**LECTURE NOTES**

**ON**

**Distributed Systems**

**III B-Tech II Semester**



**INFORAMTION TECHNOLOGY**

**CMR TECHNICAL CAMPUS**

**KANDLAKOYA (V), MEDCHAL**

**Section: III-II Semester (IT) Subject: Distributed Systems**

**UNIT-I**

Characterization of distributed systems: introduction, examples of distributed systems, resources sharing and the web, challenges. System models: Introduction, architectural models, fundamental models.

**UNIT - II**

Time and global states: Introduction, clocks events and process states, synchronizing physical clocks, logical time and logical clocks, global states, distributed debugging.

Coordination and agreement: introduction, distributed manual exclusion, elections, multicast communication, consensus and related problems.

**UNIT -III**

Inter process communication: Introduction the API for the Internet Protocols, External Data Representation and Marshalling, Client-Server Communication, Group Communication, Case Study: IPC in UNIX

**UNIT - IV**

Distributed File Systems: Introduction, File Service Architecture, Case Study

1: Sun Network File System, Case Study 2: The Andrew File System.

Name Services: Introduction, Name Services and the Domain Name System, Directory Services, Case Study of the Global Name Services.

Distributed Shared Memory: Introduction, Design and Implementation Issues, Sequential Consistency and IVY case study, Release Consistency, Munin Case Study: Other consistency Models

**UNIT** – **V**

Transactions and Concurrency control: Introduction, Transactions, Nested Transactions, Locks Optimistic Concurrency Control, Time Stamp Ordering, and Comparison of Methods for Concurrency Control.

Distributed Transactions: Introduction, Flat and Nested Distributed Transactions, Atomic Commit Protocols, Concurrency control in Distributed Transactions, Distributed Deadlocks, and Transaction Recovery.

**Suggested Books**

**TEXT BOOKS:**

**T**1. Distributed Systems, Concepts and Design, George Coulouris, J Dollimre and Tim Kind berg, Pearson Education, 4th Edition, 2009.

**REFERENCE BOOKS**

**R**1. Distributed Systems, Principles and Paradigms, Andrew S. Tanenbaum, Maarten Van Steen, 2nd Edition, PHI

**R**2. Distributed Systems, an Algorithm Approach, Sukumar Ghosh, Chapman and Hall/CRC, Taylor& Francis Group 2007.

**UNIT-I**

**Introduction to Distributed Systems**

This unit is about the introduction to distributed systems. We start with definition of distributed systems. It lists a list of examples of distributed systems. It also introduces Resource Sharing and Challenges of distributed systems.

The goal is to provide motivational examples of contemporary distributed systems, illustrating both the Pervasive role of distributed systems and the great diversity of the associated applications.

**DEFINITION:** A distributed system can be defined as a ―collection of independent computers that appear to the users of the system as a single computer‖. The computers in a distributed system are essentially independent machines. This means that, architecturally, the machines are capable of operating independently. The software enables this set of connected machines to appear as a single computer to the users of the system.

**Another definition:** A distributed system can be defined as a system of networked computers that coordinate their activity only by message passing.

In a distributed system, each computer has its own memory, has its own clock, and each computer runs its own operating systems.

Why distributed systems?

It is mainly because of availability of powerful yet cheap microprocessors (PCs, workstations, PDAs, embedded systems, etc.) and continuing advances in communication technology.

**The benefits of distributed systems:**

Price/performance ratio: You don't get twice the performance for twice the price in buying computers. Processors are only so fast and the price/performance curve becomes nonlinear and steep very quickly. With multiple CPUs, we can get (almost) double the performance for double the money (as long as we can figure out how to keep the processors busy and the overhead negligible).

**Distributing machines:** It makes sense to put the CPUs for ATM cash machines at the source, each networked with the bank. Each bank can have one or more computers networked with each other and with other banks. For computer graphics, it makes sense to put the graphics processing at the user's terminal to maximize the bandwidth between the device and processor.

**Computer supported cooperative networking**: Users that are geographically separated can now work and play together. Examples of this are electronic whiteboards, distributed document systems, audio/video teleconferencing, email, file transfer, and games such as Doom, Quake, Age of Empires, and Duke Nuke‘em, Star craft, and scores of others.

**Increased reliability**: If a small percentage of machines break, the rest of the system remains intact and can do useful work.

**Incremental growth**: A company may buy a computer. Eventually the workload is too great for the machine. The only option is to replace the computer with a faster one. Networking allows you to add on to an existing infrastructure.

**Remote services:** Users may need to access information held by others at their systems. Examples of this include web browsing, remote file access, and programs such as Napster and Gnutella to access MP3 music.

**Mobility:** Users move around with their laptop computers, Palm Pilots, and WAP phones. It is not feasible for them to carry all the information they need with them.

**Advantages and disadvantages:** A distributed system has distinct advantages over a set of non-networked smaller computers. Data can be shared dynamically – giving private copies (via floppy disk, for example) does not work if the data is changing. Peripherals can also be shared. Some peripherals are expensive and/or infrequently used so it is not justifiable to give each PC

a peripheral. These peripherals include optical and tape jukeboxes, typesetters, large format color printers and expensive drum scanners. Machines themselves can be shared and workload can be distributed amongst idle machines. The advantages are Economics, Speed, Inherent distribution, Reliability, Incremental growth.

There are many problems with distributed systems: Designing, implementing and using distributed software may be difficult. Issues of creating operating systems and/or languages that support distributed systems arise. The network may lose messages and/or become overloaded. Rewiring the network can be costly and difficult. Security becomes a far greater concern. Easy and convenient data access from anywhere creates security problems. The disadvantages are software, network, more components to fail, security

**EXAMPLES OF DISTRIBUTED SYSTEMS:**  examples of distributed systems are: the Internet and the associated World Wide Web, web search, online gaming, email, social networks, Ecommerce, etc. Distributed systems encompass many of the most significant technological developments of recent years and hence an understanding of the underlying technology is absolutely central to knowledge of modern computing. We now look at more specific examples of distributed systems to further illustrate the diversity and indeed complexity of distributed systems provision today.

**Web search:** Web search has emerged as a major growth industry in the last decade, with recent figures indicating that the global number of searches has risen to over 10 billion per calendar month. The task of a web search engine is to index the entire contents of the World Wide Web, encompassing a wide range of information styles including web pages, multimedia sources and (scanned) books. This is a very complex task, as current estimates state that the Web consists of over 63 billion pages and one trillion unique web addresses. Given that most search engines analyze the entire web content and then carryout sophisticated processing on this enormous database, this task itself represents a major challenge for distributed systems design.

**Mobile and ubiquitous computing:** Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include: Laptop computers. Handheld devices including mobile phones, Smart phones, GPS-enabled devices, pagers, personal digital assistants (PDAs), video cameras and digital cameras. Wearable devices such as smart watches with functionality similar to a PDA. Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators. The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes *mobile computing* possible. Mobile computing is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their ‗home‘intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them.

**Wide area network applications:** There are many wide area network applications, namely email (electronic mail), bbs (bulletin board systems), net news (group discussions on single subject), gopher (text retrieval service), WWW (world wide web, is the biggest example of distributed system).

**The Internet:** network of networks, global access to everybody(data, service, other actor; open ended), enormous size (open ended), no single authority

**Intranet:** it is a portion of the internet managed by an organization. A single authority, protected access.

**Key characteristics of distributed systems:** The key characteristics of distributed systems are Resource sharing, Openness, Concurrency Scalability, Fault Tolerance Transparency, No global clock, Independent failures

**RESOURCE SHARING**

Note that the users are so accustomed to the benefits of resource sharing that they may easily overlook their significance. We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.

When we look at from the point of view of hardware provision, we share equipment such as printers and disks to reduce costs. But of far greater significance to users is the sharing of the higher-level resources that play a part in their applications and in their everyday work and social activities. For example, users are concerned with sharing data in the form of a shared database or a set of web pages. Similarly, users think in terms of shared resources such as a search engine or a currency converter, without regard for the server or servers that provide these.

We use the term *service* for a distinct part of a computer system that manages a collection of related resources and presents their functionality to users and applications.

The only access we have to the service is via the set of operations that it exports.

The fact that services restrict resource access to a well-defined set of operations is in part standard software engineering practice. For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.

The term *server* refers to a running program (a *process*) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately. The requesting processes are referred to as *clients,* and the overall approach is known as *client-server computing*. In this approach, requests are sent in messages from clients to a server and replies are sent in messages from the server to the clients. A complete interaction between a client and a server, from the point when the client sends its request to when it receives the server‘s response, is called a *remote* *invocation*.

Many, but certainly not all, distributed systems can be constructed entirely in the form of interacting clients and servers. The World Wide Web, email and networked printers all fit this model.

**There are two types of resources:** they are hardware resources ( e.g., disks and printers) and software resources (e.g., files, windows, and data objects).

Hardware sharing is used for convenience and reduction of cost. Whereas Data sharing (shared usage of information) is used for consistency (compilers and libraries), exchange of information (database), and cooperative work (groupware).

Service resources are used as search engines, computer-supported cooperative working and so on.

**Resource Manager:** Software module that manages a set of resources. Each resourcerequires its own management policies and methods.

**Client server model:** server processes act as resource managers for a set of resources and a set of clients.

**Object based model:** resources are objects that can move. Object manager is movable. Request for a task on an object is sent to the current manager. Manager must be co-located with object.

Examples for distributed systems in use: ARGUS, Amoeba, Mach, Arjuna, Clouds, Emerald

**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 beenrecognized as a further important aspect of service quality.

**CHALLENGES OF DISTRIBUTED SYSTEMS:**

The main challenges of distributed systems are

**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.

**Openness**

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

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

To summarize:

1. Open systems are characterized by the fact that their key interfaces are published.
2. Open distributed systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.
3. Open distributed systems can be constructed from heterogeneous hardware and software, possibly from different vendors.

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

1. 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.

However, security challenges have not yet been fully met: for ‗*Denial of service attacks’* and ‗*Security of mobile code’*

**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.

**The design of scalable distributed systems presents the following challenges:**

Controlling the cost of physical resources, controlling the Performance loss,

Preventing software resources running out avoiding performance bottlenecks.

Some shared resources are accessed very frequently; for example, many users may access the same web page, causing a decline in performance. Ideally, the system and application software should not need to change when the scale of the system increases, but this is difficult to achieve. The issue of scale is a dominant theme in the development of distributed systems.

**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.

The following techniques for dealing with failures are discussed throughout the book:

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

*Masking failures*: Some failures that have been detected can be hidden or made lesssevere. Two examples of hiding failures: Messages can be retransmitted when they fail to arrive, 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 notbe 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 involve the users tolerating them as well.

*Recovery from failures*: Recovery involves the design of software so that the state ofpermanent 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.

Distributed systems provide a high degree of availability in the face of hardware faults. The *availability* of a system is a measure of the proportion of time that it is available for use. When one of the components in a distributed system fails, only the work that was using the failed component is affected. A user may move to another computer if the one that they were using fails; a server process can be started on another computer.

**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. In this case, their operations on the shared resource may conflict with one another and produce inconsistent results.

A shared resource in a distributed system must be responsible for ensuring that it operates correctly in a concurrent environment. This applies not only to servers but also to objects in applications. Therefore any programmer who takes an implementation of an object that was not intended for use in a distributed system must do whatever is necessary to make it safe in a concurrent environment. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

**Transparency**

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

The various transparencies are:

*Access transparency:* enables local and remote resources to be accessed using identicaloperations.

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

*Concurrency transparency:* enables several processes to operate concurrently usingshared resources without interference between them.

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

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

*Mobility transparency:* allows the movement of resources and clients within a systemwithout affecting the operation of users or programs.

*Performance transparency:* allows the system to be reconfigured to improveperformance as loads vary.

*Scaling transparency:* allows the system and applications to expand in scale withoutchange to the system structure or the application algorithms.

In this unit we introduced distributed systems. We started with definition of distributed systems. We presented a list of examples of distributed systems. This unit also introduced Resource Sharing and Challenges of distributed systems. In this unit we also discussed about different transparencies.

**KEYWORDS**

**A distributed system**: can be defined as a ―collection of independent computers thatappear to the users of the system as a single computer.

***Resource Manager:*** Software module that manages a set of resources. Each resourcerequires its own management policies and methods.

**Transparency:** is defined as the concealment from the user and the applicationprogrammer 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

**System Models**

**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. Architecture models define the main components of the system, what their roles are and how they interact (software architecture), and how they are deployed in an underlying network of computers (system architecture). Architecture model is concerned with the placement of its parts, namely how components are mapped to underlying network and the relationship between them, that is, their functional roles and patterns of communication between them. An architectural model simplifies and abstracts the functions of the individual components of a distributed system and then it considers the placement of the components across a network of an architectural model, the interrelationships between the components.

**Architectural elements:** To understand the fundamental building blocks of a distributed system, it is necessary to consider four key questions:

What are the entities that are communicating in the distributed system?

How do they communicate, or, more specifically, what communication paradigms used? What (potentially changing) roles and responsibilities do they have in the overall architecture?

