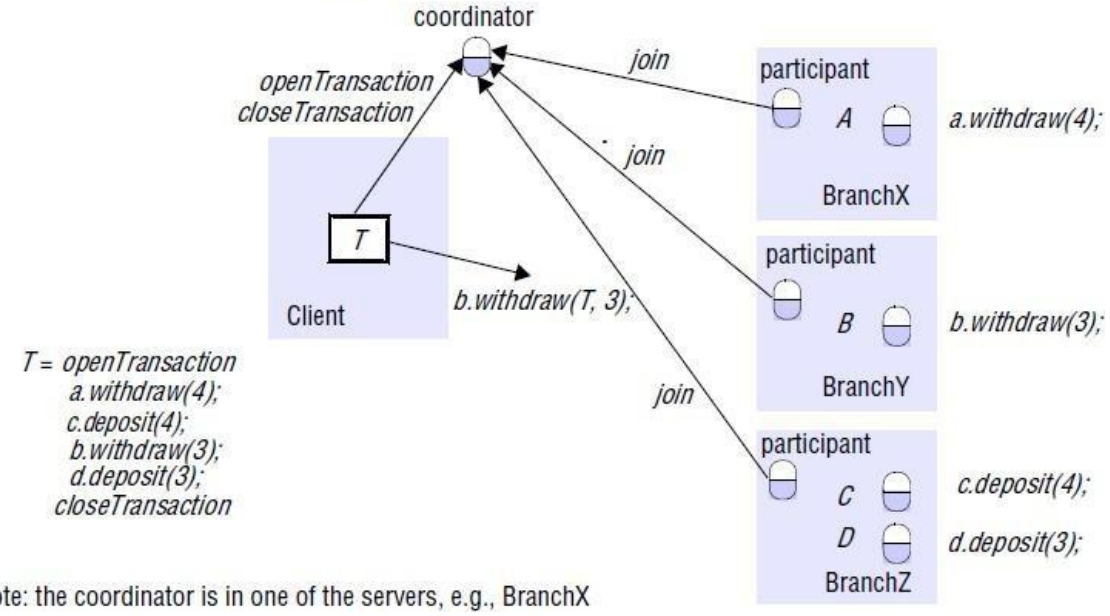


Figure 17.3 A distributed banking transaction



Note: the coordinator is in one of the servers, e.g., BranchX

Servers that execute requests as part of a distributed transaction need to be able to communicate with one another to coordinate their actions when the transaction commits. A client starts a transaction by sending an *openTransaction* request to a coordinator in any server, as described in Section 16.2. The coordinator that is contacted carries out the *openTransaction* and returns the resulting transaction identifier (TID) to the client. Transaction identifiers for distributed transactions must be unique within a distributed system. A simple way to achieve this is for a TID to contain two parts: the identifier (for example, an IP address) of the server that created it and a number unique to the server.

The coordinator that opened the transaction becomes the coordinator for the distributed transaction and at the end is responsible for committing or aborting it. Each of the servers that manages an object accessed by a transaction is a participant in the transaction and provides an object we call the *participant*. Each participant is responsible for keeping track of all of the recoverable objects at that server that are involved, in the transaction. The participants are responsible for cooperating with the coordinator in carrying out the commit protocol.

During the progress of the transaction, the coordinator records a list of references to the participants, and each participant records a reference to the coordinator.

The interface for *Coordinator* shown in Figure 13.3 provides an additional method, *join*, which is used whenever a new participant joins the transaction:

Atomic commit protocols:

Transactioncommitprotocolsweredevisedintheearly1970s,andthetwo-phasecommitprotocol appeared in Gray [1978]. The atomicity property of transactionsrequires that when adistributed transaction comes to an end, either all of its operationsare carried out or none of them.In the case of a distributed transaction, the client hasrequested operations atmore than oneserver.

