UNIT I

Characterization of Distributed Systems: Introduction, Examples of Distributed systems, Resource sharing and web, challenges.

System Models: Introduction, Architectural and Fundamental models.

Introduction

A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal.

Distributed systems Principles

A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

Centralised System Characteristics

- 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

Distributed System Characteristics

- 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

Examples of distributed systems and applications of distributed computing include the following:

- telecommunication networks:
- telephone networks and cellular networks,

  - computer networks such as the Internet,
  - wireless sensor networks,
  - routing algorithms;
  - World wide web and peer-to-peer networks,
  - massively multiplayer online games and virtual reality communities,
  - distributed databases and distributed database management systems,
  - network file systems,
  - distributed information processing systems such as banking systems and airline reservation systems;
- real-time process control:
  - aircraft control systems,
  - industrial control systems;
- parallel computation:
  - scientific computing, including cluster computing and grid computing and various volunteer computing projects (see the list of distributed computing projects),
  - distributed rendering in computer graphics.

RESOURCE SHARING

- Is the primary motivation of distributed computing
- Resources types
  - Hardware, e.g. printer, scanner, camera
  - Data, e.g. file, database, web page
  - More specific functionality, e.g. search engine, file
- *Service*
  - manage a collection of related resources and present their functionalities to users and applications
- *Server*
  - a process on networked computer that accepts requests from processes on other computers to perform a *service* and responds appropriately
- *Client*
  - the requesting process
- *Remote invocation*

THE CHALLENGES IN DISTRIBUTED SYSTEM:

Heterogeneity

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

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

Although the Internet consists of many different sorts of network, their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network.

Data types such as integers may be represented in different ways on different sorts of hardware – for example, there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware. Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

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

Middleware • The term *middleware* applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. The Common Object Request Broker (CORBA), is an example. Some middleware, such as Java Remote Method Invocation (RMI), supports only a single programming language. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems and hardware.

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

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

The Java virtual machine needs to be implemented once for each type of computer to enable Java programs to run.

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

Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures,

which are usually cumbersome and slow-moving. However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people. The designers of the Internet protocols introduced a series of documents called ‘Requests For Comments’, or RFCs, each of which is known by a number. The specifications of the Internet communication protocols were published in this series in the early 1980s, followed by specifications for applications that run over them, such as file transfer, email and telnet by the mid-1980s.

Systems that are designed to support resource sharing in this way are termed *open distributed systems* to emphasize the fact that they are extensible. They may be extended at the hardware level by the addition of computers to the network and at the software level by the introduction of new services and the reimplementation of old ones, enabling application programs to share resources.

To summarize:

• Open systems are characterized by the fact that their key interfaces are published.

• Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.

• Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors. But the conformance of each component to the published standard must be carefully tested and verified if the system is to work correctly.

Security

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

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

1. A doctor might request access to hospital patient data or send additions to that data.

2. In electronic commerce and banking, users send their credit card numbers across the Internet.

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

Denial of service attacks: Another security problem is that a user may wish to disrupt a service for some reason. This can be achieved by bombarding the service with such a large number of pointless requests that the serious users are unable to use it. This is called a *denial of service* attack. There have been several denial of service attacks on well-known web services. Currently such attacks are countered by attempting to catch and punish the perpetrators after the event, but that is not a general solution to the problem.

Security of mobile code: Mobile code needs to be handled with care. Consider someone who receives an executable program as an electronic mail attachment: the possible effects of running the program are unpredictable; for example, it may seem to display an interesting picture but in reality it may access local resources, or perhaps be part of a denial of service attack.

Scalability

Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The number of computers and servers in the Internet has increased dramatically. Figure 1.6 shows the increasing number of computers and web servers during the 12-year history of the Web up to 2005 . It is interesting to note the significant growth in both computers and web servers in this period, but also that the relative percentage is flattening out – a trend that is explained by the growth of fixed and mobile personal computing. One web server may also increasingly be hosted on multiple computers.

The design of scalable distributed systems presents the following challenges:

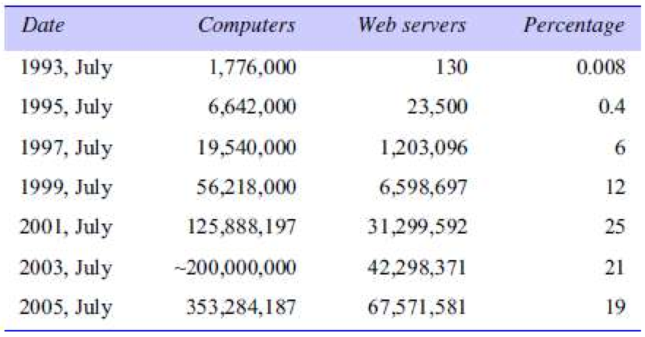
Controlling the cost of physical resources: As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it. For example, the frequency with which files are accessed in an intranet is likely to grow as the number of users and computers increases. It must be possible to add server computers to avoid the performance bottleneck that would arise if a single file server had to handle all file access requests. In general, for a system with n users to be scalable, the quantity of physical resources required to support them should be at most $O(n)$ – that is, proportional to n. For example, if a single file server can support 20 users, then two such servers should be able to support 40 users.

Controlling the performance loss: Consider the management of a set of data whose size is proportional to the number of users or resources in the system – for example, the table with the correspondence between the domain names of computers and their Internet addresses held by the Domain Name System, which is used mainly to look up DNS names such as www.amazon.com. Algorithms that use hierarchic structures scale better than those that use linear structures. But even with hierarchic structures an increase in size will result in some loss in performance: the time taken to access hierarchically structured data is O(*log n*), where *n* is the size of the set of data. For a

system to be scalable, the maximum performance loss should be no worse than this.

Preventing software resources running out: An example of lack of scalability is shown by the numbers used as Internet (IP) addresses (computer addresses in the Internet). In the late 1970s, it was decided to use 32 bits for this purpose, but as will be explained in Chapter 3, the supply of available Internet addresses is running out. For this reason, a new version of the protocol with 128-bit Internet addresses is being adopted, and this will require modifications to many software components.

Growth of the Internet (computers and web servers)



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

Avoiding performance bottlenecks: In general, algorithms should be decentralized to avoid having performance bottlenecks. We illustrate this point with reference to the predecessor of the Domain Name System, in which the name table was kept in a single master file that could be downloaded to any computers that needed it. That was

fine when there were only a few hundred computers in the Internet, but it soon became a serious performance and administrative bottleneck.

Failure handling

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

Detecting failures: Some failures can be detected. For example, checksums can be used to detect corrupted data in a message or a file. It is difficult or even impossible to detect some other failures, such as a remote crashed server in the Internet. The challenge is to manage in the presence of failures that cannot be detected but may be suspected.

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

1. Messages can be retransmitted when they fail to arrive.
2. File data can be written to a pair of disks so that if one is corrupted, the other may still be correct.

Tolerating failures: Most of the services in the Internet do exhibit failures – it would not be practical for them to attempt to detect and hide all of the failures that might occur in such a large network with so many components. Their clients can be designed to tolerate failures, which generally involves the users tolerating them as well. For example, when a web browser cannot contact a web server, it does not make the user wait forever while it keeps on trying – it informs the user about the problem, leaving them free to try again later. Services that tolerate failures are discussed in the paragraph on redundancy below.

Recovery from failures: Recovery involves the design of software so that the state of permanent data can be recovered or 'rolled back' after a server has crashed. In general, the computations performed by some programs will be incomplete when a fault occurs, and the permanent data that they update (files and other material stored
in permanent storage) may not be in a consistent state.

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

1. There should always be at least two different routes between any two routers in the Internet.
2. In the Domain Name System, every name table is replicated in at least two different servers.
3. A database may be replicated in several servers to ensure that the data remains accessible after the failure of any single server; the servers can be designed to detect faults in their peers; when a fault is detected in one server, clients are redirected to the remaining servers.

Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. The process that manages a shared resource could take one client request at a time. But that approach limits throughput.

Therefore services and applications generally allow multiple client requests to be processed concurrently. To make this more concrete, suppose that each resource is encapsulated as an object and that invocations are executed in concurrent threads. In this case it is possible that several threads may be executing concurrently within an object, in which case their operations on the object may conflict with one another and produce inconsistent results.

Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. The implications of transparency are a major influence on the design of the system software.

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

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

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

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

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

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

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

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

Quality of service

Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main nonfunctional properties of systems that affect the quality of the service experienced by clients and users are *reliability*, *security* and *performance*.

Adaptability to meet changing system configurations and resource availability has been recognized as a further important aspect of service quality.

Some applications, including multimedia applications, handle *time-critical data* – streams of data that are required to be processed or transferred from one process to another at a fixed rate. For example, a movie service might consist of a client program that is retrieving a film from a video server and presenting it on the user's screen. For a satisfactory result the successive frames of video need to be displayed to the user within some specified time limits.

In fact, the abbreviation QoS has effectively been commandeered to refer to the ability of systems to meet such deadlines. Its achievement depends upon the availability of the necessary computing and network resources at the appropriate times. This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time (for example, the task of displaying a frame of video).

INTRODUCTION TO SYSTEM MODELS

Systems that are intended for use in real-world environments should be designed to function correctly in the widest possible range of circumstances and in the face of many possible difficulties and threats .

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

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

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

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

Architectural models

The architecture of a system is its structure in terms of separately specified components and their interrelationships. The overall goal is to ensure that the structure will meet present and likely future demands on it. Major concerns are to make the system reliable, manageable, adaptable and cost-effective. The architectural design of a building has similar aspects – it determines not only its appearance but also its general structure and architectural style (gothic, neo-classical, modern) and provides a consistent frame of reference for the design.

Software layers

The concept of layering is a familiar one and is closely related to abstraction. In a layered approach, a complex system is partitioned into a number of layers, with a given layer making use of the services offered by the layer below. A given layer therefore offers a software abstraction, with higher layers being unaware of implementation details, or indeed of any other layers beneath them.

In terms of distributed systems, this equates to a vertical organization of services into service layers. A distributed service can be provided by one or more server processes, interacting with each other and with client processes in order to maintain a consistent system-wide view of the service's resources. For example, a network time service is implemented on the Internet based on the Network Time Protocol (NTP) by server processes running on hosts throughout the Internet that supply the current time to any client that requests it and adjust their version of the current time as a result of interactions with each other. Given the complexity of distributed systems, it is often helpful to organize such services into layers. the important terms *platform* and *middleware*, which define as follows:

The important terms *platform* and *middleware*, which is defined as follows:

A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates communication and coordination between processes. Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM/Symbian are majorexamples.

– Remote Procedure Calls – Client programs call procedures in server programs

– Remote Method Invocation – Objects invoke methods of objects on distributed hosts

– Event-based Programming Model – Objects receive notice of events in other objects inwhich they have interest

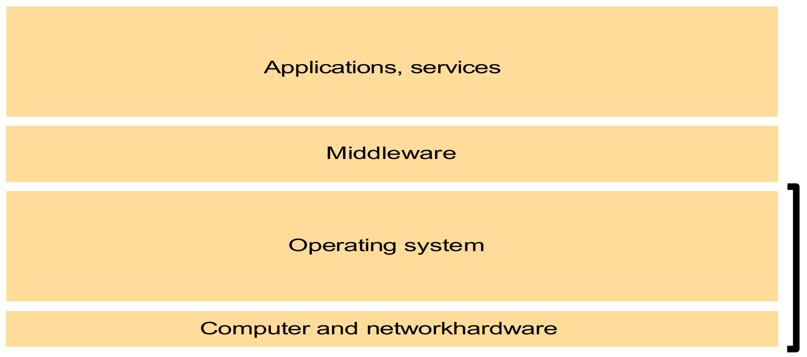
Middleware

• Middleware: software that allows a level of programming beyond processes and message passing

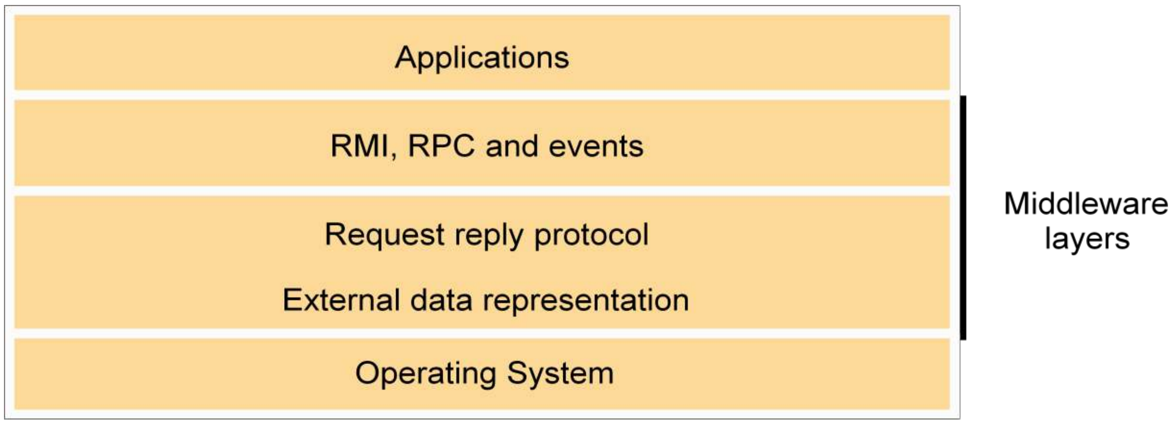
– Uses protocols based on messages between processes to provide its higher-level abstractions

– such as remote invocation and events

– Supports location transparency



– Usually uses an interface definition language (IDL) to define interfaces



Interfaces in Programming Languages

– Current PL allow programs to be developed as a set of modules that communicate with each other. Permitted interact ions between modules are defined by interfaces

– A specified interface can be implemented by different modules without the need to modify other modules using the interface

• **Interfaces in Distributed Systems**

– When modules are in different processes or on different hosts there are limitations on the interactions that can occur. Only actions with parameters that are fully specified and understood can communicate effectively to request or provide services to modules in another process.

– A service interface allows a client to request and a server to provide particular services

– A remote interface allows objects to be passed as arguments to and results from distributed modules.

• **Object Interfaces**

– An interface defines the signatures of a set of methods, including arguments, argument types, return values and exceptions. Implementation details are not included in an interface.

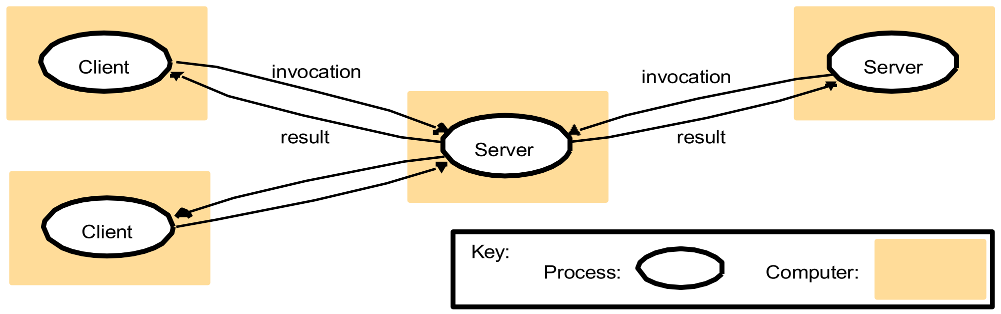
– A class may implement an interface by specifying behavior for each method in the interface. Interfaces do not have constructors.

– System architectures

– Client-server: This is the architecture that is most often cited when distributed systems are discussed. It is historically the most important and remains the most widely employed. Figure 2.3 illustrates the simple structure in which processes take on the roles of being clients or servers. In particular, client processes interact with individual server processes in potentially separate host computers in order to access the shared resources that they manage.

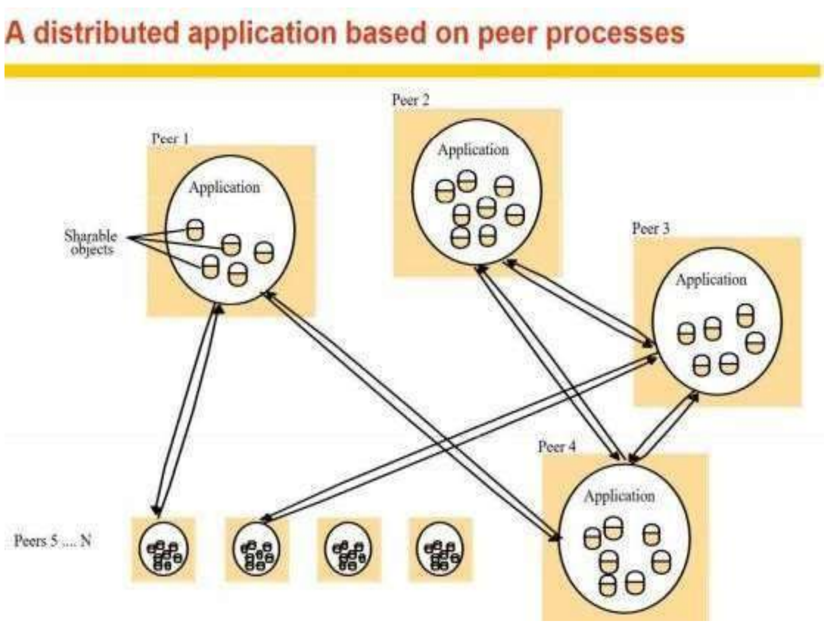
– Servers may in turn be clients of other servers, as the figure indicates. For example, a web server is often a client of a local file server that manages the files in which the web pages are stored. Web servers and most other Internet services are clients of the DNS service, which translates Internet domain names to network addresses.

Clients invoke individual servers



Another web-related example concerns *search engines*, which enable users to look up summaries of information available on web pages at sites throughout the Internet. These summaries are made by programs called *web crawlers*, which run in the background at a search engine site using HTTP requests to access web servers throughout the Internet. Thus a search engine is both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers. In this example, the server tasks (responding to user queries) and the crawler tasks (making requests to other web servers) are entirely independent; there is little need to synchronize them and they may run concurrently. In fact, a typical search engine would normally include many concurrent threads of execution, some serving its clients and others running web crawlers. In Exercise 2.5, the reader is invited to consider the only synchronization issue that does arise for a concurrent search engine of the type outlined here.

Peer-to-peer: In this architecture all of the processes involved in a task or activity play similar roles, interacting cooperatively as *peers* without any distinction between client and server processes or the computers on which they run. In practical terms, all participating processes run the same program and offer the same set of interfaces to each other. While the client-server model offers a direct and relatively simple approach to the sharing of data and other resources, it scales poorly.



A number of placement strategies have evolved in response to this problem, but none of them addresses the fundamental issue – the need to distribute shared resources much more widely in order to share the computing and communication loads incurred in accessing them amongst a much larger number of computers and network links. The key insight that led to the development of peer-to-peer systems is that the network and computing resources owned by the users of a service could also be put to use to support that service. This has the useful consequence that the resources available to run the service grow with the number of users.

Models of systems share some fundamental properties. In particular, all of them are composed of processes that communicate with one another by sending messages over a computer network. All of the models share the design requirements of achieving the performance and reliability characteristics of processes and networks and ensuring the security of the resources in the system.

About their characteristics and the failures and security risks they might exhibit. In general, such a fundamental model should contain only the essential ingredients that need to consider in order to understand and reason about some aspects of a system's behaviour. 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 generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed. The guarantees are
dependent on logical analysis and, where appropriate, mathematical proof.

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

Interaction: Computation occurs within processes; the processes interact by passing messages, resulting in communication (information flow) and coordination (synchronization and ordering of activities) between processes. In the analysis and design of distributed systems we are concerned especially with these interactions. The interaction model must reflect the facts that communication takes place with delays that are often of considerable duration, and that the accuracy with which independent processes can be coordinated is limited by these delays and by the difficulty of maintaining the same notion of time across all the computers in a distributed system.

Failure: The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs (including software faults) or in the network that connects them. Our model defines and classifies the faults. This provides a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.

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

Fundamental Models

Interaction model

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

- Multiple server processes may cooperate with one another to provide a service; the examples mentioned above were the Domain Name System, which partitions and replicates its data at servers throughout the Internet, and Sun’s Network Information Service, which keeps replicated copies of password files at several servers in a local area network.
- A set of peer processes may cooperate with one another to achieve a common goal: for example, a voice conferencing system that distributes streams of audio data in a similar manner, but with strict real-time constraints.

Most programmers will be familiar with the concept of an *algorithm* – a sequence of
steps to be taken in order to perform a desired computation. Simple programs are controlled by algorithms in which the steps are strictly sequential. The behaviour of the program and the state of the program’s variables is determined by them. Such a program is executed as a single process. Distributed systems composed of multiple processes such as those outlined above are more complex. Their behaviour and state can be described by a *distributed algorithm* – a definition of the steps to be taken by each of the processes of which the system is composed, *including the transmission of messages between them*. Messages are transmitted between processes to transfer information between them and to coordinate their activity.

Two significant factors affecting interacting processes in a distributed system:

• Communication performance is often a limiting characteristic.

• It is impossible to maintain a single global notion of time.

Performance of communication channels • The communication channels in our model are realized in a variety of ways in distributed systems – for example, by an implementation of streams or by simple message passing over a computer network. Communication over a computer network has the following performance characteristics relating to latency, bandwidth and jitter:

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

– The time taken for the first of a string of bits transmitted through a network to reach its destination. For example, the latency for the transmission of a message through a satellite link is the time for a radio signal to travel to the satellite and back.

– The delay in accessing the network, which increases significantly when the network is heavily loaded. For example, for Ethernet transmission the sending station waits for the network to be free of traffic.

– The time taken by the operating system communication services at both the sending and the receiving processes, which varies according to the current load on the operatingsystems.

• The *bandwidth* of a computer network is the total amount of information that can be transmitted over it in a given time. When a large number of communication channels are using the same network, they have to share the available bandwidth.

• *Jitter* is the variation in the time taken to deliver a series of messages. Jitter is relevant to multimedia data. For example, if consecutive samples of audio data are played with differing time intervals, the sound will be badly distorted.

Computer clocks and timing events • Each computer in a distributed system has its own internal clock, which can be used by local processes to obtain the value of the current time. Therefore two processes running on different computers can each associate timestamps with their events. However, even if the two processes read their clocks at the same time, their local clocks may supply different time values. This is because computer clocks drift from perfect time and, more importantly, their drift rates differ from one another. The term *clock drift rate* refers to the rate at which a computer clock deviates from a perfect reference clock. Even if the clocks on all the computers in a distributed system are set to the same time initially, their clocks will eventually vary quite significantly unless corrections are applied.

Two variants of the interaction model • In a distributed system it is hard to set limits on the time that can be taken for process execution, message delivery or clock drift. Two opposing extreme positions provide a pair of simple models – the first has a strong assumption of time and the second makes no assumptions about time:

Synchronous distributed systems: Hadzilacos and Toueg define a synchronous distributed system to be one in which the following bounds are defined:

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

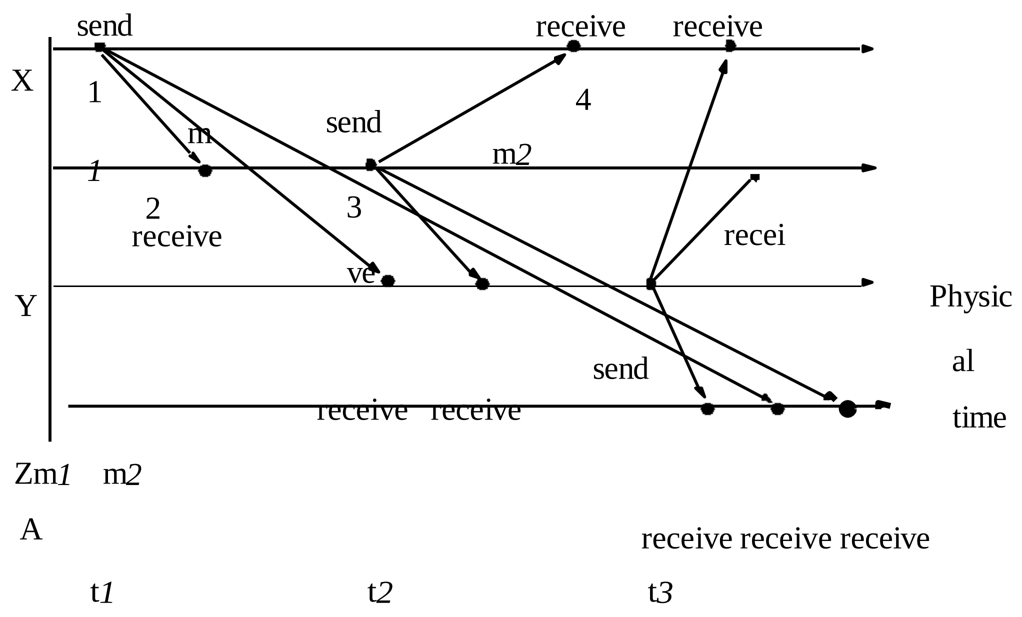
Asynchronous distributed systems: Many distributed systems, such as the Internet, are very useful without being able to qualify as synchronous systems. Therefore we need an alternative model. An asynchronous distributed system is one in which there are no bounds on:

- Process execution speeds – for example, one process step may take only a picosecond and another a century; all that can be said is that each step may take an arbitrarily long time.
- Message transmission delays – for example, one message from process A to process B may be delivered in negligible time and another may take several years. In other words, a message may be received after an arbitrarily long time.
- Clock drift rates – again, the drift rate of a clock is arbitrary.

ordering • In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after or concurrently with another event at another process. The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks. For example, consider the following set of exchanges between a group of email users, X, Y, Z and A, on a mailing list:

1. User X sends a message with the subject *Meeting*.
2. Users Y and Z reply by sending a message with the subject *Re: Meeting*.

In real time, X's message is sent first, and Y reads it and replies; Z then reads both X's message and Y's reply and sends another reply, which references both X's and Y's messages. But due to the independent delays in message delivery, the messages may be delivered as shown in the following figure and some users may view these two messages in the wrong order.


send
receive
receive
X
1
4
send
m
m2
1
2
3
receive
recei
ve
Y
Physic
al
time
send
receive
receive
Zm1
m2
A
receive receive receive
t1
t2
t3

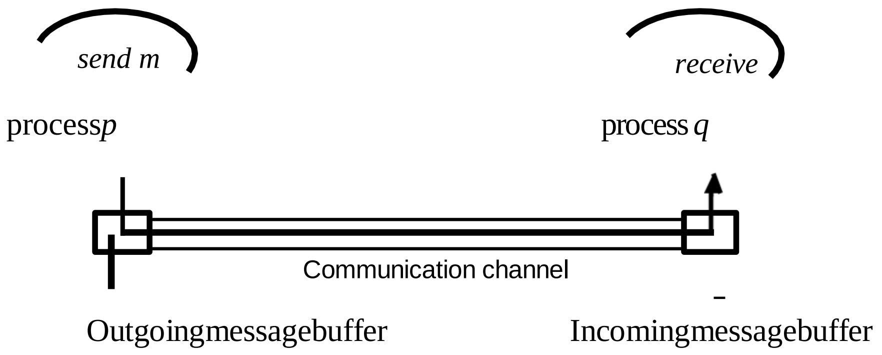
Failure model

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 behaviour. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. Hadzilacos and Toueg provide a taxonomy that distinguishes between the 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 is supposed to do.

Process omission failures: The chief omission failure of a process is to crash. When, say that a process has crashed we mean that it has halted and will not execute any further steps of its program ever. The design of services that can survive in the presence of faults can be simplified if it can be assumed that the services on which they depend crash cleanly – that is, their processes either function correctly or else stop. Other processes may be able to detect such a crash by the fact that the process repeatedly fails to respond to invocation messages. 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 forsomething to occur. In an asynchronous system a 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.

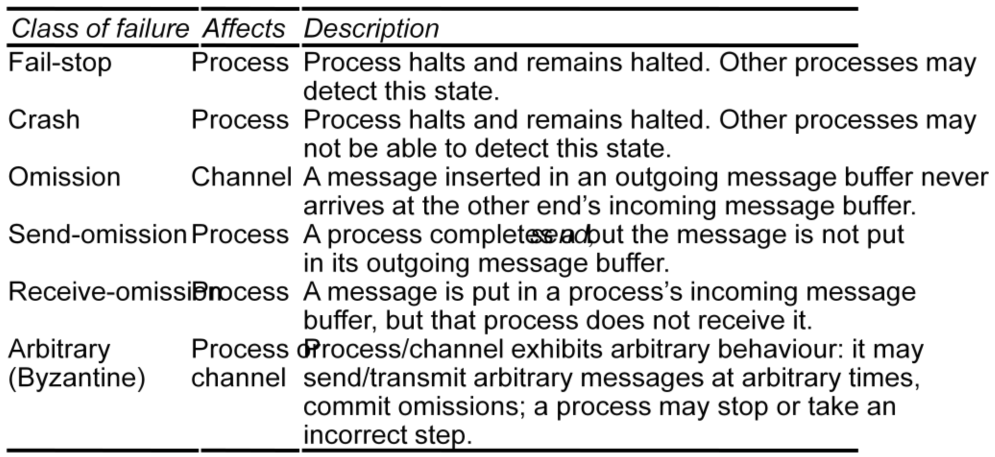
Communication omission failures: Consider the communication primitives *send* and *receive*. A process *p* performs a *send* by inserting the message *m* in its outgoing message buffer. The communication channel transports *m* to *q*'s incoming message buffer. Process *q* performs a *receive* by taking *m* from its incoming message buffer and delivering it. The outgoing and incoming message buffers are typically provided by the operating system.



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

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

takes unintended processing steps. Arbitrary failures in processes cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply. Communication channels can suffer from arbitrary failures; for example, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once. Arbitrary failures of communication channels are rare because the communication software is able to recognize them and reject the faulty
messages. For example, checksums are used to detect corrupted messages, and message sequence numbers can be used to detect nonexistent and duplicated messages.

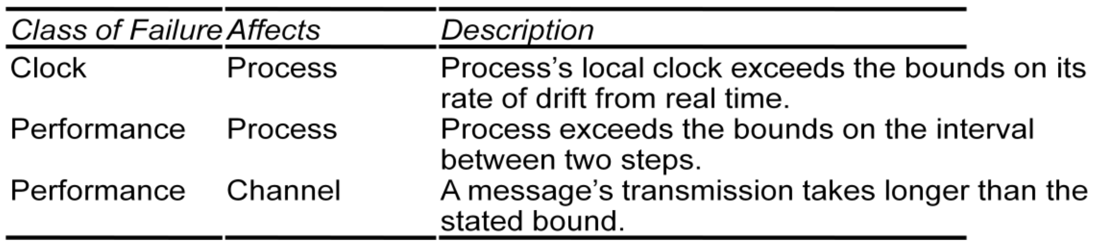


Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never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

Timing failures • Timing failures are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate. Timing failures are listed in the following figure. Any one of these failures may result in responses being unavailable to clients within a specified time interval.
In an asynchronous distributed system, an overloaded server may respond too slowly, but we cannot say that it has a timing failure since no guarantee has been offered. Real-time operating systems are designed with a view to providing timing guarantees, but they are more complex to design and may require redundant hardware.

Most general-purpose operating systems such as UNIX do not have to meet real-time constraints.

Masking failures • Each component in a distributed system is generally constructed from a collection of other components. It is possible to construct reliable services from components that exhibit failures. For example, multiple servers that hold replicas of data can continue to provide a service when one of them crashes. A knowledge of the failure characteristics of a component can enable a new service to be designed to mask the failure of the components on which it depends. A service *masks* a failure either by hiding it altogether or by converting it into a more acceptable type of failure. For an example of the latter, checksums are used to mask corrupted messages, effectively converting an arbitrary failure into an omission failure. The omission failures can be hidden by using a protocol that retransmits messages that do not arrive at their destination. Even process crashes may be masked, by replacing the process and restoring its memory from information stored on disk by its predecessor.



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

Reliability of one-to-one communication • Although a basic communication channel can exhibit the omission failures described above, it is possible to use it to build a communication service that masks some of those failures.

The term *reliable communication* is defined in terms of validity and integrity as follows:

Validity: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.

Integrity: The message received is identical to one sent, and no messages are delivered twice. The threats to integrity come from two independent sources:

• Any protocol that retransmits messages but does not reject a message that arrives twice. Protocols can attach sequence numbers to messages so as to detect those that are delivered twice.

• Malicious users that may inject spurious messages, replay old messages or tamper with messages. Security measures can be taken to maintain the integrity property in the face of such attacks.

Security model

The sharing of resources as a motivating factor for distributed systems, and in Section 2.3 we described their architecture in terms of processes, potentially encapsulating higher-level

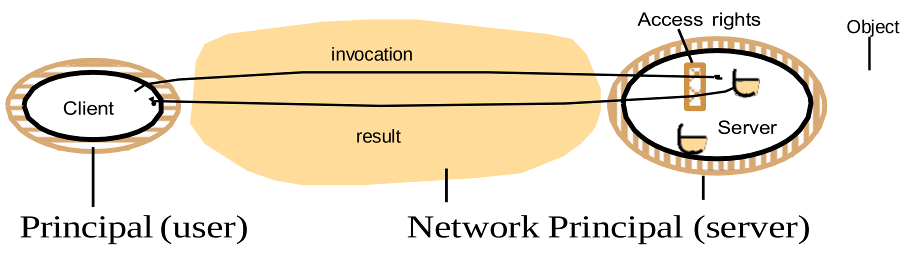
abstractions such as objects, components or services, and providing access to them through interactions with other processes. That architectural model provides the basis for our security model:the security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

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

Protecting objects :

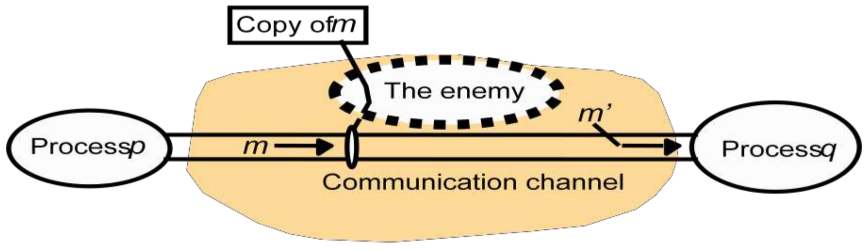
Server that manages a collection of objects on behalf of some users. The users can run client programs that send invocations to the server to perform operations on the objects. The server carries out the operation specified in each invocation and sends the result to the client.

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



Securing processes and their interactions • Processes interact by sending messages. The messages are exposed to attack because the network and the communication service that they use are open, to enable any pair of processes to interact. Servers and peer processes expose their interfaces, enabling invocations to be sent to them by any other process.

The enemy • To model security threats, we postulate an enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in the following figure. Such attacks can be made simply by using a computer connected to a network to run a program that reads network



messages addressed to other computers on the network, or a program that generates messages that make false requests to services, purporting to come from authorized users. The attack may come from a computer that is legitimately connected to the network or from one that is connected in an unauthorized manner. The threats from a potential enemy include *threats to processes* and *threats to communication channels*.

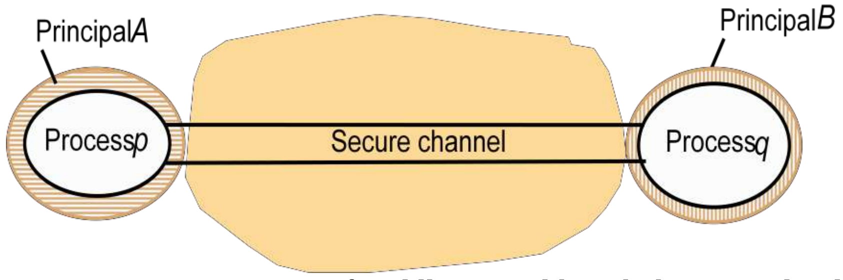
Defeating security threats

Cryptography and shared secrets: Suppose that a pair of processes (for example, a particular client and a particular server) share a secret; that is, they both know the secret but no other process in the distributed system knows it. Then if a message exchanged by that pair of processes includes information that proves the sender's knowledge of the
shared secret, the recipient knows for sure that the sender was the other process in the pair. Of course, care must be taken to ensure that the shared secret is not revealed to an enemy.
Cryptography is the science of keeping messages secure, and *encryption* is the process of scrambling a message in such a way as to hide its contents. Modern cryptography is based on encryption algorithms that use secret keys – large numbers that are difficult to guess – to transform data in a manner that can only be reversed with knowledge of the corresponding decryption key.