How are they mapped on to the physical distributed infrastructure (what is their placement).Now we examine two architectural styles stemming from the role of individual processes: client-server and peer-to-peer.

**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. The Figure 1.2.1 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.

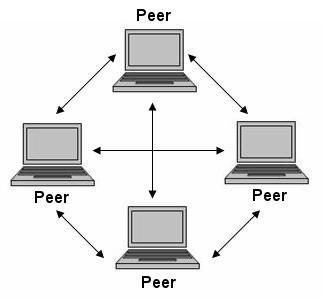
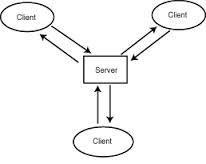


Figure1.2.1 : Client Server Model Figure1.2.2:Peer to Peer Model

**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. The centralization of service provision and management implied by placing a service at a single address does not scale well beyond the capacity of the computer that hosts the service and the bandwidth of its network connections. The Figure 1.2.2 shows Peer-to-Peer model.

**2. 4 FUNDAMENTAL MODELS**

Fundamental models deal with formal description of the properties that is common to architecture models. Since no global time in a distributed system, so the clocks on different computers do not necessarily give the same time as one another. Messages communications can be affected by delays and suffer from a variety of failures and vulnerable to security attacks. All system models have some common fundamental properties. There are three fundamental models: 1) interaction models, 2) failure models and 3) security models.

**Interaction:** Processes communicate with messages and coordinate via synchronization and ordering of activities. The message delays are often of considerable duration, the coordination between processes is limited by lack of global clock. The interaction model deals with performance and with difficulty of setting time limits.

**Failure:** The correct operation is threatened whenever a fault occurs in any of the computers and network. We should define types of faults in order to tolerate them for the system to continue to run correctly. The **failure** model attempts to give a precise specification of the faults that can be exhibited by processes and communication channels.

**Security:** The modular feature of distributed system and their openness exposes them to attack by both external and internal agents. Security model defines and classifies the forms of attack may take, providing a basis for the analysis of threats to a system and for the design of system that are able to resist them. The **security** model discusses the possible threats to processes and communication channels. It introduces the concept of a secure channel, which is secure against those threats.

**UNIT-II**

**Time and Global States**

Time is an important practical issue. For example, we require computers around the world to timestamp electronic commerce transactions consistently. But time is problematic in distributed systems. Each computer may have its own physical clock, but the clocks typically deviate, and we cannot synchronize them perfectly. We examine algorithms for synchronizing physical clocks approximately and then go on to explain logical clocks, including vector clocks, which are a tool for ordering events without knowing precisely when they occurred.

The absence of global physical time makes it difficult to find out the state of our distributed programs as they execute. We often need to know what state process A is in

when process B is in a certain state, but we cannot rely on physical clocks to know what is true at the same time.

In this unit we introduce fundamental concepts and algorithms related to monitoring distributed systems as their execution unfolds, and to timing the events that occur in their executions.

Time is an important and interesting issue in distributed systems, for several reasons. First, time is a quantity we often want to measure accurately. In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronize its clock with an authoritative, external source of time.

Second, algorithms that depend upon clock synchronization have been developed for several problems in distribution. These include maintaining the consistency of distributed data, checking the authenticity of a request sent to a server and eliminating the processing of duplicate updates.

Measuring time can be problematic due to the existence of multiple frames of reference.

**CLOCKS, EVENTS AND PROCESS STATES**

Here we to understand how to characterize the system‘s evolution as it executes, and how to timestamp the events in a system‘s execution that interest users.

We begin by considering how to order and timestamp the events that occur at a single process.

We take a distributed system to consist of a collection P of *N* processes p*i*, i= 1, 2, … *N.*

Each process executes on a single processor, and the processors do not share memory. Each process pi in P has a state S*i* 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 a message *send* or *receive* operation, or an operation that transforms Pi‘s state –one that changesone or more of the values in Si*.* In practice, we may choose to use a high-level description of the actions, according to the application. 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, *e*

→i *e'* 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 →i:

h*istory(P*i *)* = hi = *<* ei 0 ei 1 ei 2, … *>*

**Clocks:** We have seen how to order the events at a process, but not how to timestampthem – 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 standard for elapsed real time, known as *International Atomic Time*. Since 1967, 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).

**Coordinated Universal Time** –abbreviated as UTC (from the French equivalent)–is aninternational 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 land-based radio stations and satellites covering many parts of the world.

**SYNCHRONIZING PHYSICAL CLOCKS**

In order to know at what time of day events occur at the processes in our distributed system P – for example, for accountancy purposes – it is necessary to synchronize the processes‘ clocks, *C*i , with an authoritative, external source of time. This is *external* *synchronization*. And if the clocks *C*iare synchronized with one another to a knowndegree 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*ofUTC time, | *S*(*t)* – *C*i *(t)|* < *D*, for *i* = 1, 2, …*N* and for all real times *t* in *I*. Another way of saying this is that the clocks *C*i are *accurate* to within the bound *D*.

***Internal synchronization*:**For a synchronization bound*D >*0 , |*C*i*(t)*–*C*i*() | < D*for*i, j* = 1, 2, … *N* , and for all real times *t* in *I*. Another way of saying this is that the clocks

*C*i *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 P 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 ρ > 0. This

means that the error in measuring the interval between real times *t* and *t*‘ (*t’ >* *t*) is bounded:

(1 – ρ) (t‘ – *t) ≤ H(t*) – *H(t)* *≤* ( 1 + ρ ) ( 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  C(t’) > C(t)*

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 *C*i*(t)* = α *Hi(t)* + β , where we are free to choose the values of α and β .

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*.

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.

**Synchronization in a synchronous system**

We begin by considering the simplest possible case: of internal synchronization between two processes in a synchronous distributed system. In a synchronous system, bounds are known for the drift rate of clocks, the maximum message transmission delay, and the time required to execute each step of a process.

One process sends the time *t* on its local clock to the other in a message *m.* In principle, the receiving process could set its clock to the time *t* + *T* trans, where *T* trans is the time taken to transmit *m* between them. The two clocks would then agree.

Unfortunately, *T* trans is subject to variation and is unknown. In general, other processes are competing for resources with the processes to be synchronized at their respective nodes, and other messages compete with *m* for the network resources. Nonetheless, there is always a minimum transmission time, *min*, that would be obtained if no other processes executed and no other network traffic existed; *min* can be measured or conservatively estimated.

In a synchronous system, by definition, there is also an upper bound *max* on the time taken to transmit any message. Let the uncertainty in the message transmission time be *u*, so that *u* = (*max* – *min)*. If the receiver sets its clock to be *t* + *min*, then the clock skew may be as much as *u*, since the message may in fact have taken time *max* to arrive. Similarly, if it sets its clock to *t* + *max*, the skew may again be as large as *u*. If, however, it sets its clock to the halfway point, *t* + (*max* + *min)/* 2, then the skew is at most *u /* 2. In general, for a synchronous system, the optimum bound that can be achieved on clock skew when synchronizing *N* clocks is *u*(1 – 1/N).

Most distributed systems found in practice are asynchronous: the factors leading to message delays are not bounded in their effect, and there is no upper bound *max* on message transmission delays. This is particularly so for the Internet. For an asynchronous system, we may say only that *T* trans = *min* + *x*, where *x ≥* 0. The value of *x* is not known in a particular case, although a distribution of values may be measurable for a particular installation.

**Cristian’s method for synchronizing clocks**

Cristian suggested the use of a time server, connected to a device that receives signals from a source of UTC, to synchronize computers externally. Upon request, the server process *S* supplies the time according to its clock, as shown in Figure 4.4.1

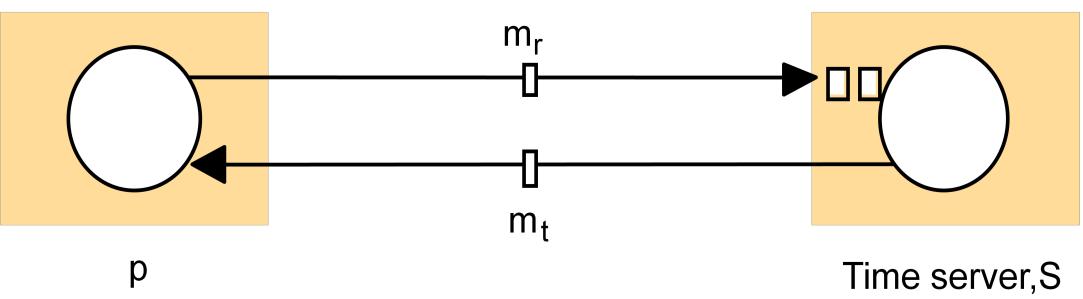


Figure 16.4.1: Clock synchronization using a time server.

Cristian observed that while there is no upper bound on message transmission delays in an asynchronous system, the round-trip times for messages exchanged between pairs of processes are often reasonably short – a small fraction of a second. He describes the algorithm as ***probabilistic*:** the method achieves synchronization only if the observed round-trip times between client and server are sufficiently short compared with the required accuracy.

A process *p* requests the time in a message *m*r, and receives the time value *t* in a message *m*t(*t* is inserted in *m*tat the last possible point before transmission from *S*‘s computer).Process *p* records the total round-trip time *Tround* taken to send the request *m*r and receive the reply *m*t. It can measure this time with reasonable accuracy if its rate of clock drift is small. For example, the round-trip time should be on the order of 1–10 milliseconds on a LAN, over which time a clock with a drift rate of 10–6 seconds/second varies by at most 10–5 milliseconds.

A simple estimate of the time to which *p* should set its clock is *t* + *Tround* / 2, which assumes that the elapsed time is split equally before and after *S* placed *t* in *m*t. This is

normally a reasonably accurate assumption, unless the two messages are transmitted over different networks. If the value of the minimum transmission time *min* is known or can be conservatively estimated, then we can determine the accuracy of this result as follows.

The earliest point at which *S* could have placed the time in *m*t was *min* after *p* dispatched *m*r. The latest point at which it could have done this was *min* before *m*tarrived at *p*. Thetime by *S*‘s clock when the reply message arrives is therefore in the range [*t* + *min*, *t* +

*Tround* – *min*]. The width of this range is *Tround* –2*min*, so the accuracy is *±*(*Tround /*2– *min)*.

Variability can be dealt with to some extent by making several requests to *S* and taking the minimum value of *Tround* to give the most accurate estimate. The greater the accuracy required, the smaller the probability of achieving it. This is because the most accurate results are those in which both messages are transmitted in a time close to *min* – an unlikely event in a busy network.

**Discussion of Cristian’s algorithm:** Cristian‘s method suffers from the problemassociated with all services implemented by a single server: that the single time server might fail and thus render synchronization temporarily impossible.

**The Berkeley algorithm**

Gusella and Zatti describe an algorithm for internal synchronization that they developed for collections of computers running Berkeley UNIX. In it, a coordinator computer is chosen to act as the *master*. Unlike in Cristian‘s protocol, this computer periodically polls the other computers whose clocks are to be synchronized, called *slaves*. The slaves send back their clock values to it. The master estimates their local clock times by observing the round-trip times (similarly to Cristian‘s technique), and it averages the values obtained(including its own clock‘s reading). The balance of probabilities is that this average cancels out the individual clocks‘ tendencies to run fast or slow. The accuracy of the protocol depends upon a nominal maximum round-trip time between the master and the slaves. The master eliminates any occasional readings associated with larger times than this maximum.Instead of sending the updated current time back to the other computers – which would introduce further uncertainty due to the message transmission time – the master sends the amount by which each individual slave‘s clock requires adjustment. This can be a positive or negative value.

The Berkeley algorithm eliminates readings from faulty clocks. Such clocks could have a significant adverse effect if an ordinary average was taken so instead the master takes a *fault-tolerant average*. That is, a subset is chosen of clocks that do not differ from oneanother by more than a specified amount, and the average is taken of readings from only these clocks.

Should the master fail, then another can be elected to take over and function exactly as its predecessor.

**The Network Time Protocol**

Cristian‘s method and the Berkeley algorithm are intended primarily for use within intranets. The Network Time Protocol (NTP) defines an architecture for a time service and a protocol to distribute time information over the Internet. NTP‘s chief design aims and features are as follows:

*To provide a service enabling clients across the Internet to be synchronized accurately to UTC*: Although large and variable message delays are encountered in Internetcommunication, NTP employs statistical techniques for the filtering of timing data and it discriminates between the quality of timing data from different servers.

*To provide a reliable service that can survive lengthy losses of connectivity*: There areredundant servers and redundant paths between the servers. The servers can reconfigure so as to continue to provide the service if one of them becomes unreachable.

*To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers*: The service is designed to scale to large numbers of clients andservers.

*To provide protection against interference with the time service, whether malicious or accidental*: The time service uses authentication techniques to check that timing dataoriginate from the claimed trusted sources. It also validates the return addresses of messages sent to it.