A transaction comes to an end when theclient requests that it be committed or aborted. A simpleway to complete the transactionin an atomic manner is for the coordinator to communicate thecommit or abort requestto all of the participants in the transaction and to keep on repeating therequest until allof them have acknowledged that they have carried it out. This is an example of aonephaseatomic commit protocol.

This simple one-phase atomic commit protocol is inadequate, though, because itdoes not allow aserver to make a unilateral decision to abort a transaction when theclient requests acommit.Reasonsthatpreventaserverfrombeingabletocommititspartofatransactiongenerallyrelatetoissuesofconcurrencycontrol.Forexample,iflockingisinuse,theresolutionofadeadlockcan lead to the aborting of a transactionwithout the client being aware unless it makes anotherrequest to the server. Also ifoptimistic concurrency control is in use, the failure of validation at aserver would causeit to decide to abort the transaction. Finally, the coordinator may not know if aserver hascrashed and been replaced during the progress of a distributed transaction – such aserverwill need to abort thetransaction.

Thetwo-phasecommitprotocolisdesignedtoallowany participanttoabortitspartofatransaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then thewhole transaction must be aborted.In the first phase of the protocol, each participant votes forthe transaction to be committed or aborted. Once a participanthas voted to commit a transaction,it is not allowed to abort it. Therefore, before a participant votes to commit a transaction, it mustensure that it will eventually be able to carry out its part of the commit protocol, even if it failsand is replaced in the interim. A participant in a transaction is said to be in a prepared state for atransaction if it willeventually be able to commit it. To make sure of this, each participant savesin permanent storage allof the objects that it has altered in the transaction, together with itsstatus– prepared.

In the second phase of the protocol, every participant in the transaction carries out the jointdecision.Ifanyoneparticipantvotestoabort,thenthedecisionmustbetoabortthetransaction. If

alltheparticipants votetocommit,then thedecisionisto committhetransaction.

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.

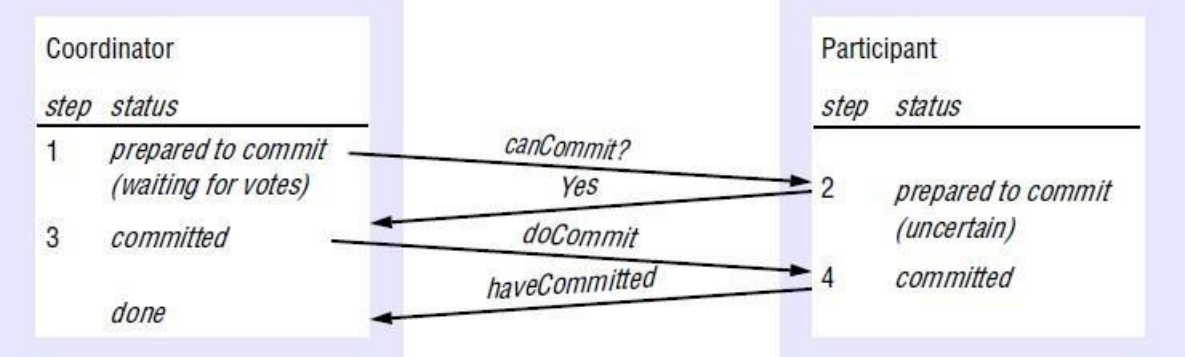
The two-phase commit protocol

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.



Concurrency control in distributed transactions

Each server manages a set of objects and is responsible for ensuring that they remain consistentwhen accessed by concurrent transactions. Therefore, each server is responsible for applyingconcurrency control to its own objects. The members of a collection of servers of distributedtransactions are jointly responsible for ensuring that they are performed in a serially equivalentmanner.

This implies that if transaction T is before transaction U in their conflicting access to objects atone of the servers, then they must be in that order at all of the servers whose objects are accessedinaconflictingmanner byboth TandU.