Authentication: The use of shared secrets and encryption provides the basis for the *authentication* of messages – proving the identities supplied by their senders. The basic authentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to guarantee its authenticity. The authentication portion of a request to a file server to read part of a file, for example, might include a representation of the requesting principal's identity, the identity of the file and the date and time of the request, all encrypted with a secret key shared between the file server and the requesting process. The server would decrypt this and check that it corresponds to the unencrypted details specified in the request.

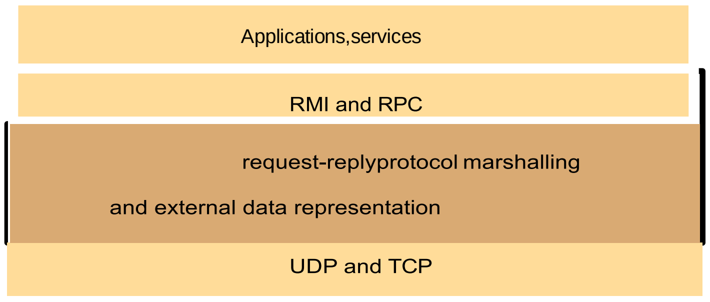
Secure channels: Encryption and authentication are used to build secure channels as a service layer on top of existing communication services. A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal, as shown in the following figure. A secure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

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing. Therefore if a client and server communicate via a secure channel, the server knows the identity of the principal behind the invocations and can check their access rights before performing an operation. This enables the server to protect its objects correctly and allows the client to be sure that it is receiving results from a *bona fide* server.
- A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it.
- Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered.



Communication aspects of middleware, although the principles discussed are more widely applicable.

This one is concerned with the design of the components shown in the darker layer in the following figure.


Applications,services
RMI and RPC
request-replyprotocol marshalling
and external data representation
UDP and TCP

UNIT II

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

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

CLOCKS, EVENTS AND PROCESS STATES

Each process executes on a single processor, and the processors do not share memory. Each process *pi* in has a state *si* that, in general, it transforms as it executes. The process's state includes the values of all the variables within it. Its state may also include the values of any objects in its local operating system environment that it affects, such as files. We assume that processes cannot communicate with one another in any way except by sending messages through the network.

So, for example, if the processes operate robot arms connected to their respective nodes in the system, then they are not allowed to communicate by shaking one another's robot hands! As each process *pi* executes it takes a series of actions, each of which is either amessage *send* or *receive* operation, or an operation that transforms *pi* 's state – one that changes one or more of the values in *si*. In practice, we may choose to use a high-leveldescription of the actions, according to the application. For example, if the processes in are engaged in an eCommerce application, then the actions may be ones such as 'client dispatched order message' or 'merchant server recorded transaction to log'. We define an event to be the occurrence of a single action that a process carries out as it executes – a communication action or a state-transforming action. The sequence of events within a single process *pi* can be placed in a single, total ordering, which we denote by the relation *i* between the events. That is, if and only if the event *e* occurs before *e* at *pi* . This ordering is well defined, whether or not the process is multithreaded, since we have assumed that the process executes on a single processor. Now we can define the *history* of process *pi* to be the series of events that take place within it, ordered as we have described by the relation

Clocks • We have seen how to order the events at a process, but not how to timestamp them – i.e., to assign to them a date and time of day. Computers each contain their own physical clocks. These clocks are electronic devices that count oscillations occurring in a crystal at a definite frequency, and typically divide this count and store the result in a counter register. Clock devices can be programmed to generate interrupts at regular intervals in order that, for example, timeslicing can be implemented; however, we shall not concern ourselves with this aspect of clock operation.

The operating system reads the node's hardware clock value, *Hit* , scales it and adds an offset to produce a software clock *Cit* = *Hit* + that approximately measures real, physical time *t* for process *pi*

In other words, when the real time in an absolute frame of reference is *t*, *Cit* is the reading on the software clock. For example, *Cit* could be the 64-bit value of the number of nanoseconds that have elapsed at time *t* since a convenient reference time. In general, the clock is not completely accurate, so *Cit* will differ from *t*. Nonetheless, if *Ci* behaves sufficiently well (we shall examine the notion of clock correctness shortly), we can use its value to timestamp any event at *pi* . Note that successive events will correspond to different timestamps only if the *clock resolution* – the period between updates of the clock value – is smaller than the time interval between successive events. The rate at which events occur depends on such factors as the length of the processor instruction cycle.

Clock skew and clock drift • Computer clocks, like any others, tend not to be in perfect agreement

Coordinated Universal Time • Computer clocks can be synchronized to external sources of highly accurate time. The most accurate physical clocks use atomic oscillators, whose drift rate is about one part in 1013. The output of these atomic clocks is used as the standard second has been defined as 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of Caesium- 133 (Cs133). Seconds and years and other time units that we use are rooted in astronomical time. They were originally defined in terms of the rotation of the Earth on its axis and its rotation about the Sun. However, the period of the Earth's rotation about its axis is gradually getting longer, primarily because of tidal friction; atmospheric effects and convection currents within the Earth's core also cause short-term increases and decreases in the period. So astronomical time and atomic time have a tendency to get out of step.

Coordinated Universal Time – abbreviated as UTC (from the French equivalent) – is an international standard for timekeeping. It is based on atomic time, but a so-called 'leap second' is inserted – or, more rarely, deleted – occasionally to keep it in step with astronomical time. UTC signals are synchronized and broadcast regularly from landbased radio stations and satellites covering many parts of the world. For example, in the USA, the radio station WWV broadcasts time signals on several shortwave frequencies.

Satellite sources include the *Global Positioning System* (GPS).Receivers are available commercially. Compared with 'perfect' UTC, the signals received from land-based stations have an accuracy on the order of 0.1–10 milliseconds,depending on the station used. Signals received from GPS satellites are accurate to about 1 microsecond. Computers with receivers attached can synchronize their clocks with these timing signals.

Synchronizing physical clocks

In order to know at what time of day events occur at the processes in our distributed system – for example, for accountancy purposes – it is necessary to synchronize the processes' clocks, *Ci* , with an authoritative, external source of time. This is *external synchronization*. And if the clocks *Ci* are synchronized with one another to a known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks, even though they are not necessarily synchronized to an external source of time. This is *internal synchronization*.We define these two modes of synchronization more closely as follows, over an interval of real time *I*:

External synchronization: For a synchronization bound *D* 0 , and for a source *S* of UTC time, *St* – *Cit* < *D*, for *i* = 1 2*N* and for all real times *t* in *I*. Another way of saying this is that the clocks *Ci* are

accurate to within the bound *D*.

Internal synchronization: For a synchronization bound *D* 0 , *Cit* – *Cjt D* for *i j* = 1 2*N* , and for all real times *t* in *I*. Another way of saying this is that he clocks *Ci agree* within the bound *D*. Clocks that are internally synchronized are not necessarily externally synchronized, since they may drift collectively from an external source of time even though they agree with one another. However, it follows from the definitions that if the system is externally synchronized with a bound *D* then the same system is internally synchronized with a bound of 2*D*. Various notions of *correctness* for clocks have been suggested. It is common to define a hardware clock *H* to be correct if its drift rate falls within a known bound (a value derived from one supplied by the manufacturer, such as 10–6 seconds/second).

This means that the error in measuring the interval between real times *t* and *t* (*t t*) is bounded: 1 – *t* – *t Ht* – *Ht* 1 + *t* – *t*

This condition forbids jumps in the value of hardware clocks (during normal operation). Sometimes we also require our software clocks to obey the condition but a weaker condition of *monotonicity* may suffice. Monotonicity is the condition that a clock *C* only ever advances: *t t Ct Ct* For example, the UNIX *make* facility is a tool that is used to compile only those source files that have been modified since they were last compiled. The modification dates of each corresponding pair of source and object files are compared to determine this condition. If a computer whose clock was running fast set its clock back after compiling a source file but before the file was changed, the source file might appear to have been modified prior to the compilation. Erroneously, *make* will not recompile the source file. We can achieve monotonicity despite the fact that a clock is found to be running fast. We need only change the rate at which updates are made to the time as given to applications. This can be achieved in software without changing the rate at which the underlying hardware clock ticks – recall that *Cit =Hit* + , where we are free to choose the values of and . A hybrid correctness condition that is sometimes applied is to require that a clock obeys the monotonicity condition, and that its drift rate is bounded between synchronization points, but to allow the clock value to jump ahead at synchronization points.

A clock that does not keep to whatever correctness conditions apply is defined to be *faulty*. A clock's *crash failure* is said to occur when the clock stops ticking altogether;any other clock failure is an *arbitrary failure*. A historical example of an arbitrary failure is that of a clock with the 'Y2K bug', which broke the monotonicity condition by registering the date after 31 December 1999 as 1 January 1900 instead of 2000; another example is a clock whose batteries are very low and whose drift rate suddenly becomes very large. Note that clocks do not have to be accurate to be correct, according to the definitions. Since the goal may be internal rather than external synchronization, the criteria for correctness are only concerned with the proper functioning of the clock's 'mechanism', not its absolute setting. We now describe algorithms for external synchronization and for internal synchronization.

Logical time and logical clocks

From the point of view of any single process, events are ordered uniquely by times shown on the local clock. However, as Lamport [1978] pointed out, since we cannot synchronize clocks perfectly across a distributed system, we cannot in general use physical time to find out the order of any arbitrary pair of events occurring within it. In general, we can use a scheme that is similar to physical causality but that applies in distributed systems to order some of the events that occur at different processes. This ordering is based on two simple and intuitively obvious points: • If two events occurred at the same process *pi i* = 1 2 *N* , then they occurred in the order in which *pi* observes them – this is the order *i* that we defined above.• Whenever a message is sent between processes, the eventof sending the message occurred before the event of receiving the message.

Lamport called the partial ordering obtained by generalizing these two relationships the *happened-before* relation. It is also sometimes known as the relation of *causal ordering* or *potential causal ordering*. We can define the happened-before relation, denoted by , as follows: HB1: If process*pi* : *e i e'*, then *e e* .HB2: For any message *m*, *send*(*m*) *receive*(*m*) – where *send*(*m*) is the event of sending the message, and *receive*(*m*)s the event of receiving it. HB3: If *e*, *e* and *e* are events such that *e e* and *e e* , then *e e* .

Totally ordered logical clocks • Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps. However, we can create a total order on the set ofevents– that is, one for which all pairs of distinct events are ordered – by taking into account the identifiers of the processes at which events occur. If *e* is an event occurring at *pi* with local timestamp *Ti* , and *e* is an event occurring at *pj* with local timestamp *Tj* , we define the global logical timestamps for these events to be *Ti i* and *Tj j* , respectively. And we define *Ti i Tj j* if and only if either *Ti Tj* , or *Ti* = *Tj* and *i j* . This ordering has no general physical significance (because process identiiers are arbitrary), but it is sometimes useful. Lamport used it, for example, to order the entry of processes to a critical section.

Vector clocks • Mattern [1989] and Fidge [1991] developed vector clocks to overcome the shortcoming of Lamport's clocks: the fact that from *Le Le* we cannot conclude that *e e*. A vector clock for a system of *N* processes is an array of *N* integers. Each process keeps its own vector clock, *Vi* , which it uses to timestamp local events. Like Lamport timestamps, processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:

VC1: Initially, *Vij* = 0 , for *i j* = 1 2 *N* .

VC2: Just before *pi* timestamps an event, it sets *Vii* :=*Vii* + 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 *Vij* := *maxVij tj* , for *j* = 1 2 *N* .

Taking the componentwise maximum of two vector timestamps in this way is known as a *merge* operation.For a vector clock *Vi* , *Vii* is the number of events that *pi* has timestamped, and *Vij j i* is the number of events that have occurred at *pj* that have potentially affected *pi* . (Process *pj* may have timestamped more events by this point, but no information has flowed to *pi* about them in messages as yet.) **Clocks, Events and Process States**

- A distributed system consists of a collection *P* of ***N*** processes ***pi***, *i = 1,2,… N*Each process *pi*

has a state ***si*** consisting of its variables (which it transforms as it executes) Processes communicate only by messages (via a network)

• **Actions** of processes*: Send, Receive,* change own state

• ***Event****:* the occurrence of a single action that a process carries out as it executes

– Events at a single process *pi*, can be placed in a total **ordering** denoted by the relation →*i* between the events. i.e.***e* →*i* *e'*** if and only if event *e* occurs before event *e'* at process *pi*

• A history of process pi: is a series of events ordered by →*i* – ***history*(pi)** = hi =<e*i*0, *ei*1, *ei*2, …> clocks

To timestamp events, use the computer's clock • At **real time, *t***, the OS reads the time on the computer's **hardware clock *Hi*(*t*)**

• It calculates the time on its **software clock *Ci*(*t*)=α*Hi*(*t*)** + β

– e.g. a 64 bit value giving nanoseconds since some base time

– **Clock resolution**: period between updates of the clock value

• In general, the clock is not completely accurate – but if *Ci* behaves well enough, it can be used to timestamp events at *pi*

Skew between computer clocks in a distributed system

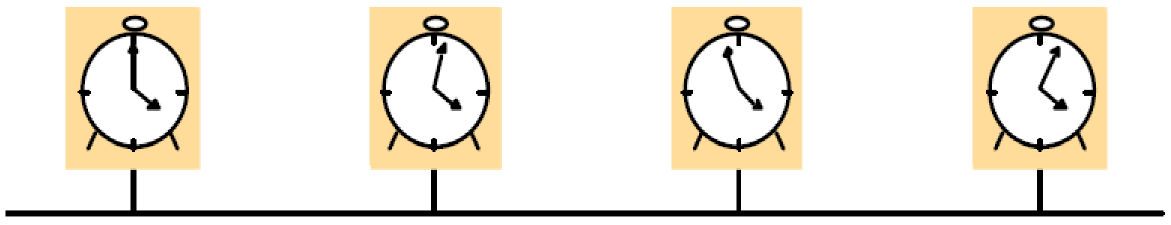


Figure 11.1

Computer clocks are not generally in perfect agreement

• ***Clock skew***: the difference between the times on two clocks (at any instant)

• Computer clocks use crystal-based clocks that are subject to physical variations

– ***Clock drift***: they count time at different rates and so diverge (frequencies of oscillation differ)

– ***Clock drift rate***: the difference per unit of time from some ideal reference clock

– Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10-6 secs/sec).

– High precision quartz clocks drift rate is about 10-7 or 10-8 secs/sec

Coordinated Universal Time (UTC)

• UTC is an international standard for time keeping

– It is based on atomic time, but occasionally adjusted to astronomical time

– International Atomic Time is based on very accurate physical clocks (drift rate 10-13)

• It is broadcast from radio stations on land and satellite (e.g.GPS)

• Computers with receivers can synchronize their clocks with these timing signals (by requesting

• time from GPS/UTC source)

– Signals from land-based stations are accurate to about 0.1-10 millisecond

– Signals from GPS are accurate to about 1 microsecond

Synchronizing physical clocks

Two models of synchronization

• External synchronization: a computer‘s clock *Ci* is synchronized with an external authoritative time source *S,* so that:

– $|S(t) - Ci(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, *I* of realtime

– The clocks *Ci* are **accurate** to within the bound *D*.

• Internal synchronization: the clocks of a pair of computers are synchronized with one another so that:

– $|Ci(t) - Cj(t)| < D$ for $i = 1, 2, \ldots N$ over an interval, *I* of realtime

– The clocks *Ci* and *Cj* **agree** within the bound *D*.

Internally synchronized clocks are not necessarily externally synchronized, as they may drift collectively

– if the set of processes *P* is synchronized externally within a bound *D*, it is also internally synchronized within bound 2*D (worst case polarity)*

Clock correctness

• ***Correct clock***: a hardware clock *H* is said to be correct if its drift rate is within a bound $\rho > 0$ (e.g. 10-6 secs/ sec)

This means that the error in measuring the interval between real times *t* and *t’* is bounded:

– $(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$ (where $t' > t$) Which forbids jumps in time readings of hardware clocks

– ***Clock monotonicity***: weaker condition of correctness – $t' > t \Rightarrow C(t') > C(t)$ e.g. required by Unix

make

– A hardware clock that runs fast can achieve monotonicity by adjusting the values of α and β

such that $Ci(t) = \alpha Hi(t) + \beta$

– ***Faulty clock***: a clock not keeping its correctness condition *crash failure* - a clock stops ticking

• *arbitrary* failure - any other failure e.g. jumps in time; Y2Kbug Synchronization in a synchronous system

A synchronous distributed system is one in which the following bounds are defined he 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 (min and max) each process has a local clock whose drift rate from real time has a known bound.

Internal synchronization in a synchronous system

- One process *p*1 sends its local time *t* to process *p*2 in a message *m*
- *p*2 could set its clock to *t* + *T*trans where *T*trans is the time to transmit *m*
- *T*trans is unknown but *min* ≤ *T*trans ≤ *max*
- uncertainty *u* = *max*-*min*. Set clock to *t* + (*max* - *min*)/2 then skew ≤*u*/2 Cristian‘s

method for an asynchronous system

- A time server *S* receives signals from a UTC source
- Process *p* requests time in *mr* and receives *t* in *mt* from *S*
- *p* sets its clock to *t* + *T*round/2
- Accuracy ± (*T*round/2 - *min*) :
- because the earliest time *S* puts *t* in message *mt* is *min* after *p* sent *mr*
- the latest time was *min* before *mt* arrived at *p*
- the time by *S*‘s clock when *mt* arrives is in the range [*t*+*min*, *t* + *T*round - *min*]
- the width of the range is *T*round + 2*min*

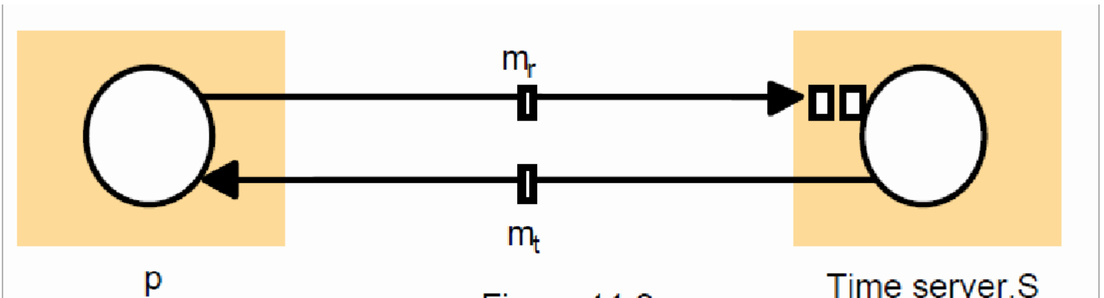


Figure 11.2

The Berkeley algorithm

- Problem with Cristian‘s algorithm
- a single time server might fail, so they suggest the use of a group of synchronized servers
- it does not deal with faulty servers
- Berkeley algorithm (also 1989)
- An algorithm for internal synchronization of a group of computers
- A *master* polls to collect clock values from the others (*slaves*)
- The master uses round trip times to estimate the slaves‘ clock values
- It takes an average (eliminating any above some average roundtrip time or with faulty clocks)
- It sends the required adjustment to the slaves (better thansending the time which depends on the round trip time)
- Measurements
- 15 computers, clock synchronization 20-25 millisecs drift rate < 2x10-5
- If master fails, can elect a new master to take over (not in bounded time)

Network Time Protocol (NTP)

- A time service for the Internet - synchronizes clients to UTC Reliability from redundant paths, scalable, authenticates time sources Architecture
- Primary servers are connected to UTC sources
- Secondary servers are synchronized to primary servers
- Synchronization subnet - lowest level servers in users‘ computers

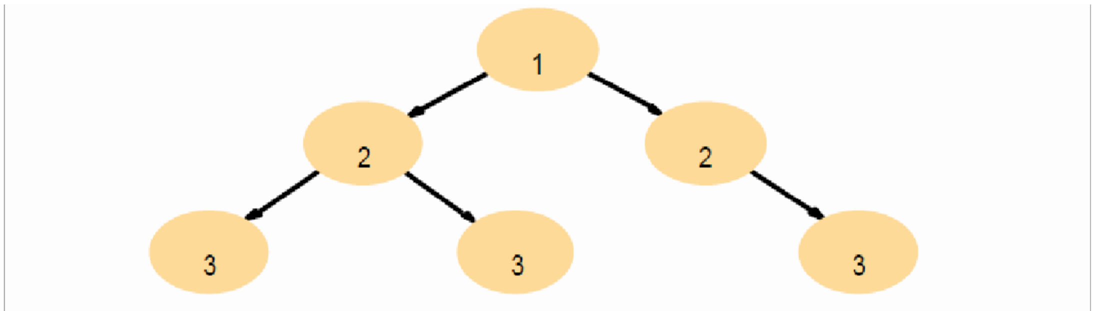


Figure 11.3 An example synchronization subnet in an NTP implementation

- strata: the hierarchy level

NTP - synchronization of servers

- The synchronization subnet can reconfigure if failures occur
- a primary that loses its UTC source can become a secondary
- a secondary that loses its primary can use another primary
- Modes of synchronization for NTP servers:
- Multicast
- A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not veryaccurate)
- Procedure call
- A server accepts requests from other computers (like Cristian‘s algorithm)
- Higher accuracy. Useful if no hardware multicast.

Messages exchanged between a pair of NTP peers

- All modes use UDP
- Each message bears timestamps of recent events:
- Local times of *Send* and *Receive* of previous message
- Local times of *Send* of current message
- Recipient notes the time of receipt T_i (we have Ti-3, Ti-2, Ti-1, Ti)
- Estimations of clock offset and message delay
- For each pair of messages between two servers, NTP estimates an offset ***oi*** (between the two clocks) and a delay ***di*** (total time for the two messages, which take t and t')
- $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$
- This gives us (by adding the equations) : $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$
- Also (by subtracting the equations)
    - $= o_i + (t' - t)/2$ where $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$
- Using the fact that $t, t'>0$ it can be shown that
- $o_i - d_i/2 \leq o \leq o_i + d_i/2$.
- Thus o_i is an estimate of the offset and d_i is a measure of the accuracy
- Data filtering
- NTP servers filter pairs $<o_i, d_i>$, estimating reliability from variation (dispersions), allowing them to select peers; and synchronization based on the lowest dispersion

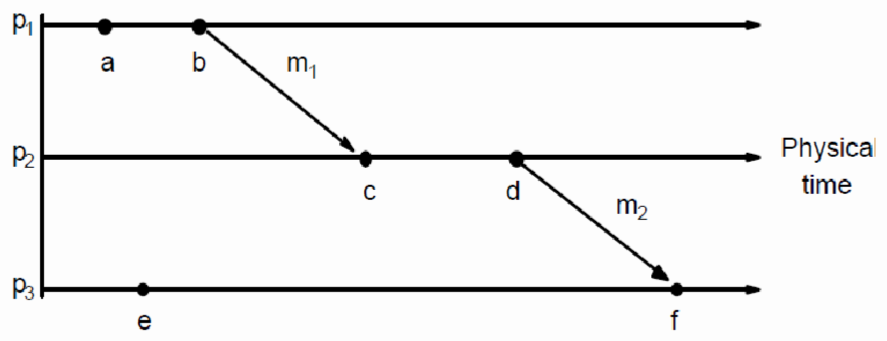
  or min d_i ok

- A relatively high filter dispersion represents relatively unreliabledata
- Accuracy of tens of milliseconds over Internet paths (1 ms onLANs)

Logical time and logical clocks

- Instead of synchronizing clocks, event ordering can be used
- If two events occurred at the same process *pi* (*i* = 1, 2, … *N*) then theyoccurred in the order observed by *pi*, that is order □ →***i***
- when a message, *m* is sent between two processes, *send(m)* happened before *receive(m)*
- Lamport[1978] generalized these two relationships into the **happened-before relation:** ***e* →*i* *e'***
- HB1: if *e* →*i* *e'* in process *p*i, then *e* → e'
- HB2: for any message *m*, send(*m*) → receive(*m*)
- HB3: if *e* → *e'* and *e'* → *e*", then *e* → *e''*

Figure 11.5 Events occurring at three processes



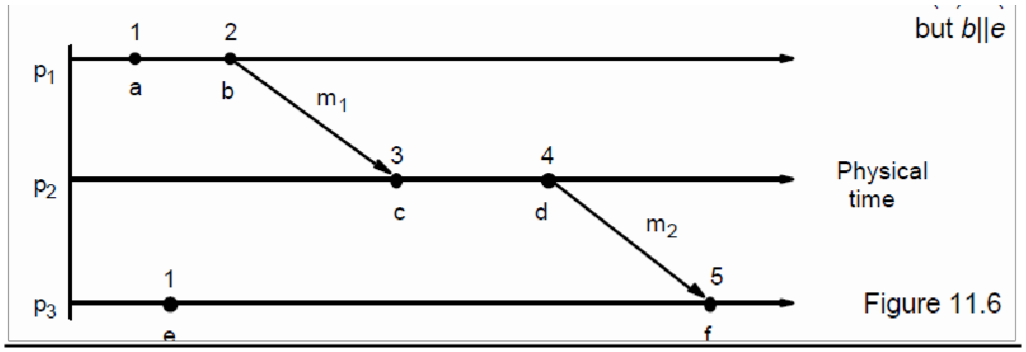
- HB1: a→b, c→d, e→f
- HB2: b→c, d→f
- HB3: a→b→c→d→f
- a||e: a and e are concurrent (neither a→e nor e→a)

Lamport‘s logical clocks

- Each process *pi* has a logical clock ***Li***
  - a monotonically increasing software counter
  - not related to a physical clock
- Apply Lamport timestamps to events with happened-beforerelation
  - LC1: *Li* is incremented by 1 before each event at process *pi*

- LC2:
- when process *pi* sends message *m*, it piggybacks *t* = *Li*
- when *pj* receives *(m,t)*, it sets *L*j := *max*(*L*j, *t*) and applies LC1before timestamping the event *receive* (*m)*

➢ *e* →*e*' implies *L*(*e*)<*L*(*e*'), but *L*(*e*)<*L*(*e*') does not imply *e*→*e*'



Totally ordered logical clocks

➢ Some pairs of distinct events, generated by different processes, may have numerically identical Lamport timestamps

➢ Different processes may have same Lamport time

➢ Totally ordered logical clocks

➢ If e is an event occurring at pi with local timestamp Ti, and if e' is an event occurring at pj with local timestamp Tj

➢ Define global logical timestamps for the events to be (*Ti, i*) and (*Tj, j*)

➢ Define (*Ti, i*) < (*Tj, j*) iff

➢ *Ti* < *Tj* or

➢ *Ti* = *Tj* and *i* < *j*

➢ No general physical significance since process identifiers are arbitrary

Vector clocks

➢ Shortcoming of Lamport clocks:

➢ *L(e)* < *L(e')* doesn't imply *e* → *e'*

➢ Vector clock: an array of N integers for a system of N processes

➢ Each process keeps its own vector clock *Vi* to timestamp local events

➢ Piggyback vector timestamps on messages

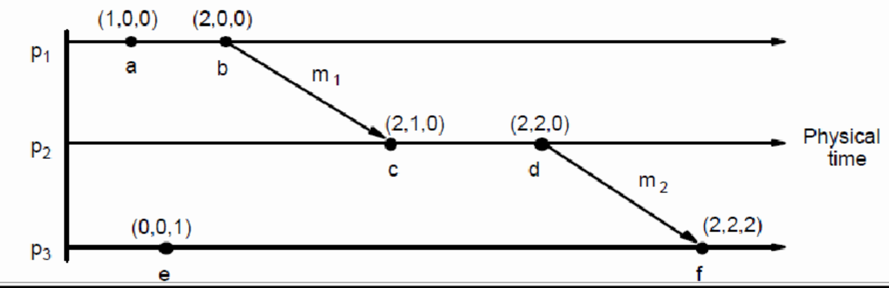
- Rules for updating vector clocks:
- *Vi*[*i*]] is the number of events that *pi* has timestamped
- *Viji*] (*j*≠ *i*) is the number of events at *pj* that *pi* has been affected

  by VC1: Initially, *Vi*[*j*] := 0 for *pi*, *j*=1.. *N* (*N* processes)
- VC2: before *pi* timestamps an event, *Vi*[*i*] := *Vi*[

  i]+1 VC3: *pi* piggybacks *t* = *Vi* on every message

  it sends
- VC4: when *pi* receives a timestamp *t*, it sets *Vi*[*j*] := max(*Vi*[*j*] , *t*[*j*]) for
- *j*=1..*N* (merge operation)

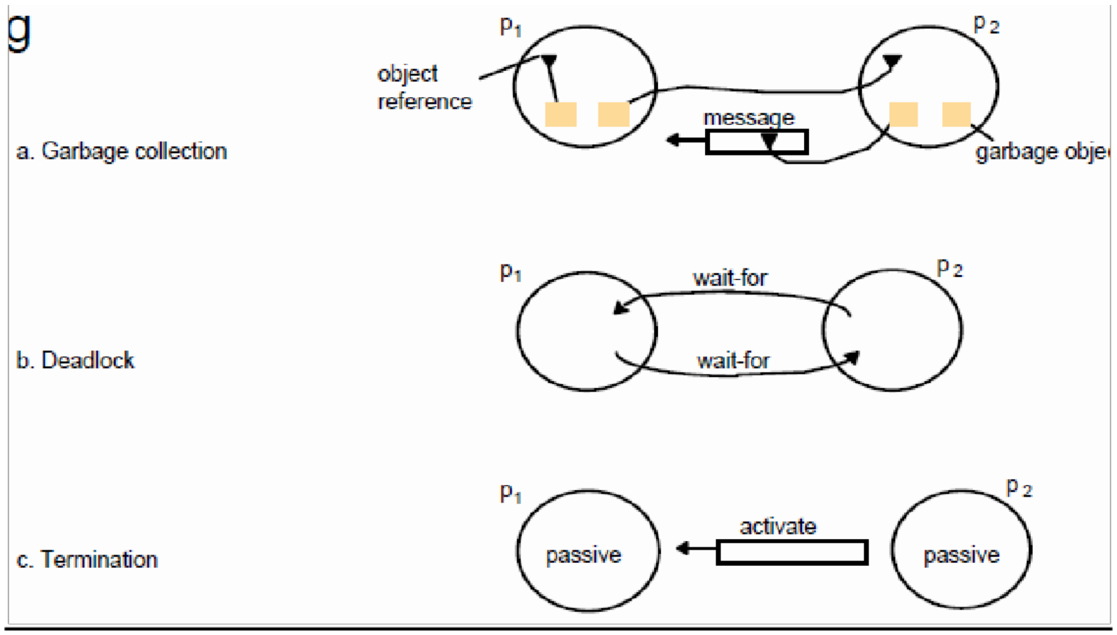
Figure 11.7 Vector timestamps for events shown in Figure 11.5



- Compare vector timestamps
- V=V‘ iff V[j] = V‘[j] for j=1..N
- V>=V‘ iff V[j] <= V‘[j] for j=1..N
- V<V‘ iff V<= V‘ ^ V!=V‘
- Figure 11.7 shows
- *a*→*f* since V(a) < V(f)
- c || e since neither V(c) <= V(e) nor V(e) <= V(c)

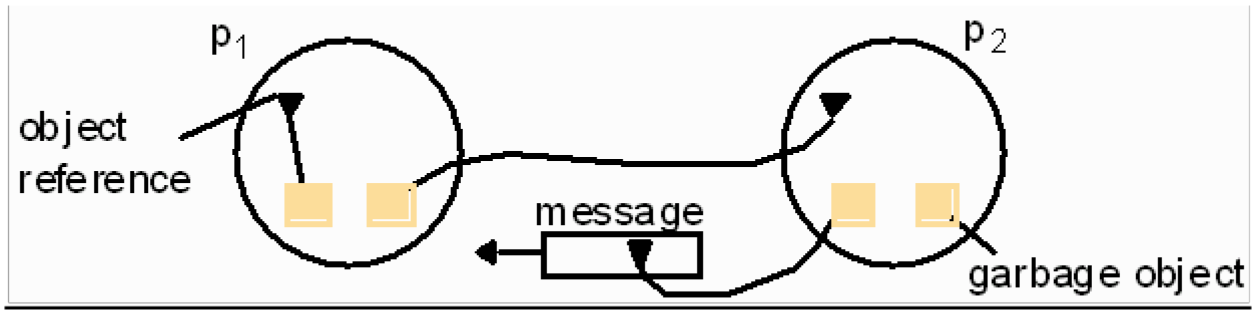
Global states

- How do we find out if a particular property is true in a distributed system? For examples, we will look at:
- Distributed Garbage Collection
- Deadlock Detection
- Termination Detection
- Debugging



Distributed Garbage Collection

- Objects are identified as ***garbage*** when there are no longer any references to them in the system
- Garbage collection reclaims memory used by thoseobjects
- In figure 11.8a, process p2 has two objects that do not have any references to other objects, but one object does have a reference to a message in transit. It is not garbage, but the other p2 object is
- Thus we must consider communication channels as well as object references to determine unreferenced objects



Deadlock Detection

- A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and there is a cycle in the graph of the ***waits-for*** relationship

- In figure 11.8b, both p1 and p2 wait for a message from the other, so both are blocked and the system cannot continue

Coordination And Agreement

Introduction

- Fundamental issue: for a set of processes, how to coordinate their actions or to agree on one or more values?
- even no fixed master-slave relationship between the components
- Further issue: how to consider and deal with failures when designing algorithms
- Topics covered
- mutual exclusion
- how to elect one of a collection of processes to perform a special role
- multicast communication
- agreement problem: consensus and byzantine agreement

Failure Assumptions and Failure Detectors

- Failure assumptions of this chapter
- Reliable communication channels
- Processes only fail by crashing unless stateotherwise
- Failure detector: object/code in a process that detects failures of other processes
- unreliable failure detector
- One of two values: unsuspected or suspected
- Evidence of possible failures
- Example: most practical systems
- Each process sends ―alive/I‘m here‖ message to everyone else
- If not receiving ―alive‖ message after timeout, it‘s suspected
- maybe function correctly, but network partitioned
- reliable failure detector
- One of two accurate values: unsuspected or failure – few practical systems

12.2 Distributed Mutual Exclusion

- Process coordination in a multitasking OS
- **Race condition**: several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place
- **critical section**: when one process is executing in a critical section, no other process is to be allowed to execute in its critical section
- **Mutual exclusion**: If a process is executing in its critical section, then no other processes can be executing in their critical sections
- Distributed mutual exclusion
- Provide critical region in a distributed environment
- message passing
- for example, locking files, locked daemon in UNIX (NFS is stateless, no file-locking at the NFS level)

Algorithms for mutual exclusion

- Problem: an asynchronous system of *N* processes
- processes don't fail
- message delivery is reliable; not share variables
- only one critical region
- application-level protocol: enter(), resourceAccesses(), exit()
- Requirements for mutual exclusion
- Essential
- [ME1] safety: only one process at a time
- [ME2] liveness: eventually enter or exit
- Additional
- [ME3] happened-before ordering: ordering of enter() is the same as HB ordering
- Performance evaluation
- overhead and bandwidth consumption: # of messages sent
- client delay incurred by a process at entry and exit
- throughput measured by synchronization delay: delay between one's exit and next's entry

A central server algorithm

- server keeps track of a token---permission to enter critical region

- a process requests the server for the token
- the server grants the token if it has the token
- a process can enter if it gets the token, otherwise waits when done, a
- process sends release and exits