The NTP service is provided by a network of servers located across the Internet. *Primary* *servers* are connected directly to a time source such as a radio clock receiving UTC; *secondary servers* are synchronized, ultimately, with primary servers. The servers areconnected in a logical hierarchy called a *synchronization subnet* (see Figure 4.4.2) whose levels are called *strata*. Primary servers occupy stratum 1: they are at the root. Stratum 2 servers are secondary servers that are synchronized directly with the primary servers; stratum 3 servers are synchronized with stratum 2 servers, and so on. The lowest-level

(leaf) servers execute in users‘ workstations.

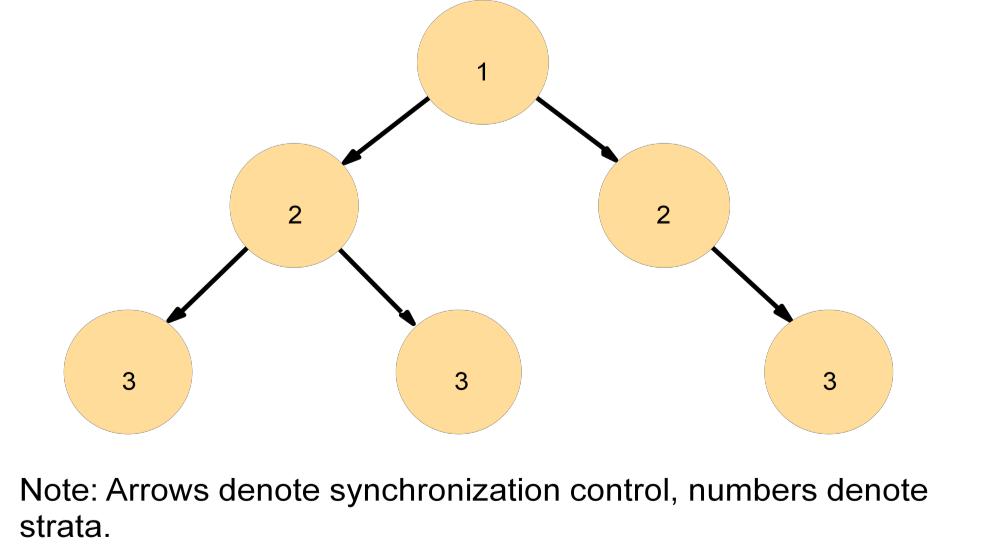


Figure 16.4.2: An example synchronization subnet in an NTP implementation

The clocks belonging to servers with high stratum numbers are liable to be less accurate than those with low stratum numbers, because errors are introduced at each level of synchronization. NTP also takes into account the total message round-trip delays to the root in assessing the quality of timekeeping data held by a particular server.

The synchronization subnet can reconfigure as servers become unreachable or failures occur. If, for example, a primary server‘s UTC source fails, then it can become a stratum 2 secondary server. If a secondary server‘s normal source of synchronization fails or becomes unreachable, then it may synchronize with another server.

NTP servers synchronize with one another in one of three modes: multicast, procedure-call and symmetric mode. *Multicast mode* is intended for use on a high-speed LAN. One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay. This mode can achieve only relatively low accuracies, but ones that nonetheless are considered sufficient for many purposes.

*Procedure-call mode* is similar to the operation of Cristian‘s algorithm. In this mode, oneserver accepts requests from other computers, which it processes by replying with its timestamp (current clock reading). This mode is suitable where higher accuracies are required than can be achieved with multicast, or where multicast is not supported in hardware. For example, file servers on the same or a neighboring LAN that need to keep accurate timing information for file accesses could contact a local server in procedure-call mode.

Finally, *symmetric mode* is intended for use by the servers that supply time information in LANs and by the higher levels (lower strata) of the synchronization subnet, where the highest accuracies are to be achieved. A pair of servers operating in symmetric mode exchanges messages bearing timing information. Timing data are retained as part of an association between the servers that is maintained in order to improve the accuracy of their synchronization over time.

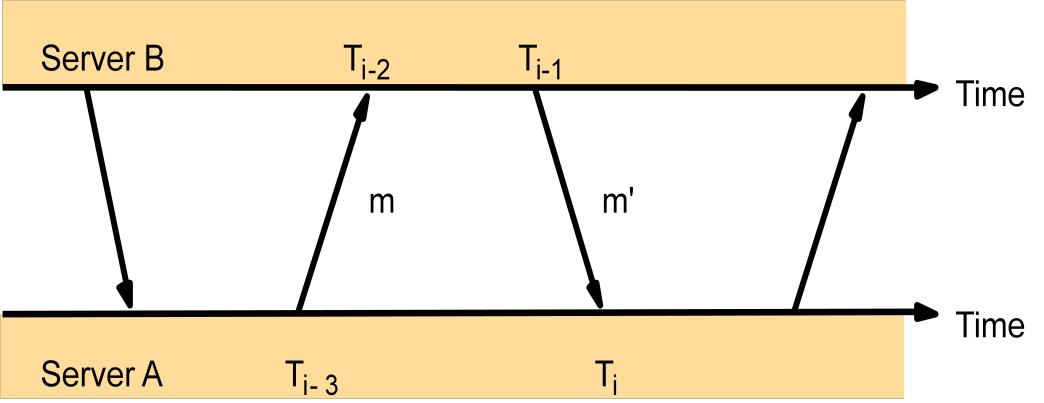
In all modes, messages are delivered unreliably, using the standard UDP Internet transport protocol. In procedure-call mode and symmetric mode, processes exchange pairs of messages. Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted. The recipient of the NTP message notes the local time when it receives the message. The four times *T*i-3, *T*i-2, *T*i-1 and *T*i are shown in Figure 4.4.3 for the messages *m* and *m'* sent between servers *A* and *B*. Note that in symmetric mode, unlike in Cristian‘s algorithm, there can be a nonnegligible delay between the arrival of one message and the dispatch of the next. Also, messages may be lost, but the three timestamps carried by each message are nonetheless valid.

Figure 16.4.3: Messages exchanged between a pair of NTP peers

For each pair of messages sent between two servers the NTP calculates an *offset o*i ,

which is an estimate of the actual offset between the two clocks, and a *delay d*i ,

which is the total transmission time for the two messages.

If the true offset of the clock at *B* relative to that at *A* is *o*,

and if the actual transmission times for *m* and *m'* are *t* and *t'*, respectively, then we have:

*T*i-2= *T*i-3+ *t* + *o and T*i= *T*i-1+ *t*‘ – *o*

This leads to:

*d*i= *t* + *t*‘ = *T*i-2– *T*i-3+ *T*i-1 and:

*o* = *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 ≤ *o* ≤ *o*i + *d*i */*2 .

Thus *o*i is an estimate of the offset, and *d*i is a measure of the accuracy of this estimate.

NTP servers apply a data filtering algorithm to successive pairs *<o*i, *d* i*>,* which estimates the offset *o* and calculates the quality of this estimate as a statistical quantity called the *filter dispersion*. A relatively high filter dispersion represents relatively unreliable data.The eight most recent pairs *<o*i *d*i*>* are retained. As with Cristian‘s algorithm, the value of *o*j that corresponds to the minimum value *d*j is chosen to estimate *o*.

The value of the offset derived from communication with a single source is not necessarily used by itself to control the local clock, however. In general, an NTP server engages in message exchanges with several of its peers. In addition to data filtering applied to exchanges with each single peer, NTP applies a peer-selection algorithm. This examines the values obtained from exchanges with each of several peers, looking for relatively unreliable values. The output from this algorithm may cause a server to change the peer that it primarily uses for synchronization.

Peers with lower stratum numbers are more favored than those in higher strata because they are ‗closer‘to the primary time sources. Also, those with the lowest *synchronization* *dispersion* are relatively favored. This is the sum of the filter dispersions measuredbetween the server and the root of the synchronization subnet.

NTP employs a phase lock loop model, which modifies the local clock‘s update frequency in accordance with observations of its drift rate. To take a simple example, if a clock is discovered always to gain time at the rate of, say, four seconds per hour, then its frequency can be reduced slightly (in software or hardware) to compensate for this. The clock‘s drift in the intervals between synchronization is thus reduced.

**GLOBAL STATES**

We examine the problem of finding out whether a particular property is true of a distributed system as it executes. We begin by giving the examples of distributed garbage collection, deadlock detection, termination detection and debugging:

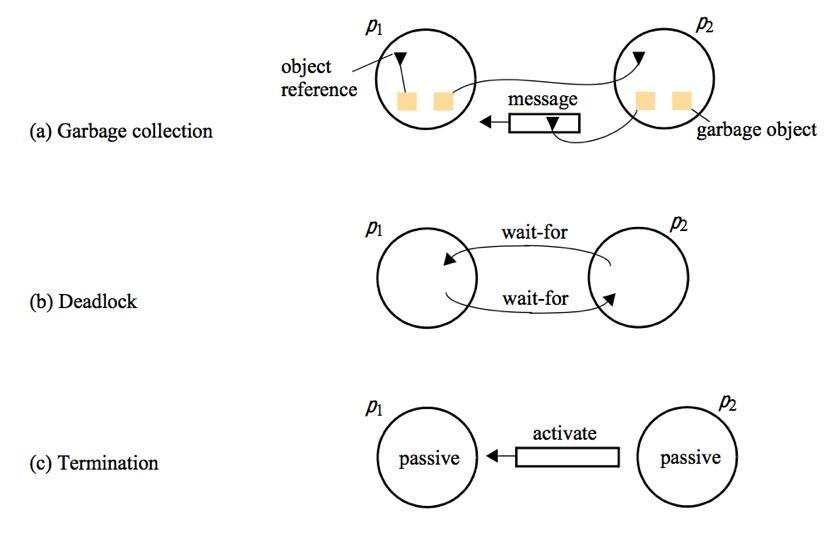


Figure 16.4.4: Detecting global properties

***Distributed garbage collection*:**An object is considered to be garbage if there are nolonger any references to it anywhere in the distributed system. The memory taken up by that object can be reclaimed once it is known to be garbage. To check that an object is garbage, we must verify that there are no references to it anywhere in the system. In Figure 4.4.4(a), process *p*1 has two objects that both have references – one has a reference within *p*1 itself, and *p*2 has a reference to the other. Process *p*2 has one garbage object, with no references to it anywhere in the system. It also has an object for which neither *p*1 nor *p*2 has a reference, but there is a reference to it in a message that is in transit between the processes. This shows that when we consider properties of a system, we must include the state of communication channels as well as the state of the processes.

***Distributed deadlock detection*:**A distributed deadlock occurs when each of a collectionof processes waits for another process to send it a message, and where there is a cycle in

the graph of this ‗waits-for‘ relationship. Figure 4.4.4(b) shows that processes *p*1 and *p*2 are each waiting for a message from the other, so this system will never make progress.

***Distributed termination detection*:**The problem here is how to detect that a distributedalgorithm has terminated. Detecting termination is a problem that sounds deceptively easy to solve: it seems at first only necessary to test whether each process has halted. To see that this is not so, consider a distributed algorithm executed by two processes *p*1 and *p*2, each of which may request values from the other. Instantaneously, we may find that aprocess is either active or passive – a passive process is not engaged in any activity of its own but is prepared to respond with a value requested by the other. Suppose we discover that *p*1 is passive and that *p*2 is passive (Figure 4.4.4c). To see that we may not conclude that the algorithm has terminated, consider the following scenario: when we tested *p*1 for passivity, a message was on its way from *p*2, which became passive immediately after sending it. On receipt of the message, *p*1 became active again – after we had found it to be passive. The algorithm had not terminated.

The phenomena of termination and deadlock are similar in some ways, but they are different problems. First, a deadlock may affect only a subset of the processes in a system, whereas all processes must have terminated. Second, process passivity is not the same as waiting in a deadlock cycle: a deadlocked process is attempting to perform a further action, for which another process waits; a passive process is not engaged in any activity.

***Distributed debugging*:**Distributed systems are complex to debug, and care needs to betaken in establishing what occurred during the execution. For example, suppose Smith has written an application in which each process *p*i contains a variable *x*i ( *i* = 1, 2 …, *N* ). The variables change as the program executes, but they are required always to be within a value of one another.

Unfortunately, there is a bug in the program, and Smith suspects that under certain circumstances |*x*i – *x* j *|* for some *i* and *j*, breaking her consistency constraints. Her problem is that this relationship must be evaluated for values of the variables that occur at the same time.

**Global states**

It is possible in principle to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes – is much harder to address.

The essential problem is the absence of global time. If all processes had perfectly synchronized clocks, then we could agree on a time at which each process would record its state – the result would be an actual global state of the system. From the collection of process states we could tell, for example, whether the processes were deadlocked. But we cannot achieve perfect clock synchronization, so this method is not available to us.

So we might ask whether we can assemble a meaningful global state from local states recorded at different real times. The answer is a qualified ‗yes‘.