Locking:

Inadistributedtransaction,thelocksonanobjectare heldlocally(inthesameserver).Thelocallockmanagercandecidewhethertograntalockormakether equestingtransactionwait.However,itcannotreleaseanylocksuntilitknowsthatthetransactionhasb eencommittedorabortedatalltheserversinvolvedinthetransaction.Whenlockingisusedforconcurr encycontrol,theobjectsremainlockedandareunavailableforothertransactionsduringtheatomicco mmit protocol, although an aborted transaction releases its locks after phase 1 of the protocol.Aslockmanagersindifferentserverssettheirlocksindependentlyofoneanother,itispossibl ethatdifferent servers mayimposedifferent orderings on transactions.

ConsiderthefollowinginterleavingoftransactionsTand Uatservers X and Y:

ThetransactionTlocksobjectAatserverX,andthentransactionUlocksobjectBatserverY.After that,TtriestoaccessBatserverYandwaitsforU'slock.Similarly,transactionUtriesto

accessAatserverXandhastowaitforT'slock.

Therefore,wehaveTbeforeUinoneserverandUbeforeTin theother.Thesedifferent orderings can

leadtocyclicdependencies betweentransactions,givingrisetoadistributeddeadlocksituation.

T			*U*		
write(A)	at *X*	locks *A*			
			write(B)	at *Y*	locks *B*
read(B)	at *Y*	waits for *U*			
			read(A)	at *X*	waits for *T*

Distributed dead locks:

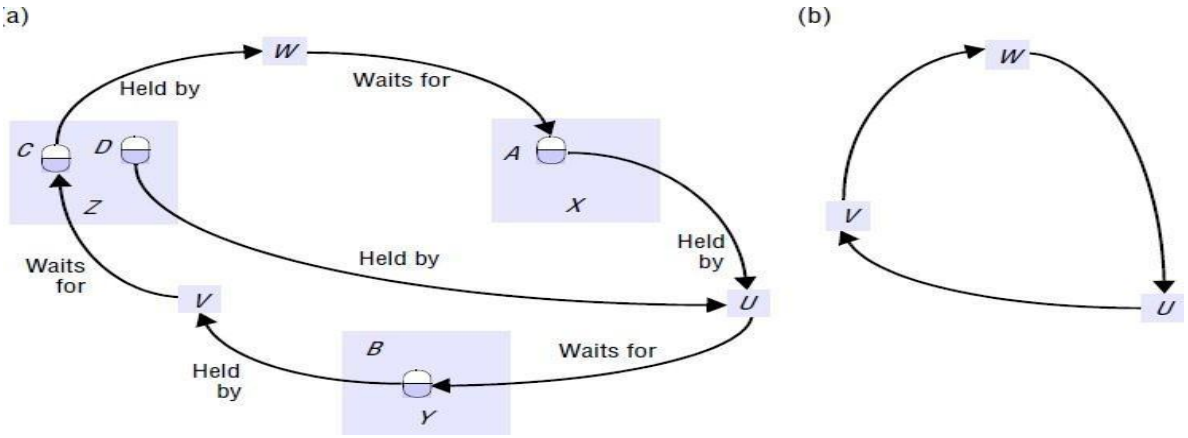
Deadlocks can arise within a single server when locking is used for concurrency control. Serversmusteitherpreventordetectandresolvedeadlocks.Usingtimeoutstoresolvepossibledeadlock.It is difficult to choose an appropriate timeout interval, and transactions may be abortedunnecessarily.Withdeadlockdetectionschemes,atransactionisabortedonlywhenitisinvolvedinadeadlock.Mostdeadlockdetectionschemesoperateby finding cyclesinthetransaction wait-for graph. In a distributed system involving multiple servers being accessed bymultiple 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, therecan be a *distributed deadlock*. Recall that the wait-for graph is a directed graph in which nodesrepresent transactions and objects, and edges represent either an object held by a transaction or atransaction waiting for an object. There is a deadlock if and only if there is a cycle in the wait-forgraph.