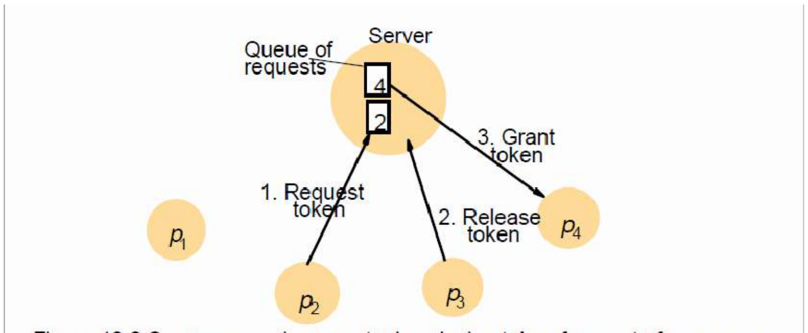


Figure 12.2 Server managing a mutual exclusion token for a set of processes

A central server algorithm: discussion

- Properties
- safety, why?
- liveness, why?
- HB ordering not guaranteed, why?
- Performance
- enter overhead: two messages (request and grant)
- enter delay: time between request and grant
- exit overhead: one message (release)
- exit delay: none
- synchronization delay: between release and grant
- centralized server is the bottleneck

A ring-based algorithm

- Arrange processes in a logical ring to rotate a token
- Wait for the token if it requires to enter the critical section
- The ring could be unrelated to the physical configuration
- *pi* sends messages to *p*(*i*+1) mod *N*
- when a process requires to enter the critical section, waits for the token
- when a process holds the token
- If it requires to enter the critical section, it can enter

- when a process releases a token (exit), it sends to its neighbor
- If it doesn‘t, just immediately forwards the token to its neighbor

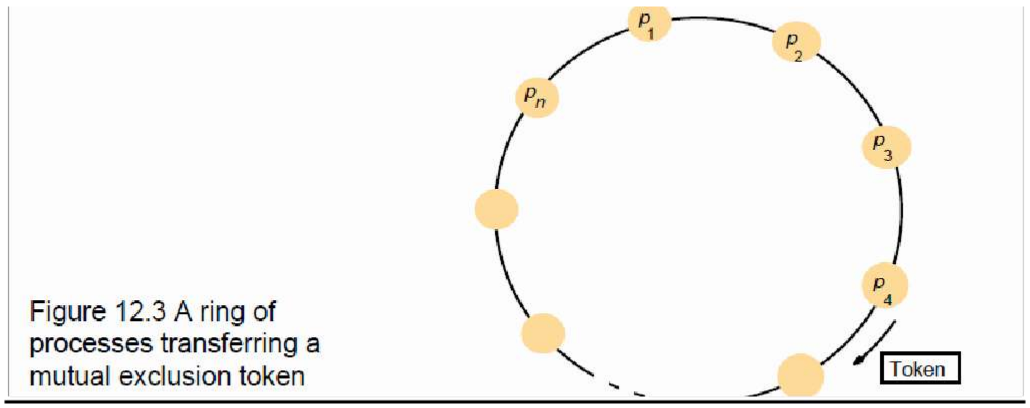


Figure 12.3 A ring of processes transferring a mutual exclusion token

An algorithm using multicast and logical clocks

- Multicast a request message for the token (Ricart and Agrawala [1981])
- enter only if all the other processes reply
- totally-ordered timestamps: <*T*, *pi* >
- Each process keeps a *state: RELEASED*, *HELD*, *WANTED*
- if all have *state* = *RELEASED*, all reply, a process can hold the token and enter
- if a process has *state* = *HELD*, doesn't reply until it exits
- if more than one process has *state* = *WANTED*, process with the lowest timestamp will get all

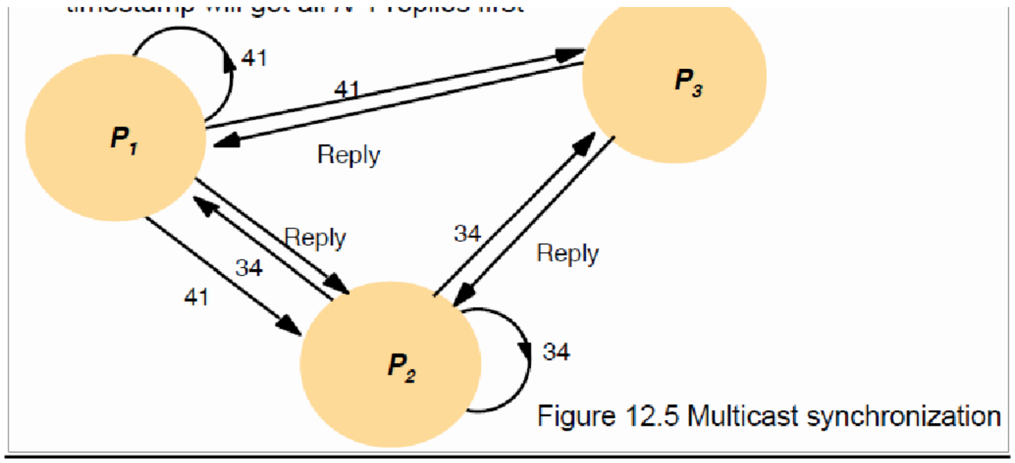


Figure 12.5 Multicast synchronization

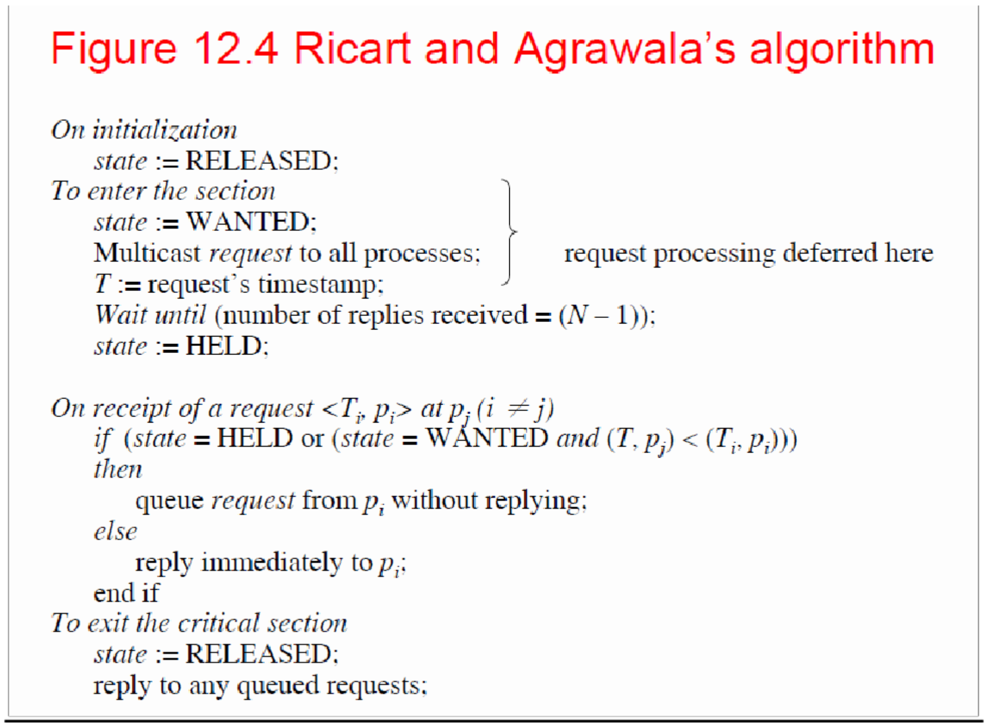


Figure 12.4 Ricart and Agrawala's algorithm

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

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

An algorithm using multicast: discussion

- •Properties
- safety, why?
- liveness, why?
- HB ordering, why?
- Performance
- bandwidth consumption: no token keeps circulating
- entry overhead: 2(N-1), why? [with multicast support: 1 + (N -1) = N]
- entry delay: delay between request and getting all replies
- exit overhead: 0 to N-1 messages
- exit delay: none
- synchronization delay: delay for 1 message (one last reply from the previous holder)

Maekawa's voting algorithm

- •Observation: not all peers to grant it access
- Only obtain permission from subsets, overlapped by any two processes
- •Maekawa's approach
- subsets Vi,Vj for process Pi, Pj
- Pi ∈ Vi, Pj ∈ Vj
- Vi ∩ Vj ≠ ∅ , there is at least one common member
- subset |Vi|=K, to be fair, each process should have the same size
- Pi cannot enter the critical section until it has received all K reply messages
- Choose a subset

- Simple way (2√N): place processes in a √N by √N matrix and let Vi be the union of the row and column containing Pi
- If P1, P2 and P3 concurrently request entry to the critical section, then its possiblethat each process has received one (itself) out of two replies, and none can proceed
- adapted and solved by [Saunders 1987]

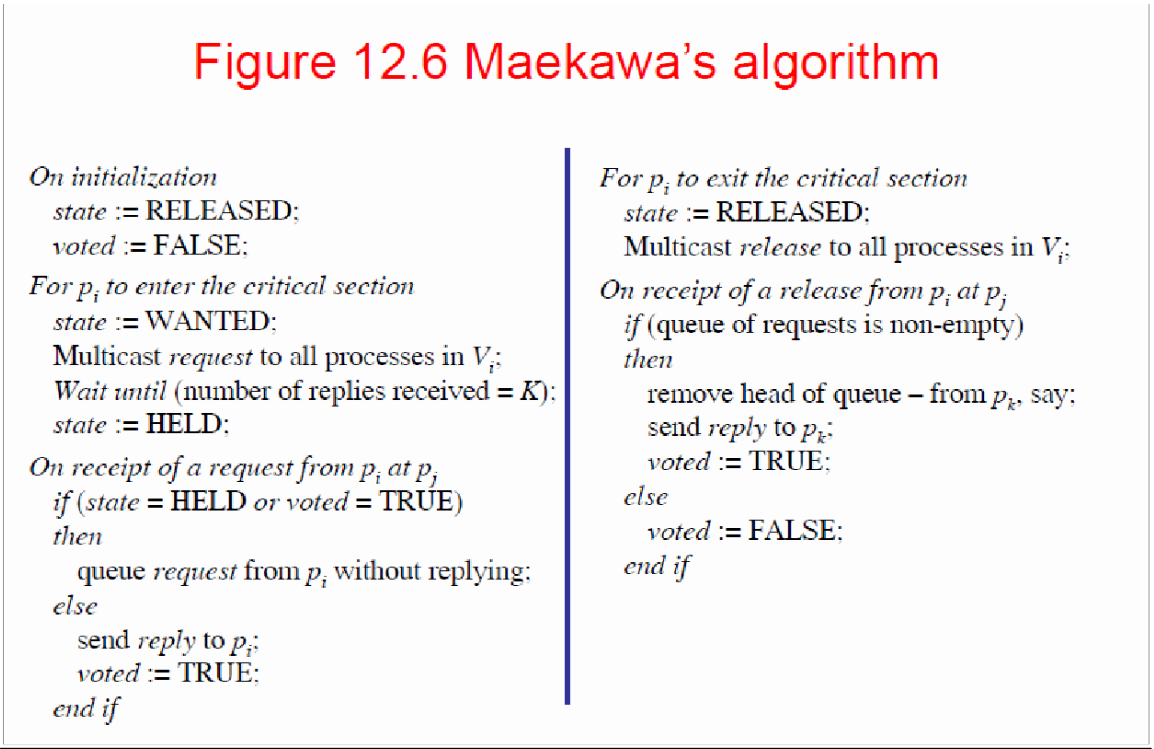


Figure 12.6 Maekawa's algorithm

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

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

Elections

Election: choosing a unique process for a particular role

- All the processes agree on the ***unique*** choice
- For example, server in dist. Mutex assumptions
- Each process can call only one election at a time multiple concurrent elections can be called by different processes
- Participant: engages in an election each process *pi* has variable *electedi* = ? (don't know) initially process with the *largest* identifier wins.
- The (unique) identifier could be any useful value Properties
- [E1] *electedi* of a ―participant‖ process must be P (elected process=largestid) or ⊥ (undefined)
- [E2] liveness: all processes participate and eventually set *electedi* != ⊥(or crash) Performance
- overhead (bandwidth consumption): # of messages
- turnaround time: # of messages to complete an election

A ring-based election algorithm

- Arrange processes in a logical ring
    - *pi* sends messages to *p*(*i*+1) mod *N*
    - It could be unrelated to the physical configuration
    - Elect the coordinator with the largest id

- Assume no failures

➢ Initially, every process is a non-participant. Any process can call an election

- Marks itself as participant
- Places its id in an *election* message
- Sends the message to its neighbor
- Receiving an election message

➢ if *id* > *myid*, forward the msg, mark participant

➢ if *id* < *myid*

- non-participant: replace *id* with *myid*: forward the msg, mark participant
- participant: stop forwarding (why? Later, multiple elections)

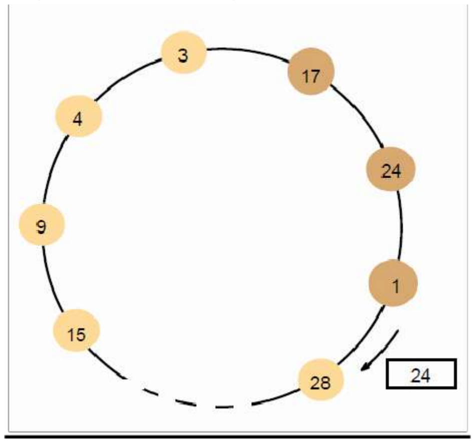
➢ if *id* = *myid*, coordinator found, mark non-participant, *electedi* := *id*, send *elected*

- message with *myid*
- Receiving an elected message

➢ *id* != *myid*, mark non-participant, *electedi* := *id* forward the msg

➢ if *id* = *myid*, stop forwarding

Figure 12.7 A ring-based election in progress



➢ Receiving an election message:

➢ if *id* > *myid*, forward the msg, mark participant

➢ if *id* < *myid*

➢ non-participant: replace *id* with *myid*: forward the msg, mark participant

➢ participant: stop forwarding (why? Later, multiple elections)

➢ if *id* = *myid*, coordinator found, mark non-participant, *electedi* := *id*, send *elected*

message with *myid*

➢ Receiving an elected message: – *id* != *myid*, mark non-participant,

➢ *electedi* := *id* forward the msg

➢ if *id* = *myid*, stop forwarding

A ring-based election algorithm: discussion

➢ •Properties

- safety: only the process with the largest id can send an *elected* message
- liveness: every process in the ring eventually participates in the election; extra elections are stopped
- Performance
- one election, best case, when?
- *N election* messages
- *N elected* messages
- turnaround: 2*N* messages
- one election, worst case, when?
- 2*N* - 1 election messages
- *N elected* messages
- turnaround: 3*N* - 1 messages
- can't tolerate failures, not very practical

The bully election algorithm

• Assumption

– Each process knows which processes have higher identifiers, and that it can communicate with all such processes

• Compare with ring-based election

– Processes can crash and be detected by timeouts

• synchronous

• timeout *T* = 2*Ttransmitting* (max transmission delay) + *Tprocessing* (max processing delay)

• Three types of messages

– Election: announce an election

– Answer: in response to Election

– Coordinator: announce the identity of the elected process The bully election algorithm: how to

• Start an election when detect the coordinator has failed or begin to replace the coordinator, which has lower identifier

– Send an election message to all processes with higher id's and waits for answers (exceptthe failed coordinator/process)

• If no answers in time *T*

– Considers it is thecoordinator

– sends coordinator message (with its id) to all processes with lower id's

• else

– waits for a coordinator message and starts an election if T‘ timeout

– To be a coordinator, it has to start an election

• A higher id process can replace the current coordinator (hence ―bully‖)

– The highest one directly sends a coordinator message to all process with lower identifiers

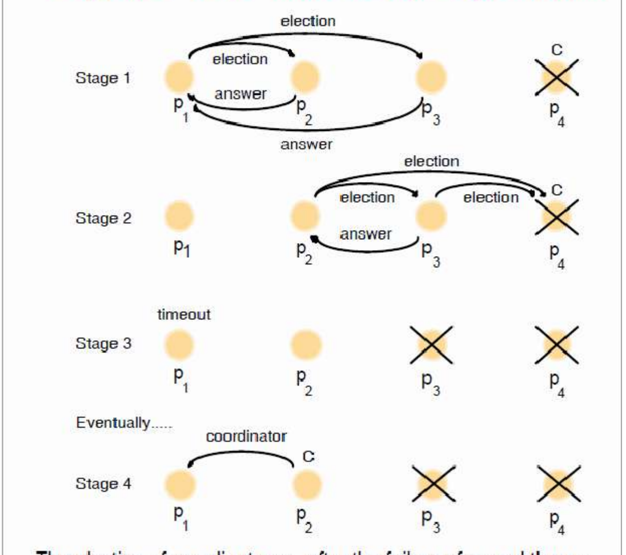
• Receiving an election message

– sends an answer message back

– starts an election if it hasn't started one—send election messages to all higher-id processes (including the ―failed‖ coordinator—the coordinator might be up by now)

• Receiving a coordinator message – set *electedi* to the new coordinator

Figure 12.8 The bully algorithm



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

The bully election algorithm: discussion

- Properties
- safety:
- a lower-id process always yields to a higher-id process
- However, it‘s guaranteed
- if processes that have crashed are replaced by processes with the same identifier since message delivery order might not be guaranteed and
- failure detection might be unreliable
- liveness: all processes participate and know the coordinator at the end
- Performance

- best case: when?
- overhead: *N*-2 *coordinator* messages
- turnaround delay: no *election/answer* messages

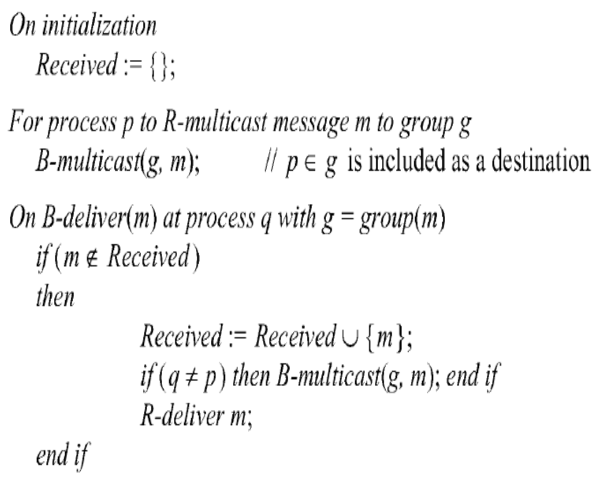
Multicast Communication

- Group (multicast) communication: for each of a groupof processes to receive copies of the messages sent to the group, often with deliveryguarantees
- The set of messages that every process of the group shouldreceive
- On the delivery ordering across the group members
- Challenges
- Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
- Delivery guarantees ensure that operations are completed
- Types of group
- Static or dynamic: whether joining or leaving is considered Closed or open
- A group is said to be closed if only members of the group can multicast to it. Reliable

Multicast

- Simple basic multicasting (B-multicast) is sending a message to every process that is a member of a defined group
- B-multicast (g, m) for each process p $\in$ group g, send (p, message m)
- On receive (m) at p: B-deliver (m) at p
- Reliable multicasting (R-multicast) requires these properties
- Integrity: a correct process sends a message to only a member of the group
- Validity: if a correct process sends a message, it will eventually bedelivered
- Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

Figure 12.10 Reliable multicast algorithm



```
On initialization
    Received := {};

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

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

Types of message ordering Three types of message ordering

– ***FIFO (First-in, first-out) ordering***: if a correct process delivers a message beforeanother, every correct process will deliver the first message before the other

– ***Casual ordering***: any correct process that delivers the second message will deliver the previous message first

– ***Total ordering***: if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first

• Note that

– FIFO ordering and casual ordering are only partial orders

– Not all messages are sent by the same sending process

– Some multicasts are concurrent, not able to be ordered by happened before

– Total order demands consistency, but not a particular order Figure 12.12 Total, FIFO and causal ordering of multicast messages

Notice

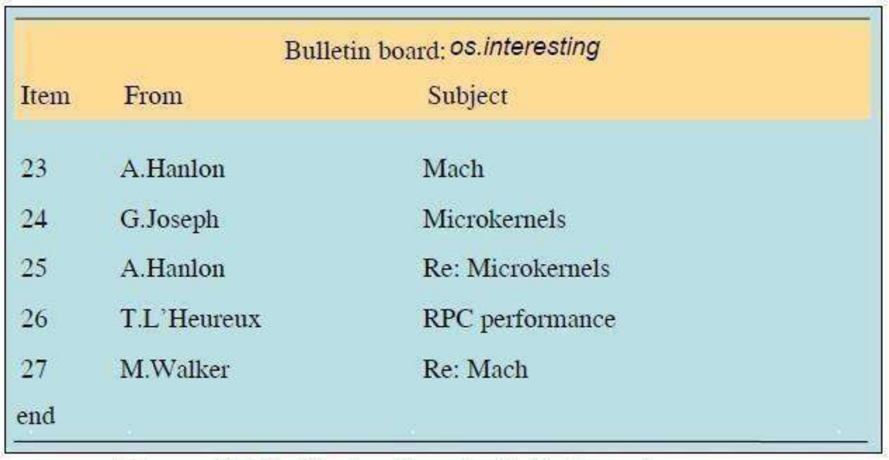
- the consistent ordering of totally ordered messages *T1* and *T2*,
- the FIFO-related messages *F1* and *F2* and
- the causally related messages *C1* and *C3* and
- the otherwise arbitrary delivery ordering ofmessages

Note that *T1* and *T2* are delivered in opposite order to the physical time of message creation

Bulletin board example (FIFO ordering)

• A bulletin board such as Web Board at NJIT illustrates the desirability of consistency and FIFO ordering. A user can best refer to preceding messages if they are delivered in order. Message 25 in Figure 12.13 refers to message 24, and message 27 refers to message 23.

• Note the further advantage that Web Board allows by permitting messages to begin threads by replying to a particular message. Thus messages do not have to be displayed in the same order they are delivered



Bulletin board: *os.interesting*

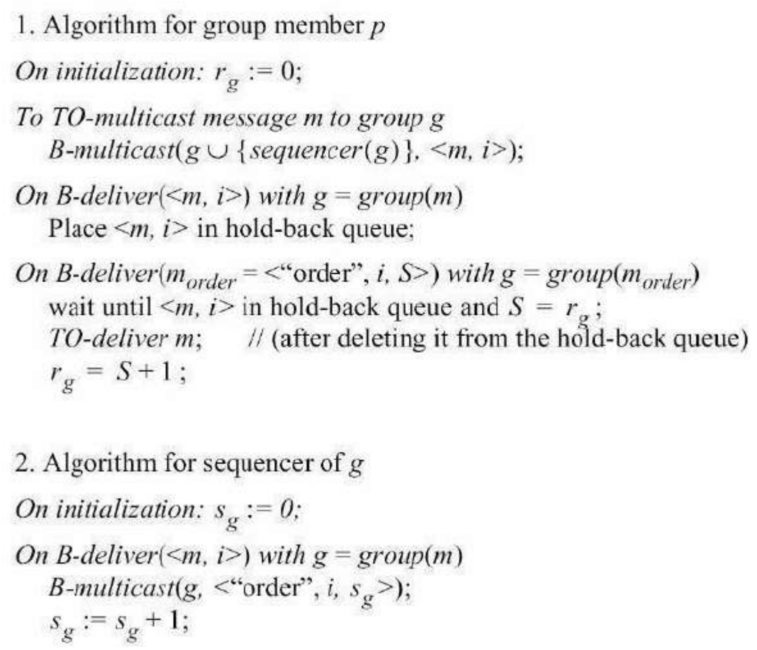
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 12.13 Display from bulletin board program

Implementing total ordering

- The normal approach to total ordering is to assign totally ordered identifiers tomulticast messages, using the identifiers to make ordering decisions.
- One possible implementation is to use a sequencer process to assign identifiers. See Figure 12.14. A drawback of this is that the sequencer can become a bottleneck.
- An alternative is to have the processes collectively agree on identifiers. A simple algorithmis shown in Figure 12.15.

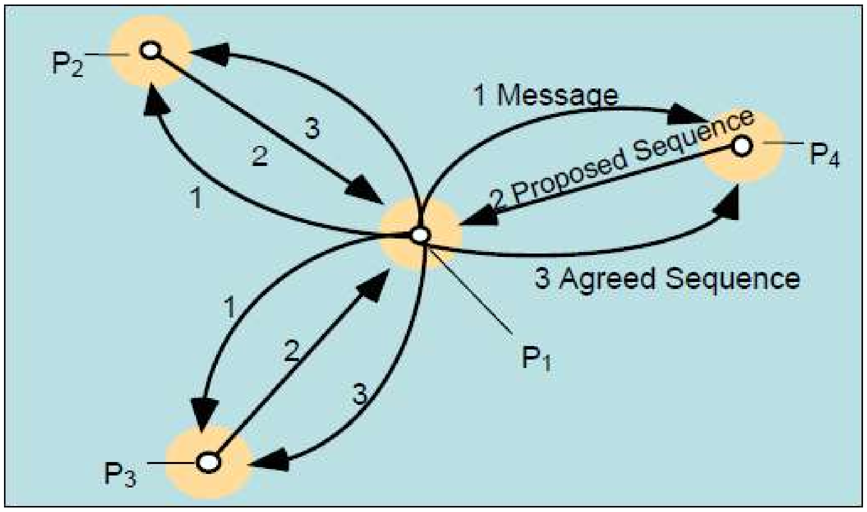
Figure 12.14 Total ordering using a sequencer



```
1. Algorithm for group member p
On initialization: $r_g := 0$;
To TO-multicast message m to group g
    B-multicast($g \cup \{sequencer(g)\}$, <m, i>);
On B-deliver(<m, i>) with g = group(m)
    Place <m, i> in hold-back queue;
On B-deliver($m_{order}$ = <"order", i, S>) with g = group($m_{order}$)
    wait until <m, i> in hold-back queue and $S = r_g$;
    TO-deliver m;        // (after deleting it from the hold-back queue)
    $r_g = S + 1$;

2. Algorithm for sequencer of g
On initialization: $s_g := 0$;
On B-deliver(<m, i>) with g = group(m)
    B-multicast(g, <"order", i, $s_g$>);
    $s_g := s_g + 1$;
```

Figure 12.15 The ISIS algorithm for total ordering



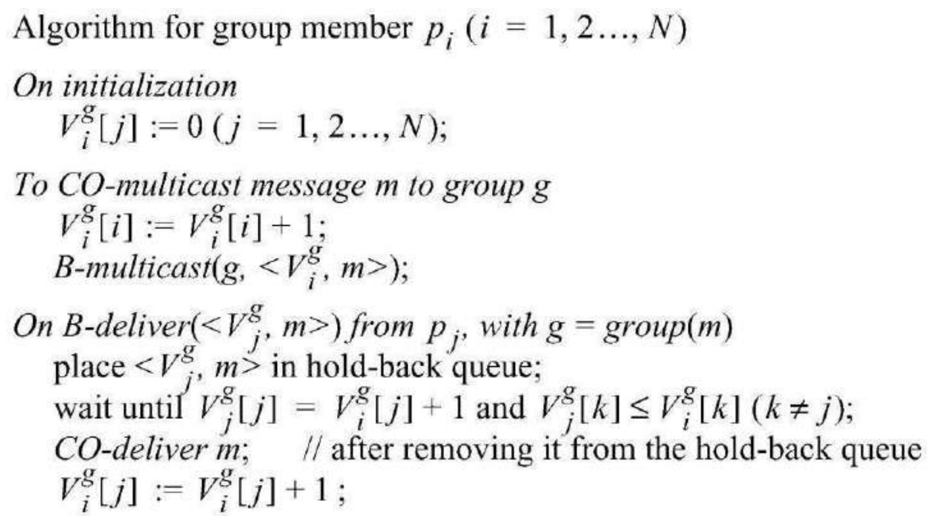
Each process q in group g keeps

• Aq g: the largest agreed sequence number it has observed so far for the group g

• Pq g: its own largest proposed sequence number Algorithm for process p to multicast a message m to group g

1. B-multicasts <m, i> to g, where i is a unique identifier for m

2. Each process q replies to the sender p with a proposal for the message's agreed sequence number of Pq g :=Max(Aq g, Pq g)+1
3. Collects all the proposed sequence numbers and selects the largest one a as the next agreed sequence number. It then B-multicasts <i, a> to g.
4. Each process q in g sets Aq g := Max(Aq g, a) and attaches a to the message identified by i Implementing casual ordering
• Causal ordering using vector timestamps (Figure 12.16)
– Only orders multicasts, and ignores one-to-one messages between processes
– Each process updates its vector timestamp before delivering a message to maintain the count of precedent messages



```
Algorithm for group member $p_i$ ($i = 1, 2 \ldots, N$)

On initialization
    $V_i^g[j] := 0$ ($j = 1, 2 \ldots, N$);

To CO-multicast message m to group g
    $V_i^g[i] := V_i^g[i] + 1$;
    B-multicast(g, $<V_i^g, m>$);