**Coordination and Agreement**

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

1. Consider a mission critical application that requires several computers to communicate and decide whether to proceed with or abort a mission. Clearly, all must come to agreement about the fate of the mission.
2. Consider the Berkeley algorithm for time synchronization. One of the participate computers serves as the coordinator. Suppose that coordinator fails. The remaining computers must elect a new coordinator.
3. Broadcast networks like Ethernet and wireless must agree on which nodes can send at any given time. If they do not agree, the result is a collision and no message is transmitted successfully.
4. Like other broadcast networks, sensor networks face the challenging of agreeing which nodes will send at any given time. In addition, many sensor network algorithms require that nodes elect coordinators that take on a server-like responsibility. Choosing these nodes is particularly challenging in sensor networks because of the battery constraints of the nodes.
5. Many applications, such as banking, require that nodes coordinate their access of a shared resource. For example, a bank balance should only be accessed and updated by one computer at a time.

**Failure Assumptions and Detection**

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

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

**Failure Detection**

One possible algorithm for detecting failures is as follows:

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

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

**Mutual Exclusion**

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

Each of the following algorithms attempt to ensure the following:

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

**Central Server**

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

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

Requests are serviced in FIFO order.

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

**Token Ring**

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

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

**Multicast and Logical Clocks**

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

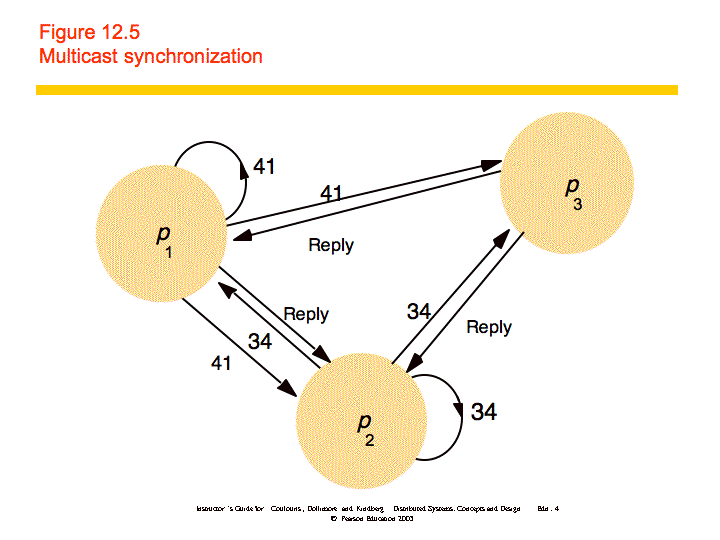
* sets its state to waiting
* sends a message to all other processes containing its ID and timestamp
* once all other processes respond, it can enter the CS

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

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

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

* sets its state to released
* replies to queued requests



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

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

**Election**

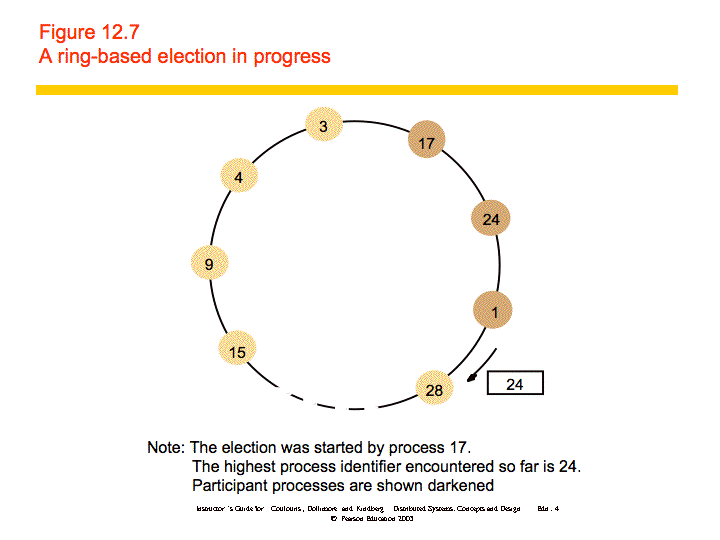
An election algorithm determines which process will play the role of coordinator or server. All processes need to agree on the selected process. Any process can start an election, for example if it notices that the previous coordinator has failed. The requirements of an election algorithm are as follows:

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

**Ring-based**

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

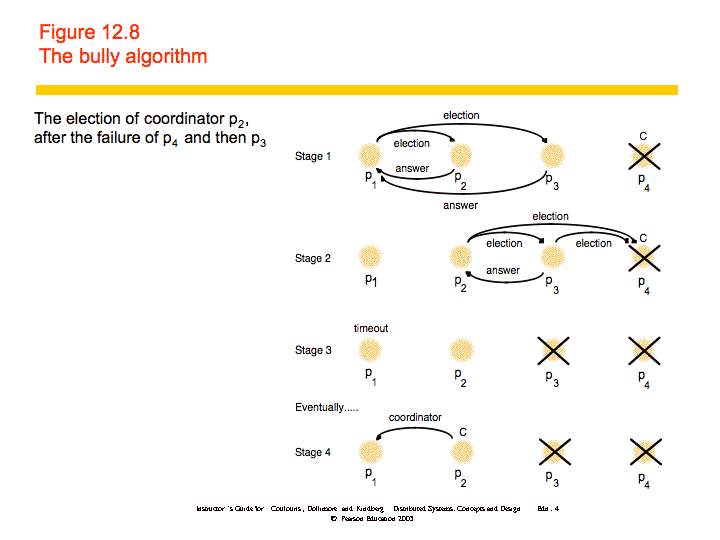
* If the value is greater that its own, it saves the ID and forwards the value to its neighbor.
* Else if its own value is greater and the it has not yet participated in the election, it replaces the ID with its own, the value with its own, and forwards the message.
* Else if it has already participated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



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

**Bully**

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



**Consensus**

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

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

The goals of consensus are as follows:

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

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

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

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

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

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

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

**UNIT-III**

**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 eachmessage 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* isissued. 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 mustseparately 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 in 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 non blocking form of *receive*.

**Message destinations:** In the Internet protocols, messages are sent to (*Internet address*,*local port*) pairs. A local port is a message destination within a computer, specified as aninteger. A port has exactly one receiver (multicast ports are an exception) 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. If the client uses a fixed Internet address to refer to a service, then that service must always run on the same computer for its address to remain valid. This can be avoided by using the following approach to providing location transparency:**•** Client programs refer to services by name and use a name server or binder totranslate their names into server locations at runtime. This allows services to be relocated but not to migrate – that is, to be moved while the system is running.

**Reliability:** Reliable communication in defined terms of validity and integrity. As far asthe 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 described as 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*–thatis, 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 communication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Inter process communication consists of transmitting a message between a socket in one process and a socket in another process. For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number (216) of possible port numbers for use by local processes for receiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. (Processes using IP multicast are an exception in that they do share ports) 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 sentto 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 anInternet 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 clientbinds 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 sizein 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* fordatagram 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 hasa socket bound to the destination port.

The method *receive* blocks until a datagram is received, unless a timeout has been set on the socket. If the process that invokes the *receive* method has other work to do while waiting for the message, it should arrange to use a separate thread. For example, when a server receives a message from a client, the message may specify work to do, in which case the server will use separate threads to do the work and to wait for messages from other clients*.*

*Timeouts*: The *receive* that blocks forever is suitable for use by a server that is waiting toreceive 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 therecipient 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:** Reliable communication is defined in terms of twoproperties: 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 checksumerror 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 usingUDP data grams 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 tooccasional 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 data grams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

1. The need to store state information at the source and destination;
   1. The transmission of extra messages;
2. Latency for the sender.

**Java API for UDP datagrams:** The Java API provides datagram communication bymeans of two classes: *DatagramPacket* and *DatagramSocket*.

*DatagramPacket*: This class provides a constructor that makes an instance out of an arrayof 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 length of message Internet address port number message

An instance of *DatagramPacket* may be transmitted between processes when one process *sends* it and another *receives* it.

This class provides another constructor for use when receiving a message. Its arguments specify an array of bytes in which to receive the message and the length of the array. A received message is put in the *DatagramPacket* together with its length and the Internet address and port of the sending socket.

The message can be retrieved from the *DatagramPacket* by means of the method *getData.* The methods *getPort* and *getAddress* access the port and Internet address.

*import java.net.\*; import java.io.\*;*

*public class UDPClient{*

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

*// args give message contents and server hostname DatagramSocket aSocket = null;*

*try {*

*aSocket = new DatagramSocket(); byte [] m = args[0].getBytes();*

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

*DatagramPacket request =*

*new DatagramPacket(m, m.length(), aHost, serverPort); aSocket.send(request);*

*byte[] buffer = new byte[1000];*

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

*System.out.println("Reply: " + new String(reply.getData()));*

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

*} finally { if(aSocket != null) aSocket.close();}*

*}*

*}*

Figure 5.1.1 UDP client sends a message to the server and gets a reply

*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.

The class *DatagramSocket* provides methods that include the following:

*send* and *receive*: These methods are for transmitting datagrams between a pair ofsockets. The argument of *send* is an instance of *DatagramPacket* containing a message and its destination. The argument of *receive* is an empty *DatagramPacket* in which to put the message, its length and its origin. The methods *send* and *receive* can throw

*IOExceptions*.

*setSoTimeout*: This method allows a timeout to be set. With a timeout set, the *receive* method will block for the time specified and then throw an *InterruptedIOException*.

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

*import java.net.\*; import java.io.\*;*

*public class 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();}*

*}*

*}*

**Figure 5.1.2** UDP server repeatedly receives a request and sends it back to the client

Figure 2.1.1 shows the program for a client that creates a socket, sends a message to a server at port 6789 and then waits to receive a reply. The arguments of the *main* method supply a message and the DNS hostname of the server. The message is converted to an array of bytes, and the DNS hostname is converted to an Internet address. Figure 2.1.2 shows the program for the corresponding server, which creates a socket bound to its server port (6789) and then repeatedly waits to receive a request message from a client, to which it replies by sending back the same message.

**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 readsfrom 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 asimple 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 cuts down on the number of acknowledgement messages required.

***Flow control*:**The TCP protocol attempts to match the speeds of the processes that readfrom 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 IPpacket, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.

***Message destinations*:**A pair of communicating processes establishes a connectionbefore 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.

The API for stream communication assumes that when a pair of processes are establishing a connection, one of them plays the client role and the other plays the server role, but thereafter they could be peers. The client role involves creating a stream socket bound to any port and then making a *connect* request asking for a connection to a server at its server port. The server role involves creating a listening socket bound to a server port and waiting for clients to request connections. The listening socket maintains a queue of incoming connection requests. In the socket model, when the server *accept*s a

connection, a new stream socket is created for the server to communicate with a client, meanwhile retaining its socket at the server port for listening for *connect* requests from other clients.

The pair of sockets in the client and server are connected by a pair of streams, one in each direction. Thus each socket has an input stream and an output stream. One of the pair of processes can send information to the other by writing to its output stream, and the other process obtains the information by reading from its input stream. When an application *closes* a socket, this indicates that it will not write any more data to its output stream. Anydata in the output buffer is sent to the other end of the stream and put in the queue at the destination socket, with an indication that the stream is broken. The process at the destination can read the data in the queue, but any further reads after the queue is empty will result in an indication of end of stream. When a process exits or fails, all of its sockets are eventually closed and any process attempting to communicate with it will discover that its connection has been broken. The following are some outstanding issues related to stream communication:

***Matching of data items*:**Two communicating processes need to agree as to the contentsof the data transmitted over a stream. For example, if one process writes an *int* followed by a *double* to a stream, then the reader at the other end must read an *int* followed by a *double.* When a pair of processes does not cooperate correctly in their use of a stream, thereading process may experience errors when interpreting the data or may block due to insufficient data in the stream.

***Blocking*:**The data written to a stream is kept in a queue at the destination socket. Whena process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available. The process that writes data to a stream may be blocked by the TCP flow-control mechanism if the socket at the other end is queuing as much data as the protocol allows.

***Threads*:**When a server accepts a connection, it generally creates a new thread in whichto communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients. In an environment in which threads are not provided, an alternative is to test whether input is available from a stream before attempting to read it; for example, in a UNIX environment the *select* system call may be used for this purpose.

**Failure model:** To satisfy the integrity property of reliable communication, TCP streamsuse checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets. For the sake of the validity property, TCP streams use timeouts and retransmissions to deal with lost packets. Therefore, messages are guaranteed to be delivered even when some of the underlying packets are lost. But if the packet loss over a connection passes some limit or the network connecting a pair of communicating processes is severed or becomes severely congested, the TCP software responsible for sending messages will receive no acknowledgements and after a time will declare the connection to be broken. Thus TCP does not provide reliable communication, because it does not guarantee to deliver messages in the face of all possible difficulties.

When a connection is broken, a process using it will be notified if it attempts to read or write. This has the following effects:

The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection.

The communicating processes cannot tell whether the messages they have sent recently have been received or not.

**Use of TCP:** Many frequently used services run over TCP connections, with reservedport numbers. These include the following:

*HTTP*: The Hypertext Transfer Protocol is used for communication between webbrowsers and web servers.

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

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

*SMTP*: The Simple Mail Transfer Protocol is used to send mail between computers.

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

***Server Socket* and *Socket*:**

*ServerSocket*: This class is intended for use by a server to create a socket at a server portfor 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 aconstructor 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 *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error occurs.