DISTRIBUTEDTRANSACTIONS:

U		*V*		*W*	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B*		
a.deposit(20)	lock *A*		at *Y*		
	at *X*				
				c.deposit(30)	lock *C*
b.withdraw(30)	wait at *Y*				at *Z*
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*

DISTRIBUTEDDEADLOCK:

Whenitfindsacycle,it makesadecisiononhowtoresolvethe deadlockandtellstheservers whichtransactiontoabort.



Centralizeddeadlockdetectionisnotagoodidea,becauseitdependsonasingleservertocarryitout.Its uffersfromthe usual problems associated with centralizedsolutions in distributed systems – poor availability, lack of faulttolerance and no abilityto scale. In addition, the cost of the frequent transmission of local wait-for graphs ishigh. Iftheglobal graphiscollectedlessfrequently,deadlocksmaytake longertobedetected.

Phantom deadlocks • A deadlock that is 'detected' but is not really a deadlock iscalledaphantom deadlock. In distributed deadlock detection, information about wait-forrelationshipsbetweentransactionsistransmittedfromoneservertoanother.Ifthereisadeadlock,t henecessary information will eventually be collected in one place and acycle will be detected. Asthis procedure will take some time, there is a chance that oneof the transactions that holds a lockwill meanwhile havereleasedit, in whichcase thedeadlock will nolonger exist.

Edgechasing•Adistributedapproachtodeadlockdetectionusesatechniquecallededgechasing or path pushing. In this approach, the global wait-for graph is notconstructed, but each ofthe servers involved has knowledge about some of its edges.The servers attempt to find cycles byforwardingmessagescalledprobes,whichfollowtheedgesofthegraphthroughoutthedistributed system. A probe message consists oftransaction wait-for relationships representing apathin theglobal wait-for graph.

Transactionrecovery:

The atomic property of transactions requires that all the effects of committedtransactions andnoneoftheeffectsofincompleteorabortedtransactionsarereflectedintheobjectstheyaccessed.

Thispropertycanbedescribedintermsoftwoaspects:durabilityandfailureatomicity.Durability requires that objects are saved in permanentstorage and will be available indefinitelythereafter. Therefore an acknowledgement ofa client's commit request implies that all the effectsof the transaction have beenrecorded in permanent storage as well as in the server's (volatile)objects.

Failureatomicityrequiresthateffectsoftransactionsareatomicevenwhen theservercrashes.

Recovery is concerned with ensuring that a server's objects are durable and that theserviceprovides failureatomicity.

Therequirements fordurabilityandfailureatomicityarenot reallyindependent

of one another and can be dealt with by a single mechanism – the *recovery manager*.Thetasks ofarecoverymanagerare:

- tosaveobjectsin permanent storage(inarecoveryfile)forcommittedtransactions;
- torestoretheserver'sobjects afteracrash;
- toreorganizetherecoveryfiletoimprovetheperformanceofrecovery;
- toreclaimstoragespace(intherecoveryfile).

Logging:

- In the logging technique, the recovery file represents a log containing the history of allthetransactions performed by a server. The history consists of values of objects,transaction status entriesandtransactionintentions lists.Theorder oftheentriesinthelogreflectstheorderinwhichtransactionshaveprepared, committed and abortedat thatserver.
- In practice, the recovery file will contain a recent snapshot of the values of all theobjects in the serverfollowed by a history of transactions postdating the snapshot.During the normal operation of a server,its recovery manager is called whenevera transaction prepares to commit, commits or aborts atransaction. When the server isprepared to commit a transaction, the recovery manager appends all theobjects in itsintentions list to the recovery file, followed by the current status of thattransaction(prepared) together with its intentions list.