On B-deliver($<V_j^g, m>$) from $p_j$, with $g = group(m)$
    place $<V_j^g, m>$ in hold-back queue;
    wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k]$ ($k \ne j$);
    CO-deliver m;      // after removing it from the hold-back queue
    $V_i^g[j] := V_i^g[j] + 1$;
```

Consensus and related problems

• Problems of agreement

– For processes to agree on a value (consensus) after one or more of the processes has proposed what that value should be

– Covered topics: byzantine generals, interactive consistency, totally ordered multicast

• The byzantine generals problem: a decision whether multiple armies should attack or retreat, assuming that united action will be more successful than some attacking and some retreating

• Another example might be space ship controllers deciding whether to proceed or abort. Failure handling during consensus is a key concern

• Assumptions

– communication (by message passing) is reliable

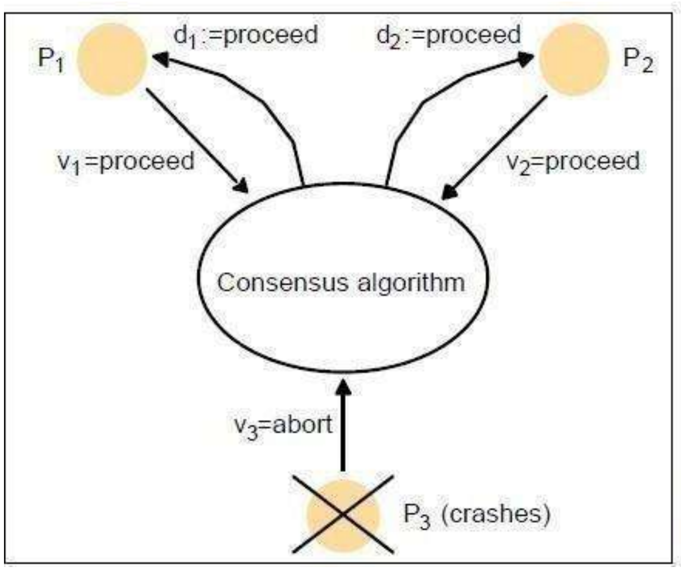
– processes may fail

• Sometimes up to f of the N processes are faulty Consensus Process

1. Each process pi begins in an undecided state and proposes a single value vi, drawn from a set D (i=1…N)

2. Processes communicate with each other, exchanging values

3. Each process then sets the value of a decision variable di and enters the decided state



Two processes propose "proceed." One proposes "abort," but then crashes. The two remaining processes decide proceed.

Figure 12.17 Consensus for three processes

Requirements for Consensus

• Three requirements of a consensus algorithm

– ***Termination***: Eventually every correct process sets its decision variable

– ***Agreement***: The decision value of all correct processes is the same: if pi and pj are correctand have entered the *decided* state, then di=dj

(i,j=1,2, …, N)

– ***Integrity***: If the correct processes all proposed the same value, then any correct process inthe

decided state has chosen that value The byzantine generals problem

• Problem description

– Three or more generals must agree to *attack* or to *retreat*

– One general, the *commander*, issues the order

– Other generals, the *lieutenants*, must decide to attack or retreat

– One or more generals may be treacherous

• A *treacherous general* tells one general to attack and another to retreat

• Difference from consensus is that a single process supplies the value to agree on

• Requirements

– *Termination*: eventually each correct process sets its decision variable

– *Agreement*: the decision variable of all correct processes is the same

– *Integrity*: if the commander is correct, then all correct processes agree on the value that the commander has proposed (but the commander need not be correct)

The interactive consistency problem

• Interactive consistency: all correct processes agree on a vector of values, one for each process.

This is called the decision vector

– Another variant of consensus

• Requirements

– *Termination*: eventually each correct process sets its decision variable

– *Agreement*: the decision vector of all correct processes is the same

– *Integrity*: if any process is correct, then all correct processes decide the correct value for that process

Relating consensus to other problems

• Consensus (C), Byzantine Generals (BG), and Interactive Consensus (IC) are all problems concerned with making decisions in the context of arbitrary or crash failures

• We can sometimes generate solutions for one problem in terms of another. For example

– We can derive IC from BG by running BG N times, once for each process with thatprocess acting as commander

– We can derive C from IC by running IC to produce a vector of values at each process, then

– applying a function to the vector‘s values to derive a single value.

– We can derive BG from C by

• Commander sends proposed value to itself and each remaining process

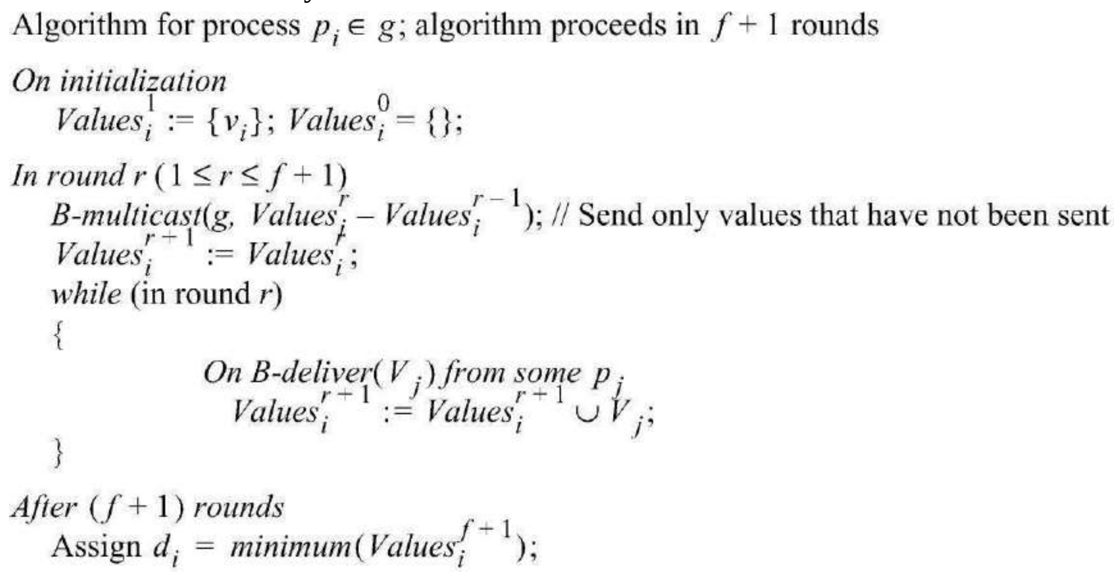
• All processes run C with received values

• They derive BG from the vector of C values Consensus in a Synchronous System

• Up to f processes may have crash failures, all failures occurring during f+1 rounds. During each round, each of the correct processes multicasts the values amongthemselves

• The algorithm guarantees all surviving correct processes are in a position to agree

• Note: any process with f failures will require at least f+1 rounds to agree Limits for solutions to Byzantine Generals



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

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

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

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

• Some cases of the Byzantine Generals problems have no solutions

– Lamport *et al* found that if there are only 3 processes, there is no solution

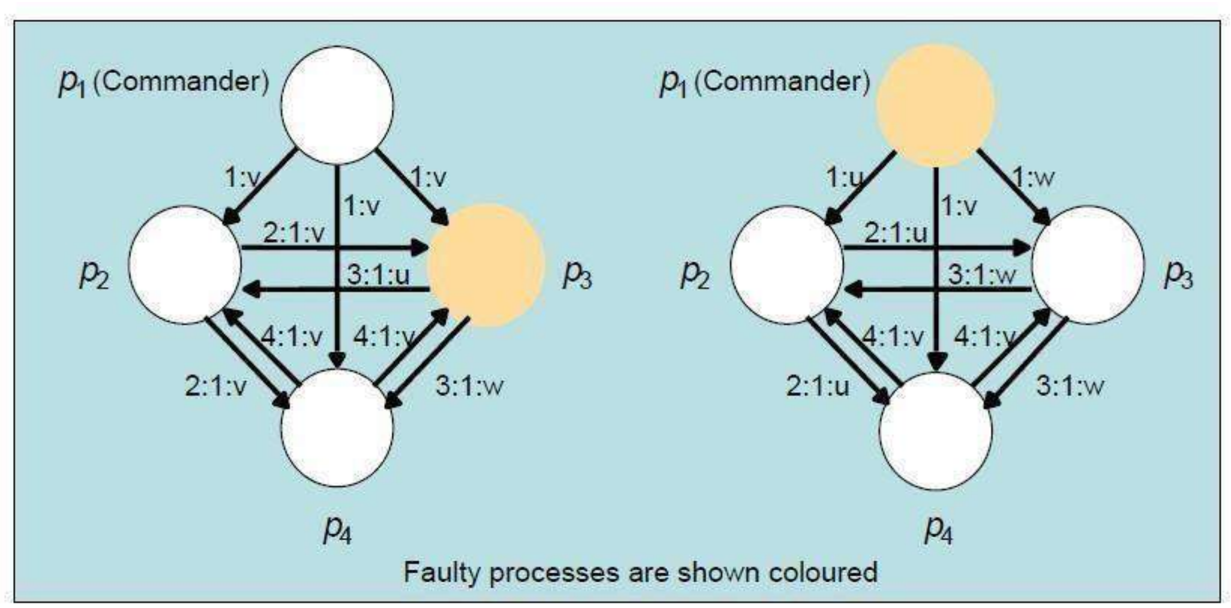
– Pease *et al* found that if the total number of processes is less than three times the number of failures plus one, there is no solution

• Thus there is a solution with 4 processes and 1 failure, if there are two rounds

– In the first, the commander sends the values

– while in the second, each lieutenant sends the values it received

Figure 12.20 Four Byzantine generals



Asynchronous Systems

• All solutions to consistency and Byzantine generals problems are limited to synchronous systems

• Fischer *et al* found that there are no solutions in an asynchronous system with even one failure

• This impossibility is circumvented by *masking faults* or using *failure detection*

• There is also a partial solution, assuming an *adversary* process, based on *introducing random values* in the process to prevent an effective thwarting strategy. This does not always reach consensus

UNIT III

Inter Process Communication: Introduction, characteristics of inter process communication, External Data Representation and Marshalling, Client-Server Communication, Group Communication, **Distributed Objects and Remote Invocation**: Introduction, Communication between Distributed Objects, Remote Procedure Call, Events and Notifications,

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 be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving processes may be either synchronous or asynchronous. 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.

In the *asynchronous* form of communication, the use of the *send* operation is *nonblocking* 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 *receive* operation can have blocking and non-blocking variants. In the non-blocking variant, 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.

In a system environment such as Java, which supports multiple threads in a single process, the blocking *receive* has no disadvantages, for it can be issued by one thread while other threads the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage. Non-blocking communication appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control. For these reasons, today's systems do not generally provide the nonblocking form of *receive*.

Message destinations • Chapter 3 explains that 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. A port has exactly one receiver but can have many senders. Processes may use multiple ports to 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.

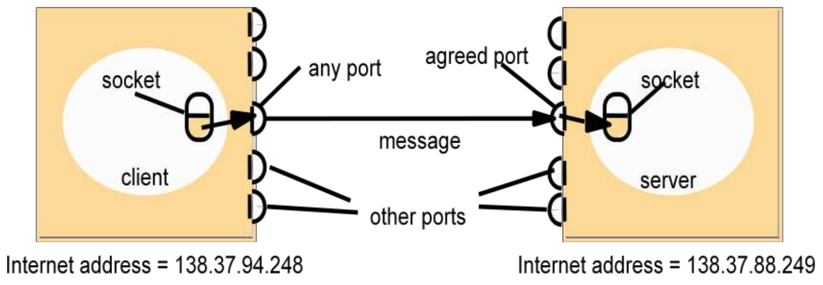
Reliability • As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a 'reasonable' number of packets being dropped or lost. In contrast, a point-to-point message service can be describedas unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

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 an endpoint for ommunication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh

OS. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process, is shown in the following figure.


socket
client
any port
agreed port
message
other ports
socket
server
Internet address = 138.37.94.248
Internet address = 138.37.88.249

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 bya process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number(216) of possible port numbers for use by local processes for receiving messages. Any processmay make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

Java API for Internet addresses • As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, *InetAddress*, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames. For example, instances of *InetAddress* that contain Internet addresses can be created by calling a static method of *InetAddress*, giving a DNS hostname as the argument. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host whose DNS name is *bruno.dcs.qmul.ac.uk*, use:

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

This method can throw an *UnknownHostException*. Note that the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet addresses – 4 bytes in IPv4 and 16 bytes in IPv6.

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 8 kilobytes. Any application requiring messages larger than the maximum must fragment them into chunks of that size.

Generally, an application, for example DNS, will decide on a size that is not excessively large but is adequate for its intended use.

Blocking: 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 a 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 receive requests 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, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

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 to check where the message came from. It is possible to connect a datagram socket 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.

Failure model for UDP datagrams • 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 datagrams suffer from the following failures:

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

Ordering: Messages can sometimes be delivered out of sender order. Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require.

A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements.

Use of UDP • For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP. UDP datagrams are sometimes an attractive choice because they do notsuffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

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

• the transmission of extra messages;

• latency for the sender.

Java API for UDP datagrams • The Java API provides datagram communication by means of two classes: DatagramPacket and DatagramSocket. DatagramPacket:

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 sends

it and another receives it. UDP server repeatedly receives a request and send sit back to the client

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

DatagramSocket: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to 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 *SocketException* if the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.

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

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

TCP stream communication

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:

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

Lost messages: The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required.

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

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

Message destinations: A pair of communicating processes establish 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 considerable overhead for a single client-server request and reply.

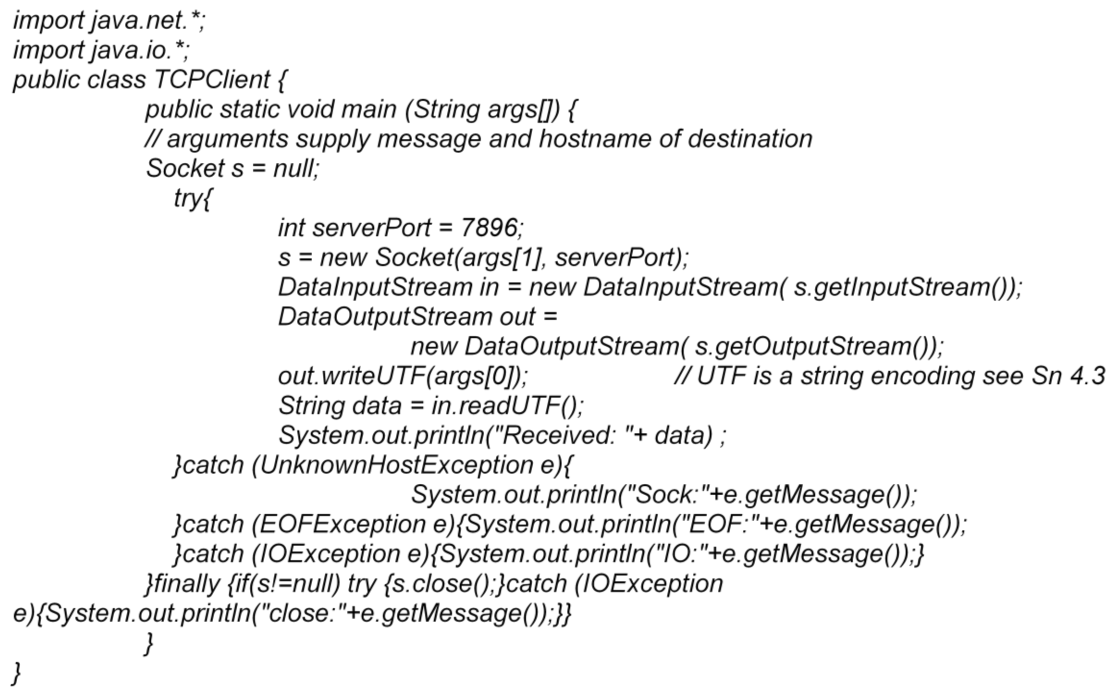
Java API for TCP streams • The Java interface to TCP streams is provided in the classes

ServerSocket and *Socket*:

ServerSocket: This class is intended for use by a server to create a socket at a server port for listening for *connect* requests from clients. Its *accept* method gets a *connect* request from the queue or, if the queue is empty, blocks until one arrives. The result of executing *accept* is an instance of *Socket* – a socket to use for communicating with the client.

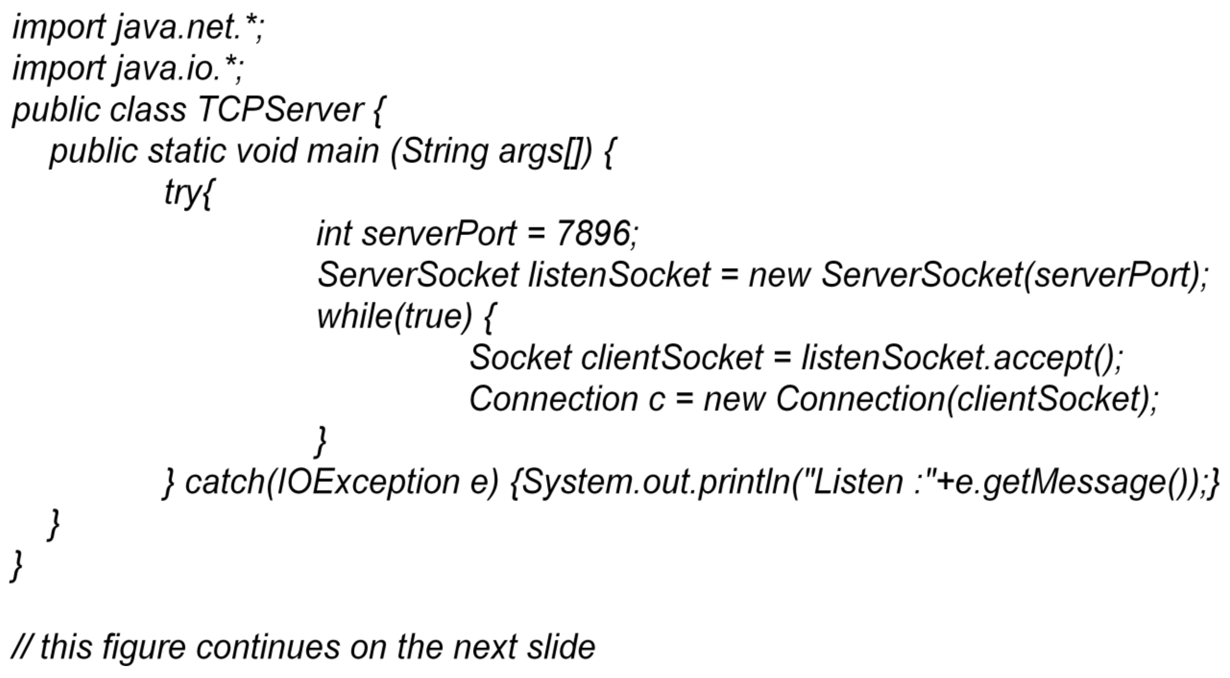
Socket: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and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

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



```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
          try{
                int serverPort = 7896;
                s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =
                        new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);          // UTF is a string encoding see Sn 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
          }catch (UnknownHostException e){
                        System.out.println("Sock:"+e.getMessage());
          }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
          }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException
e){System.out.println("close:"+e.getMessage());}}
        }
}
```

TCP server 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
```

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

External data representation and marshalling

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

One of the following methods can be used to enable any two computers to exchange binary data values:

- The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.
- The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary. Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. An agreed standard for the representation of data structures and primitive values is called an *external data representation*. *Marshalling* is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. *Unmarshalling* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Thus marshalling consists of the translation of structured data items and

primitive values into an external data representation. Similarly, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

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

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

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

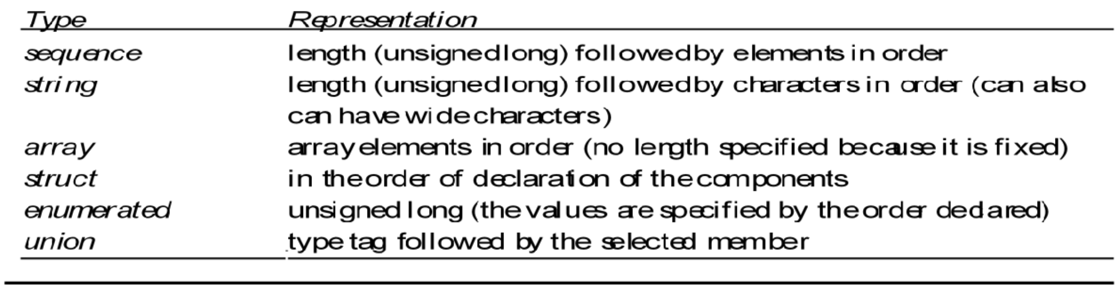
• XML (Extensible Markup Language), which defines a textual fomat for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages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

In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer. Even in the case of XML, which is textual and therefore more accessible to hand-encoding, software for marshalling and unmarshalling is available for all commonly used platforms and programming environments. Because marshalling requires the consideration of all the finest details of the representation of the primitive components of composite objects, the process is likely to be error-prone if carried out by hand. Compactness is another issue that can be addressed in the design of automatically generated marshalling procedures.

In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation. The HTTP protocol, which is described in Chapter 5, is another example of the textualapproach. Another issue with regard to the design of marshalling methods is whether the marshalled data should include information concerning the type of its contents. For example, CORBA’s representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information, but in different ways. Java puts all of the required type information into the serialized form, but XML documents may refer to externally defined sets of names (with types) called *namespaces*.

Although we are interested in the use of an external data representation for the arguments and results of RMIs and RPCs, it does have a more general use for representing data structures, objects or structured documents in a form suitable for transmission in messages or storing in files.

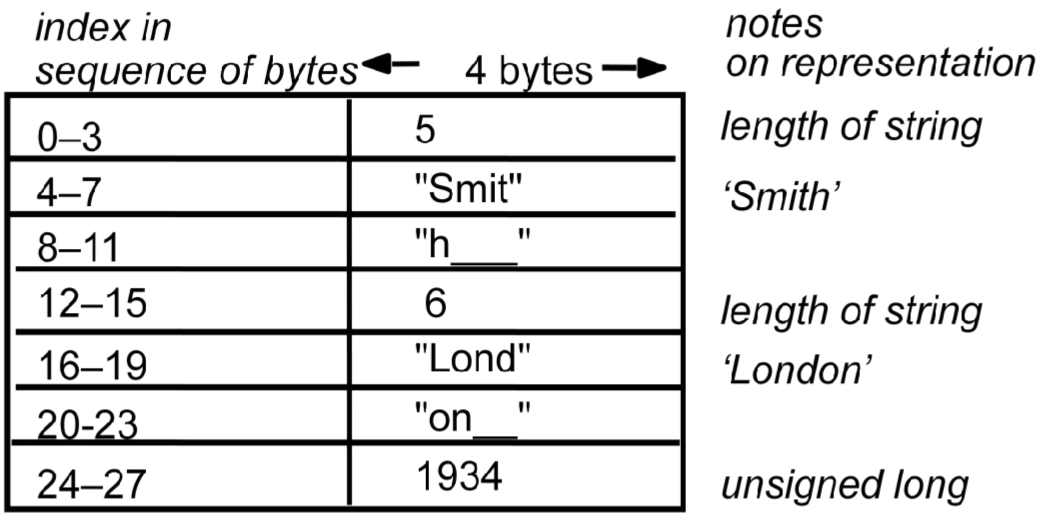
CORBA CDR for constructed types



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

COBRBA's Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0. 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*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types, which are described in Figure 4.7. Each argument or result in a remote invocation is
represented by a sequence of bytes in the invocation or result message.



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

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Marshalling in CORBA • Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL (see Section 8.3.1), which provides a notation for describing the types of the arguments and results of RMI methods.

Java object serialization

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

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

Extensible Markup Language (XML)

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

XML data items are tagged with 'markup' strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. That is, in XML, the tags relate to the structure of the text that they enclose, in contrast to HTML, in which the tags specify how a browser could display the text. For a specification of XML, see the pages on XML provided by W3C [www.w3.org VI].

XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services. However, XML is also used in many other ways, including in archiving and retrieval systems – although an XML archive may be larger than a binary one, it has the advantage of being readable on any computer.
Other examples of uses of XML include for the specification of user interfaces and the encoding of configuration files in operating systems.

XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them. For example, clients usually use SOAP messages to communicate with web
services. SOAP is an XML format whose tags are published for use by web services and their clients.

Some external data representations (such as CORBA CDR) do not need to be self describing, because it is assumed that the client and server exchanging a message have prior knowledge of the order and the types of the information it contains. However, XML was intended to be used by multiple applications for different purposes. The provision of tags, together with the use of namespaces to define the meaning of the tags, has made this possible. In addition, the use of tags enables applications to select just those parts of a document it needs to process: it will not be affected by the addition of information relevant to other applications.

-

<u>XML definition of the Person structure</u>

```
<person id="123456789">
            <name>Smith</name>
            <place>London</place>
            <year>1934</year>
            <!-- a comment -->
</person >
```

Remote object references

Java and CORBA that support the distributed object model. It is not relevant to XML. When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A *remote object reference* is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked. Chapter 5 explains that remote object references are also passed as arguments and returned as results of remote method invocations, that each remote object has a single remote object reference and that remote object references can be compared to see whether they refer to the same remote object. Here, we discuss the external representation of remote object references.

Client-server communication

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments) sends a request message to the remote object and returns the reply.

The arguments specify the remote object, the method to be invoked and the arguments of that method.

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

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

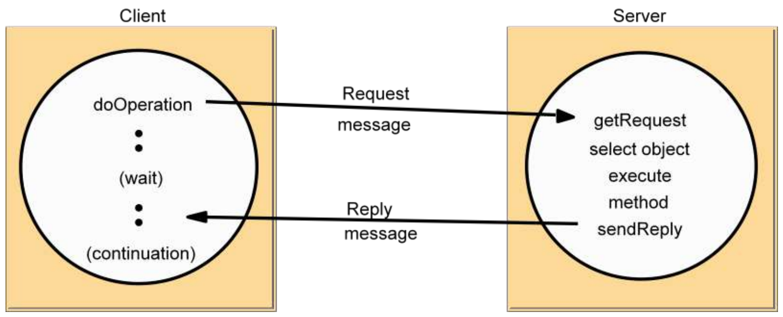
RPC exchange protocols

HTTP request message

HTTP reply message

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

Request-reply communication



Group communication

A *multicast operation* is more appropriate – this is an operation that sends a single message from one process to
each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired behaviour of a multicast. The simplest multicast rotocol provides no guarantees about message delivery or ordering.

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

1. *Fault tolerance based on replicated services*: A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.
2. *Discovering services in spontaneous networking*: Section 1.3.2 defines service discovery in the context of spontaneous networking. Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
3. *Better performance through replicated data*: Data are replicated to increase the performance

of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value is multicast to the processes managing the replicas.

4. *Propagation of event notifications*: Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publishsubscribe protocols may make use of group multicast to disseminate events to subscribers (see Chapter 6).

IP multicast – An implementation of multicast communication

IP multicast • *IP multicast* is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers – ports belong to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. A *multicast group* is specified by a Class D Internet address – that is, an address whose first 4 bits are 1110 in IPv4.

At the application programming level, IP multicast is available only via UDP. An application program performs multicasts by sending UDP datagrams with multicast addresses and ordinary port numbers. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group. At the IP level, a computer belongs to a multicast group when one or more of its processes has sockets that belong to that group. When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

Multicast routers: IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet.

Internet multicasts make use of multicast routers, which forward single datagrams 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. To understand how routers know which other routers have members of a multicast group.

Multicast address allocation: As discussed in Chapter 3, Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA). The management of this address space is reviewed annually, with current practice documented in RPC 3171. This document defines a partitioning of this address space into a number of blocks, including:

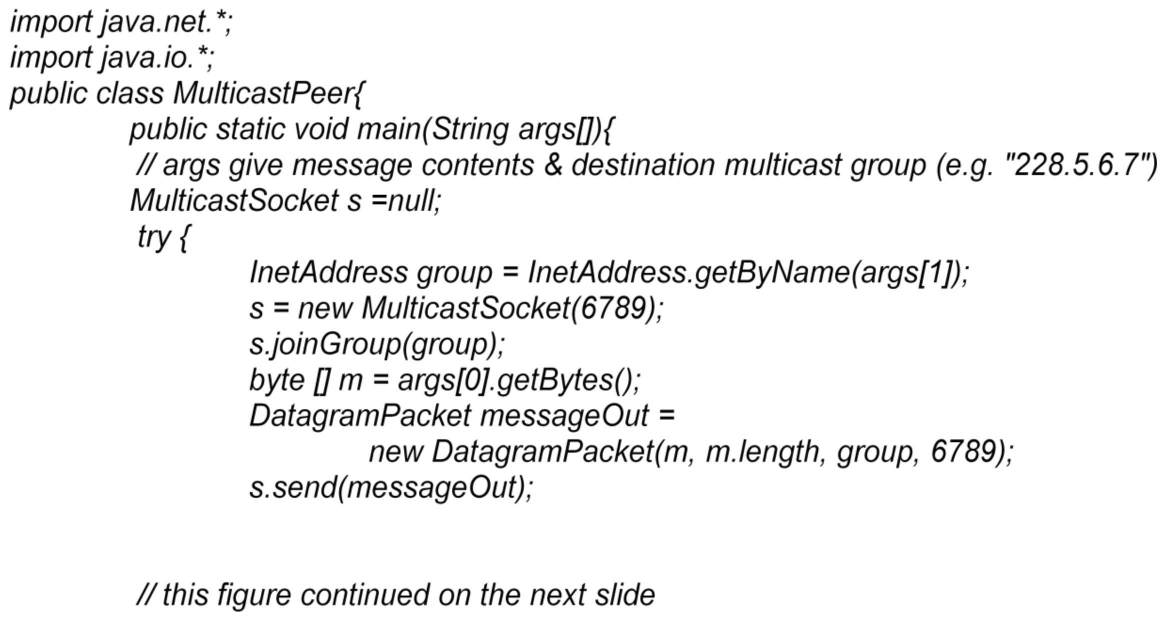
- Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.
- Internet Control Block (224.0.1.0 to 224.0.1.225).
- Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.
- Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic (to constrain propagation).

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

Java API to IP multicast • The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*, which is a subclass of *DatagramSocket* with the additional capability of being able to join multicast groups. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a or any free local port. A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will

receive datagrams sent by processes on other computers to that group at that port.A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket.

Multicast peer joins a group and sends and receives datagrams



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


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

Reliability and ordering of multicast

The effect of the failure semantics of IP multicast on the four examples of the use of replication

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

Communication between Distributed Objects

The Object Model

Five Parts of the Object Model

– An object-oriented program consists of a collection of interacting objects

• Objects consist of a set of data and a set of methods

• In DS, object’s data should be accessible only via methods

Object References

– Objects are accessed by object references

– Object references can be assigned to variables, passed as arguments, and returned as the result of a method

– Can also specify a method to be invoked on that object

Interfaces

– Provide a definition of the signatures of a set of methods without specifying their implementation

– Define types that can be used to declare the type of variables or of the parameters andreturn values of methods

Actions

– Objects invoke methods in other objects

– An invocation can include additional information as arguments to perform the behavior specified by the method

– Effects of invoking a method

1. The state of the receiving object may be changed
2. A new object may be instantiated
3. Further invocations on methods in other objects may occur
4. An exception may be generated if there is a problem encountered

Exceptions

– Provide a clean way to deal with unexpected events or errors

– A block of code can be defined to throw an exception when errors or unexpectedconditions occur. Then control passes to code that catches the exception

Garbage Collection

– Provide a means of freeing the space that is no longer needed

– Java (automatic), C++ (user supplied)

Distributed Objects

• Physical distribution of objects into different processes or computers in a distributed system

– Object state consists of the values of its instance variables

– Object methods invoked by remote method invocation (RMI)

– Object encapsulation: object state accessed only by the object methods

Usually adopt the client-server architecture

– Basic model

• Objects are managed by servers and

• Their clients invoke their methods using RMI

– Steps

1. The client sends the RMI request in a message to the server
2. The server executes the invoked method of the object
3. The server returns the result to the client in another message

– Other models

• Chains of related invocations: objects in servers may become clients of objects in other servers

• Object replication: objects can be replicated for fault tolerance and performance

• Object migration: objects can be migrated to enhancing performance and availability

The Distributed Object Model

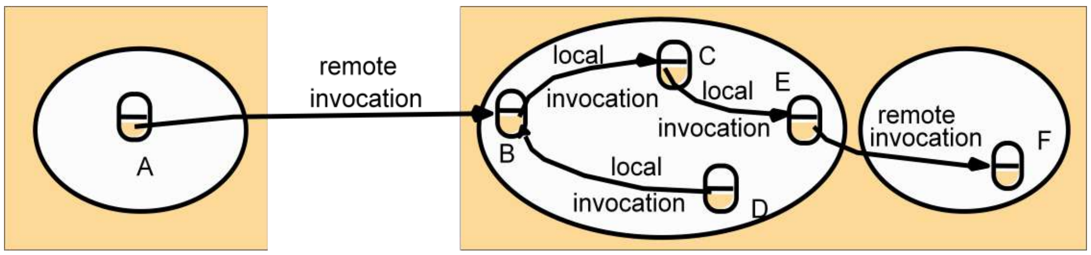
Two fundamental concepts: Remote Object Reference and Remote Interface

– Each process contains objects, some of which can receive remote invocations are called remote objects (B, F), others only local invocations

– Objects need to know the remote object reference of an object in another process in order to invoke its methods, called remote method invocations

– Every remote object has a remote interface that specifies which of its methods can be invoked remotely

Remote and local method invocations



Five Parts of Distributed Object Model

• Remote Object References

– accessing the remote object

– identifier throughout a distributed system

– can be passed as arguments

• Remote Interfaces

– specifying which methods can be invoked remotely

– name, arguments, return type

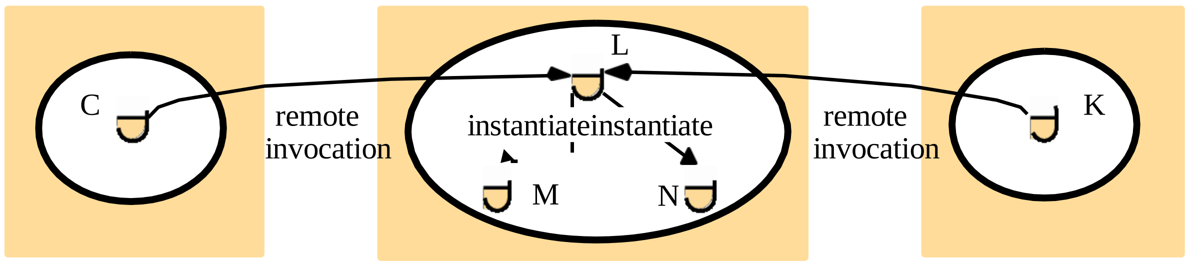
– Interface Definition Language (IDL) used for defining remote interface

Remote Object and Its remote Interface

• Actions

– An action initiated by a method invocation may result in further invocations on methods in other objects located indifference processes or computers

– Remote invocations could lead to the instantiation of new objects, ie. objects M and Nof following figure.



• Exceptions

– More kinds of exceptions: i.e. timeout exception

- RMI should be able to raise exceptions such as timeouts that are due to distribution as wellas those raised during the execution of the method invoked

• Garbage Collection

- Distributed garbage collections is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting

Design Issues for RMI

• Two design issues that arise in extension of local method invocation for RMI

– The choice of invocation semantics

• Although local invocations are executed exactly once, this cannot always be the case for RMI due to transmission error

– Either request or reply message may be lost

– Either server or client may be crashed

– The level of transparency

• Make remote invocation as much like local invocation as possible

RMI Design Issues: Invocation Semantics

• Error handling for delivery guarantees

– Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed

– Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server

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

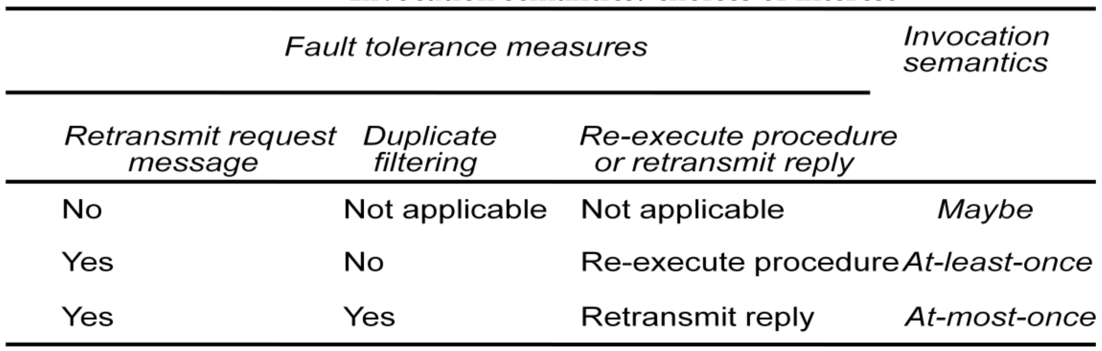
• Choices of invocation semantics

– Maybe: the method executed once or not at all (no retry nor retransmit)

– At-least-once: the method executed at least once

– At-most-once: the method executed exactly once

Invocation semantics: choices of interest



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*

RMI Design Issues: Transparency

• Transparent remote invocation: like a local call

– marshalling/unmarshalling

– locating remote objects

– accessing/syntax

• Differences between local and remote invocations

– latency: a remote invocation is usually several order of magnitude greater than that ofa local one

– availability: remote invocation is more likely to fail

– errors/exceptions: failure of the network? server? hard to tell

• syntax might need to be different to handle different local vs remote errors/exceptions(e.g. Argus)

– consistency on the remote machine:

• Argus: incomplete transactions, abort, restore states [as if the call was never made]

Implementation of RMI

• Communication module

– Two cooperating communication modules carry out the request-reply protocols: message type, request ID, remote object reference

• Transmit request and reply messages between client and server

• Implement specific invocation semantics

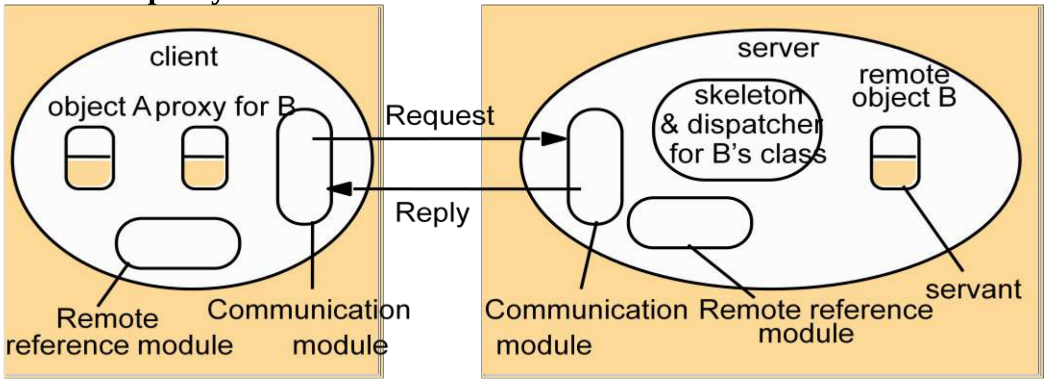
– The communication module in the server

• selects the dispatcher for the class of the object to be invoked,

• passes on local reference from remote reference module,

• returns request

The role of proxy and skeleton in remote method invocation



• **Remote reference module**

– Responsible for translating between local and remote object references and for creatingremote object references

– remote object table: records the correspondence between local and remote object references

• remote objects held by the process (B onserver)

• local proxy (B onclient)

– When a remote object is to be passed for the first time, the module is asked to create a remote object reference, which it adds to its table

• **Servant**

– An instance of a class which provides the body of a remoteobject

– handles the remote requests

•**RMI software**

– Proxy: behaves like a local object, but represents the remote object

– Dispatcher: look at the methodID and call the corresponding method in the skeleton

– Skeleton: implements the method

Generated automatically by an interface compiler

Implementation Alternatives of RMI

• **Dynamic invocation**

– Proxies are static—interface complied into client code

– Dynamic—interface available during run time

• Generic invocation; more info in ―Interface Repository‖ (COBRA)

• Dynamic loading of classes (Java RMI)

•**Binder**

– A separate service to locate service/object by name through table mapping for namesand remote object references

• **Activation of remote objects**

– Motivation: many server objects not necessarily in use all of the time

• Servers can be started whenever they are needed by clients, similar to inetd

– Object status: active or passive

• active: available for invocation in a running process

• passive: not running, state is stored and methods are pending

– Activation of objects:

• creating an active object from the corresponding passive object by creatinga new instance of its class

• initializing its instance variables from the stored state

– Responsibilities of activator

• Register passive objects that are available for activation

• Start named server processes and activate remote objects in them

• Keep track of the locations of the servers for remote objects that it has already activated

• **Persistent object stores**

– An object that is guaranteed to live between activations of processes is called a persistent object

– Persistent object store: managing the persistent objects

• stored in marshaled from on disk for retrieval

• saved those that were modified

– Deciding whether an object is persistent or not:

• persistent root: any descendent objects are persistent (persistent Java, PerDiS)

• some classes are declared persistent (Arjuna system)

• Object location

– specifying a location: ip address, port #, ...