The *Socket* class provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket. The return types of these methods are

*InputStream* and *OutputStream*, respectively–abstract classes that define methods forreading and writing bytes. The return values can be used as the arguments of constructors for suitable input and output streams. Our example uses *DataInputStream* and

*DataOutputStream*, which allow binary representations of primitive data types to be readand written in a machine-independent manner

*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;*

*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){/\*close failed\*/}}*

*}}*

**Figure 5.1.3** TCP client makes connection to server, sends request and receives reply

Figure 2.1.3 shows a client program in which the arguments of the *main* method supply a message and the DNS hostname of the server. The client creates a socket bound to the hostname and server port 7896. It makes a *DataInputStream* and a *DataOutputStream* from the socket‘s input and output streams, then writes the message to its output stream and waits to read a reply from its input stream. The server program in Figure 2.1.4 opens a server socket on its server port (7896) and listens for *connect* requests. When one arrives, it makes a new thread in which to communicate with the client. The new thread creates a *DataInputStream* and a *DataOutputStream* from its socket‘s input and output streams and then waits to read a message and write the same on e back.

*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());}*

*}}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\*/}}*

*}*

*}*

**Figure 5.1.4** TCPserver makes a connection for each client and then echoes the client‘srequest

Message consists of a string, the client and server processes use the method *writeUTF* of *DataOutputStream* to write it to the output stream and the method *readUTF* of *DataInputStream* to read it from the input stream. UTF-8 is an encoding that represents strings in a particular format.

When a process has closed its socket, it will no longer be able to use its input and output streams. The process to which it has sent data can read the data in its queue, but any further reads after the queue is empty will result in an *EOFException*. Attempts to use a closed socket or to write to a broken stream result in an *IOException.*

**EXTERNAL DATA REPRESENTATION**

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.

**EXTERNAL DATA REPRESENTATION AND MARSHALLING**

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 aform 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:

1. 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.
2. 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.
3. XML (Extensible Markup Language), which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

In the first two cases, the marshalling and un marshalling activities are intended to be 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 un marshalling 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 is another example of the textual approach.

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.

Two other techniques for external data representation are worthy of mention. Google uses an approach called *protocol buffers* to capture representations of both stored and transmitted data. There is also considerable interest in JSON (JavaScript Object Notation) as an approach to external data representation. Protocol buffers and JSON represent a step towards more lightweight approaches to data representation (when compared, for example, to XML).CORBA‘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 2.2.1. Each argument or result in a remote invocation is represented by a sequence of bytes in the invocation or result message.

|  |  |
| --- | --- |
| *Type* | *Representation* |
|  |  |
| *Sequence* | length (unsigned long) followed by elements in order |
|  |  |
| *String* | length (unsigned long) followed by characters in order (can also have |
|  | wide characters) |
|  |  |
| *Array* | array elements in order (no length specified because it is fixed) |
|  |  |
| *Struct* | in the order of declaration of the components |
|  |  |
| *Enumerated* | unsigned long (the values are specified by the order declared) |
|  |  |
| *Union* | type tag followed by the selected member |
|  |  |

.**Figure 6.2.1:** CORBA CDR for constructed types

***Primitive types*:**CDR defines a representation for both big-endian and little-endianorderings. Values are transmitted in the sender‘s ordering, which is specified in each message. The recipient translates if it requires a different ordering. For example, a 16-bit *short* occupies two bytes in the message, and for big-endian ordering, the most significantbits occupy the first byte and the least significant bits occupy the second byte. Each primitive value is placed at an index in the sequence of bytes according to its size. Suppose that the sequence of bytes is indexed from zero upwards. Then a primitive value of size *n* bytes (where *n* = 1, 2, 4 or 8) is appended to the sequence at an index that is a multiple of *n* in the stream of bytes. Floating-point values follow the IEEE standard, in which the sign, exponent and fractional part are in bytes 0–*n* for big-endian ordering and the other way round for little-endian. Characters are represented by a code set agreed between client and server.

***Constructed types*:**The primitive values that comprise each constructed type are added toa sequence of bytes in a particular order, as shown in Figure 2.2.1.

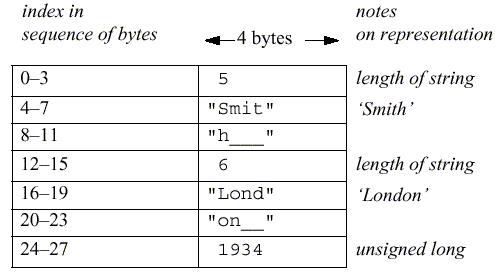


Figure 6.2.2: CORBA CDR message

Figure 6.2.2 shows a message in CORBA CDR that contains the three fields of a *struct* whose respective types are *string*, *string* and *unsigned long*. The figure shows the sequence of bytes with four bytes in each row. The representation of each string consists of an *unsigned long* representing its length followed by the characters in the string. For simplicity, we assume that each character occupies just one byte. Variable-length data is padded with zeros so that it has a standard form, enabling marshalled data or its checksum to be compared. Note that each *unsigned long*, which occupies four bytes, starts at an index that is a multiple of four. The figure does not distinguish between the big- and little-endian orderings. Although the example in Figure 6.2.2 is simple, CORBA CDR can represent any data structure that can be composed from the primitive and constructed types, but without using pointers.

Another example of an external data representation is the Sun XDR standard, which is specified in RFC. It was developed by Sun for use in the messages exchanged between clients and servers in Sun NFS.

The type of a data item is not given with the data representation in the message in either the CORBA CDR or the Sun XDR standard. This is because it is assumed that the sender and recipient have common knowledge of the order and types of the data items in a message. In particular, for RMI or RPC, each method invocation passes arguments of particular types, and the result is a value of a particular type.

**Marshalling in CORBA •** Marshalling operations can be generated automatically fromthe 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, which provides a notation for describing the types of the arguments and results of RMI methods. For example, we might use CORBA IDL to describe the data structure in the message in Figure 2.2.2 as follows:

*struct Person{ string name; string place;*

*unsigned long year; };*

The CORBA interface compiler generates appropriate marshalling and unmarshalling operations for the arguments and results of remote methods from the definitions of the types of their parameters and results.

**Remote object references**

This section applies only to languages such as 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.

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. Remote object references must be generated in a manner that ensures uniqueness over space and time. In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system. Even after the remote object associated with a given remote object reference is deleted, it is important that the remote object reference is not reused, because its potential invokers may retain obsolete remote object references. Any attempt to invoke a deleted object should produce an error rather than allow access to a different object.

There are several ways to ensure that a remote object reference is unique. One way is to construct a remote object reference by concatenating the Internet address of its host computer and the port number of the process that created it with the time of its creation and a local object number. The local object number is incremented each time an object is created in that process.

The port number and time together produce a unique process identifier on that computer. With this approach, remote object references might be represented with a format such as that shown in Figure 2.2.3. In the simplest implementations of RMI, remote objects live only in the process that created them and survive only as long as that process continues to run. In such cases, the remote object reference can be used as the address of the remote object. In other words, invocation messages are sent to the Internet address in the remote reference and to the process on that computer using the given port number

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *32 bits* | *32 bits* |  | *32 bits* | *32 bits* |  |
|  |  |  |  |  |  |
| Internet address | port number | time |  | object number | interface of |
|  |  |  |  |  | remote object |
|  |  |  | |  |  |
| **Figure 6.2.3:** Representation of a remote object reference | | | | |  |

To allow remote objects to be relocated into a different process on a different computer, the remote object reference should not be used as the address of the remote object.

The peer-to-peer overlay systems use a form of remote object reference that is completely independent of location. Messages are routed to resources by means of a distributed routing algorithm.

The last field of the remote object reference shown in Figure 2.2.3 contains some information about the interface of the remote object, for example, the interface name. This information is relevant to any process that receives a remote object reference as an argument or as the result of a remote invocation, because it needs to know about the methods offered by the remote object.

**CLIENT-SERVER COMMUNICATION :**The client-server communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server. Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later. It is often built over UDP datagrams. Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary. So, it has no connection establishment overhead.

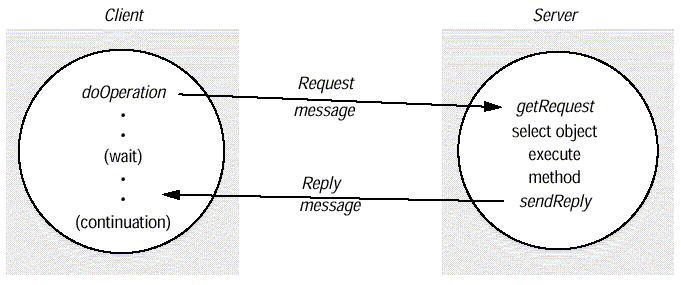
There is no need for flow control due to small amounts of data being transferred. The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in Figure 2.2.4.

Figure 2.2.4: Request-reply communication

The designed request-reply protocol matches requests to replies. If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message. Figure 2.2.5 outlines the three communication primitives.

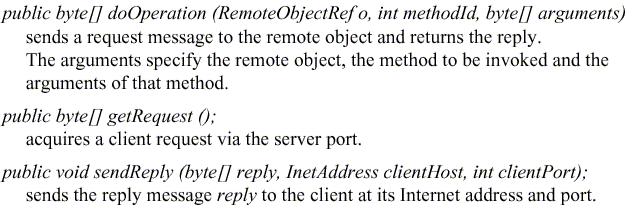
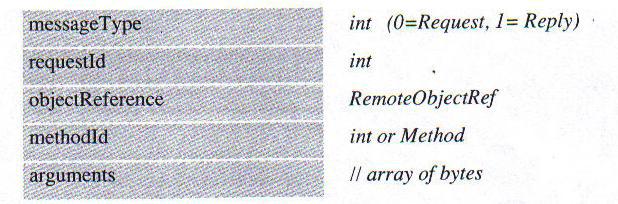


Figure 2.2.5: Operations of the request-reply protocol

The information to be transmitted in a request message or a reply message is shown in Figure 6.2.6.

In a protocol message:

The first field indicates whether the message is a request or a reply message. The second field request id contains a message identifier.

The third field is a remote object reference.

The forth field is an identifier for the method to be invoked.

Message identifier: A message identifier consists of two parts:

A requestId, which is taken from an increasing sequence of integers by the sending process

An identifier for the sender process, for example its port and Internet address. Failure model of the request-reply protocol: If the three primitive doOperation, getRequest, and sendReply are implemented over UDP datagram, they have the same communication failures.

Omission failure

Messages are not guaranteed to be delivered in sender order.

RPC exchange protocols: Three protocols are used for implementing various types of RPC.

The request (R) protocol.

The request-reply (RR) protocol.

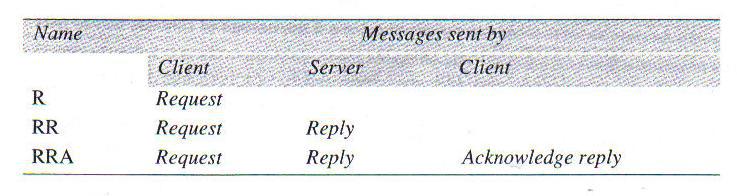
The request-reply-acknowledge (RRA) protocol.

Figure 6.2.7: RPC exchange protocols

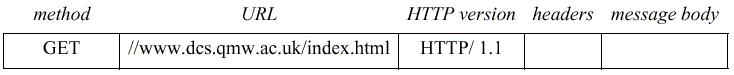
In the R protocol, a single request message is sent by the client to the server. The R protocol may be used when there is no value to be returned from the remote method. The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol. RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply. HTTP is an example of a request-reply protocol. HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.

HTTP protocol has the following steps:

Connection establishment between client and server at the default server port or at a port specified in the URL client sends a request server sends a reply connection closure HTTP 1.1 uses persistent connections. Persistent connections are connections that remain open over a series of request-reply exchanges between client and server. Resources can have MIME-like structures in arguments and results. A Mime type specifies a type and a subtype, for example text/plain, text/html, image/gif, image/jpeg.

The following are the HTTP methods:

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

Figure 2.2.8: HTTP request message

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

POST: It specifies the URL of a resource (for instance, a server program) that can deal with the data supplied with the request. This method is designed to deal with:

Providing a block of data to a data-handling process

Posting a message to a bulletin board, mailing list or news group. Extending a dataset with an append operation

PUT: It stores the supplied data in the given URL as its identifier.

DELETE: The server deletes an identified resource by the given URL on the server. OPTIONS: A server supplies the client with a list of methods. It allows to be applied to the given URL.

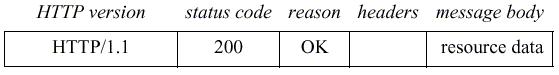
TRACE: The server sends back the request message. A reply message specifies. The protocol version, A status code, Reason, Some headers, An optional message body

Figure 2.2.9 HTTP reply message

**GROUP COMMUNICATION**

Group communication provides an example of an indirect communication paradigm.