MALLAREDDYCOLLEGEOFENGINEERING&TECHNOLOGY

(AutonomousInstitution–UGC,Govt.ofIndia)

IIIB.TechIISemester,Model Paper-I

DistributedSystems

(CSE&IT)

RollNo										

Time:3 hours **Max.Marks:70**

Note: ThisquestionpaperConsistsof5Sections.Answer**FIVE**Questions,ChoosingONEQuestionfrom eachSECTIONandeachQuestioncarries14marks.

*****SECTION-I**

1	a)Discusshowdistributedsystemsaremorescalablethanthecentralizedsystems	**[7M]**
	b)Demonstratethedesignrequirementsfordistributedarchitectures	**[7M]**
	OR	
2	Explaindifferenttypes offailureswithexamples.	**[14M]**
	SECTION-II	
3	a)Explainexternalsynchronizationandinternalsynchronization.	**[7M]**
	b)DefineConsistent-globalstates,consistentcutandconsistentrun	**[7M]**
	OR	
4	a)describeindetailaboutfailuredetectors b)ExplainaboutMaekawa'salgorithm.	**[14M]**
	SECTION-III	
5	a)Writeaboutgroupcommunication.	**[8M]**
	b)What isamiddleware?Explainthevariouslayerspresentinit.	**[6M]**
	OR	
6	a)WriteshortnotesExternaldatarepresentation	**[8M]**
	b)DiscussindetailaboutCORBA'scommondatarepresentation	**[6M]**
	SECTION-IV	
7	a)Discussvariousfilesystemoperations.	**[7M]**
	b)DiscussindetailaboutCORBA'scommondatarepresentation	**[7M]**
	OR	
8	writeabout groupcommunication.	**[7M]**
	Whatisamiddleware?Explainthevariouslayers presentinit.	**[7M]**
	SECTION-V	
9	a)ExplainaboutOptimisticconcurrencycontrol. b)ExplainTimestampordering	**[14M]**
	OR	
10	Explaindistributeddeadlockdetectionmechanismwithexample.	**[14M]**

****

R17

CodeNo:XXXXXX

MALLAREDDYCOLLEGEOFENGINEERING&TECHNOLOGY

(AutonomousInstitution–UGC,Govt.ofIndia)

IIIB.TechIISemester,ModelPaper-II

DistributedSystems

(CSE&IT)

RollNo										

Time:3 hours **Max.Marks:70**

Note: This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question fromeachSECTIONandeachQuestioncarries14marks.

*****SECTI ON-I**

1	a)Discussinbriefthemain featuresofHTTP	**[7M]**
	b)Listandexplainthetechniquesusedfordealingwithfailures	**[7M]**
	OR	
2	Explainbrieflyaboutarchitecturalmodels	**[14M]**

SECTION-II

3	1.)Discussinbriefabout, i. Mobileagents ii. ThinClients iii. NetworkComputers	**[14M]**
	OR	
4	Whatissignificanceoffailuremodels?Explainin detailthetaxonomythat distinguishesbetweenhe failuresofprocessesandcommunicationchannels.	**[7M]**
	Whataretheproblemsofdistributedsystems?	**[7M]**

SECTION-III

5	ExplaintheElectionalgorithmswith examples.	**[14M]**
	OR	
6	Explainthealgorithm formutualexclusionusingmulticast andlogicalclocks **[6M]**What is meant by interprocess communication? How interprocess communication **[8M]**isusedin distributed systems	

SECTION-IV

7	WhataretheSixbuildingblocksofanXMLdocument?GiveExamples.	**[14M]**
	OR	
8	a)DrawandexplainthearchitectureofSUNNetworksFileSystem	**[7M]**
	b)Whatarethe variousoperations providedbyNFS Server	**[7M]**

SECTION-V

9	a)Discussinbriefaboutthe “ACID” PropertiesofTransactions	**[7M]**
	b)Explainwithanexamplehowtwotransactionsareinterleavedwhichareserially equivalentateachserverbutisnotseriallyequivalentglobally?	**[7M]**
	OR	
10	ExplainconcurrencycontrolinDistributedtransactions.	**[14M]**