– location service for migratable objects

• Map remote object references to their probable current locations

• Cache/broadcast scheme (similar to ARP)

– Cache locations

– If not in cache, broadcast to find it

• Improvement: forwarding (similar to mobile IP)

Distributed Garbage Collection

• Aim: ensure that an object

– continues to exist if a local or remote reference to it is still held anywhere

– be collected as soon as no object any longer holds a reference to it

• General approach: reference count

• Java's approach

– the server of an object (B) keeps track of proxies

– when a proxy is created for a remote object

• addRef(B) tells the server to add an entry

– when the local host's garbage collector removes the proxy

• removeRef(B) tells the server to remove the entry

– when no entries for object B, the object on server is deallocated

Remote Procedure Call

• client: "stub" instead of "proxy" (same function, different names)

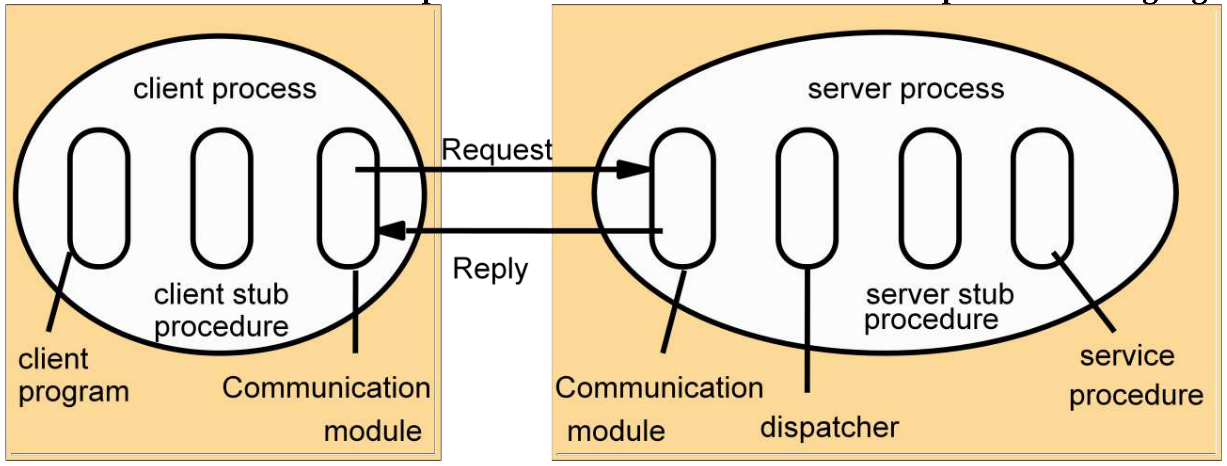
– local call, marshal arguments, communicate the request

• server:

– dispatcher

– "stub": unmarshal arguments, communicate the results back

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



Case Study: Sun RPC

• Sun RPC: client-server in the SUN NFS (network file system)

– UDP or TCP; in other unix OS as well

– Also called ONC (Open Network Computing) RPC

• Interface Definition Language (IDL)

– initially XDR is for data representation, extended to be IDL

– less modern than CORBA IDL and Java

• program numbers instead of interface names

• procedure numbers instead of procedure names

• single input parameter (structs)

– rpcgen: compiler for XDR

• client stub; server main procedure, dispatcher, and server stub

• XDR marshalling, unmarshaling

• Binding (registry) via a local binder - portmapper

– Server registers its program/version/port numbers with portmapper

– Client contacts the portmapper at a fixed port with program/version numbers to get the server port

– Different instances of the same service can be run on different computers at different ports

• Authentication

– request and reply have additional fields

– unix style (uid, gid), shared key for signing, Kerberos

```
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;
```

Files interface in Sun XDR

Events and Notifications

• Idea behind the use of events

– One object can react to a change occurring in another object

• Events

– Notifications of events: objects that represent events

• asynchronous and determined by receivers what events are interested

– event types

• each type has attributes (information in it)

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

• Publish-subscribe paradigm

– publish events to send

– subscribe events to receive

• Main characteristics in distributed event-based systems

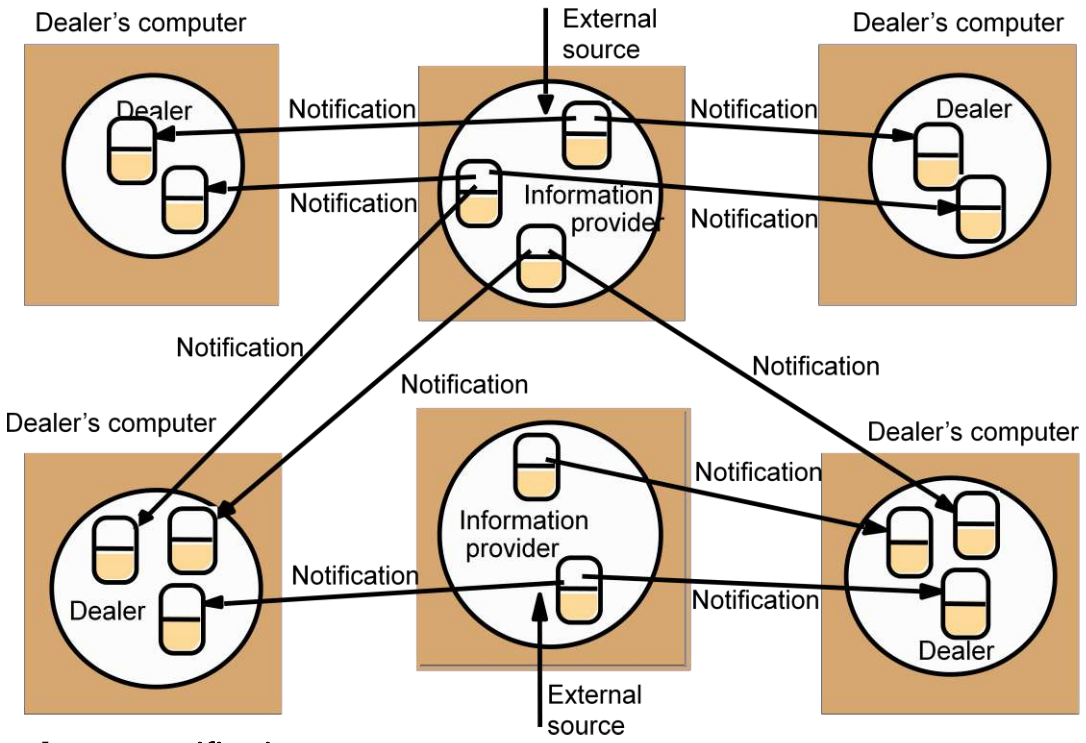
– Heterogeneous: a way to standardize communication inheterogeneous systems

• not designed to communicate directly

– Asynchronous: notifications are sent asynchronously

• no need for a publisher to wait for each subscriber--subscribers come and go

Dealing room system: allow dealers using computers to see the latest information about the market prices of the stocks they deal in



Distributed Event Notification

• Distributed event notification

– decouple publishers from subscribers via an event service (manager)

• Architecture: roles of participating objects

– object of interest (usually changes in states are interesting)

– event

– notification

– subscriber

– observer object (proxy) [reduce work on the object of interest]

• forwarding

• filtering of events types and content/attributes

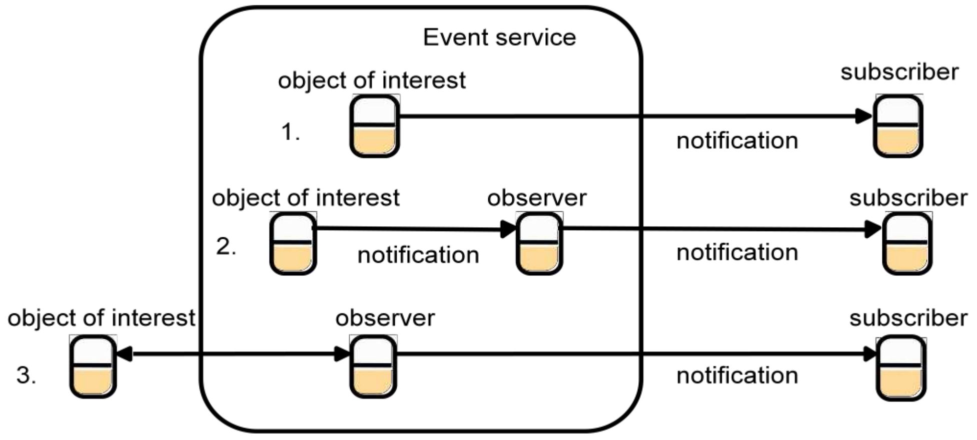
• patterns of events (occurrence of multiple events, not just one)

• mailboxes (notifications in batch es, subscriber might not be ready)

– publisher (object of interest or observer object)

• generates event notifications

Example: Distributed Event Notification



• Three cases

– Inside object without an observer: send notifications directly to the subscribers

– Inside object with an observer: send notification via the observer to the subscribers

– Outside object (with an observer)

1. An observer queries the object of interest in order to discover when events occur
2. The observer sends notifications to the subscribers

UNIT IV

Distributed File Systems: Introduction, File service Architecture, Case Study1: Sun Network File System, Case Study 2: The Andrew File System.

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

DISTRIBUTED FILE SYSTEMS

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 I/O requests.

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 scale gracefully to huge ones (hundreds through tens of thousands of systems).

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

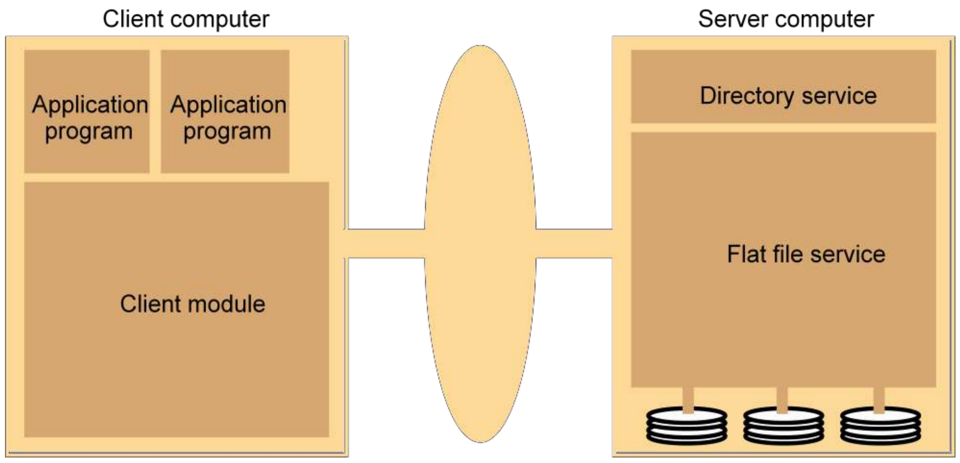
Migration transparency: Files should be able to move around without the client's knowledge. Support fine-grained distribution of data: To optimize performance, we may wish to locate individual objects near the processes that use them

Tolerance for network partitioning: The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

File Service Architecture

- An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components:
  - ➢ A flat file service
  - ➢ A directory service
  - ➢ A client module.
- The relevant modules and their relationship is shown in Figure 5.

Figure 5. File service architecture



- The Client module implements exported interfaces by flat file and directory services on server side.
- Responsibilities of various modules can be defined as follows:

➢ Flat file service:

❖ Concerned with the implementation of operations on the contents of file. Unique File Identifiers (UFIDs) are used to refer to files in all requests for

flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.

➢ Directory service:

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

➢ Client module:

❖ It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.

❖ It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.

- Flat file service interface:
- Figure 6 contains a definition of the interface to a flat file service.

Figure 6. Flat file service operations

Read(FileId, i, n) -> Data -throws BadPosition	if *1≤i≤Length(File):* Reads a sequence of up to n items from a file starting at item *i* and returns it in *Data.*
Write(FileId, i, Data) -throws BadPosition	if *1≤i≤Length(File)+1*: Write a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() -> FileId	Creates a new file of length0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 3.)

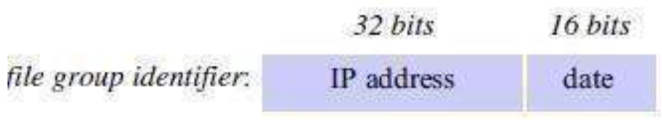
- Access control
  - In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.
- Directory service interface
  - Figure 7 contains a definition of the RPC interface to a directory service.

Figure 7. Directory service operations

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

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

To construct globally unique ID we use some unique attribute of the machine on which it is created. E.g: IP number, even though the file group may move subsequently.



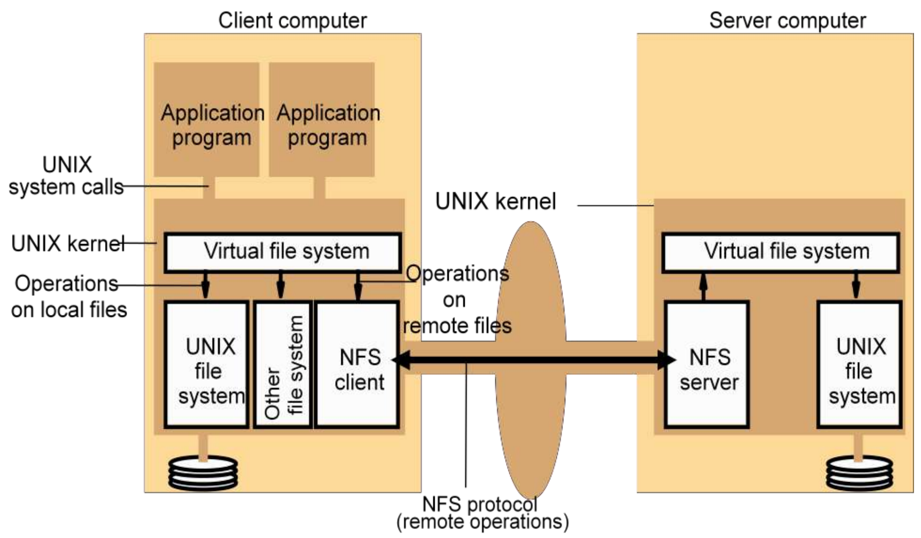
DFS: Case Studies

- NFS (Network File System)
  - Developed by Sun Microsystems (in 1985)
  - Most popular, open, and widely used.
  - NFS protocol standardized through IETF (RFC 1813)
- AFS (Andrew File System)
  - Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
  - A research project to create campus wide file system.
  - Public domain implementation is available on Linux (LinuxAFS)
  - It was adopted as a basis for the DCE/DFS file system in the Open Software Foundation (OSF, www.opengroup.org) DEC (Distributed Computing Environment

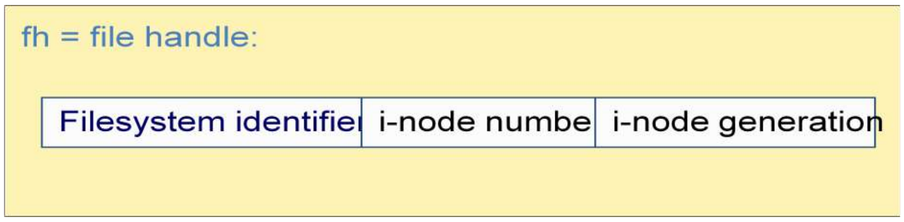
Sun Network File System

NFS architecture

Figure 8 shows the architecture of Sun NFS



- The file identifiers used in NFS are called file handles.


fh = file handle:

Filesystem identifier	i-node number	i-node generation

- A simplified representation of the RPC interface provided by NFS version 3 servers is shown in Figure 9.

Figure 9. NFS server operations (NFS Version 3 protocol, simplified)

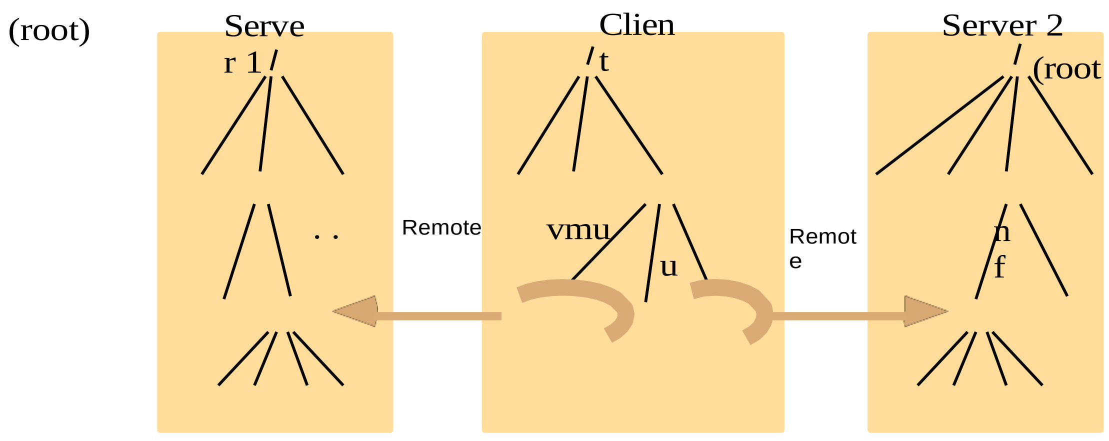
- NFS access control and authentication
  - ➢ The NFS server is stateless server, so the user's identity and access rights must be checked by the server on each request.
    - ❖ In the local file system they are checked only on the file’s access permission attribute.
  - ➢ Every client request is accompanied by the userID and groupID
    - ❖ It is not shown in the Figure 8.9 because they are inserted by the RPC system.
  - ➢ Kerberos has been integrated with NFS to provide a stronger andmore comprehensive security solution.

- Mount service
  - ➢ Mount operation:

    mount(remotehost, remotedirectory, localdirectory)
  - ➢ Server maintains a table of clients who have mounted filesystems at that server.
  - ➢ Each client maintains a table of mounted file systems holding:

    < IP address, port number, file handle>
  - ➢ Remote file systems may be hard-mounted or soft-mounted in a client computer.
  - ➢ Figure 10 illustrates a Client with two remotely mounted file stores.

Figure 10. Local and remote file systems accessible on an NFS client



- Automounter
    - ➢ The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever an ‘empty’ mount point is referenced by a client.
        - ❖ Automounter has a table of mount points with a reference to one or more NFS servers listed against each.
        - ❖ it sends a probe message to each candidate server and then uses the mount service to mount the file system at the first server to respond.
    - ➢ Automounter keeps the mount table small.
    - ➢ Automounter Provides a simple form of replication for read-only file systems.
        - ❖ E.g. if there are several servers with identical copies of /usr/lib then each server will have a chance of being mounted at some clients.
- 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 remote case it doesn't offer necessary synchronization guarantees to clients.
    - ➢ NFS v3 servers offers two strategies for updating the disk:
        - ❖ Write-through - altered pages are written to disk as soon as they are received at the server. When a write() RPC returns, the NFS client knows that the page is on the disk.
        - ❖ Delayed commit - pages are held only in the cache until a commit() call is received for the relevant file. This is the default mode used by NFS v3 clients. A commit() is issued by the client whenever a file is closed.
- Client caching
    - ➢ Server caching does nothing to reduce RPC traffic between client and server
        - ❖ further optimization is essential to reduce server load in large networks.

❖ NFS client module caches the results of read, write, getattr, lookup and readdir operations

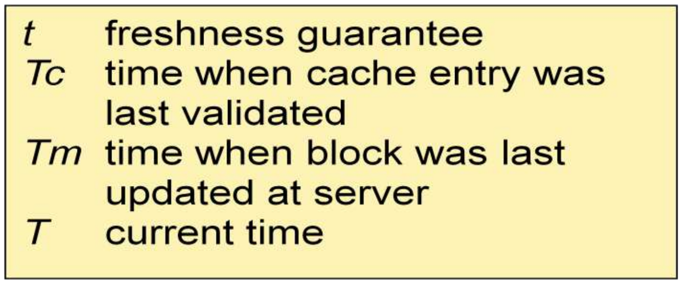
❖ synchronization of file contents (one-copy semantics) is not guaranteed when two or more clients are sharing the same file.

➢ Timestamp-based validity check

❖ It reduces inconsistency, but doesn't eliminate it.

❖ It is used for validity condition for cache entries at the client:

(T - Tc < t) v (Tmclient = Tmserver)



t	freshness guarantee
Tc	time when cache entry was last validated
Tm	time when block was last updated at server
T	current time

- ❖ it is configurable (per file) but is typically set to 3 seconds for files and 30 secs. for directories.
- ❖ it remains difficult to write distributed applications that share files with NFS.

❖ Other NFS optimizations
  - ❖ Sun RPC runs over UDP by default (can use TCP if required).
  - ❖ Uses UNIX BSD Fast File System with 8-kbyte blocks.
  - ❖ reads() and writes() can be of any size (negotiated between client and server).
  - ❖ The guaranteed freshness interval t is set adaptively for individual files to reduce getattr() calls needed to update Tm.
  - ❖ File attribute information (including Tm) is piggybacked in replies to all file requests.

❖ NFS performance
  - ❖ Early measurements (1987) established that:
    - ❖ Write() operations are responsible for only 5% of server calls in typical UNIX environments.
      - ❖ hence write-through at server is acceptable.
    - ❖ Lookup() accounts for 50% of operations -due to step-by-step pathname

resolution necessitated by the naming and mounting semantics.

- ❖ More recent measurements (1993) show high performance.
  - ❖ see www.spec.org for more recent measurements.

❖ NFS summary

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

❖Achievement of transparencies are other goals of NFS:

❖ Access transparency:

❖ The API is the UNIX system call interface for both localand remote files.

❖ Location transparency:

❖ Naming of filesystems is controlled by client mount operations, but transparency can be ensured by an appropriate system configuration.

❖ Mobility transparency:

❖ Hardly achieved; relocation of files is not possible, relocation of filesystems is possible, but requires updates to client configurations.

❖ Scalability transparency:

❖ File systems (file groups) may be subdivided and allocated to separate servers.

❖ Replication transparency:

– Limited to read-only file systems; for writable files, the SUN Network Information Service (NIS) runs over NFS and is used to replicate essential system files.

❖ Hardware and software operating system heterogeneity:

– NFS has been implemented for almost every known operating system and hardware platform and is supported by a variety of filling systems.

❖ Fault tolerance:

– Limited but effective; service is suspended if a server fails. Recovery from failures is aided by the simple statelessdesign.

❖ Consistency:

– It provides a close approximation to one-copy semantics andmeets the needs of the vast majority of applications.

– But the use of file sharing via NFS for communication or close coordination between processes on different computers cannot be recommended.

❖ Security:

– Recent developments include the option to use a secure RPC implementation for authentication and the privacy and security of the data transmitted with read and write operations.

– Efficiency:

❖ NFS protocols can be implemented for use in situations that generate very heavy loads.

Case Study: The Andrew File System (AFS)

AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

Whole-file serving: The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).

Whole file caching: Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' *open* requests in preference to remote copies whenever possible.

▪ Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations.

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

Scenario • Here is a simple scenario illustrating the operation of AFS:

1. When a user process in a client computer issues an *open* system call for a file in the shared

-file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.

2. The copy is stored in the local UNIX file system in the client computer. The copy is then

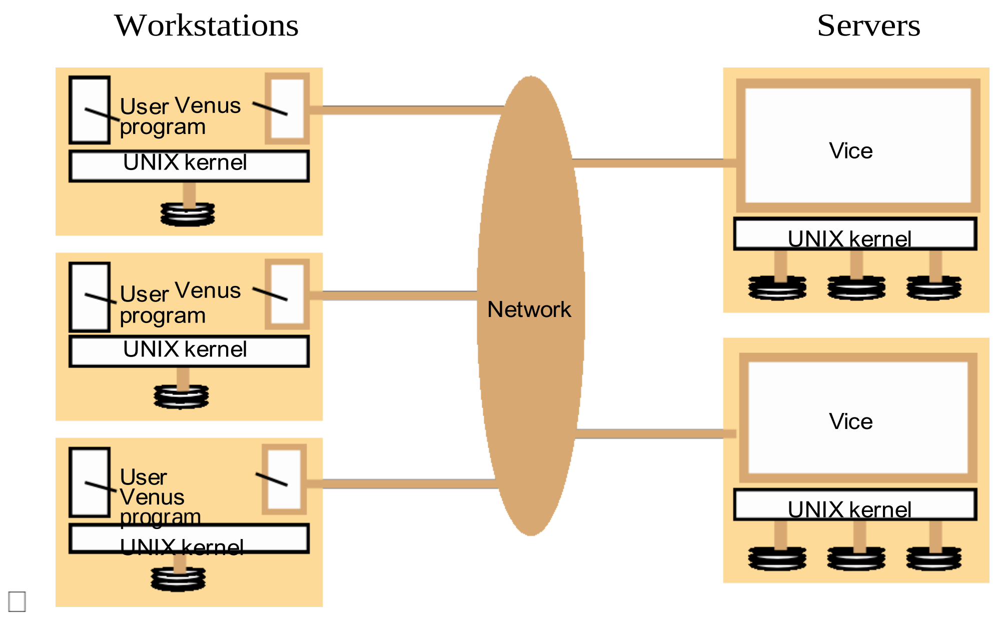
*open*ed and the resulting UNIX file descriptor is returned to the client.

3. Subsequent *read*, *write* and other operations on the file by processes in the client computer are applied to the local copy.

4. When the process in the client issues a *close* system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client’s local disk is retained in caseisneeded again by

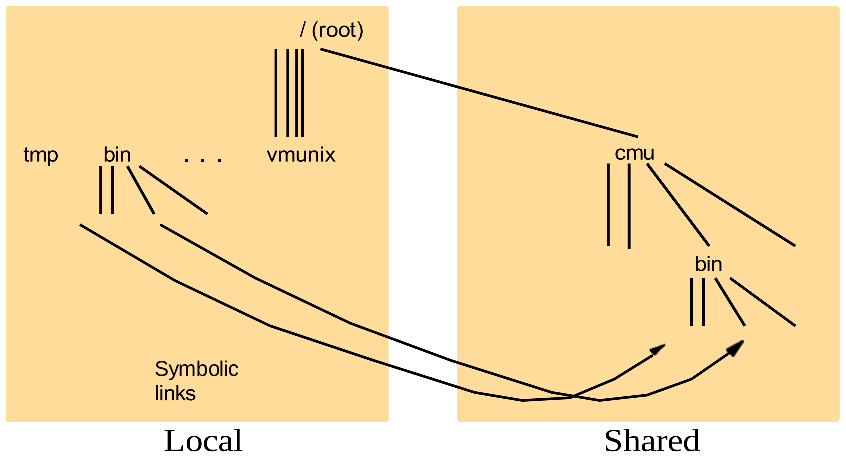
a user-level process on the same workstation.

Figure 11. Distribution of processes in the Andrew File System



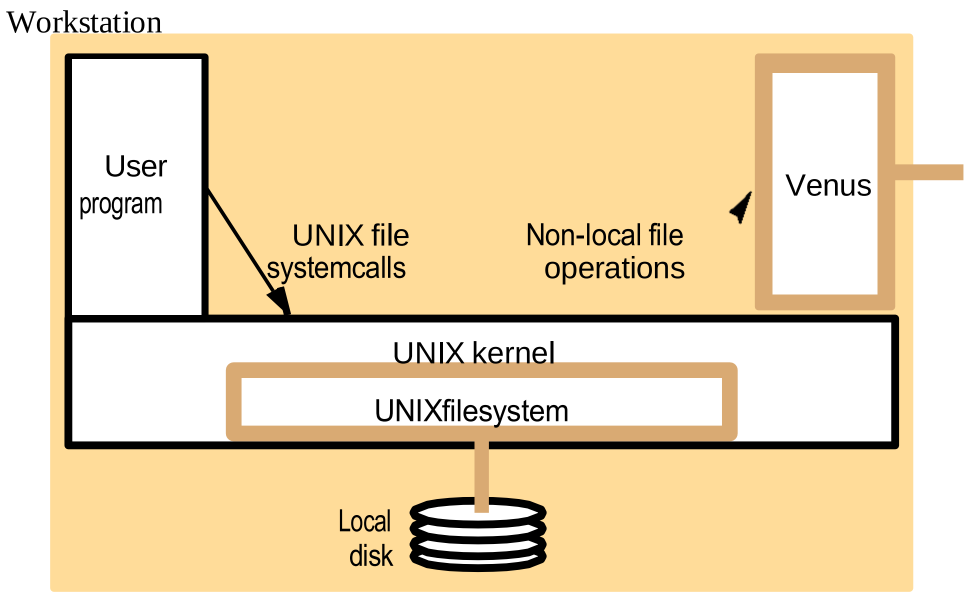
- The files available to user processes running on workstations are either local or shared.
- Local files are handled as normal UNIX files.
- They are stored on the workstation’s disk and are available only to local user processes.
- Shared files are stored on servers, and copies of them are cached on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

Figure 12. File name space seen by clients of AFS



- The UNIX kernel in each workstation and server is a modified version of BSD UNIX.
- The modifications are designed to intercept open, close and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer. (Figure 13)

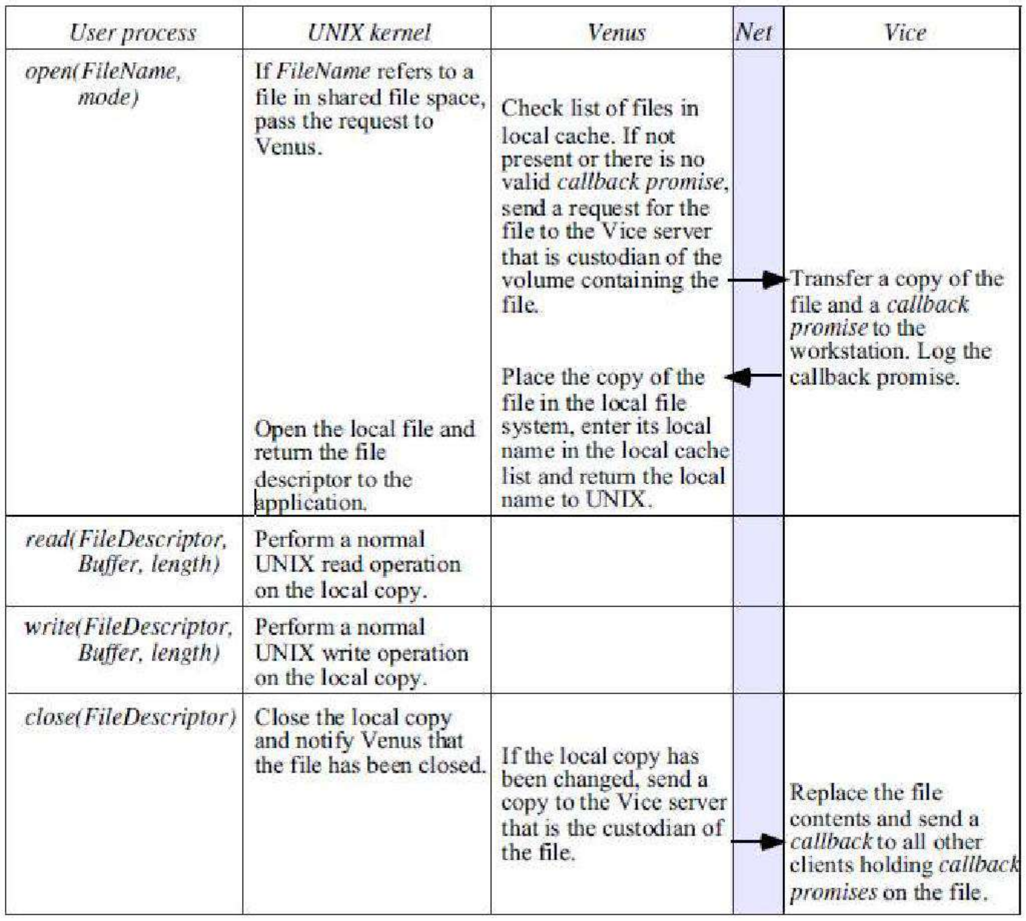
Figure 13. System call interception in AFS



- Figure 14 describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues system calls.

Figure 14. implementation of file system calls in AFS

Figure 12.14 Implementation of file system calls in AFS



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

Cache consistency

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

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

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

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

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

1. UNIX kernel modifications
2. Location database
3. Threads

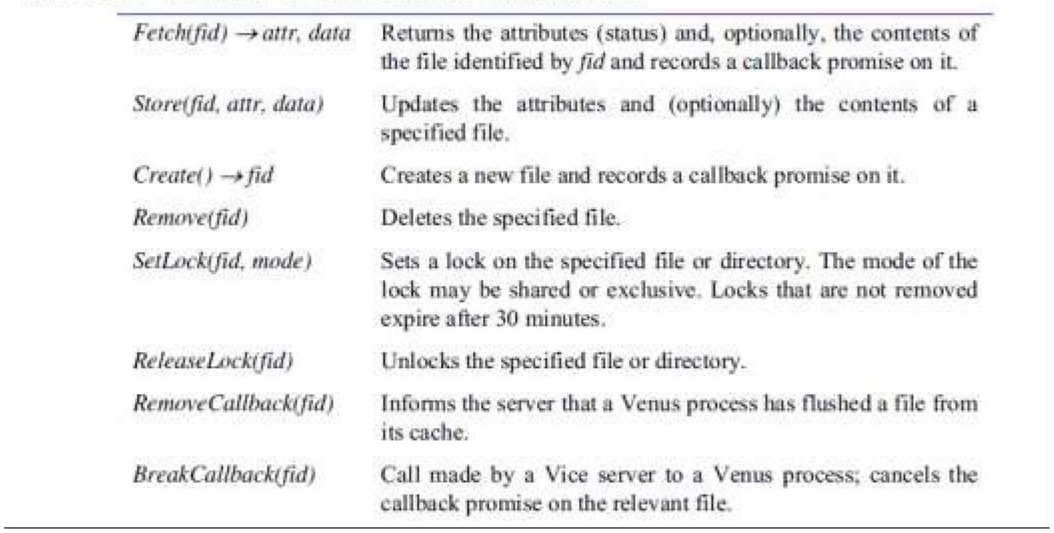


Figure 12.15 The main components of the Vice service interface

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

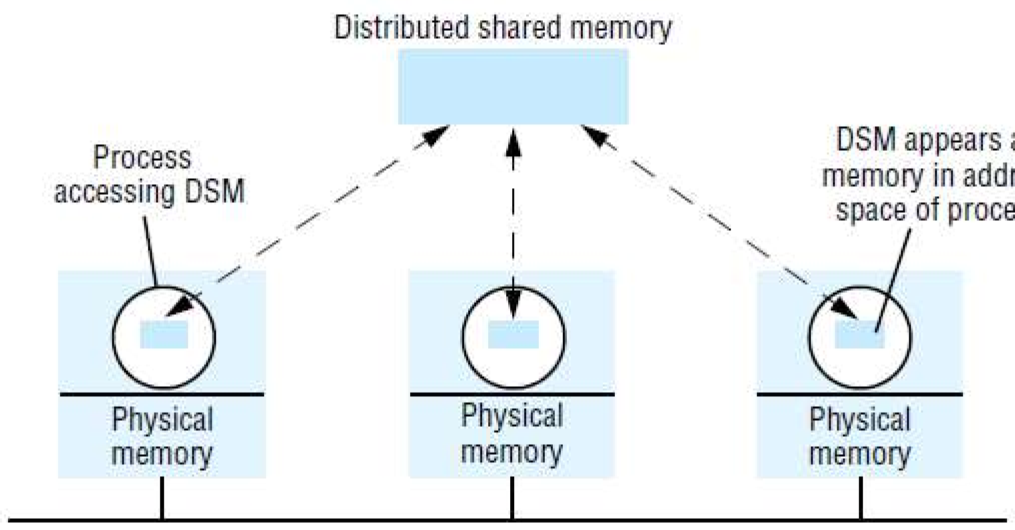
4. Read-only replicas
5. Bulk transfers
6. Partial file caching
7. Performance
8. Wide area support

DISTRIBUTED SHARED MEMORY

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another.

The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client-server systems, where clients normally view server-held resources as abstract data and access them by request(for reasons of modularity and protection).

The distributed shared memory abstraction


Distributed shared memory
Process accessing DSM
DSM appears as memory in address space of process
Physical memory
Physical memory
Physical memory

Message passing cannot be avoided altogether in a distributed system: in the absence if physically shared memory, the DSM runtime support has to send updates in messages between computers. DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access.

In distributed memory multiprocessors and clusters of off-the-shelf computing components (see Section 6.3), the processors do not share memory but are connected by a very high-speed network. These systems, like general-purpose distributed systems, can scale to much greater numbers of processors than a shared-memory multiprocessor’s 64 or so. A central question that has been pursued by the DSM and multiprocessor research communities is whether the investment in knowledge of shared memory algorithms and the associated software can be directly transferred to a more scalable distributed memory architecture.

Message passing versus DSM

As a communication mechanism, DSM is comparable with message passing rather than with request- reply-based communication, since its application to parallel processing, in particular, entails the use of asynchronous communication. The DSM and message passing approaches to programming can be contrasted as follows:

Programming model:

Under the message passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory

the processes involved share variables directly, so no marshalling is necessary – even of pointers to shared variables – and thus no separate communication operations are necessary.

Efficiency :

Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message passing platforms on the same hardware – at least in the case of relatively small numbers of computers (ten or so). However, this result cannot be generalized. The performance of a program based on DSM depends upon many factors, as we shall discuss below – particularly the pattern of data sharing. **Implementation approaches to DSM** Distributed shared memory is implemented using one or a combination of specialized hardware, conventional paged virtual memory or middleware:

*Hardware:*Shared-memory multiprocessor architectures based on a NUMA architecture rely on specialized hardware to provide the processors with a consistent view of shared memory. They handle

memory LOAD and STORE instructions by communicating with remote memory and cache modules as necessary to store and retrieve data.