*Group communication* offers a service whereby a message is sent to a group and then thismessage is delivered to all members of the group. In this action, the sender is not aware of the identities of the receivers. Group communication represents an abstraction over multicast communication and may be implemented over IP multicast or an equivalent overlay network, adding significant extra value in terms of managing group membership, detecting failures and providing reliability and ordering guarantees. With the added guarantees, group communication is to IP multicast what TCP is to the point-to-point service in IP.

Group communication is an important building block for distributed systems, and particularly reliable distributed systems, with key areas of application including:

1. The reliable dissemination of information to potentially large numbers of clients, including in the financial industry, where institutions require accurate and up-todate access to a wide variety of information sources;
2. Support for collaborative applications, where again events must be disseminated to multiple users to preserve a common user view – for example, in multiuser games.
3. Support for a range of fault-tolerance strategies, including the consistent update of replicated data or the implementation of highly available (replicated) servers;
4. Support for system monitoring and management, including for example load balancing strategies.

We look at group communication in more detail below, examining the programming model offered and the associated implementation issues. We examine the JGroups toolkit as a case study of a group communication service.

**The programming model**

In group communication, the central concept is that of a *group* with associated *group-membership*, whereby processes may *join* or *leave* the group. Processes can then send amessage to this group and have it propagated to all members of the group with certain guarantees in terms of reliability and ordering. Thus, group communication implements *multicast* communication, in which a message is sent to all the members of the group by asingle operation. Communication to *all* processes in the system, as opposed to a subgroup of them, is known as *broadcast*, whereas communication to a single process is known as *unicast*.

The essential feature of group communication is that a process issues only one multicast operation to send a message to each of a group of processes (in Java this operation is *aGroup.send(aMessage)*) instead of issuing multiple send operations to individualprocesses.

The use of a single multicast operation instead of multiple send operations amounts to much more than a convenience for the programmer: it enables the implementation to be efficient in its utilization of bandwidth. It can take steps to send the message no more than once over any communication link, by sending it over a distribution tree; and it can use network hardware support for multicast where this is available. The implementation can also minimize the total time taken to deliver the message to all destinations, as compared with transmitting it separately and serially.

To see these advantages, compare the bandwidth utilization and the total transmission time taken when sending the same message from a computer in London to two computers on the same Ethernet in Palo Alto, (a) by two separate UDP sends, and (b) by a single IP

multicast operation. In the former case, two copies of the message are sent independently, and the second is delayed by the first. In the latter case, a set of multicast aware routers forward a single copy of the message from London to a router on the destination LAN in California. That router then uses hardware multicast (provided by the Ethernet) to deliver the message to both destinations at once, instead of sending it twice.

The use of a single multicast operation is also important in terms of delivery guarantees. If a process issues multiple independent send operations to individual processes, then there is no way for the implementation to provide guarantees that affect the group of processes as a whole. If the sender fails halfway through sending, then some members of the group may receive the message while others do not. In addition, the relative ordering of two messages delivered to any two group members is undefined.

Group communication, however, has the potential to offer a range of guarantees in terms of reliability and ordering.

Group communication has been the subject of many research projects, including the V-system, Chorus, Amoeba, Trans/Total, Delta-4, Isis, Horus, Totem and Transis.

**Process groups and object groups:** Most work on group services focuses on the conceptof *process groups*, that is, groups where the communicating entities are processes. Such services are relatively low-level in that:

1. Messages are delivered to processes and no further support for dispatching is provided.
2. Messages are typically unstructured byte arrays with no support for marshalling of complex data types.

The level of service provided by process groups is therefore similar to that of sockets. In contrast, *object groups* provide a higher-level approach to group computing. An object group is a collection of objects (normally instances of the same class) that process the same set of invocations concurrently, with each returning responses. Client objects need not be aware of the replication. They invoke operations on a single, local object, which acts as a proxy for the group. The proxy uses a group communication system to send the invocations to the members of the object group. Object parameters and results are marshalled as in RMI and the associated calls are dispatched automatically to the right destination objects/methods.

Electra is a CORBA-compliant system that supports object groups. An Electra group can be interfaced to any CORBA-compliant application. Electra was originally built on top of the Horus group communication system, which it uses to manage the membership of the group and to multicast invocations. In ‗transparent mode‘, the local proxy returns the first available response to a client object. In ‗non-transparent mode‘, the client object can access all the responses returned by the group members. Electra uses an extension of the standard CORBA Object Request Broker interface, with functions for creating and destroying object groups and managing their membership. Eternal and the Object Group Service also provide CORBA-compliant support for object groups.

Despite the promise of object groups, however, process groups still dominate in terms of usage. For example, the popular JGroups toolkit, is a classical process group approach.

**Other key distinctions:** A wide range of group communication services has beendeveloped, and they vary in the assumptions they make:

*Closed and open groups*: A group is said to be *closed* if only members of the group maymulticast to it. A process in a closed group delivers to itself any message that it multicasts to the group. A group is *open* if processes outside the group may send to it. (The categories ‗open‘ and ‗closed‘ also apply with analogous meanings to mailing lists.)

Closed groups of processes are useful, for example, for cooperating servers to send messages to one another that only they should receive. Open groups are useful, for example, for delivering events to groups of interested processes.

*Overlapping and non-overlapping groups*: In *overlapping groups*, entities (processes orobjects) may be members of multiple groups, and non-overlapping groups imply that membership does not overlap (that is, any process belongs to at most one group). Note that in real-life systems, it is realistic to expect that group membership will overlap.

*Synchronous and asynchronous systems*: There is a requirement to consider groupcommunication in both environments. Such distinctions can have a significant impact on the underlying multicast algorithms. For example, some algorithms assume that groups are closed. The same effect as openness can be achieved with a closed group by picking a member of the group and sending it a message (one-to-one) for it to multicast to its group. Rodrigues *et al.* discuss multicast to open groups. Issues related to open and closed groups arise, when algorithms for reliability and ordering are considered. We must also consider the impact of overlapping groups and whether the system is synchronous or asynchronous on such protocols.

**Marshalling:** It is the process of taking a collection of data items and assembling theminto a form suitable for transmission in a message.

**Unmarshallin:** It is the process of disassembling them on arrival to produce anequivalent collection of data items at the destination.

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.

**Closed and open groups:** A group is said to be*closed*if only members of the group maymulticast to it. A process in a closed group delivers to itself any message that it multicasts to the group. A group is *open* if processes outside the group may send to it.

**UNIT-IV**

**FILE SERVICE ARCHITECTURE**

A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks. Here, we define a simple architecture for file systems and describe two basic distributed file service implementations with contrasting designs that have been in widespread use for over two decades:

1. The Sun Network File System, NFS;
2. The Andrew File System, AFS.

Each emulates the UNIX file system interface, with differing degrees of scalability, fault tolerance and deviation from the strict UNIX one-copy file update semantics.

It has been identified the sharing of resources as a key goal for distributed systems. The sharing of stored information is perhaps the most important aspect of distributed resource sharing. Mechanisms for data sharing take many forms. Web servers provide a restricted form of data sharing in which files stored locally, in file systems at the server or in servers on a local network, are made available to clients throughout the Internet. The design of large-scale wide area read-write file storage systems poses problems of load balancing, reliability, availability and security, whose resolution is the goal of the peer-to-peer file storage systems focuses on replicated storage systems that are suitable for applications requiring reliable access to data stored on systems where the availability of individual hosts cannot be guaranteed.

The requirements for sharing within local networks and intranets lead to a need for a different type of service – one that supports the persistent storage of data and programs of all types on behalf of clients and the consistent distribution of up-to-date data. The purpose of this unit is to describe the architecture and implementation of these *basic* distributed file systems. We use the word ‗basic‘ here to denote distributed file systems whose primary purpose is to emulate the functionality of a non-distributed file system for client programs running on multiple remote computers. They do not maintain multiple persistent replicas of files, nor do they support the bandwidth and timing guarantees required for multimedia data. Basic distributed file systems provide an essential underpinning for organizational computing based on intranets.

File systems were originally developed for centralized computer systems and desktop computers as an operating system facility providing a convenient programming interface to disk storage. They subsequently acquired features such as access-control and file-locking mechanisms that made them useful for the sharing of data and programs. Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet. A well

designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks. Their design is adapted to the performance and reliability characteristics of local networks, and hence they are most effective in providing shared persistent storage for use in intranets.

A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet. The concentration of persistent storage at a few servers reduces the need for local disk storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization. Other services, such as the name service, the user authentication service and the print service, can be more easily implemented when they can call upon the file service to meet their needs for persistent storage. Web servers are reliant on filing systems for the storage of the web pages that they serve. In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects. One way to achieve this is to serialize objects and to store and retrieve the serialized objects using files. But this method for achieving persistence and distribution is impractical for rapidly changing objects, so several more direct approaches have been developed. Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | *Sharing* | *Persistence* |  | *Distributed* | |  |
|  | *Consistency* | *Example* |  |  |  |  |
|  |  | *cache/replicas* | | *Maintenance* | |  |
| Main memory | × | × | × | 1 | RAM |  |
| File system | × | √ | × | 1 | UNIX | file |
| system |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Distributed file system | √ | √ | √ | √ | Sun NFS |
| Web | √ | √ | √ | × | Web server |
| Distributed shared memory | √ | × | √ | √ | Ivy |
| Remote objects (RMI/ORB) | √ | × | × | 1 | CORBA |
| Persistent object store | √ | √ | × | 1 | CORBA |
| Persistent |  |  |  |  |  |
|  |  |  |  |  | State |
|  |  |  |  |  | Service |
| Peer-to-peer storage system | √ | √ | √ | 2 | OceanStore |

Types of consistency:

1: strict one-copy √: slightly weaker guarantees 2: considerably weaker guarantees

**Figure 13.1.1 Storage systems and their properties**

Figure 13.1.1 provides an overview of types of storage system. The table includes distributed shared memory (DSM) systems and persistent object stores. It provides an emulation of a shared memory by the replication of memory pages or segments at each host, but it does not necessarily provide automatic persistence. Persistent object stores aim to provide persistence for distributed shared objects. Examples include the CORBA Persistent State Service and persistent extensions to Java. Some research projects have developed in platforms that support the automatic replication and persistent storage of objects. Peer-to-peer storage systems offer scalability to support client loads much larger than the systems, but they incur high performance costs in providing secure access control and consistency between updatable replicas.

The *consistency* column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur. Virtually all storage systems rely on the use of caching to optimize the performance of programs. Caching was first applied to main memory and non-distributed file systems, and for those the consistency is strict (denoted by a ‗1‘, for one-copy consistency in Figure 4.1.1) – programs cannot observe any discrepancies between cached copies and stored data after an update. When distributed replicas are used, strict consistency is more difficult to achieve. Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – this is indicated by a tick (√) in the consistency column of Figure 4.1.1.

The Web uses caching extensively both at client computers and at proxy servers maintained by user organizations. The consistency between the copies stored at web proxies and client caches and the original server is only maintained by explicit user actions. Clients are not notified when a page stored at the original server is updated; they must perform explicit checks to keep their local copies up-to-date. This serves the purposes of web browsing adequately, but it does not support the development of cooperative applications such as a shared distributed whiteboard. Persistent object systems vary considerably in their approach to caching and consistency. The CORBA and Persistent Java schemes maintain single copies of persistent objects, and remote invocation is required to access them, so the only consistency issue is between the persistent copy of an object on disk and the active copy in memory, which is not visible to remote clients. The PerDiS and Khazana projects that we mentioned above maintain cached replicas of objects and employ quite elaborate consistency mechanisms to produce forms of consistency similar to those found in DSM systems.

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

**Figure 13.1.2: File system modules**

Having introduced some wider issues relating to storage and distribution of persistent and non-persistent data, we now return to the main topic– the design of basic distributed file systems. We describe some relevant characteristics of (non-distributed) file systems.

**Characteristics of file systems**

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media.

|  |
| --- |
| File length |
| Creation timestamp |
| Read timestamp |
| Write timestamp |
| Attribute timestamp |
| Reference count |
| Owner |
| File type |
| Access control list |

**Figure 13.1.3: File attribute record structure**

Files contain both *data* and *attributes*. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner‘s identity and access control lists. A typical attribute record structure is illustrated in Figure 13.1.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.

File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. The naming of files is supported by the use of directories. A *directory* is a file, often of a special type, that provides a mapping from text names to internal file identifiers. Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part *pathnames* for files used in UNIX and other operating systems. File systems also take responsibility for the control of access to files, restricting access to files according to users‘ authorizations and the type of access requested (reading, updating, executing and so on).

The term *metadata* is often used to refer to all of the extra information stored by a file system that is needed for the management of files. It includes file attributes, directories and all the other persistent information used by the file system. Figure 4.1.2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system. Each layer depends only on the layers below it. The implementation of a distributed file service requires all of the components shown there, with additional components to deal with client-server communication and with the distributed naming and location of files.

|  |  |  |
| --- | --- | --- |
| *filedes = open(name, mode)* | Opens an existing file with the given *name*. | |
| *filedes = creat(name, mode)* | Creates a new file with the given *name*. | |
|  | Both operations deliver a file descriptor referencing the | |
|  | openfile. The *mode* is *read*, *write* or both. | |
| *status = close(filedes)* | Closes the open file *filedes.* | |
| *count = read(filedes, buffer, n)* | | Transfers *n* bytes from the file referenced by *filedes* |
| to *buffer* |  |  |
| *count = write(filedes, buffer, n)* | | Transfers *n* bytes to the file referenced by *filedes* |
|  |  | from *buffer*. |
|  |  | Both operations deliver the number of bytes |

actually

transferred and advance the read-write pointer.