****

R17

CodeNo:XXXXXX

MALLAREDDYCOLLEGEOFENGINEERING&TECHNOLOGY

(AutonomousInstitution–UGC,Govt.ofIndia)

IIIB.TechIISemester,ModelPaper-III

DistributedSystems(

CSE&IT)

RollNo										

Time:3 hours **Max.Marks:70**

Note: ThisquestionpaperConsistsof5Sections.Answer**FIVE**Questions,ChoosingONEQuestionfrom eachSECTIONandeachQuestioncarries14marks.

***

SECTION-I

1 Explaindifferentchallengesfacedbydistributedsystemswithexamples **[14M]**

OR

2 Writeshortnotesonfundamentalmodels **[14M]**

SECTION-II

3 a)Defineinteractingprocesses.Alsodiscusstwosignificantfactors effectinginteractionprocessesinDistributedsystem. **[7M]**

b)ExplainindetailHTML. **[7M]**

OR

4 (a)Whatisaneedofelectionalgorithm.Explainringbasedelectionalgorithm. **[7M]**

(b)Whataretheessentialfeaturesofmulticastcommunication? **[7M]**

SECTION-III

5 a)Writeabouttheorderingofmessages. **[6M]**

b)Explainthealgorithmtosolveconsensusprobleminasynchronoussystem. **[8M]**

OR

6 (a)Discussindetailaboutrequest replyprotocolandRPC. **[7M]**

(b)WriteaboutinterprocesscommunicationinUNIX. **[7M]**

SECTION-IV

7 (a)WhatarethedesigncharacteristicsofAndrewfilesystem.Howisthedistrib utionof processesdoneinAFS? **[7M]**

****

R17

CodeNo:XXXXXX

MALLAREDDYCOLLEGEOFENGINEERING&TECHNOLOGY

(AutonomousInstitution–UGC,Govt.ofIndia)

IIIB.TechIISemester,ModelPaper-IV

DistributedSystems

(CSE&IT)

RollNo										

Time:3 hours **Max.Marks:70**

Note: This question paper Consists of 5 Sections. Answer **FIVE** Questions, Choosing ONE Question fromeachSECTIONandeachQuestioncarries14marks.

***

SECTION-I

1	a)Whatarethedifferentmethodsofsharingresourcesindistributedsystems.	**[7M]**
	b) Explainaboutmobileandubiquitouscomputing.	**[7M]**
	OR	
2	a)Explaininteractionmodels.	**[7M]**
	b)ExplaindifferentvariationsinClientServermodel.	**[7M]**

SECTION-II

3	(a)Whatisthe importanceoftimeindistributedsystems	**[7M]**
	(b)Describethealgorithmforexternalsynchronization	**[7M]**
	OR	
4	(a)WhatisConsensusProblem	**[14M]**
	(b)Discussthetwo implementationofreliablemulticast	

SECTION-III

5	ExplainRPCwithaneatexample.	**[14M]**
	OR	
6	a)Listanddiscussthecharacteristicsofnetworkthatarehiddenbythestream obstruction	**[8M]**
	b)DiscussindetailaboutHTTPProtocol	**[6M]**

SECTION-IV

7	a)Give anOverViewofTypesofStorageSystemsandtheir properties	**[7M]**
	b) Explainthefileservicearchitecturewithaneatdiagram	**[7M]**
	OR	
8	a)Whatarethe requirementsforthedesignofdistributedfile system	**[8M]**
	b)Writeabout (i) HierarchicFileSystems (ii) FileGroups	**[6M]**

SECTION-V

9	a)Discussinbriefaboutthe "ACID" PropertiesofTransactions	**[6M]**
	b)Writeshortnotesonlocksforconcurrencycontrol.	**[8M]**
	OR	
10	Explaindifferenttransactionrecoverymechanismsindistributedtransactions.	**[14M]**