Paged virtual memory:

Many systems, including Ivy and Mether , implement DSM as a region of virtual
memory occupying the same address range in the address space of every
participating process.

```
}
#include "world.h"
struct shared { int
a, b; }; Program
Writer:
main()
{
struct shared *p;
methersetup(); /* Initialize the Mether
runtime */ p = (struct shared
*)METHERBASE;
/* overlay structure on METHER
segment */

p->a = p->b = 0; /* initialize fields to
zero */
while(TRUE){ /* continuously update structure
fields */ p –>a = p –>a + 1;
p –>b = p –>b - 1;
}
}
```

Program Reader:

```
main()
{
struct shared *p;
methersetup();
p = (struct shared *)METHERBASE;
while(TRUE){ /* read the fields once every second */
printf("a = %d, b = %d\n", p –>a, p –>b);
sleep(1);
}
}
```

Middleware:

Some languages such as Orca, support forms of DSM without any hardware or paging support, in a platform-neutral way. In this type of implementation, sharing is implemented by communication between instances of the user-level support layer in clients and servers. Processes make calls to this layer when they access data items in DSM. The instances of this layer at the different computers access local data items and communicate as necessary to maintain consistency.

Design and implementation issues

The synchronization model used to access DSM consistently at the application level; the DSM consistency model, which governs the consistency of data values accessed from different computers; the update options for communicating written values between computers; the granularity of sharing in a DSM implementation; and the problem of thrashing.

Structure

A DSM system is just such a replication system. Each application process is presented with some abstraction of a collection of objects, but in this case the 'collection' looks more or less like memory. That is, the objects can be addressed in some fashion or other. Different approaches to DSM vary in what they consider to be an 'object' and in how objects are addressed. We consider three approaches, which view DSM as being composed respectively of contiguous bytes, language-level objects or immutable data items.

Byte-oriented

This type of DSM is accessed as ordinary virtual memory – a contiguous array of bytes. It is the

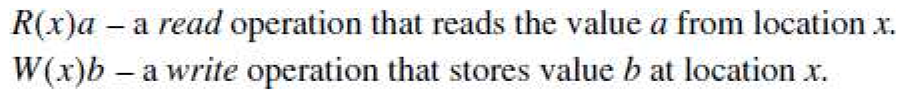
view illustrated above by the Mether system. It is also the view of many other DSM systems, including Ivy.It allows applications (and language implementations) to impose whatever data structures they want on the shared memory. The shared objects are directly addressible memory locations (in practice, the shared locations may be multi-byte words rather than individual bytes). The only operations upon those objects are *read* (or LOAD) and *write* (or STORE). If *x* and *y* are two memory locations, then we denote instances of these operations as follows:

Object-oriented

The shared memory is structured as a collection of language-level objects with higher-level semantics than simple *read* / *write* variables, such as stacks and dictionaries. The contents of the shared memory are changed only by invocations upon these objects and never by direct access to their member variables. An advantage of viewing memory in this way is that object semantics can be utilized when enforcing consistency.

Immutable data

When reading or taking a tuple from tuple space, a process provides a tuple specification and the tuple space returns any tuple that matches that specification – this is a type of



$R(x)a$ – a *read* operation that reads the value a from location x.
$W(x)b$ – a *write* operation that stores value b at location x.

associative addressing. To enable processes to synchronize their activities, the *read* and *take*

operations both block until there is a matching tuple in the tuple space.

Synchronization model

Many applications apply constraints concerning the values stored in shared memory. This is as true of applications based on DSM as it is of applications written for sharedmemory multiprocessors (or indeed for any concurrent programs that share data, such as operating system kernels and multi- threaded servers). For example, if *a* and *b* are two variables stored in DSM, then a constraint might be that *a*=*b* always. If two or moreprocesses execute the following code:

a:= *a* + 1;

b := *b* + 1;

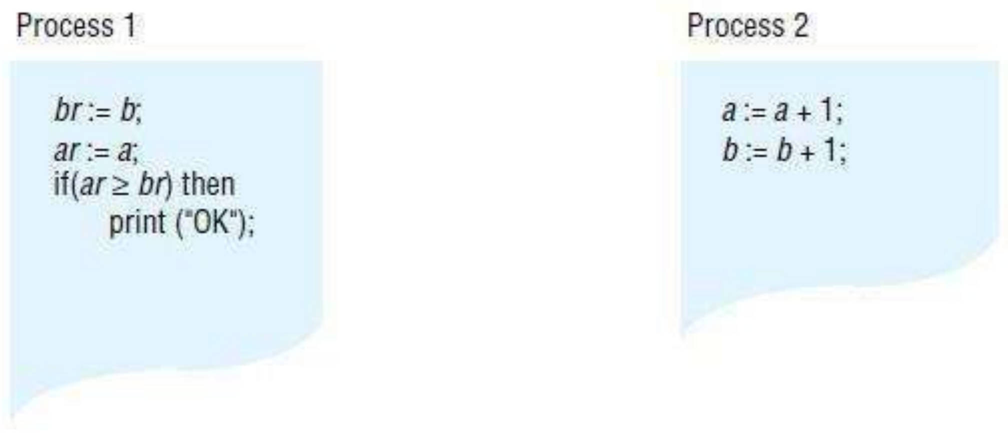
then an inconsistency may arise. Suppose *a* and *b* are initially zero and that process 1gets as far as setting *a* to 1. Before it can increment *b*, process 2 sets *a* to 2 and *b* to 1.

Consistency model

The local replica manager is implemented by a combination of middleware (the DSM runtime layer in each process) and the kernel. It is usual for middleware to perform the majority of DSM processing. Even in a page-based DSM implementation, the kernel usually provides only basic page mapping, page-fault handling and communication mechanisms and middleware is

responsible for implementing the page-sharing policies. If DSM segments are persistent, then one or more storage servers (for example, file servers) will also act as replica managers.

Two processes accessing shared variables



```
br := b;
ar := a;
if(ar ≥ br) then
      print ("OK");
```

```
a := a + 1;
b := b + 1;
```

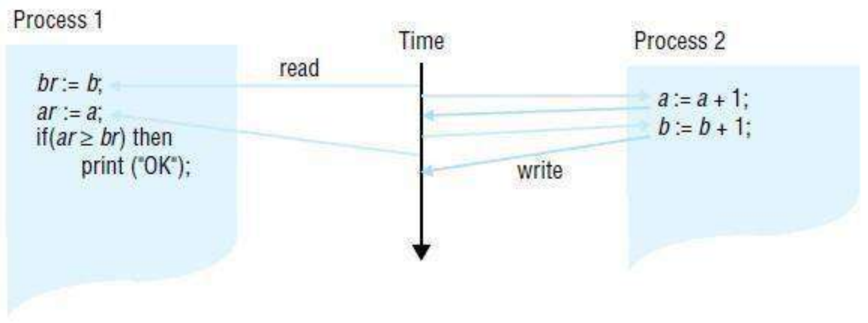
Sequential consistency

A DSM system is said to be sequentially consistent if *for any execution* there is some interleaving of the series of operations issued by all the processes that satisfies the following two criteria:

SC1: The interleaved sequence of operations is such that if R(x) a occurs in the sequence, then either the last write operation that occurs before it in the interleaved sequence is W(x) a, or no write operation occurs before it and *a* is the initial value of *x*.

SC2: The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

Interleaving under sequential consistency



Coherence

Coherence is an example of a weaker form of consistency. Under coherence, every process agrees on the order of write operations to the same location, but they do not necessarily agree on the ordering of write operations to different locations. We can think of coherence as sequential consistency on a locationby- location basis. Coherent DSM can be implemented by taking a protocol for implementing sequential consistency and applying it separately to each unit of replicated data – for example, each page.

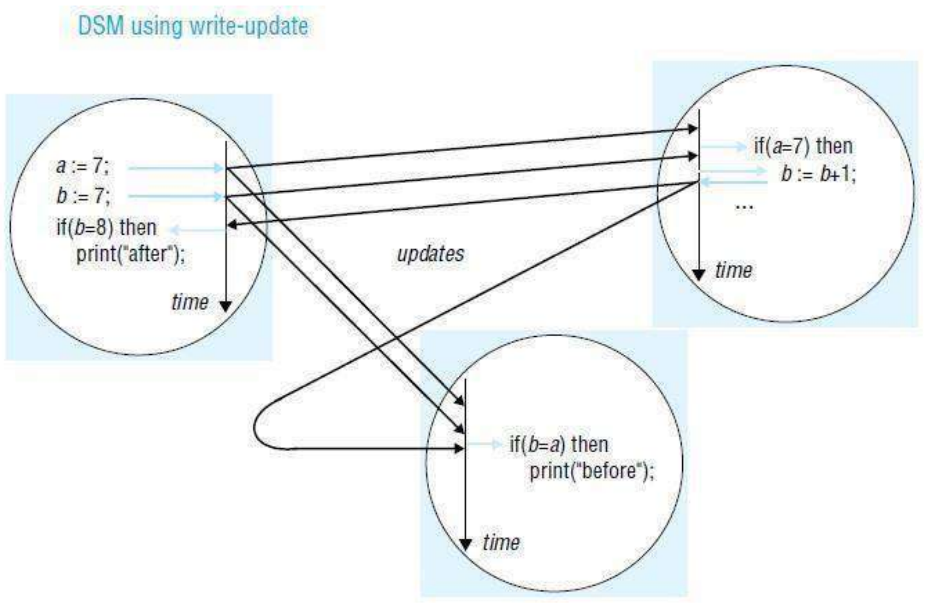
Weak consistency

This model exploits knowledge of synchronization operations in order to relax memory consistency, while appearing to the programmer to implement sequential consistency (at least, under certain conditions that are beyond the scope of this book). For example, if the programmer uses a lock to implement a critical section, then a DSM system can assume that no other process may access the data items accessed under mutual exclusion within it. It is therefore redundant for the DSM system to propagate updates to these items until the process leaves the critical section. While items are left with 'inconsistent' values some of the time, they are not accessed at those points; the execution appears to be sequentially consistent.

Update options

Two main implementation choices have been devised for propagating updates made by one process to the others: write-update and write-invalidate. These are applicable to a variety of DSM consistency models, including sequential consistency. In outline, the options are as follows:

Write-update: The updates made by a process are made locally and multicast to all other replica managers possessing a copy of the data item, which immediately modify the data read by local processes. Processes read the local copies of data items, without the need for communication. In addition to allowing multiple readers, several processes may write the same data item at the same time; this is known as multiple-reader/multiple-writer sharing.



Write-invalidate: This is commonly implemented in the form of multiple-reader/ single-writer sharing. At any time, a data item may either be accessed in read-only mode by one or more processes, or it may be read and written by a single process. An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is first sent to all other copies to invalidate them and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data (that is, data that are not up to date). Any processes attempting to access the data item are blocked if a writer exists.

Granularity

An issue that is related to the structure of DSM is the granularity of sharing. Conceptually, all processes share the entire contents of a DSM. As programs sharing DSM execute, however, only certain parts of the data are actually shared and then only for certain times during the execution. It would clearly be very wasteful for the DSM implementation always to transmit the entire contents of DSM as processes access and update it.

Thrashing

A potential problem with write-invalidate protocols is thrashing. Thrashing is said to occur where the DSM runtime spends an inordinate amount of time invalidating and transferring shared data compared with the time spent by application processes doing useful work. It occurs when several processes compete for the same data item, or for falsely shared data items.

CONSISTENCY MODELS

Models of memory consistency can be divided into *uniform models*, which do not distinguish between types of memory access, and *hybrid models*, which do distinguish between ordinary and synchronization accesses (as well as other types of access).
Other uniform consistency models include:

Causal consistency: Reads and writes may be related by the happened-before relationship . This is defined to hold between memory operations when either (a) they are made by the same process; (b) a process reads a value written by another process; or (c) there exists a sequence of such operations linking the two operations. The model's constraint is that the value returned by a read must be consistent with the happened-before relationship.
Processor consistency: The memory is both coherent and adheres to the pipelined RAM model (see below). The simplest way to think of processor consistency is that the memory is coherent and that all processes agree on the ordering of any two write accesses made by the same process that is, they agree with its program order.

UNIT-V

.

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

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

Introduction

The goal of transactions is to ensure that all of the objects managed by a server remain in a consistent state when they are accessed by multiple transactions and in the presence of server crashes Objects that can be recovered after their server crashes are called recoverable objects. In general, the objects managed by a server may be stored in volatile memory(for example, RAM) or persistent memory (for example, a hard disk). Even if objects are stored in volatile memory, the server may use persistent memory to store sufficient information for the state of the objects to be recovered if the server process crashes. This enables servers to make objects recoverable. A transaction is specified by a client as a set of operations on objects to be performed as an indivisible unit by the servers

managing those objects. The servers must guarantee that either the entire transaction is carried out and the results recorded in permanent storage or, in the case that one or more of them crashes, its effects are completely erased. The next chapter discusses issues related to transactions that involve several servers, in particular how they decide on the outcome of a distributed transaction.

Simple synchronization (without transactions)

One of the main issues of this chapter is that unless a server is carefully designed, its operations performed on behalf of different clients may sometimes interfere with one another. Such interference may result in incorrect values in the objects. In this section, we discuss how client operations may be synchronized without recourse to transactions.

Atomic operations at the server •

multiple threads is beneficial to performance in many servers. We have also noted that the use of threads allows operations from multiple clients to run concurrently and possibly access the same objects. Therefore, the methods of objects should be designed for use in a multi-threaded context.

For example, if the methods deposit and withdraw are not designed for use in a multi-threaded program, then it is possible that theservers managing those objects. The servers must guarantee that either the entire transaction is carried out and the results recorded in permanent storage or, in the case that one or more of them crashes, its effects are completely erased. The next chapter discusses issues related to transactions that involve several servers, in particular how they decide on the outcome of a distributed transaction.

Simple synchronization (without transactions)

One of the main issues of this chapter is that unless a server is carefully designed, its operations performed on behalf of different clients may sometimes interfere with one another. Such interference may result in incorrect values in the objects. In this section, we discuss how client operations may be synchronized without recourse to transactions.

Atomic operations at the server •

multiple threads is beneficial to performance in many servers. We have also noted that

the use of threads allows operations from multiple clients to run concurrently and possibly access the same objects. Therefore, the methods of objects should be designed for use in a multi-threaded context. For example, if the methods deposit and withdraw are not designed for use in a multi- threaded program, then it is possible that the actions of two or more concurrent executions of the method could be interleaved arbitrarily and have strange effects on the instance variables of the account objects.

Figure 16.1 Operations of the Account interface

deposit(amount)

deposit amount in the account withdraw(amount)

withdraw amount from the account getBalance()-> amount

return the balance of the account setBalance(amount)

set the balance of the account to amount

Operations of the Branch interface

create(name)-> account

create a new account with a given name lookUp(name)-> account

return a reference to the account with the given name branchTotal()-> amount

return the total of all the balances at the branch

the synchronized keyword, which can be applied to methods in Java to ensure that only one thread at a time can access an object. In our example, the class that implements the Account interface will be able to declare the

methods as synchronized. For example:

```
  public synchronized void deposit(int amount) throws RemoteException{
// adds amount to the balance of the account
 }
```

If one thread invokes a synchronized method on an object, then that object is effectively locked, and another thread that invokes one of its synchronized methods will be blocked until the lock is released. Thisformof synchronization forces theserversmanaging those objects. The servers must guarantee that either the entire transaction is carried out and the results recorded in permanent storage or, in the case that one or more of them crashes, its effects are completely erased. The next chapter discusses issues

related to transactions that involve several servers, in particular how they decide on the outcome of a distributed transaction.

Simple synchronization (without transactions)

One of the main issues of this chapter is that unless a server is carefully designed, its operations performed on behalf of different clients may sometimes interfere with one another. Such interference may result in incorrect values in the objects. In this section,we discuss how client operations may be synchronized without recourse to transactions.

Atomic operations at the server •

multiple threads is beneficial to performance in many servers. We have also noted that

the use of threads allows operations from multiple clients to run concurrently and possibly access the same objects. Therefore, the methods of objects should be designed for use in a multi-threaded context. For example, if the methods deposit and withdraw are not designed for use in a multi-threaded program, then it is possible that the actions of two or more concurrent executions of the method could be interleaved arbitrarily and have strange effects on the instance variables of the account objects.

Figure 16.1 Operations of the Account interface

deposit(amount)

deposit amount in the account withdraw(amount)

withdraw amount from the account getBalance()-> amount

return the balance of the account setBalance(amount)

set the balance of the account to amount **Operations of the Branch interface** create(name)-> account

create a new account with a given name lookUp(name)-> account

return a reference to the account with the given name branchTotal()-> amount

return the total of all the balances at the branch

the synchronized keyword, which can be applied to methods in Java to ensure that only one thread at a time can access an object. In our example, the class that implements the Account interface will be able to declare the methods as synchronized. For example:

```
public synchronized void deposit(int amount) throws RemoteException{
    // adds amount to the balance of the account
     }
```

If one thread invokes a synchronized method on an object, then that object is effectively locked, and another thread that invokes one of its synchronized methods will be blocked until the lock is released. This form of synchronization forces the execution of threads to be separated in time and ensures that the instance variables of a single object are accessed in a consistent manner. Without synchronization, two separate deposit invocations might read the balance before either has incremented it – resulting in an incorrect value. Any method that accesses an instance variable that can vary should be synchronized.

Operations that are free from interference from concurrent operations being performed in other threads are called atomic operations. The use of synchronized methods in Java is one way of achieving atomic operations. But in other programming environments for multi-threaded servers the operations on objects still need to have atomic operations in order to keep their objects consistent. This may be achieved by the use of any available mutual exclusion mechanism, such as a mutex.Enhancing client cooperation by synchronization of server operations •

Clients may use a server as a means of sharing some resources. This is achieved by some clients using operations to update the server's objects and other clients using operations to access them. The above scheme for synchronized access to objects provides all that is required in many applications – it prevents threads interfering with one another. However, some applications require a way for threads to communicate with each other.

For example, a situation may arise in which the operation requested by one client cannot be completed until an operation requested by another client has been performed. This can happen when some clients are producers and others are consumers – the consumers may have to wait until a producer has supplied some more of the commodity
in question. It can also occur when clients are sharing a resource – clients needing the resource may have to wait for other clients to release it. The Java wait and notify methods allow threads to communicate with one another in a manner that solves the above problems. They must be used within synchronized methods of an object. A thread calls wait on an object so as to suspend itself and to allow another thread to execute a method of that object. A thread calls notify to inform any thread waiting on that object that it has changed some of its data. Access to an object is still atomic when threads wait for one another: a thread that calls wait gives up its lock and suspends itself as a single atomic action; when a thread is restarted after being notified it acquires a new lock on the object and resumes execution from after its wait. A thread that calls notify (from within a synchronized method) completes the execution of that method before releasing the lock on the object. Consider the implementation of a shared Queue object with two methods: first removes and returns the first object in the queue, and append adds a given object to the end of the queue. The method first will test whether the queue is empty, in which case it will call wait on the queue. If a client invokes first when the queue is empty, it will not get a reply until another client has added something to the queue – the append operation will call notify when it has added an object to the queue. This allows one of the threads waiting on the queue object to resume and to return the first object in the queue to it

client. When threads can synchronize their actions on an object by means of wait and notify, the server holds onto requests that cannot immediately be satisfied and the client waits for a reply until another client has produced whatever it needs.

Failure model for transactions Lampson [1981] proposed a fault model for distributed transactions that accounts for failures of disks, servers and communication. In this model, the claim is that the algorithms work correctly in the presence of predictable faults, but no claims are made about their behaviour when a disaster occurs. Although errors may occur, they can be detected and dealt with before any incorrect behaviour results. The model states the following:

• Writes to permanent storage may fail, either by writing nothing or by writing a wrong value – for example, writing to the wrong block is a disaster. File storage may also decay. Reads from permanent storage can detect (by a checksum) when a actions of two or more concurrent executions of the method could be interleaved arbitrarily and have strange effects on the instance variables of the account objects.

Figure 16.1 Operations of the Account interface

deposit(amount)

deposit amount in the account withdraw(amount)

withdraw amount from the account getBalance()-> amount

return the balance of the account setBalance(amount)

set the balance of the account to amount **Operations of the Branch interface** create(name)-> account

create a new account with a given name lookUp(name)-> account

return a reference to the account with the given name branchTotal()-> amount

return the total of all the balances at the branch the synchronized keyword, which can be applied to methods in Java to ensure that only one thread at a time can access an object. In our example, the class that implements the Account interface will be able to declare the methods as synchronized.

For example:

```
  public synchronized void deposit(int amount) throws RemoteException{
  // adds amount to the balance of the account
 }
```

If one thread invokes a synchronized method on an object, then that object is effectively locked, and another thread that invokes one of its synchronized methods will be blocked until the lock is released. This form of synchronization forces the execution of threads to be separated in time and ensures that the instance variables of a single object areaccessed in a consistent manner. Without synchronization, two separate deposit invocations might read the balance before either has incremented it – resulting in an incorrect value. Any method that accesses an instance variable that can vary should be synchronized. Operations that are free from interference from concurrent operations being performed in other threads are called atomic operations. The use of synchronized methods in Java is one way of achieving atomic operations. But in other programming environments for multi-threaded servers the operations on objects still need to have atomic operations in order to keep their objects consistent. This may be achieved by the use of any available mutual exclusion mechanism, such as a mutex.Enhancing client cooperation by synchronization of server operations.

•Clients may use a server as a means of sharing some resources. This is achieved by some clients using operations to update the server's objects and other clients using operations to access them. The above scheme for synchronized access to objects provides all that is required in many applications – it prevents threads interfering with one another. However, some applications require a way for threads to communicate with each other.

For example, a situation may arise in which the operation requested by one client cannot be completed until an operation requested by another client has been performed. This can happen when some clients are producers and others are consumers – the consumers may have to wait until a producer has supplied some more of the commodity

in question. It can also occur when clients are sharing a resource – clients needing the resource may have to wait for other clients to release it. The Java wait and notify methods allow threads to communicate with one another in a manner that solves the above problems. They must be used within synchronized methods of an object. A thread calls wait on an object so as to suspend itself and to allow another thread to execute a method of that object. A thread calls notify to inform any thread waiting on that object that it has changed some of its data. Access to an object is still atomic when threads wait for one another: a thread that calls wait gives up its lock and suspends itself as a single atomic action; when a thread is restarted after being notified it acquires a new lock on the object and resumes execution from after its wait. A thread that calls notify (from within a synchronized method) completes the execution of that method before releasing the lock on the object. Consider the implementation of a shared Queue object with two methods: first removes and returns the first object in the queue, and append adds a given object to the end of the queue. The method first will test whether the queue is empty, in which case it will call wait on the queue. If a client invokes first when the queue is empty, it will not get a reply until another client has added something to the queue – the append operation will call notify when it has added an object to the queue. This allows one of the threads waiting on the queue object to resume and to return the first object in the queue to its client. When threads can synchronize their actions on an object by means of wait and notify, the server holds onto requests that cannot immediately be satisfied and the client waits for a reply until another client has produced whatever it needs.

Failure model for transactions Lampson [1981] proposed a fault model for distributed transactions that accounts for failures of disks, servers and communication. In this model, the claim is that the algorithms work correctly in the presence of predictable faults, but no claims are made about their behaviour when a disaster occurs. Although errors may occur, they can be detected and dealt with before any incorrect behaviour results. The model states the following:

•Writes to permanent storage may fail, either by writing nothing or by writing a wrong value – for example, writing to the wrong block is a disaster. File storage may also decay. Reads from permanent storage can detect (by a checksum) when a block of data is bad. • Servers may crash occasionally. When a crashed server is replaced by a new process, its volatile memory is first set to a state in which it knows none of the values (for example, of objects) from before the crash. After that it carries out a recovery procedure using information in permanent storage and obtained from other processes to set the values of objects including those related to the two-phase commit protocol When a processor is faulty, it is made to crash so that it is prevented from sending erroneous messages and from writing wrong values to permanent storage – that is, so it cannot produce arbitrary failures.

Crashes can occur at any time; in particular, they may occur during recovery. • There may be an arbitrary delay before a message arrives. A message may be lost, duplicated or corrupted. The recipient can detect corrupted messages using a checksum. Both forged messages and undetected corrupt messages are regarded as disasters.The fault model for permanent storage, processors and communications was used to design a stable system whose components can survive any single fault and present a simple failure model. In particular, stable storage provided an atomic write operation inthe presence of a single fault of the write operation or a crash failure of the process. This was achieved by replicating each block on two disk blocks. A write operation wasapplied to the pair of disk blocks, and in the case of a single fault, one good block wasalways available. A stable processor used stable storage to enable it to recover itsobjects after a crash.

Communication errors were masked by using a reliable remote procedure calling mechanism.

Transactions

In some situations, clients require a sequence of separate requests to a server to be atomic in the sense that:

1. They are free from interference by operations being performed on behalf of other concurrent clients.
2. Either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

client's banking transaction

Transaction T:

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

We return to our banking example to illustrate transactions. A client that performs a sequence of operations on a particular bank account on behalf of a user will first lookup the account by name and then apply the deposit, withdraw and getBalance operations directly to the relevant account. In our examples, we use accounts with names A, B and C. The client looks them up and stores references to them in variables a, b and c of type Account. The details of looking up the accounts by name and the declarations of the variables are omitted from the examples.example of a simple client transaction specifying a series of related actions involving the bank accounts A, B and C. The first two actions transfer $100 from A to B and the second two transfer $200 from C to B. A client achieves atransfer operation by doing a withdrawal followed by a deposit. In all of these contexts, a transaction applies to recoverable objects and is intended to be atomic. It is often called an atomic transaction. There are two aspects to atomicity:All or nothing: A transaction either completes successfully, in which case the effects of all of its operations are recorded in the objects, or (if it fails or is deliberately aborted) has no effect at all. This all-or-nothing effect has two further aspects of its own:

Failure atomicity: The effects are atomic even when the server crashes Durability: After a transaction has completed successfully, all its effects are saved in permanent storage. We use the term 'permanent storage' to refer to files held on disk or another permanent medium. Data saved in a file will survive if the server process crashes.

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

To support the requirement for failure atomicity and durability, the objects must be *recoverable*; that is, when a server process crashes unexpectedly due to a hardware fault or a software error, the changes due to all completed transactions must be available in permanent storage so that when the server is replaced by a new process, it can recover the objects to reflect the all-or-nothing effect. By the time a server acknowledges the completion of a client's transaction, all of the transaction's changes to the objects must have been recorded in permanent storage.

server that supports transactions must synchronize the operations sufficiently to ensure that the isolation requirement is met. One way of doing this is to perform the transactions serially – one at a time, in some arbitrary order. Unfortunately, this solution would generally be unacceptable for servers whose resources are shared by multiple interactive users. For instance, in our banking example it is desirable to allow several bank clerks to perform online banking transactions at the same time as one another.

The aim for any server that supports transactions is to maximize concurrency. Therefore transactions are allowed to execute concurrently if this would have the same effect as a serial execution – that is, if they are *serially equivalent* or *serializable*.

Operations in the *Coordinator* interface

openTransaction() o trans;

Starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans)o (commit, abort);

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

abortTransaction(trans);

Aborts the transaction.

Transaction capabilities can be added to servers of recoverable objects. Each transaction is created and managed by a coordinator, which implements the *Coordinator* interface shown in Figure 16.3. The coordinator gives each transaction an identifier, or TID. The client invokes the *openTransaction* method of the coordinator to introduce a new transaction – a transaction identifier or TID is allocated and returned. At the end of a transaction, the client invokes the *closeTransaction* method to indicate its end – all of the recoverable objects accessed by the transaction should be saved. If, for some reason, the client wants to abort a transaction, it invokes the *abortTransaction* method – all of its effects should be removed from sight. transaction is achieved by cooperation between a client program, some recoverable objects and a coordinator. The client specifies the sequence of invocations on recoverable objects that are to comprise a transaction. To achieve this, the client sends with each invocation the transaction identifier returned by *openTransaction*. One way to make this possible is to include an extra argument in each operation of a recoverable object to carry the TID. For example, in the banking service the *deposit* operation might be defined:

deposit(trans, amount)

Deposits *amount* in the account for transaction with TID *trans*

When transactions are provided as middleware, the TID can be passed implicitly with all remote invocations between *openTransaction* and *closeTransaction* or *abortTransaction*. This is what the CORBA Transaction Service does. We shall not show TIDs in our examples.

Normally, a transaction completes when the client makes a *closeTransaction* request. If the transaction has progressed normally, the reply states that the transaction is *committed* – this constitutes a promise to the client that all of the changes requested in the transaction are permanently recorded and that any future transactions that access the same data will see the results of all of the changes made during the transaction.

Alternatively, the transaction may have to *abort* for one of several reasons related to the nature of the transaction itself, to conflicts with another transaction or to the crashing of a process or computer.

When a transaction is aborted the parties involved (the recoverable objects and the coordinator) must ensure that none of its effects are visible to future transactions, either in the objects or in their copies in permanent storage.

Figure 16.4
Transaction life histories

Successful	*Aborted by client*		*Aborted by server*
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•	server aborts	•
•	•	*transaction o*	•
			operation ERROR
operation	*operation*		
			reported to clien
	abortTransactio		

closeTransaction *n*

shows these three alternative life histories for transactions. We refer to a transaction as *failing* in both of the latter cases.

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

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

Concurrency control

This section illustrates two well-known problems of concurrent transactions in the context of the banking example – the 'lost update' problem and the 'inconsistent retrievals' problem. We then show how both of these problems can be avoided by using serially equivalent executions of transactions. We assume throughout that each of the operations *deposit*, *withdraw*, *getBalance* and *setBalance* is a synchronized operation – that is, that its effects on the instance variable that records the balance of an account are atomic.

The lost update problem • The lost update problem is illustrated by the following pair of transactions on bank accounts *A*, *B* and *C*, whose initial balances are $100, $200 and $300, respectively. Transaction *T* transfers an amount from account *A* to account *B*. Transaction *U* transfers an amount from account *C* to account *B*. In both cases, the amount transferred is calculated to increase the balance of *B* by 10%. The net effects on account *B* of executing the transactions *T* and *U* should be to increase the balance of account *B* by 10% twice, so its final value is $242.

Now consider the effects of allowing the transactions *T* and *U* to run concurrently, as in

Figure 16.5. Both transactions get the balance of *B* as $200 and then deposit $20. The result is incorrect, increasing the balance of account *B* by $20 instead of $42. This is an illustration of the 'lost update' problem. *U*'s update is lost because *T* overwrites it without seeing it. Both transactions have read the old value before either writes the new value.

In Figure onwards, we show the operations that affect the balance of an account on successive lines down the page, and the reader should assume that an operation on a particular line is executed at a later time than the one on the line above it.

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance();		*balance = b.getBalance();*	
*b.setBalance(balance*1.1);*		*b.setBalance(balance*1.1);*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
		c.withdraw(balance/10)	$280
a.withdraw(balance/10)	$80		

amount transferred is calculated to increase the balance of *B* by 10%. The net effects on account *B* of executing the transactions *T* and *U* should be to increase the balance of account *B* by 10% twice, so its final value is $242.

Now consider the effects of allowing the transactions *T* and *U* to run concurrently, as in Figure 16.5. Both transactions get the balance of *B* as $200 and then deposit $20. The result is incorrect, increasing the balance of account *B* by $20 instead of $42. This is an illustration of the 'lost update' problem. *U*'s update is lost because *T* overwrites it without seeing it. Both transactions have read the old value before either writes the new value.

In Figure 16.5 onwards, we show the operations that affect the balance of an account on successive lines down the page, and the reader should assume that an operation on a particular line is executed at a later time than the one on the line above it.

Inconsistent retrievals • Figure 16.6 shows another example related to a bank account in which transaction *V* transfers a sum from account *A* to *B* and transaction *W* invokes the *branchTotal* method to obtain the sum of the balances of all the accounts in the bank.

The inconsistent retrievals problem

Transaction *V*:		**Transaction *W*:**	

a.withdraw(100)
 b.deposit(100)
		aBranch.branchTotal()	

a.withdraw(100);	$100		
		total = a.getBalance( )	$100
		total = total + b.getBalance()	$300
		total = total + c.getBalance()	
		b.deposit(100)	$300

•
•

A serially equivalent interleaving of *T* and *U*

Transaction *T*:		**Transaction *U*:**	

balance = b.getBalance()
 *b.setBalance(balance*1.1)*
 a.withdraw(balance/10)
		balance = b.getBalance()
		 *b.setBalance(balance*1.1)*
		 c.withdraw(balance/10)	

balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
		c.withdraw(balance/10)	$278
a.withdraw(balance/10)	$80		

The balances of the two bank accounts, *A* and *B*, are both initially $200. The result of *branchTotal* includes the sum of *A* and *B* as $300, which is wrong. This is an illustration of the 'inconsistent retrievals' problem. *W*'s retrievals are inconsistent because *V* has performed only the withdrawal part of a transfer at the time the sum is calculated.

Serial equivalence • If each of several transactions is known to have the correct effect when it is done on its own, then we can infer that if these transactions are done one at a time in some order the combined effect will also be correct. An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order is a *serially equivalent* interleaving. When we say that two different transactions have the *same effect* as one another, we mean that the *read* operations return the same values and that the instance variables of the objects have the same values at the end.

The use of serial equivalence as a criterion for correct concurrent execution prevents the occurrence of lost updates and inconsistent retrievals.

The lost update problem occurs when two transactions read the old value of a variable and then use it to calculate the new value. This cannot happen if one transaction is performed before the other, because the later transaction will read the value written by the earlier one. As a serially equivalent interleaving of two transactions produces the same effect as a serial one, we can solve the lost update problem by means of serial equivalence. Figure 16.7 shows one such interleaving in which the operations that affect the shared account, *B*, are actually serial, for transaction *T* does all its operations on *B* before transaction *U* does. Another interleaving of *T* and *U* that has this property is one in which transaction *U* completes its operations on account *B* before transaction *T* starts.

We now consider the effect of serial equivalence in relation to the inconsistent retrievals problem, in which transaction *V* is transferring a sum from account *A* to *B* and transaction *W* is obtaining the sum of all the balances (see Figure 16.6). The inconsistent retrievals problem can occur when a retrieval transaction runs concurrently with an update transaction. It cannot occur if the retrieval transaction is performed before or after the update transaction. A serially equivalent interleaving of a retrieval transaction and an update transaction, for example as in Figure 16.8, will prevent inconsistent retrievals occurring.

A serially equivalent interleaving of *V* and *W*

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal( )*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance( )	$100
		total = total + b.getBalance()	$400
		total = total + c.getBalance()	
		...	

Conflicting operations • When we say that a pair of operations *conflicts* we mean that their combined effect depends on the order in which they are executed. To simplify matters we consider a pair of operations, *read* and *write*. *read* accesses the value of an object and *write* changes its value. The *effect* of an operation refers to the value of an object set by a *write* operation and the result returned by a *read* operation. The conflict rules for *read* and *write* operations are given in Figure 16.9.

For any pair of transactions, it is possible to determine the order of pairs of conflicting operations on objects accessed by both of them. Serial equivalence can be defined in terms of operation conflicts as follows:

For two transactions to be *serially equivalent,* it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

Figure 16.9 *Read* and *write* operation conflict rules

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

Figure 16.10 A non–serially-equivalent interleaving of operations of transactions *T* and *U*

Transaction *T*:	**Transaction *U*:**

x = read(i)
write(i, 10)
y = read(j)
write(j, 30)
write(j, 20)
z = read (i)

Consider as an example the transactions *T* and *U*, defined as follows:

T: x = *read(i)*; *write(i, 10)*; *write(j, 20)*;

U: y = *read(j)*; *write(j, 30)*; *z = read (i)*;

Then consider the interleaving of their executions, shown in Figure 16.10. Note that each transaction's access to objects *i* and *j* is serialized with respect to one another, because *T* makes all of its accesses to *i* before *U* does and *U* makes all of its accesses to *j* before *T* does. But the ordering is not serially equivalent, because the pairs of conflicting operations are not done in the same order at both objects. Serially equivalent orderings require one of the following two conditions:

T accesses *i* before *U* and *T* accesses *j* before *U*. accesses *i* before *T* and *U* accesses *j* before *T*.