*os = lseek(filedes, offset,* Moves the read-write pointer to *offset* (relative or

absolute,

*whence)* depending on *whence*).

*status = unlink(name)* Removes the file *name* from the directory structure. If the

file has no other names, it is deleted.

*status = link(name1, name2)* Adds a new name (*name2*) for a file (*name1*). *status = stat(name, buffer)* Puts the file attributes for file *name* into *buffer*.

**Figure 13.1.4: UNIX file system operations**

**File system operations:** Figure 4.1.4 summarizes the main operations on files that areavailable to applications in UNIX systems. These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes. We give the primitives here as an indication of the operations that file services are expected to support and for comparison with the file service interfaces that we shall introduce below.

The UNIX operations are based on a programming model in which some file state information is stored by the file system for each running program. This consists of a list of currently open files with a read-write pointer for each, giving the position within the file at which the next read or write operation will be applied.

The file system is responsible for applying access control for files. In local file systems such as UNIX, it does so when each file is opened, checking the rights allowed for the user‘s identity in the access control list against the *mode* of access requested in the *open* system call. If the rights match the mode, the file is opened and the *mode* is recorded in the open file state information.

**Distributed file system requirements:** Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems. Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent phases of development. We discuss these and related requirements in the following subsections.

**Transparency:** The file service is usually the most heavily loaded service in an intranet,so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems. The design must balance the flexibility and scalability that derive from transparency against software complexity and performance. The following forms of transparency are partially or wholly addressed by current file services:

***Access transparency*:**Client programs should be unaware of the distribution of files. Asingle set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

***Location transparency*:**Client programs should see a uniform file name space. Files orgroups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

***Mobility transparency*:**Neither client programs nor system administration tables in clientnodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

***Performance transparency*:**Client programs should continue to perform satisfactorilywhile the load on the service varies within a specified range.

***Scaling transparency*:**The service can be expanded by incremental growth to deal with awide range of loads and network sizes.

**Concurrent file updates:** Changes to a file by one client should not interfere with theoperation of other clients simultaneously accessing or changing the same file. This is the

well-known issue of concurrency control. The need for concurrency control for access to shared data in many applications is widely accepted and techniques are known for its implementation, but they are costly. Most current file services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking.

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

**Hardware and operating system heterogeneity:** The service interfaces should bedefined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of openness.

**Fault tolerance:** The central role of the file service in distributed systems makes itessential that the service continue to operate in the face of client and server failures. Fortunately, a moderately fault-tolerant design is straightforward for simple servers. To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics; or it can use the simpler *at-least-once* semantics with a server protocol designed in terms of *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files. The servers can be *stateless*, so that they can be restarted and the service restored after a failure without any need to recover previous state. Tolerance of disconnection or server failures requires file replication, which is more difficult to achieve.

**Consistency:** Conventional file systems such as that provided in UNIX offer*one-copy**update semantics*. This refers to a model for concurrent access to files in which the filecontents seen by all of the processes accessing or updating a given file are those that they

would see if only a single copy of the file contents existed. When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

**Security:** Virtually all file systems provide access-control mechanisms based on the useof access control lists. In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data.

**Efficiency:** A distributed file service should offer facilities that are of at least the samepower and generality as those found in conventional file systems and should achieve a comparable level of performance. Birrell and Needham expressed their design aims for the Cambridge File Server (CFS) in these terms:

We would wish to have a simple, low-level file server in order to share an expensive resource, namely a disk, whilst leaving us free to design the filing system most appropriate to a particular client, but we would wish also to have available a high-level system shared between clients.

The changed economics of disk storage have reduced the significance of their first goal, but their perception of the need for a range of services addressing the requirements of clients with different goals remains and can best be addressed by a modular architecture of the type outlined above.

The techniques used for the implementation of file services are an important part of the design of distributed systems. A distributed file system should provide a service that is comparable with, or better than, local file systems in performance and reliability. It must be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently.

**CASE STUDIES**

We have constructed an abstract model for a file service to act as an introductory example, separating implementation concerns and providing a simplified model. We describe the Sun Network File System in some detail, drawing on our simpler abstract model to clarify its architecture.

**File service architecture:** This is an abstract architectural model that underpins bothNFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a *stateless* implementation of the server module.

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

The client-server relationship is symmetrical: each computer in an NFS network can act as both a client and a server, and the files at every machine can be made available for remote access by other machines. Any computer can be a server, exporting some of its files, and a client, accessing files on other machines. But it is common practice to configure larger installations with some machines as dedicated servers and others as workstations.

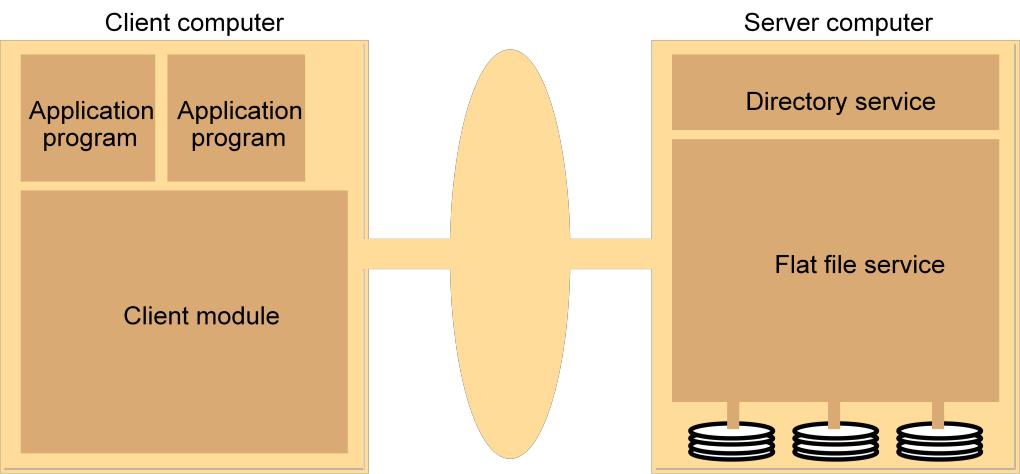
An important goal of NFS is to achieve a high level of support for hardware and operating system heterogeneity. The design is operating system–independent: client and server implementations exist for almost all operating systems and platforms, including all versions of Windows, Mac OS, Linux and every other version of UNIX. Implementations of NFS on high-performance multiprocessor hosts have been developed by several vendors, and these are widely used to meet storage requirements in intranets with many concurrent users.

**Andrew File System:** Andrew is a distributed computing environment developed atCarnegie Mellon University (CMU) for use as a campus computing and information system. The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version.

AFS was initially implemented on a network of workstations and servers running BSD UNIX and the Mach operating system at CMU and was subsequently made available in commercial and public-domain versions. A public-domain implementation of AFS is available in the Linux operating system [Linux AFS]. AFS was adopted as the basis for the DCE/DFS file system in the Open Software Foundation‘s Distributed Computing Environment (DCE) [www.opengroup.org]. The design of DCE/DFS went beyond AFS in several important respects.

**FILE SERVICE ARCHITECTUR**

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* and a *client module*. The relevant modules and their relationships areshown in Figure 4.1.5. The flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files.

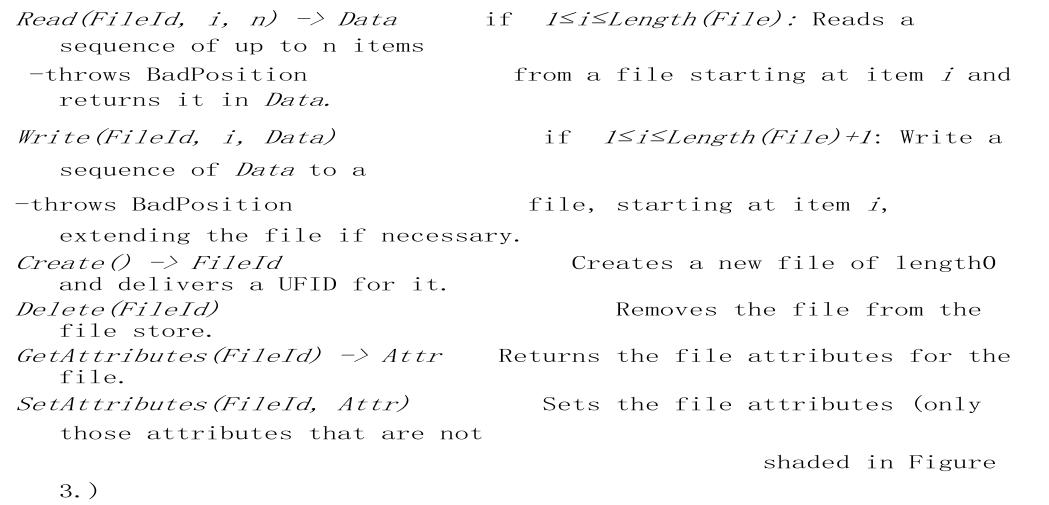
**Flat file service:** The flat file service is concerned with implementing operations on thecontents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

**Figure 13.1.5** File service architecture

**Directory service:** The directory service provides a mapping between*text names*for filesand their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

**Client module:** A client module runs in each client computer, integrating and extendingthe operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

**Flat file service interface:** Figure 4.1.6 contains a definition of the interface to a flat fileservice. This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A *FileId* is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* throw exceptions if the *FileId* argument contains an invalid UFID or the user doesn‘t have sufficient access rights. These exceptions are omitted from the definition for clarity. The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter *i* specifying a position in the file. The *Read* operation copies the sequence of *n* data items beginning at item *i* from the specified file into *Data*, which is then returned to the client. The *Write* operation copies the sequence of data items in *Data* into the specified file beginning at item *i*, replacing the previous contents of the file at the corresponding position and extending the file if necessary. *Create* creates a new, empty file and returns the UFID that is generated. *Delete* removes the specified file.



**Figure 13.1.6 Flat file service operations**

*GetAttributes* and *SetAttributes* enable clients to access the attribute record. *GetAttributes* is normally available to any client that is allowed to read the file. Access to the

*SetAttributes* operation would normally be restricted to the directory service that providesaccess to the file. The values of the length and timestamp portions of the attribute record are not affected by *SetAttributes*; they are maintained separately by the flat file service itself.

Comparison with UNIX: Our interface and the UNIX file system primitives are functionally equivalent. It is a simple matter to construct a client module that emulates the UNIX system calls in terms of our flat file service and the directory service operations described in the next section.

In comparison with the UNIX interface, our flat file service has no *open* and *close* operations – files can be accessed immediately by quoting the appropriate UFID. The

*Read* and *Write* requests in our interface include a parameter specifying a starting pointwithin the file for each transfer, whereas the equivalent UNIX operations do not. In UNIX, each *read* or *write* operation starts at the current position of the read-write pointer, and the read-write pointer is advanced by the number of bytes transferred after each *read* or *write*. A *seek* operation is provided to enable the read-write pointer to be explicitly repositioned.

The interface to our flat file service differs from the UNIX file system interface mainly for reasons of fault tolerance:

***Repeatable operations*:**With the exception of*Create*, the operations are*idempotent*,allowing the use of *at-least-once* RPC semantics – clients may repeat calls to which they receive no reply. Repeated execution of *Create* produces a different new file for each call.

***Stateless servers*:**The interface is suitable for implementation by*stateless*servers.Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

The UNIX file operations are neither idempotent nor consistent with the requirement for a stateless implementation. A read-write pointer is generated by the UNIX file system whenever a file is opened, and it is retained, together with the results of access-control checks, until the file is closed. The UNIX *read* and *write* operations are not idempotent; if an operation is accidentally repeated, the automatic advance of the read-write pointer results in access to a different portion of the file in the repeated operation. The read-write pointer is a hidden, client-related state variable. To mimic it in a file server, *open* and *close* operations would be needed, and the read-write pointer‘s value would have to beretained by the server as long as the relevant file is open. By eliminating the read-write pointer, we have eliminated most of the need for the file server to retain state information on behalf of specific clients.

**Access control:** In the UNIX file system, the user‘s access rights are checked against theaccess *mode* (read or write) requested in the *open* call and the file is opened only if the user has the necessary rights. The user identity (UID) used in the access rights check is retrieved during the user‘s earlier authenticated login and cannot be tampered with in non-distributed implementations. The resulting access rights are retained until the file is closed, and no further checks are required when subsequent operations on the same file are requested.