Serial equivalence is used as a criterion for the derivation of concurrency control protocols. These protocols attempt to serialize transactions in their access to objects. Three alternative approaches to concurrency control are commonly used: locking, optimistic concurrency control and timestamp ordering. However, most practical systems use locking, which is discussed in

Section 16.4. When locking is used, the server sets a lock, labelled with the transaction identifier, on each object just before it is accessed and removes these locks when the transaction has completed. While an object is locked, only the transaction that it is locked for can access that object; other transactions must either wait until the object is unlocked or, in some cases, share the lock. The use of locks can lead to deadlocks, with transactions waiting for each other to release locks – for example, when a pair of transactions each has an object locked that the other needs to access. We discuss the deadlock problem and some remedies for it in Section 16.4.1.

Optimistic concurrency control is described in Section 16.5. In optimistic schemes, a transaction proceeds until it asks to commit, and before it is allowed to commit the server performs a check to discover whether it has performed operations on any objects that conflict with the operations of other concurrent transactions, in which case the server aborts it and the client may restart it. The aim of the check is to ensure that all the objects are correct.

Timestamp ordering is described in Section 16.6. In timestamp ordering, a server records the most recent time of reading and writing of each object and for each

Figure 16.11 A dirty read when transaction *T* aborts

Transaction *T*: *a.getBalance()* *a.setBalance(balance + 10)*		**Transaction *U*:** *a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

operation, the timestamp of the transaction is compared with that of the object to determine whether it can be done immediately or must be delayed or rejected. When an operation is delayed, the transaction waits; when it is rejected, the transaction is aborted.

Basically, concurrency control can be achieved either by clients' transactions waiting for one another or by restarting transactions after conflicts between operations have been detected, or by a combination of the two.

Recoverability from aborts

Servers must record all the effects of committed transactions and none of the effects of aborted transactions.They must therefore allow for the fact that a transaction may abort by preventing it affecting other concurrent transactions if it does so.

This section illustrates two problems associated with aborting transactions in the context of the banking example. These problems are called 'dirty reads' and 'premature writes', and both of them can occur in the presence of serially equivalent executions of transactions. These issues are concerned with the effects of operations on objects such as the balance of a bank account. To simplify things, operations are considered in two categories: *read* operations and *write* operations. In our illustrations, *getBalance* is a *read* operation and *setBalance* a *write* operation.

Dirty reads •

The isolation property of transactions requires that transactions do not see the uncommitted state of other transactions. The 'dirty read' problem is caused by the interaction between a *read* operation in one transaction and an earlier *write* operation in another transaction on the same object. Consider the executions illustrated in Figure 16.11, in which *T* gets the balance of account *A* and sets it to $10 more, then *U* gets the balance of account *A* and sets it to $20 more, and the two executions are serially equivalent. Now suppose that the transaction *T* aborts after *U* has committed. Then the transaction *U* will have seen a value that never existed, since *A* will be restored to its original value. We say that the transaction *U* has performed a *dirty read*. As it has committed, it cannot be undone.

Figure 16.12 Overwriting uncommitted values

Transaction *T*:		**Transaction *U*:**
a.setBalance(105)		*a.setBalance(110)*
	$100	
a.setBalance(105)	$105	
		a.setBalance(110)

Recoverability of transactions • If a transaction (like *U*) has committed after it has seen the effects of a transaction that subsequently aborted, the situation is not recoverable. To ensure that such situations will not arise, any transaction (like *U*) that is in danger of having a dirty read delays its commit operation. The strategy for recoverability is to delay commits until after the commitment of any other transaction whose uncommitted state has been observed. In our example, *U* delays its commit until after *T* commits. In the case that *T* aborts, then *U* must abort as well.

Cascading aborts • In Figure 16.11, suppose that transaction *U* delays committing until after *T* aborts. As we have said, *U* must abort as well. Unfortunately, if any other transactions have seen the effects due to *U*, they too must be aborted. The aborting of these latter transactions may cause still further transactions to be aborted. Such situations are called *cascading aborts*. To avoid cascading aborts, transactions are only allowed to read objects that were written by committed transactions. To ensure that this is the case, any *read* operation must be delayed until other transactions that applied a *write* operation to the same object have committed or aborted. The avoidance of cascading aborts is a stronger condition than recoverability.

Premature writes • Consider another implication of the possibility that a transaction may abort. This one is related to the interaction between *write* operations on the same object belonging to different transactions. For an illustration, we consider two *setBalance* transactions, *T* and *U*, on account *A*, as shown in Figure 16.12. Before the transactions, the balance of account A was $100. The two executions are serially equivalent, with *T* setting the balance to $105 and *U* setting it to $110. If the transaction *U* aborts and *T* commits, the balance should be $105.

Some database systems implement the action of *abort* by restoring 'before images' of all the *writes* of a transaction. In our example, *A* is $100 initially, which is the 'before image' of *T*'s *write*; similarly, $105 is the 'before image' of *U*'s *write*. Thus if *U* aborts, we get the correct balance of $105.

Now consider the case when *U* commits and then *T* aborts. The balance should be $110, but as the 'before image' of *T*'s *write* is $100, we get the wrong balance of $100. Similarly, if *T* aborts and then *U* aborts, the 'before image' of *U*'s *write* is $105 and we get the wrong balance of $105 – the balance should revert to $100.

To ensure correct results in a recovery scheme that uses before images, *write* operations must be delayed until earlier transactions that updated the same objects have either committed or aborted.

Strict executions of transactions • Generally, it is required that transactions delay both their *read* and *write* operations so as to avoid both dirty reads and premature writes. The executions of transactions are called *strict* if the service delays both *read* and *write* operations on an object until all transactions that previously wrote that object have either committed or aborted. The strict execution of transactions enforces the desired property of isolation.

Tentative versions • For a server of recoverable objects to participate in transactions, it must be designed so that any updates of objects can be removed if and when a transaction aborts. To make this possible, all of the update operations performed during a transaction are done in tentative versions of objects in volatile memory. Each transaction is provided with its own private set of tentative versions of any objects that it has altered. All the update operations of a transaction store values in the transaction's own private set. Access operations in a transaction take values from the transaction's own private set if possible, or failing that, from the objects.The tentative versions are transferred to the objects only when a transaction commits, by which time they will also have been recorded in permanent storage. This is performed in a single step, during which other transactions are excluded from access to the objects that are being altered. When a transaction aborts, its tentative versions are deleted.

Nested transactions

Nested transactions extend the above transaction model by allowing transactions to be composed of other transactions. Thus several transactions may be started from within a transaction, allowing transactions to be regarded as modules that can be composed as required.

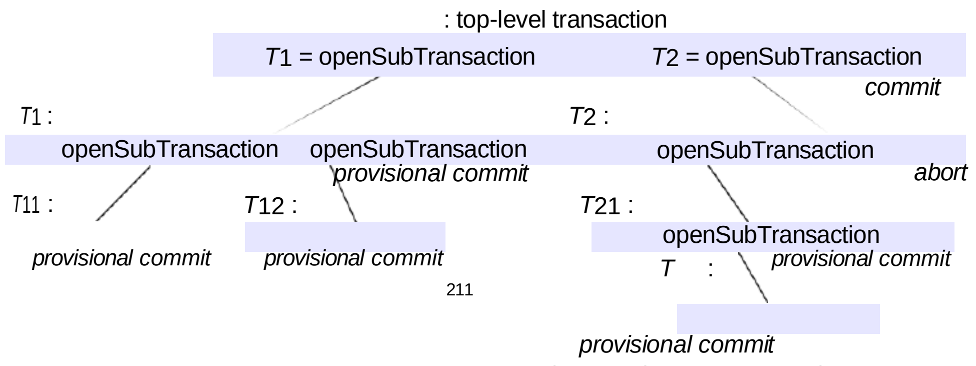
The outermost transaction in a set of nested transactions is called the *top-level* transaction.

Transactions other than the top-level transaction are called *subtransactions*. For example, in Figure 16.13, *T* is a top-level transaction that starts a pair of subtransactions, T_1 and T_2. The subtransaction T_1 starts its own pair of subtransactions, T_{11} and T_{22}. Also, subtransaction T_2 starts its own subtransaction, T_{21}, which starts another subtransaction, T_{211}.

A subtransaction appears atomic to its parent with respect to transaction failures and to concurrent access. Subtransactions at the same level, such as T_1 and T_2, can run concurrently, but their access to common objects is serialized – for example, by the locking scheme described in Section 16.4. Each subtransaction can fail independently of its parent and of the other subtransactions. When a subtransaction aborts, the parent transaction can sometimes choose an alternative subtransaction to complete its task. For example, a transaction to deliver a mail message to a list of recipients could be structured as a set of subtransactions, each of which delivers the message to one of the recipients. If one or more of the subtransactions fails, the parent transaction could record the fact and then commit, with the result that all the successful child transactions commit. It could then start another transaction to attempt to redeliver the messages that were not sent the first time.

\

Figure 16.13 Nested transactions



When we need to distinguish our original form of transaction from nested ones, we use the term *flat* transaction. It is flat because all of its work is done at the same level between an *openTransaction* and a *commit* or *abort*, and it is not possible to commit or abort parts of it. Nested transactions have the following main advantages:

Subtransactions at one level (and their descendants) may run concurrently with other subtransactions at the same level in the hierarchy. This can allow additional concurrency in a transaction. When subtransactions run in different servers, they can work in parallel. For example, consider the *branchTotal* operation in our banking example. It can be implemented by invoking *getBalance* at every account in the branch. Now each of these invocations may be performed as a subtransaction, in which case they can be performed concurrently. Since each one applies to a different account, there will be no conflicting operations among the subtransactions.

Subtransactions can commit or abort independently. In comparison with a single transaction, a set of nested subtransactions is potentially more robust. The above example of delivering mail shows that this is so – with a flat transaction, one transaction failure would cause the whole transaction to be restarted. In fact, a parent can decide on different actions according to whether a subtransaction has aborted or not.

The rules for committing of nested transactions are rather subtle:

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

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

When a parent aborts, all of its subtransactions are aborted. For example, if T_2 aborts then T_{21} and T_{211} must also abort, even though they may have provisionally committed.

When a subtransaction aborts, the parent can decide whether to abort or not. In our example, T decides to commit although T_2 has aborted.

If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted. In our example, T's commitment allows T_1, T_{11} and T_{12} to commit, but not T_{21} and T_{211} since their parent, T_2, aborted. Note that the effects of a subtransaction are not permanent until the top-level transaction commits.

In some cases, the top-level transaction may decide to abort because one or more of its subtransactions have aborted. As an example, consider the following *Transfer* transaction:

Transfer $100 from *B* to *A* *a.deposit*(100) *b.withdraw*(100)

This can be structured as a pair of subtransactions, one for the *withdraw* operation and the other for *deposit*. When the two subtransactions both commit, the *Transfer* transaction can also commit. Suppose that a *withdraw* subtransaction aborts whenever an account is overdrawn. Now consider the case when the *withdraw* subtransaction aborts and the *deposit* subtransaction commits – and recall that the commitment of a child transaction is conditional on the parent transaction committing. We presume that the top-level (*Transfer*) transaction will decide to abort. The aborting of the parent transaction causes the subtransactions to abort – so the *deposit* transaction is aborted and all its effects are undone.

Locks

Transactions must be scheduled so that their effect on shared data is serially equivalent. A server can achieve serial equivalence of transactions by serializing access to the objects. Figure 16.7 shows an example of how serial equivalence can be achieved with some degree of concurrency – transactions *T* and *U* both access account *B*, but *T* completes its access before *U* starts accessing it.

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

Figure 16.14 illustrates the use of exclusive locks. It shows the same transactions as Figure 16.7, but with an extra column for each transaction showing the locking, waiting and unlocking. In this example, it is assumed that when transactions *T* and *U* start, the balances of the accounts *A*, *B* and *C* are not yet locked. When transaction *T* is about to use account *B*, it is locked for *T*. When transaction *U* is about to use *B* it is still

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

Transaction *T*:
balance = b.getBalance()
*b.setBalance(bal*1.1)*
a.withdraw(bal/10)
		Transaction *U*:
		balance = b.getBalance()
		*b.setBalance(bal*1.1)*
		c.withdraw(bal/10)	

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

locked for *T*, so transaction *U* waits. When transaction *T* is committed, *B* is unlocked, where upon transaction *U* is resumed. The use of the lock on *B* effectively serializes the access to *B*. Note that if, for example, *T* released the lock on *B* between its *getBalance* and *setBalance* operations, transaction *U*'s *getBalance* operation on *B* could be interleaved between them.

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

We saw that because transactions may abort, strict executions are needed to prevent dirty reads and premature writes. Under a strict execution regime, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted. To enforce this rule, any locks applied during the progress of a transaction are held until the transaction commits or aborts. This is called *strict two-phase locking*. The presence of the locks prevents other transactions reading or writing the objects. When a transaction commits, to ensure recoverability, the locks must be held until all the objects it updated have been written to permanent storageserver generally contains a large number of objects, and a typical transaction accesses only a few of them and is unlikely to clash with other current transactions. The *granularity* with which concurrency control can be applied to objects is an important issue, since the scope for concurrent access to objects in a server will be limited severely if concurrency control (for example, locks) can only be applied to all the objects at once. In our banking example, if locks were applied to all customer accounts at a branch, only one bank clerk could perform an online banking transaction at any time – hardly an acceptable constraint!

The portion of the objects to which access must be serialized should be as small as possible; that is, just that part involved in each operation requested by transactions. In our banking example, a branch holds a set of accounts, each of which has a balance. Each banking operation affects one or more account balances – *deposit* and *withdraw* affect one account balance, and *branchTotal* affects all of them.

The description of concurrency control schemes given below does not assume any particular granularity. We discuss concurrency control protocols that are applicable to objects whose operations can be modelled in terms of *read* and *write* operations on the objects. For the protocols to work correctly, it is essential that each *read* and *write* operation is atomic in its effects on objects.

Concurrency control protocols are designed to cope with *conflicts* between operations in different transactions on the same object. In this chapter, we use the notion of conflict between operations to explain the protocols. The conflict rules for *read* and *write* operations are given in Figure 16.9, which shows that pairs of *read* operations from different transactions on the same object do not conflict. Therefore, a simple exclusive lock that is used for both *read* and *write* operations reduces concurrency more than is necessary.

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

As pairs of *read* operations from different transactions do not conflict, an attempt to set a read lock on an object with a read lock is always successful. All the transactions reading the same object share its read lock – for this reason, read locks are sometimes called *shared locks*.

The operation conflict rules tell us that:

If a transaction *T* has already performed a *read* operation on a particular object, then a concurrent transaction

U must not *write* that object until *T* commits or aborts.

If a transaction *T* has already performed a *write* operation on a particular object, then a concurrent transaction

U must not *read* or *write* that object until *T* commits or aborts.

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

Figure 16.15
Lock compatibility

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Figure 16.15 shows the compatibility of read locks and write locks on any particular object. The entries to the left of the first column in the table show the type of lock already set, if any. The entries above the first row show the type of lock requested. The entry in each cell shows the effect on a transaction that requests the type of lock given above when the object has been locked in another transaction with the type of lock on the left.

Inconsistent retrievals and lost updates are caused by conflicts between *read* operations in one transaction and *write* operations in another without the protection of a concurrency control scheme such as locking. Inconsistent retrievals are prevented by performing the retrieval transaction before or after the update transaction. If the retrieval transaction comes first, its read locks delay the update transaction. If it comes second, its request for read locks causes it to be delayed until the update transaction has completed.

Lost updates occur when two transactions read a value of an object and then use it to calculate a new value. Lost updates are prevented by making later transactions delay their reads until the earlier ones have completed. This is achieved by each transaction setting a read lock when it reads an object and then *promoting* it to a write lock when it writes the same object – when a subsequent transaction requires a read lock it will be delayed until any current transaction has completed.

A transaction with a read lock that is shared with other transactions cannot promote its read lock to a write lock, because the latter would conflict with the read locks held by the other transactions. Therefore, such a transaction must request a write lock and wait for the other read locks to be released.

Lock promotion refers to the conversion of a lock to a stronger lock – that is, a lock that is more exclusive. The lock compatibility table in Figure 16.15 shows the relative exclusivity of locks. The read lock allows other read locks, whereas the write lock does not. Neither allows other write locks. Therefore, a write lock is more exclusive than a read lock. Locks may be promoted because the result is a more exclusive lock. It is not safe to demote a lock held by a transaction before it commits, because the result will be more permissive than the previous one and may allow executions by other transactions that are inconsistent with serial equivalence.

The rules for the use of locks in a strict two-phase locking implementation are summarized in Figure 16.16. To ensure that these rules are adhered to, the client has no access to operations for locking or unlocking items of data. Locking is performed when the requests for *read* and *write* operations are about to be applied to the recoverable objects, and unlocking is performed by the *commit* or *abort* operations of the transaction coordinator.

For example, the CORBA Concurrency Control Service [OMG 2000b] can be used to apply concurrency control on behalf of transactions or to protect objects without using transactions. It provides a means of associating a collection of locks (called a *lockset*) with a resource such as a recoverable object. A lockset allows locks to be acquired or released. A lockset's *lock* method will acquire a lock or block until the lock is free; other methods allow locks to be promoted or released. Transactional locksets support the same methods as locksets, but their methods require transaction identifiers as arguments. We mentioned earlier that the CORBA transaction service tags all client requests in a transaction with the transaction identifier. This enables a suitable lock to be acquired before each of the recoverable objects is accessed during a transaction. The transaction coordinator is responsible for releasing the locks when a transaction commits or aborts.

The rules given in Figure 16.16 ensure strictness, because the locks are held until a transaction has either committed or aborted. However, it is not necessary to hold read locks to ensure strictness. Read locks need only be held until the request to commit or abort arrives.

Lock implementation • The granting of locks will be implemented by a separate object in the server that we call the *lock manager*. The lock manager holds a set of locks, for example in a hash table. Each lock is an instance of the class *Lock* and is associated with a particular object. The class *Lock* is shown in Figure 16.17. Each instance of *Lock* maintains the following information in its instance variables:the identifier of the locked object;

the transaction identifiers of the transactions that currently hold the lock (shared locks can have several holders);

a lock type.

Figure 16.17 Lock class

```
public class Lock {
    private Object object;     // the object being protected by the lock
    private Vector holders; // the TIDs of current holders private LockType
    lockType; // the current type

    public synchronized void acquire(TransID trans, LockType aLockType ){ while(/*another
        transaction holds the lock in conflicting mode*/) {
            try {
                wait();
            }catch ( InterruptedException e){/*...*/ }
        }
        if (holders.isEmpty()) { // no TIDs hold lock
            holders.addElement(trans); lockType = aLockType;
        } else if (/*another transaction holds the lock, share it*/ ) ){
            if (/* this transaction not a holder*/) holders.addElement(trans);
        } else if (/* this transaction is a holder but needs a more exclusive lock*/)
                lockType.promote();
        }
    }

    public synchronized void release(TransID trans ){ holders.removeElement(trans); //
        remove this holder
          set locktype to none notifyAll();
    }
}
```

The methods of *Lock* are synchronized so that the threads attempting to acquire or release a lock will not interfere with one another. But, in addition, attempts to acquire the lock use the *wait* method whenever they have to wait for another thread to release it.

The *acquire* method carries out the rules given in Figure 16.15 and Figure 16.16. Its arguments specify a transaction identifier and the type of lock required by that transaction. It tests whether the request can be granted. If another transaction holds the lock in a conflicting mode, it invokes *wait*, which causes the caller's thread to be suspended until a corresponding *notify*. Note that the *wait* is enclosed in a *while*, because all waiters are notified and some of them may not be able to proceed. When, eventually, the condition is satisfied, the remainder of the method sets the lock appropriately:

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

Figure 16.18 *LockManager* class

```
public class LockManager {
   private Hashtable theLocks;

   public void setLock(Object object, TransID trans, LockType lockType){ Lock
      foundLock; synchronized(this){
             find the lock associated with object
             if there isn't one, create it and add it to the hashtable
        }
      foundLock.acquire(trans, lockType);
   }

     synchronize this one because we want to remove all entries public synchronized void
   unLock(TransID trans) {
      Enumeration e = theLocks.elements(); while(e.hasMoreElements()){
         Lock aLock = (Lock)(e.nextElement());
         if(/* trans is a holder of this lock*/ ) aLock.release(trans);
      }
   }
}
```

The *release* method's arguments specify the transaction identifier of the transaction that is releasing the lock. It removes the transaction identifier from the holders, sets the lock type to *none* and calls *notifyAll*. The method notifies all waiting threads in case there are multiple transactions waiting to acquire read locks – all of them may be able to proceed.

The class *LockManager* is shown in Figure 16.18. All requests to set locks and to release them on behalf of transactions are sent to an instance of *LockManager*:

The *setLock* method's arguments specify the object that the given transaction wants to lock and the type of lock. It finds a lock for that object in its hashtable or, if necessary, creates one. It then invokes the *acquire* method of that lock.

The *unLock* method's argument specifies the transaction that is releasing its locks. It finds all of the locks in the hashtable that have the given transaction as a holder. For each one, it calls the *release* method.

The reader is invited to consider the following:

What is the consequence for *write* transactions in the presence of a steady trickle of requests for read locks? Think of an alternative implementation.

When the holder has a write lock, several readers and writers may be waiting. The reader should consider the effect of *notifyAll* and think of an alternative implementation. If a holder of a read lock tries to promote the lock when the lock is shared, it will be blocked. Is there any solution to this difficulty?

Locking rules for nested transactions • The aim of a locking scheme for nested transactions is to serialize access to objects so that:

Each set of nested transactions is a single entity that must be prevented from observing the partial effects of any other set of nested transactions.

Each transaction within a set of nested transactions must be prevented from observing the partial effects of the other transactions in the set.

The first rule is enforced by arranging that every lock that is acquired by a successful subtransaction is *inherited* by its parent when it completes. Inherited locks are also inherited by ancestors. Note that this form of inheritance passes from child to parent! The top-level transaction eventually inherits all of the locks that were acquired by successful subtransactions at any depth in a nested transaction. This ensures that the locks can be held until the top-level transaction has committed or aborted, which prevents members of different sets of nested transactions observing one another's partial effects.

The second rule is enforced as follows:

Parent transactions are not allowed to run concurrently with their child transactions. If a parent transaction has a lock on an object, it *retains* the lock during the time that its child transaction is executing. This means that the child transaction temporarily acquires the lock from its parent for its duration.

Subtransactions at the same level are allowed to run concurrently, so when they access the same objects, the locking scheme must serialize their access.

The following rules describe lock acquisition and release

For a subtransaction to acquire a read lock on an object, no other active transaction can have a write lock on that object, and the only retainers of a write lock are its ancestors.

For a subtransaction to acquire a write lock on an object, no other active transaction can have a read or write lock on that object, and the only retainers of read and write locks on that object are its ancestors.

When a subtransaction commits, its locks are inherited by its parent, allowing the parent to retain the locks in the same mode as the child.

When a subtransaction aborts, its locks are discarded. If the parent already retains the locks, it can continue to do so.Note that subtransactions at the same level that access the same object will take turns to acquire the locks retained by their parent. This ensures that their access to a common object is serialized.

As an example, suppose that subtransactions T_1, T_2 and T_{11} in Figure 16.13 all access a common object, which is not accessed by the top-level transaction T. Suppose that subtransaction T_1 is the first to access the object and successfully acquires a lock,

Figure 16.19 Deadlock with write locks

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

which it passes on to T_{11} for the duration of its execution, getting it back when T_{11} completes. When T_1 completes its execution, the top-level transaction T inherits the lock, which it retains until the set of nested transactions completes. The subtransaction T_2 can acquire the lock from T for the duration of its execution.

Definition of deadlock • Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock. A *wait-for graph* can be used to represent the waiting relationships between current transactions. In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions – there is an edge from node T to node U when transaction T is waiting for transaction U to release a lock.. Recall that the deadlock arose because transactions T and U both attempted to acquire an object held by the other. Therefore T waits for U and U waits for T. The dependency between transactions is indirect, via a dependency on objects. The diagram on the right shows the objects held by and waited for by transactions T and U. As each transaction can wait for only one object, the objects can be omitted from the wait-for graph – leaving the simple graph on the left.

Deadlock prevention • One solution is to prevent deadlock. An apparently simple but not very good way to overcome the deadlock problem is to lock all of the objects used by a transaction when it starts. This would need to be done as a single atomic step so as to avoid deadlock at this stage. Such a transaction cannot run into deadlocks with other transactions, but this approach unnecessarily restricts access to shared resources. In addition, it is sometimes impossible to predict at the start of a transaction which objects will be used.

This is generally the case in interactive applications, for the user would have to say in advance exactly which objects they were planning to use – this is inconceivable in browsing-style applications, which allow users to find objects they do not know about in advance. Deadlocks can also be prevented by requesting locks on objects in a predefined order, but this can result in premature locking and a reduction in concurrency.

Deadlock detection • Deadlocks may be detected by finding cycles in the wait-for graph. Having detected a deadlock, a transaction must be selected for abortion to break the cycle.

The software responsible for deadlock detection can be part of the lock manager. It must hold a representation of the wait-for graph so that it can check it for cycles from time to time. Edges are added to the graph and removed from the graph by the lock manager's *setLock* and *unLock* operations.

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

Note that when lock is shared, several edges may be added. An edge *T* o *U* is deleted whenever *U* releases a lock that *T* is waiting for and allows *T* to proceed. See Exercise 16.14 for a more detailed discussion of the implementation of deadlock detection. If a transaction shares a lock, the lock is not released, but the edges leading to a particular transaction are removed.

The presence of cycles may be checked each time an edge is added, or less frequently to avoid unnecessary overhead. When a deadlock is detected, one of the transactions in the cycle must be chosen and then be aborted. The corresponding node and the edges involving it must be removed from the wait-for graph. This will happen when the aborted transaction has its locks removed.

The choice of the transaction to abort is not simple. Some factors that may be taken into account are the age of the transaction and the number of cycles in which it is involved.

For one object		*Lock to be set*		
		read	*write*	*commit*
Lock already set	*none*	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	–
	commit	wait	wait	–

transactions are aborted because deadlocks have occurred and a choice can be made as to which transaction to abort.

Using lock timeouts, we can resolve the deadlock as shown in the above Figure in which the write lock for *T* on *A* becomes vulnerable after its timeout period. Transaction *U* is waiting to acquire a write lock on *A*. Therefore, *T* is aborted and it releases its lock on *A*, allowing *U* to resume and complete the transaction.

When transactions access objects located in several different servers, the possibility of distributed deadlocks arises. In a distributed deadlock, the wait-for graph can involve objects at multiple locations

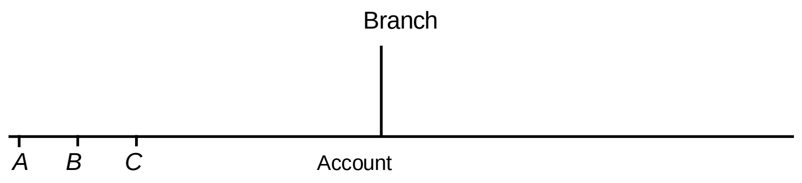
Increasing concurrency in locking schemes

Even when locking rules are based on the conflicts between *read* and *write* operations and the granularity at which they are applied is as small as possible, there is still some scope for increasing concurrency. We discuss two approaches that have been used to deal with this issue. In the first approach (two-version locking), the setting of exclusive locks is delayed until a transaction commits. In the second approach (hierarchic locks), mixed-granularity locks are used.

Two-version locking • This is an optimistic scheme that allows one transaction to write tentative versions of objects while other transactions read from the committed versions of the same objects. *read* operations only wait if another transaction is currently committing the same object. This scheme allows more concurrency than read-write locks, but writing transactions risk waiting or even rejection when they attempt to commit. Transactions cannot commit their *write* operations immediately if other uncompleted transactions have read the same objects. Therefore, transactions that request to commit in such a situation are made to wait until the reading transactions have

completed. Deadlocks may occur when transactions are waiting to commit. Therefore, transactions may need to be aborted when they are waiting to commit, to resolve deadlocks.
This variation on strict two-phase locking uses three types of lock: a read lock, a write lock and a commit lock. Before a transaction's *read* operation is performed, a read lock must be set on the object – the attempt to set a read lock is successful unless the object has a commit lock, in which case the transaction waits. Before a transaction's

Lock hierarchy for the banking example



write operation is performed, a write lock must be set on the object – the attempt to set
write lock is successful unless the object has a write lock or a commit lock, in which case the transaction waits.

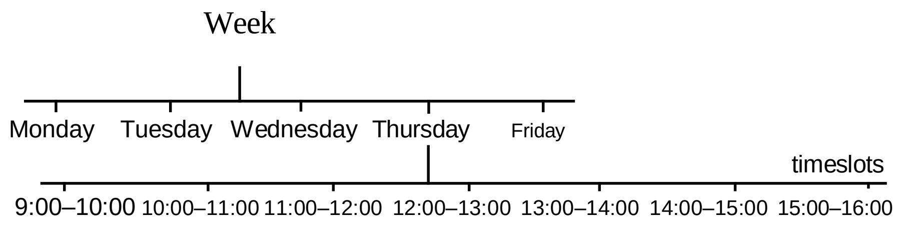
When the transaction coordinator receives a request to commit a transaction, it attempts to convert all that transaction's write locks to commit locks. If any of the objects have outstanding read locks, the transaction must wait until the transactions that set these locks have completed and the locks are released. The compatibility of read, write and commit locks is shown in Figure 16.24.

There are two main differences in performance between the two-version locking scheme and an ordinary read-write locking scheme. On the one hand, *read* operations in the two-version locking scheme are delayed only while the transactions are being committed, rather than during the entire execution of transactions – in most cases, the commit protocol takes only a small fraction of the time required to perform an entire transaction. On the other hand, *read* operations of one transaction can cause delays in committing other transactions.

Hierarchic locks • In some applications, the granularity suitable for one operation is not appropriate for another operation. In our banking example, the majority of the operations require locking at the granularity of an account. The *branchTotal* operation is different – it reads the values of all the account balances and would appear to require ead lock on all of them. To reduce locking overhead, it would be useful to allow locks of mixed granularity to coexist.
Gray [1978] proposed the use of a hierarchy of locks with different granularities. At each level, the setting of a parent lock has the same effect as setting all the equivalent child locks. This

economizes on the number of locks to be set. In our banking example, the branch is the parent and the accounts are children (see Figure 16.25).

Mixed-granularity locks could be useful in a diary system in which the data could be structured with the diary for a week being composed of a page for each day and the Lock hierarchy for a diary



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

latter subdivided further into a slot for each hour of the day, as shown in Figure 16.26. The operation to view a week would cause a read lock to be set at the top of this hierarchy, whereas the operation to enter an appointment would cause a write lock to be set on a given time slot. The effect of a read lock on a week would be to prevent write operations on any of the substructures – for example, the time slots for each day in that week.

In Gray's scheme, each node in the hierarchy can be locked, giving the owner of the lock explicit access to the node and giving implicit access to its children. In our example, a read-write lock on the branch implicitly read-write locks all the accounts. Before a child node is granted a read-write lock, an intention to read-write lock is set on the parent node and its ancestors (if any). The intention lock is compatible with other intention locks but conflicts with read and write locks according to the usual rules. Figure 16.27 gives the compatibility table for hierarchic locks. Gray also proposed a third type of intention lock – one that combines the properties of a read lock with an intention to write lock.

In our banking example, the *branchTotal* operation requests a read lock on the branch, which implicitly sets read locks on all the accounts. A *deposit* operation needs to set a write lock on a balance, but first it attempts to set an intention to write lock on the branch. These rules prevent these operations running concurrently.

Hierarchic locks have the advantage of reducing the number of locks when mixed-granularity locking is required. The compatibility tables and the rules for promoting locks are more complex.

The mixed granularity of locks could allow each transaction to lock a portion whose size is chosen according to its needs. A long transaction that accesses many objects could lock the whole collection, whereas a short transaction can lock at finer granularity.

The CORBA Concurrency Control Service supports variable-granularity locking with intention to read and intention to write lock types. These can be used as described above to take advantage the opportunity to apply locks at differing granularities in hierarchically structured data.

Optimistic concurrency control

Lock maintenance represents an overhead that is not present in systems that do not support concurrent access to shared data. Even read-only transactions (queries), which cannot possibly affect the integrity of the data, must, in general, use locking in order to guarantee that the data being read is not modified by other transactions at the same time. But locking may be necessary only in the worst case.

For example, consider two client processes that are concurrently incrementing the values of *n* objects. If the client programs start at the same time and run for about the same amount of time, accessing the objects in two unrelated sequences and using a separate transaction to access and increment each item, the chances that the two programs will attempt to access the same object at the same time are just 1 in *n* on average, so locking is really needed only once in every *n* transactions.

The use of locks can result in deadlock. Deadlock prevention reduces concurrency severely, and therefore deadlock situations must be resolved either by the use of timeouts or by deadlock detection. Neither of these is wholly satisfactory for use in interactive programs.

To avoid cascading aborts, locks cannot be released until the end of the transaction. This may reduce significantly the potential for concurrency.

The alternative approach proposed by Kung and Robinson is ‘optimistic’ because it is based on the observation that, in most applications, the likelihood of two clients’ transactions accessing the same object is low. Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until the client completes its task and issues a *closeTransaction* request. When a conflict arises, some transaction is generally aborted and will need to be restarted by the client. Each transaction has the following phases:

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

tentative version for that transaction already exists, a *read* operation accesses it; otherwise, it accesses the most recently committed value of the object. *write* operations record the new values of the objects as tentative values (which are invisible to other transactions). When there are several concurrent transactions, several different tentative values of the same object may coexist. In addition, two records are kept of the objects accessed within a transaction: a *read set* containing the objects read by the transaction and a *write set* containing the objects written by the transaction. Note that as all *read* operations are performed on committed versions of the objects (or copies of them), dirty reads cannot occur.

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

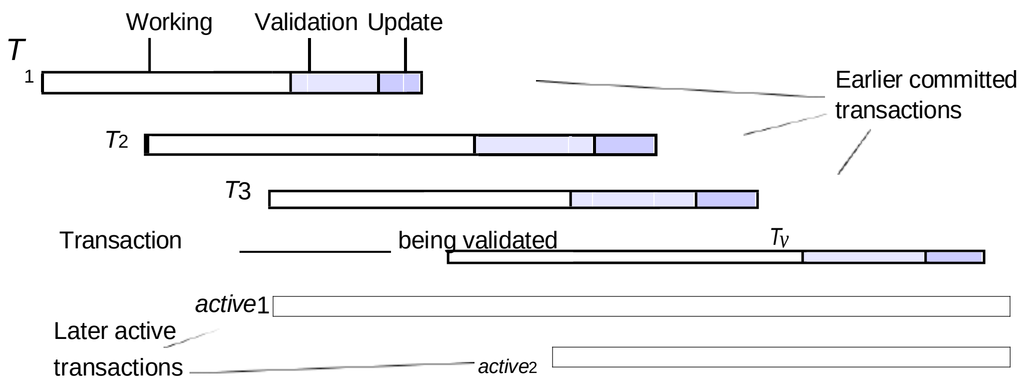
Update phase: If a transaction is validated, all of the changes recorded in its tentative versions are made permanent. Read-only transactions can commit immediately after passing validation. Write transactions are ready to commit once the tentative versions of the objects have been recorded in permanent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

Validation of transactions • Validation uses the read-write conflict rules to ensure that the scheduling of a particular transaction is serially equivalent with respect to all other *overlapping* transactions – that is, any transactions that had not yet committed at the time this transaction started. To assist in performing validation, each transaction is assigned a transaction number when it enters the validation phase (that is, when the client issues*closeTransaction*). If the transaction is validated and completes successfully, it retains this number; if it fails the validation checks and is aborted, or if the transaction is read only, the number is released for reassignment. Transaction numbers are integers assigned in ascending sequence; the number of a transaction therefore defines its position in time – a transaction always finishes its working phase after all transactions with lower numbers. That is, a transaction with the number T_i always precedes a transaction with the number T_j if $i < j$. (If the transaction number were to be assigned at the beginning of the working phase, then a transaction that reached the end of the working phase before one with a lower number would have to wait until the earlier one had completed before it could be validated.)The validation test on transaction T_v is based on conflicts between operations in pairs of transactions T_i and T_v. For a transaction T_v to be serializable with respect to an overlapping transaction T_i, their operations must conform to the following rules:

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

As the validation and update phases of a transaction are generally short in duration compared with the working phase, a simplification can be achieved by making the rule that only one transaction may be in the validation and update phase at one time. When no two transactions may overlap in the update phase, rule 3 is satisfied. Note that this restriction on *write* operations, together with the fact that no dirty reads can occur, produces strict executions. To prevent overlapping, the entire validation and update phases can be implemented as a critical section so that only one client at a time can execute it. In order to increase concurrency, part of the validation and updating may be

Figure 16.28 Validation of transactions


Working
Validation
Update
T_1
Earlier committed transactions
T_2
T_3
Transaction being validated
T_v
*active*1
Later active transactions
*active*2

implemented outside the critical section, but it is essential that the assignment of transaction numbers is performed sequentially. We note that at any instant, the current transaction number is like a pseudo-clock that ticks whenever a transaction completes successfully.

The validation of a transaction must ensure that rules 1 and 2 are obeyed by testing for overlaps between the objects of pairs of transactions T_v and T_i. There are two forms of validation – backward and forward Backward validation checks the transaction undergoing validation with other preceding overlapping transactions – those that entered the validation phase before it. Forward validation checks the transaction undergoing validation with other later transactions, which are still active.

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

Let *startTn* be the biggest transaction number assigned (to some other committed transaction) at the time when transaction T_v started its working phase and *finishTn* be the biggest transaction number assigned at the time when T_v entered the validation phase. The following program describes the algorithm for the validation of T_v:

```
boolean valid = true;
for (int $T_i$ = $startTn+1$; $T_i$ <= $finishTn$; $T_i$++){
    if (read set of $T_v$ intersects write set of $T_i$) valid = false;


}
```

Figure 16.28 shows overlapping transactions that might be considered in the validation of a transaction T_v. Time increases from left to right. The earlier committed transactions are T_1, T_2 and T_3. T_1 committed before T_v started. T_2 and T_3 committed before T_v finished its working phase. *StartTn* + 1 = T_2 and *finishTn* = T_3. In backward validation, the read set of T_v must be compared with the write sets of T_2 and T_3.In backward validation, the read set of the transaction being validated is compared with the write sets of other transactions that have already committed. Therefore, the only way to resolve any conflicts is to abort the transaction that is undergoing validation.

In backward validation, transactions that have no *read* operations (only *write* operations) need not be checked.

Optimistic concurrency control with backward validation requires that the write sets of old committed versions of objects corresponding to recently committed transactions are retained until there are no unvalidated overlapping transactions with which they might conflict. Whenever a transaction is successfully validated, its transaction number, *startTn* and write set are recorded in a preceding transactions list that is maintained by the transaction service. Note that this list is ordered by transaction number. In an environment with long transactions, the retention of old write sets of objects may be a problem. For example, in Figure
16.28 the write sets of T_1, T_2, T_3 and T_v must be retained until the active transaction $active_1$ completes. Note that the although the active transactions have transaction identifiers, they do not yet have transaction numbers.

Forward validation • In forward validation of the transaction T_v, the write set of T_v is compared with the read sets of all overlapping active transactions – those that are still in their working phase (rule 1). Rule 2 is automatically fulfilled because the active transactions do not write until after T_v has completed. Let the active transactions have (consecutive) transaction identifiers $active_1$ to $active_N$. The following program describes the algorithm for the forward validation of T_v:

```
boolean valid = true;
for (int $T_{id}$ = $active_1$; $T_{id}$ <= $active_N$; $T_{id}$++){
    if (write set of $T_v$ intersects read set of $T_{id}$) valid = false;

}
```

In Figure 16.28, the write set of transaction T_v must be compared with the read sets of the transactions with identifiers $active_1$ and $active_2$. (Forward validation should allow for the fact that read sets of active transactions may change during validation and writing.) As the read sets of the transaction being validated are not included in the check, read-only transactions always pass the validation check. As the transactions being compared with the validating transaction are still active, we have a choice of whether to abort the validating transaction or to pursue some alternative way of resolving the conflict. Härder [1984] suggests several alternative strategies:

Defer the validation until a later time when the conflicting transactions have finished. However, there is no guarantee that the transaction being validated will fare any better in the future. There is always the chance that further conflicting active transactions may start before the validation is achieved.

Abort all the conflicting active transactions and commit the transaction being validated.

Abort the transaction being validated. This is the simplest strategy but has the disadvantage that future conflicting transactions may be going to abort, in which case the transaction under validation has aborted unnecessarily.

Comparison of forward and backward validation • We have already seen that forward validation allows flexibility in the resolution of conflicts, whereas backward validation allows only one choice – to abort the transaction being validated. In general, the read sets of transactions are much larger than the write sets. Therefore, backward validation compares a possibly large read set against the old write sets, whereas forward validation checks a small write set against the read sets of active transactions. We see that backward validation has the overhead of storing old write sets until they are no longer needed.

On the other hand, forward validation has to allow for new transactions starting during the validation process.

Starvation • When a transaction is aborted, it will normally be restarted by the client program. But in schemes that rely on aborting and restarting transactions, there is no guarantee that a particular transaction will ever pass the validation checks, for it may come into conflict with other transactions for the use of objects each time it is restarted. The prevention of a transaction ever being able to commit is called starvation.

Occurrences of starvation are likely to be rare, but a server that uses optimistic concurrency control must ensure that a client does not have its transaction aborted repeatedly. Kung and Robinson suggest that this could be done if the server detects a transaction that has been aborted several times. They suggest that when the server detects such a transaction it should be given exclusive access by the use of a critical section protected by a semaphore.

Timestamp ordering

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

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

This rule assumes that there is only one version of each object and restricts access to one transaction at a time. If each transaction has its own tentative version of each object it accesses, then multiple concurrent transactions can access the same object. The timestamp ordering rule is refined to ensure that each transaction accesses a consistent set of versions of the objects. It must also ensure that the tentative versions of each object are commtted in the order determined by the timestamps of the transactions that made them. This is achieved by transactions waiting, when necessary, for earlier transactions to complete their writes. The *write* operations may be performed after the *closeTransaction* operation has returned, without making the client wait. But the client must wait when *read* operations need to wait for earlier transactions to finish. This

Figure 16.29 Operation conflicts for timestamp ordering

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

cannot lead to deadlock, since transactions only wait for earlier ones (and no cycle could occur in the wait-for graph).

Timestamps may be assigned from the server's clock or, as in the previous section,'pseudo-time' may be based on a counter that is incremented whenever a timestamp value is issued. As usual, the write operations are recorded in tentative versions of objects and are invisible to other transactions until a closeTransaction request is issued and the transaction is committed. Every object has a write timestamp and a set of tentative versions, each of which has a write timestamp associated with it; each object also has a set of read timestamps. The write timestamp of the (committed) object is earlier than that of any of its tentative versions, and the set of read timestamps can be represented by its maximum member. Whenever a transaction's write operation on an object is accepted, the server creates a new tentative version of the object with its write timestamp set to the transaction timestamp. A transaction's read operation is directed to the version with the maximum write timestamp less than the transaction timestamp. Whenever a transaction's read operation on an object is accepted, the timestamp of the transaction is added to its set of read timestamps. When a transaction is committed, the values of the tentative versions become the values of the objects, and the

timestamps of the tentative versions become the timestamps of the corresponding objects. In timestamp ordering, each request by a transaction for a read or write operation on an object is checked to see whether it conforms to the operation conflict rules.
A request by the current transaction Tc can conflict with previous operations done by other transactions, Ti, whose timestamps indicate that they should be later than Tc. These rules are shown in Figure 16.29, in which Ti > Tc means Ti is later than Tc and Ti < Tc means Ti, is earlier than Tc.

Timestamp ordering write rule: By combining rules 1 and 2 we get the following rule for deciding whether to accept a *write* operation requested by transaction T_c on object *D*:

```
if ($T_c$ • maximum read timestamp on $D$ &&
$T_c$ > write timestamp on committed version of $D$)
        perform write operation on tentative version of $D$ with write timestamp $T_c$ else /*
        write is too late */
            Abort transaction $T_c$
```

If a tentative version with write timestamp Tc already exists, the write operation is addressed to it; otherwise, a new tentative version is created and given write timestamp Tc. Note that any write that ‘arrives too late’ is aborted – it is too late in the sense that a transaction with a later timestamp has already read or written the object. Figure 16.30 illustrates the action of a *write* operation by transaction T_3 in cases where T_3 maximum read timestamp on the object (the read timestamps are not shown). In cases (a) to (c), T_3 > write timestamp on the committed version of the object and a tentative version with write timestamp T_3 is inserted at the appropriate place in the list of tentative versions ordered by their transaction timestamps. In case (d), T_3 < write timestamp on the committed version of the object and the transaction is aborted.

Timestamp ordering read rule: By using rule 3 we arrive at the following rule for deciding whether to accept immediately, to wait or to reject a *read* operation requested by transaction T_c on object D:

```
if ( $T_c$ > write timestamp on committed version of $D$) {
    let $D_{selected}$ be the version of $D$ with the maximum write timestamp ð $T_c$ if
    ($D_{selected}$ is committed)
            perform read operation on the version $D_{selected}$
    else
        wait until the transaction that made version $D_{selected}$ commits or aborts then
        reapply the read rule
} else
    Abort transaction $T_c$
```

Note:

If transaction T_c has already written its own version of the object, this will be used.

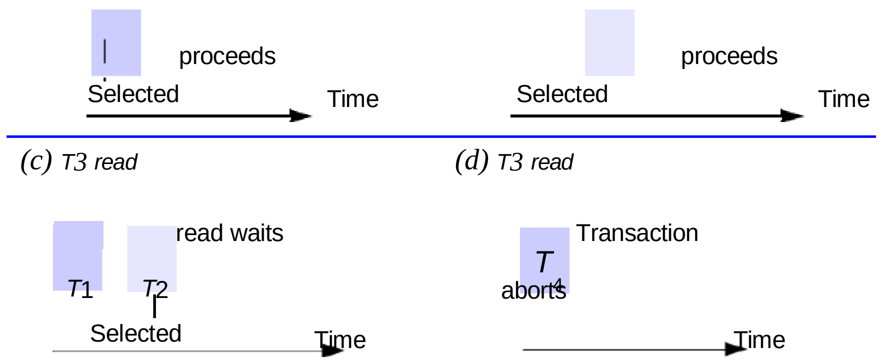
A *read* operation that arrives too early waits for the earlier transaction to complete. If the earlier transaction commits, then T_c will read from its committed version. If it aborts, then T_c will repeat the read rule (and select the previous version). This rule prevents dirty reads.

A *read* operation that 'arrives too late' is aborted – it is too late in the sense that a transaction with a later timestamp has already written the object.

Figure 16.31 illustrates the timestamp ordering read rule. It includes four cases labeled to (d), each of which illustrates the action of a *read* operation by transaction T_3. In each case, a version whose write timestamp is less than or equal to T_3 is selected. If such a version exists, it is indicated with a line. In cases (a) and (b) the *read* operation is directed to a committed version – in (a) it is the only version, whereas in (b) there is a tentative version belonging to a later transaction. In case (c) the *read* operation is directed to a tentative version and must wait until the transaction that made it commits or aborts. In case (d) there is no suitable version to read and transaction $T3$ is aborted.When a coordinator receives a request to commit a transaction, it will always be able to do so because all the operations of transactions are checked for consistency with those of earlier transactions before being carried out. The committed versions of each object must be created in timestamp order. Therefore, a coordinator sometimes needs to wait for earlier transactions to complete before writing all the committed versions of the objects accessed by a particular transaction, but there is no need for the client to wait. In order to make a transaction recoverable after a server crash, the tentative versions of objects and the fact that the transaction has committed must be written to permanent storage before acknowledging the client's request to commit the transaction.

Note that this timestamp ordering algorithm is a strict one – it ensures strict executions of transactions (see Section 16.2). The timestamp ordering read rule delays a transaction's *read* operation on any object until all transactions that had previously written that object have committed or aborted. The arrangement to commit versions in order ensures that the execution of a transaction's *write* operation on any object is delayed until all transactions that had previously written that object have committed or aborted.

Read operations and timestamps

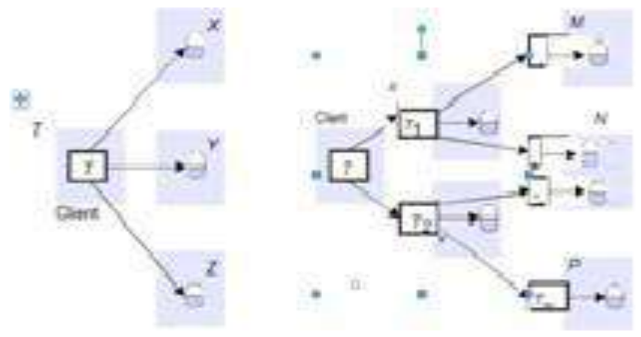


Flat and nested distributed transactions

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

In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting. Figure 17.1(b)

shows a client transaction *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 further subtransactions T_{11}, T_{12}, T_{21}, and T_{22}, which access objects at servers *M*, *N* and *P*. In the nested case, subtransactions at the same level can run concurrently, so T_1 and T_2 are concurrent, and as they invoke objects in different servers, they can run in parallel. The four subtransactions T_{11}, T_{12}, T_{21} and T_{22} also run concurrently



Consider a distributed transaction in which a client transfers $10 from account *A* to *C* and then transfers $20 from *B* to *D*. Accounts *A* and *B* are at separate servers *X* and *Y* and accounts *C* and *D* are at server *Z*. If this transaction is structured as a set of four nested transactions, as shown in Figure 17.2, the four requests (two *deposits* and two *withdraws*) can run in parallel and the overall effect can be achieved with better performance than a simple transaction in which the four operations are invoked sequentially.

Atomic commit protocols:

A transaction comes to an end when the client requests that it be committed or aborted. A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they have carried it out. This is an example of a *one-phase atomic commit protocol*.

This simple one-phase atomic commit protocol is inadequate, though, because it does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit. Reasons that prevent a server from being able to commit its part of a transaction generally relate to issues of concurrency control. For example, if locking is in use, the resolution of a deadlock can lead to the aborting of a transaction without the client being aware unless it makes another request to the server. Also if optimistic concurrency control is in use, the failure of validation at a server would cause it to decide to abort the transaction. Finally, the coordinator may not know if a server has crashed and been replaced during the progress of a distributed transaction – such a server will need to abort the transaction.The *two-phase commit protocol* is designed to allow any participant to abort its part of a transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must be aborted. In the first phase of the protocol, each participant votes for the transaction to be committed or aborted. Once a participant has voted to commit a transaction, it is not allowed to abort it. Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to

be in a *prepared* state for a transaction if it will eventually be able to commit it. To make sure of this, each participant saves in permanent storage all of the objects that it has altered in the transaction, together with its status – prepared.

In the second phase of the protocol, every participant in the transaction carries out the joint decision. If any one participant votes to abort, then the decision must be to abort the transaction. If all the participants vote to commit, then the decision is to commit the transaction.

The problem is to ensure that all of the participants vote and that they all reach the same decision. This is fairly simple if no errors occur, but the protocol must work correctly even when some of the servers fail, messages are lost or servers are temporarily unable to communicate with one another.

The two-phase commit protocol

During the progress of a transaction, there is no communication between the coordinator and the participants apart from the participants informing the coordinator when they join the transaction. A client’s request to commit (or abort) a transaction is directed to the coordinator. If the client requests *abortTransaction*, or if the transaction is aborted by one of the participants, the coordinator informs all participants immediately. It is when the client asks the coordinator to commit the transaction that the two-phase commit protocol comes into use.

In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit; in the second, it tells them to commit (or abort) the transaction. If a participant can commit its part of a transaction, it will agree as soon as it has recorded the changes it has made (to the objects) and its status in

Figure 17.4 Operations for two-phase commit protocol

canCommit?(trans)o 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) o 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.

permanent storage and is therefore prepared to commit. The coordinator in a distributed transaction communicates with the participants to carry out the two-phase commit protocol by means of the operations summarized in Figure 17.4. The methods *canCommit*, *doCommit* and *doAbort* are methods in the interface of the participant. The methods *haveCommitted* and *getDecision* are in the coordinator interface.

The two-phase commit protocol consists of a voting phase and a completion phase, as shown in Figure 17.5. By the end of step 2, the coordinator and all the participants that voted *Yes* are prepared to commit. By the end of step 3, the transaction is effectively completed. At step 3a the coordinator and the participants are committed, so the coordinator can report a decision to commit to the client. At 3b the coordinator reports a decision to abort to the client.

At step 4 participants confirm that they have committed so that the coordinator knows when the information it has recorded about the transaction is no longer needed.

This apparently straightforward protocol could fail due to one or more of the servers crashing or due to a breakdown in communication between the servers. To deal with the possibility of crashing, each server saves information relating to the two-phase commit protocol in permanent storage. This information can be retrieved by a new process that is started to replace a crashed server. The recovery aspects of distributed transactions are discussed in Section 17.6.

The exchange of information between the coordinator and participants can fail when one of the servers crashes, or when messages are lost. Timeouts are used to avoid processes blocking forever. When a timeout occurs at a process, it must take an appropriate action. To allow for this the protocol includes a timeout action for each step at which a process may block. These actions are designed to allow for the fact that in an asynchronous system, a timeout may not necessarily imply that a server has failed

The two-phase commit protocol

Phase 1 (voting phase):

The coordinator sends a *canCommit?* request to each of the participants in the transaction. 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):

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

(b) transaction and sends a *doCommit* request to each of the participants.

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

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

Locking

In a distributed transaction, the locks on an object are held locally (in the same server). The local lock manager can decide whether to grant a lock or make the requesting transaction wait. However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction. When locking is used for concurrency control, the objects remain locked and are unavailable for other transactions during the atomic commit protocol, although an aborted transaction releases its locks after phase 1 of the protocol.

As lock managers in different servers set their locks independently of one another, it is possible that different servers may impose different orderings on transactions. Consider the following interleaving of transactions *T* and *U* at servers *X* and *Y*:

T			*U*
write(A)	at *X*	locks *A*	

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

The transaction *T* locks object *A* at server *X*, and then transaction *U* locks object *B* at server *Y*. After that, *T* tries to access *B* at server *Y* and waits for *U*'s lock. Similarly, transaction *U* tries to access *A* at server *X* and has to wait for *T*'s lock. Therefore, we have *T* before *U* in one server and *U* before *T* in the other. These different orderings can lead to cyclic dependencies between transactions, giving rise to a distributed deadlock situation. The detection and resolution of distributed deadlocks is discussed in Section 17.5. When a deadlock is detected, a transaction is aborted to resolve the deadlock. In this case, the coordinator will be informed and will abort the transaction at the participants involved in the transaction.

Timestamp ordering concurrency control

In a single server transaction, the coordinator issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them. In distributed transactions, we require that each coordinator issue globally unique timestamps. A globally unique transaction timestamp is issued to the client by the first coordinator accessed by a transaction. The transaction timestamp is passed to the coordinator at each server whose objects perform an operation in the transaction.

The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. For example, if the version of an object accessed by transaction *U* commits after the version accessed by *T* at one server, if *T* and *U* access the same object as one another at other servers they must commit them in the same order. To achieve the same ordering at all the servers, the coordinators must agree as to the ordering of their timestamps. A timestamp consists of a <*local timestamp, server-id*> pair. The agreed ordering of pairs of timestamps is based on a comparison in which the *server-id* part is less significant.

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

When timestamp ordering is used for concurrency control, conflicts are resolved as each operation is performed using the rules given in Section 16.6. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the participants. Therefore any transaction that reaches the client request to commit should always be able to commit, and participants in the two-phase commit protocol will normally agree to commit. The only situation in which a participant will not agree to commit is if it has crashed during the transaction.

Distributed deadlocks

With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock. Most deadlock detection schemes operate by finding cycles in the transaction wait-for graph. In a distributed

system involving multiple servers being accessed by multiple transactions, a global

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*

wait-for graph can in theory be constructed from the local ones. There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a *distributed deadlock*. Recall that the wait-for graph is a directed graph in which nodes represent transactions and objects, and edges represent either an object held by a transaction or a transaction waiting for an object. There is a deadlock if and only if there is a cycle in the wait-for graph.

Figure 17.12 shows the interleavings of the transactions *U*, *V* and *W* involving the objects *A* and *B* managed by servers *X* and *Y* and objects *C* and *D* managed by server *Z*.

The complete wait-for graph in Figure 17.13(a) shows that a deadlock cycle consists of alternate edges, which represent a transaction waiting for an object and an object held by a transaction. As any transaction can only be waiting for one object at a time, objects can be left out of wait-for graphs, as shown in Figure 17.13(b).

Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions.

Local wait-for graphs can be built by the lock manager at each server, as discussed in Chapter 16. In the above example, the local wait-for graphs of the servers are:

server *Y*: *U* o *V* (added when *U* requests *b.withdraw(30)*)

server *Z*: *V* o *W* (added when *V* requests *c.withdraw(20)*) server *X*: *W* o *U* (added when *W* requests *a.withdraw(20)*)

As the global wait-for graph is held in part by each of the several servers involved, communication between these servers is required to find cycles in the graph.

A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for graph to the global deadlock detector, which amalgamates the information in the local graphs in order to construct a global wait-for graph. The global deadlock detector checks for cycles in the global wait-for graph When it finds a cycle, it makes a decision on how to resolve the deadlock and tells the servers which transaction to abort.

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

Phantom deadlocks • A deadlock that is 'detected' but is not really a deadlock is called *phantom deadlock*. In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected. As this procedure will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

Transaction recovery

The atomic property of transactions requires that all the effects of committed transactions and none of the effects of incomplete or aborted transactions are reflected in the objects they accessed. This property can be described in terms of two aspects: durability and failure atomicity. Durability requires that objects are saved in permanent storage and will be available indefinitely thereafter. Therefore an acknowledgement of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's (volatile) objects. Failure atomicity requires that effects of transactions are atomic even when the server crashes. Recovery is concerned with ensuring that a server's objects are durable and that the service provides failure atomicity.

Although file servers and database servers maintain data in permanent storage, other kinds of servers of recoverable objects need not do so except for recovery purposes. In this chapter, we assume that when a server is running it keeps all of its objects in its volatile memory and records its committed objects in a *recovery file* or files. Therefore recovery consists of restoring the server with the latest committed versions of its objects from permanent storage. Databases need to deal with large volumes of data. They generally hold the objects in stable storage on disk with a cache in volatile memory.

The requirements for durability and failure atomicity are not really independent of one another and can be dealt with by a single mechanism – the *recovery manager*. The tasks of a recovery manager are: to save objects in permanent storage (in a recovery file) for committed transactions; to restore the server's objects after a crash; to reorganize the recovery file to improve the performance of recovery; to reclaim storage space (in the recovery file).

In some cases, we require the recovery manager to be resilient to media failures. Corruption during a crash, random decay or a permanent failure can lead to failures of the recovery file, which can result in some of the data on the disk being lost. In such cases we need another copy of the recovery file. Stable storage, which is implemented so as to be very unlikely to fail by using mirrored disks or copies at a different location may be used for this purpose.

Intentions list • Any server that provides transactions needs to keep track of the objects accessed by clients' transactions. when a client opens a transaction, the server first contacted provides a new transaction identifier and

Types of entry in a recovery file

Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared*, *committed*, *aborted*) and other status values used for the two-phase commit protocol.
	Transaction identifier and a sequence of intentions, each of which

Intentions list	consists of <*objectID*, *P*i>, where Pi is the position in the recoverle of the value of the object.

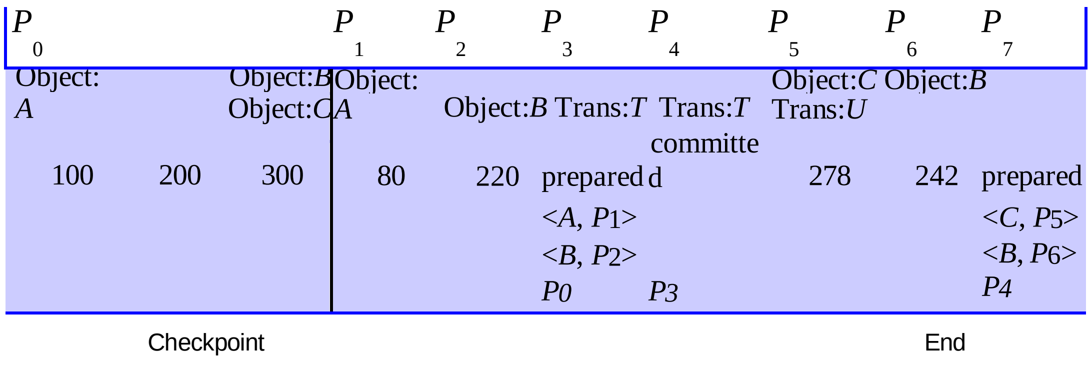
returns it to the client. Each subsequent client request within a transaction up to anincluding the *commit* or *abort* request includes the transaction identifier as an argument. During the progress of a transaction, the update operations are applied to a private set of tentative versions of the objects belonging to the transaction.

At each server, an *intentions list* is recorded for all of its currently active transactions – an intentions list of a particular transaction contains a list of the references and the values of all the objects that are altered by that transaction. When a transaction is committed, that transaction's intentions list is used to identify the objects it affected. The committed version of each object is replaced by the tentative version made by that transaction, and the new value is written to the server's recovery file. When a transaction aborts, the server uses the intentions list to delete all the tentative versions of objects made by that transaction. Recall also that a distributed transaction must carry out an atomic commit protocol before it can be committed or aborted. Our discussion of recovery is based on the two-phase commit protocol, in which all the participants involved in a transaction first say whether they are prepared to commit and later, if all the participants agree, carry out the actual commit actions. If the participants cannot agree to commit, they must abort the transaction.At the point when a participant says it is prepared to commit a transaction, its recovery manager must have saved both its intentions list for that transaction and the objects in that intentions list in its recovery file, so that it will be able to carry out the commitment later, even if it crashes in the interim.When all the participants involved in a transaction agree to commit it, the coordinator informs the client and then sends messages to the participants to commit their part of the transaction. Once the client has been informed that a transaction has committed, the recovery files of the participating servers must contain sufficient information to ensure that the transaction is committed by all of the servers, even if some of them crash between preparing to commit and committing.

Entries in recovery file • To deal with recovery of a server that can be involved in distributedtransactions, further information in addition to the values of the objects is stored in the recovery file. This information concerns the *status* of each transaction – whether it is *committed, aborted* or *prepared* to commit

Logging: In the logging technique, the recovery file represents a log containing the history of all the transactions performed by a server. The history consists of values of objects, transaction status entries and transaction intentions lists. The order of the entries in the log reflects the order in which transactions have prepared, committed and aborted at that server. In practice, the recovery file will contain a recent snapshot of the values of all the objects in the server followed by a history of transactions postdating the snapshot.

During the normal operation of a server, its recovery manager is called whenever a transaction prepares to commit, commits or aborts a transaction. When the server is prepared to commit a transaction, the recovery manager appends all the objects in its intentions list to the recovery file, followed by the current status of that transaction (*prepared*) together with its intentions list. When a transaction is eventually committed or aborted, the recovery manager appends the corresponding status of the transaction to its recovery file. It is assumed that the append operation is atomic in the sense that it writes one or more complete entries to the recovery file. If the server fails, only the last write can be incomplete. To make efficient use of the disk, several subsequent writes can be buffered and then written to disk as a single write. An additional advantage of the logging technique is that sequential writes to disk are faster than writes to random locations. After a crash, any transaction that does not have a *committed* status in the log is aborted. Therefore when a transaction commits, its *committed* status entry must be *forced* to the log – that is, written to the log together with any other buffered entries. The recovery manager associates a unique identifier with each object so that the successive versions of an object in the recovery file may be associated with the server's objects. For example, a durable form of a remote object reference such as a CORBA persistent reference will do as an object identifier Figure 17.19 illustrates the log mechanism for the banking service transactions *T* and *U* in Figure 16.7. The log was recently reorganized, and entries to the left of the double line represent a snapshot of the values of *A*, *B* and *C* before transactions *T* and *U* started. In this diagram, we use the names *A*, *B* and *C* as unique identifiers for objects. We show the situation when transaction *T* has committed and transaction *U* has prepared but not committed. When transaction *T* prepares to commit, the values of objects *A* and *B* are written at positions P_1 and P_2 in the log, followed by a prepared transaction status entry for *T* with its intentions list (< *A*, P_1 >, < *B*, P_2 >). When transaction *T* commits, a committed transaction status entry for *T* is put at position P_4. Then when transaction *U* prepares to commit, the values of objects *C* and *B* are written at positions P_5 and P_6 in the log, followed by a prepared transaction status entry for *U* with its intentions list (< *C*, P_5 >, < *B*, P_6 >).



Recovery of objects • When a server is replaced after a crash, it first sets default initial values for its objects and then hands over to its recovery manager. The recovery manager is responsible for restoring the server's objects so that they include all the effects of the committed transactions performed in the correct order and none of the effects of incomplete or aborted transactions.

The most recent information about transactions is at the end of the log. There are two approaches to restoring the data from the recovery file. In the first, the recovery manager starts at the beginning and restores the values of all of the objects from the most recent checkpoint (discussed in the next section). It then reads in the values of each of the objects, associates them with their transaction's intentions lists and for committed transactions replaces the values of the objects. In this approach, the transactions are replayed in the order in which they were executed and there could be a large number of them. In the second approach, the recovery manager will restore a server's objects by 'reading the recovery file backwards'. The recovery file has been structured so that there is a backwards pointer from each transaction status entry to the next. The recovery manager uses transactions with *committed* status to restore those objects that have not yet been restored. It continues until it has restored all of the server's objects. This has the advantage that each object is restored once only To recover the effects of a transaction, a recovery manager gets the corresponding intentions list from its recovery file. The intentions list contains the identifiers and positions in the recovery file of values of all the objects affected by the transaction. If the server fails at the point reached in Figure 17.19, its recovery manager will recover the objects as follows. It starts at the last transaction status entry in the log (at P_7) and concludes that transaction *U* has not committed and its effects should be ignored. It then moves to the previous transaction status entry in the log (at P_4) and concludes that transaction *T* has committed. To recover the objects affected by transaction T, it moves to the previous transaction status entry in the log (at P_3) and finds the intentions list for *T* ($< A, P_1 >, < B, P_2 >$). It then restores objects *A* and *B* from the values at P_1 and P_2. As it has not yet restored *C*, it moves back to P_0, which is a checkpoint, and restores *C*. To help with subsequent reorganization of the recovery file, the recovery manager notes all the prepared transactions it finds during the process of restoring the server's objects. For each prepared transaction, it adds an aborted transaction status to the recovery file. This ensures that in the recovery file, every transaction is eventually shown as either committed or aborted. The server could fail again during the recovery procedures. It is essential that recovery be idempotent, in the sense that it can be done any number of times with the same effect. This is straightforward under our assumption that all the objects are restored to volatile memory. In the case of a database, which keeps its objects in permanent storage with a cache in volatile memory, some of the objects in permanent storage will be out of date when a server is replaced after a crash. Therefore the recovery manager has to restore the objects in permanent storage. If it fails during recovery, the partially restored objects will still be there. This makes idempotence a little harder to achieve.

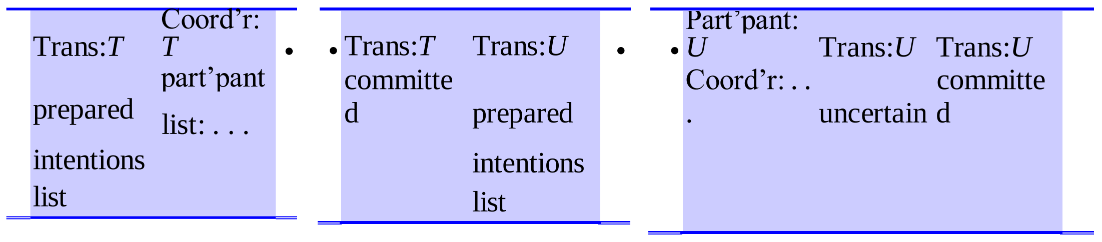
Recovery of the two-phase commit protocol In a distributed transaction, each server keeps its own recovery file. The recovery management described in the previous section must be extended to deal with any transactions that are performing the two-phase commit protocol at the time when a server fails. The recovery managers use two new status values for this purpose: *done* and *uncertain*. These status values are shown in Figure 17.6. A coordinator uses *committed* to indicate that the outcome of the vote is *Yes* and *done* to indicate that the two-phase commit protocol is complete. A participant uses *uncertain* to indicate that it has voted *Yes* but does not yet know the outcome of the vote. Two additional types of entry allow a coordinator to record a list of participants and a participant to record its coordinator:

Type of entry	*Description of contents of entry*
Coordinator	Transaction identifier, list of participants
Participant	Transaction identifier, coordinator

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

In phase 2 of the protocol, the recovery manager of the coordinator adds either a *committed* or an *aborted* transaction status to its recovery file, according to the decision. This must be a forced write (that is, it is written immediately to the recovery file). Recovery managers of participants add a *commit* or *abort* transaction status to their recovery files according to the message received from the coordinator. When a coordinator has received a confirmation from all of its participants, its recovery manager adds a *done* transaction status to its recovery file – this need not be forced. The *done* status entry is not part of the protocol but is used when the recovery file is reorganized. Figure 17.21 shows the entries in a log for transaction *T*, in which the server played the coordinator role, and for transaction *U*, in which the server played the participant role. For both transactions, the *prepared* transaction status entry comes first. In the case of a coordinator it is followed by a coordinator entry and a *committed* transaction status entry. The *done* transaction status entry is not shown in Figure 17.21. In the case of a participant, the *prepared* transaction status entry is followed by a participant entry whose state is *uncertain* and then a *committed* or *aborted* transaction status entry.

Figure 17.21 Log with entries relating to two-phase commit protocol



When a server is replaced after a crash, the recovery manager has to deal with the two-phase commit protocol in addition to restoring the objects. For any transaction where the server has played the coordinator role, it should find a coordinator entry and a set of transaction status entries. For any transaction where the server played the participant role, it should find a participant entry and a set of transaction status entries. In both cases, the most recent transaction status entry – that is, the one nearest the end of the log – determines the transaction status at the time of failure. The action of the recovery manager with respect to the two-phase commit protocol for any transaction depends on whether the server was the coordinator or a participant and on its status at the time of failure, as shown in Figure 17.22.

Reorganization of recovery file • Care must be taken when performing a checkpoint to ensure that coordinator entries of transactions without status *done* are not removed from the recovery file. These entries must be retained until all the participants have confirmed that they have completed their transactions. Entries with status *done* may be discarded. Participant entries with transaction state *uncertain* must also be retained.

Recovery of nested transactions • In the simplest case, each subtransaction of a nested transaction accesses a different set of objects. As each participant prepares to commit during the two-phase commit protocol, it writes its objects and intentions lists to the local recovery file, associating them with the transaction identifier of the top-level transaction. Although nested transactions use a special variant of the two-phase commit protocol, the recovery manager uses the same transaction status values as for flat transactions.

However, abort recovery is complicated by the fact that several subtransactions at the same and different levels in the nesting hierarchy can access the same object. Section 16.4 describes a locking scheme in which parent transactions inherit locks and subtransactions acquire locks from their parents. The locking scheme forces parent transactions and subtransactions to access common data objects at different times and ensures that accesses by concurrent subtransactions to the same objects must be serialized. Objects that are accessed according to the rules of nested transactions are made recoverable by providing tentative versions for each subtransaction. The relationship between the tentative versions of an object used by the subtransactions of a nested transaction is similar to the relationship between the locks. To support recovery from aborts, the server of an object shared by transactions at multiple levels provides a stack of tentative versions – one for each nested transaction to use.