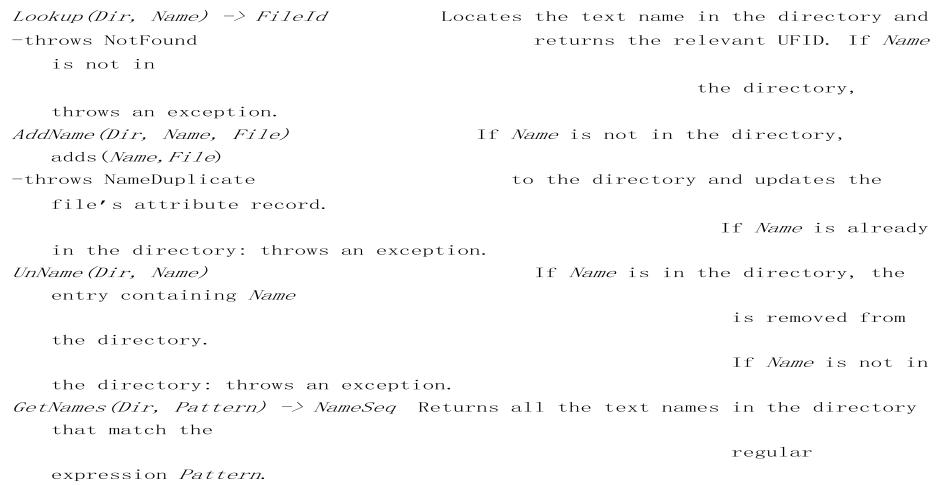
In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files. A user identity has to be passed with requests, and the server is vulnerable to forged identities. Furthermore, if the results of an access rights check were retained at the server and used for future accesses, the server would no longer be stateless. Two alternative approaches to the latter problem can be adopted:

1. An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability, which is returned to the client for submission with subsequent requests.
2. A user identity is submitted with every client request, and access checks are performed by the server for every file operation.

Both methods enable stateless server implementation, and both have been used in distributed file systems. The second is more common; it is used in both NFS and AFS. Neither of these approaches overcomes the security problem concerning forged user identities, but this can be addressed by the use of digital signatures. Kerberos is an effective authentication scheme that has been applied to both NFS and AFS.

In our abstract model, we make no assumption about the method by which access control is implemented. The user identity is passed as an implicit parameter and can be used whenever it is needed.

**Directory service interface:** Figure 4.1.7 contains a definition of the RPC interface to adirectory service. The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.



**Figure 13.1.7 Directory service operations**

We define only operations on individual directories. For each operation, a UFID for the file containing the directory is required (in the *Dir* parameter). The *Lookup* operation in the basic directory service performs a single *Name → UFID* translation. It is a building block for use in other services or in the client module to perform more complex translations, such as the hierarchic name interpretation found in UNIX. As before, exceptions caused by inadequate access rights are omitted from the definitions.

There are two operations for altering directories: *AddName* and *UnName*. *AddName* adds an entry to a directory and increments the reference count field in the file‘s attribute record.

*UnName* removes an entry from a directory and decrements the reference count. If thiscauses the reference count to reach zero, the file is removed. *GetNames* is provided to enable clients to examine the contents of directories and to implement pattern-matching operations on file names such as those found in the UNIX shell. It returns all or a subset of the names stored in a given directory. The names are selected by pattern matching against a regular expression supplied by the client.

The provision of pattern matching in the *GetNames* operation enables users to determine the names of one or more files by giving an incomplete specification of the characters in the names. A regular expression is a specification for a class of strings in the form of an expression containing a combination of literal substrings and symbols denoting variable characters or repeated occurrences of characters or substrings.

**Hierarchic file system:** A hierarchic file system such as the one that UNIX providesconsists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a *pathname* – a multi-part name that represents a path through the tree. The root has a distinguished name, and each file or directory has a name in a directory. The UNIX file-naming scheme is not a strict hierarchy – files can have several

names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a ‗well-known‘ UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

A function can be provided in the client module that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path. In a hierarchic directory service, the file attributes associated with files should include a type field that distinguishes between ordinary files and directories. This is used when following a path to ensure that each part of the name, except the last, refers to a directory.

**File groups:** A*file group*is a collection of files located on a given server. A server mayhold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refers to a software component that provides access to files.) File groups wereoriginally introduced to support facilities for moving collections of files stored on removable media between computers. In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers. In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.

File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | *32 bits* |  | *16 bits* | |  |
|  |  | |  |  |  |  |
| *file* | *group* | IP address |  | Date |  | *identifier*: |
|  |  |  |  |  |  |  |

Note that the IP address *cannot* be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

**NAME SERVICES**

In a distributed system, names are used to refer to a wide variety of resources such as computers, services, remote objects and files, as well as to users. Names facilitate communication and resource sharing. A name is needed to request a computer system to act upon a specific resource chosen out of many. Names are not the only useful means of identification: descriptive attributes are another.

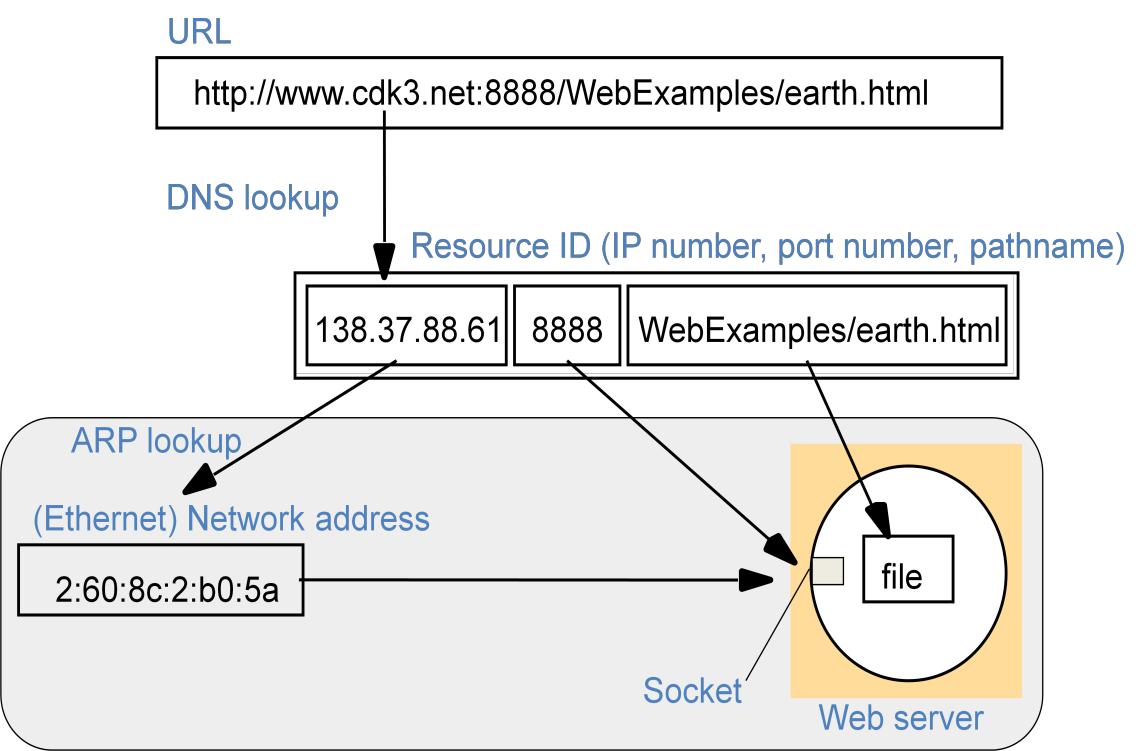
**Names, addresses and other attributes**

Any process that requires access to a specific resource must possess a name or an identifier for it. Examples of human-readable names are file names such as */etc/passwd*, URLs such as *http://www.cdk5.net/* and Internet domain names such as [*www.cdk5.net.*](http://www.cdk5.net/)

The term *identifier* is sometimes used to refer to names that are interpreted only by programs. Remote object references and NFS file handles are examples of identifiers. Identifiers are chosen for the efficiency with which they can be looked up and stored by software.

We say that a name is *resolved* when it is translated into data about the named resource or object, often in order to invoke an action upon it. The association between a name and an object is called a *binding*. In general, names are bound to *attributes* of the named objects, rather than the implementation of the objects themselves. An attribute is the value of a property associated with an object. A key attribute of an entity that is usually relevant in a distributed system is its address. For example:

1. The DNS maps domain names to the attributes of a host computer: its IP address, the type of entry (for example, a reference to a mail server or another host) and, for example, the length of time the host‘s entry will remain valid.
2. The X500 directory service can be used to map a person‘s name onto attributes including their email address and telephone number.
3. The CORBA Naming Service and Trading Service: The Naming Service maps the name of a remote object onto its remote object reference, whereas the Trading Service maps the name of a remote object onto its remote object reference, together with an arbitrary number of attributes describing the object in terms understandable by human users

****

**Figure 14.2.1 Composed naming domains used to access a resource from a URL**

Figure 4.2.1 shows the domain name portion of a URL resolved first via the DNS into an IP address and then, at the final hop of Internet routing, via ARP to an Ethernet address for the web server. The last part of the URL is resolved by the file system on the web server to locate the relevant file.

**Names and services:** Many of the names used in a distributed system are specific tosome particular service. Also, a client may use a service-specific name when requesting a service to perform an operation upon a named object or resource that it manages. For example, a file name is given to the file service when requesting that the file be deleted, and a process identifier is presented to the process management service when requesting that it be sent a signal. These names are used only in the context of the service that manages the objects named, except when clients communicate about shared objects.

Names are also sometimes needed to refer to entities in a distributed system that is beyond the scope of any single service. The major examples of these entities are users

(with proper names and email addresses), computers (with *hostnames* such as *www.cdk5.net*) and services themselves (such as *file service* or *printer service*).

**Uniform Resource Identifiers:** *Uniform Resource Identifiers*(URIs) came about fromthe need to identify resources on the Web, and other Internet resources such as electronic mailboxes. An important goal was to identify resources in a coherent way, so that they could all be processed by common software such as browsers. URIs are ‗uniform‘ in that their syntax incorporates that of indefinitely many individual types of resource identifiers (that is, URI *schemes*), and there are procedures for managing the global namespace of schemes. The advantage of uniformity is that it eases the process of introducing new types of identifier, as well as using existing types of identifier in new contexts, without disrupting existing usage.

**Uniform Resource Locators:** Some URIs contain information that can be used to locateand access a resource; others are pure resource names. The familiar term *Uniform* *Resource Locator* (URL) is often used for URIs that provide location information andspecify the method for accessing the resource. For example, *http://www.cdk5.net/* identifies a web page at the given path (‗/‘) on the host *www.cdk5.net*, and specifies that the HTTP protocol be used to access it. Another example is a ‗mailto‘ URL, such as *mailto:fred@flintstone.org*, which identifies the mailbox at the given address.

URLs are efficient identifiers for accessing resources. But they suffer from the disadvantage that if a resource is deleted or if it moves, say from one web site to another, there may be dangling links to the resource containing the old URL. If a user clicks on a dangling link to a web resource, then the web server will either respond that the resource is not found or – worse, perhaps – supply a different resource that now occupies the same location.

**Uniform Resource Names:** *Uniform Resource Names*(URNs) are URIs that is used aspure resource names rather than locators. For example, the URI: *mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com* is a URN that identifies the email message containing it in its ‗Message-Id‘ field. The URI distinguishes that message from any other email message. But it does not provide the message‘s address in any store, so a lookup operation is needed to find it.

A special sub-tree of URIs beginning with *urn:* has been reserved for URNs – although, as the *mid:* example shows, not all URNs are *urn*: URIs. The latter *urn-*prefixed URIs are all of the form *urn:nameSpace:nameSpace-specificName*.

**THE DOMAIN NAME SYSTEM**

A *name service* stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects. The collection is often subdivided into one or more naming *contexts*: individual subsets of the bindings that are managed as a unit. The major operation that a name service supports is to resolve a name – that is, to look up attributes from a given name. Operations are also required for creating new bindings, deleting bindings and listing bound names, and adding and deleting contexts.

The Domain Name System is a name service design whose main naming database is used across the Internet. It was devised principally by Mockapetris and specified in RFC 1034 and RFC 1035. DNS replaced the original Internet naming scheme, in which all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them.

This original scheme was soon seen to suffer from three major shortcomings:

1. It did not scale to large numbers of computers.
2. Local organizations wished to administer their own naming systems.
3. A general name service was needed – not one that serves only for looking up computer addresses.

The objects named by the DNS are primarily computers – for which mainly IP addresses are stored as attributes.

**Domain names:** The DNS is designed for use in multiple implementations, each ofwhich may have its own name space. In practice, however, only one is in widespread use, and that is the one used for naming across the Internet. The Internet DNS name space is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called *generic domains*) in use across the Internet were

|  |  |  |
| --- | --- | --- |
| *com* | – | Commercial organizations |
| *edu* | – | Universities and other educational institutions |
| *gov* | – | US governmental agencies |
| *mil* | – | US military organizations |
| *net* | – | Major network support centres |
| *org* | – | Organizations not mentioned above |
| *int* | – | International organizations |

New top-level domains such as *biz* and *mobi* have been added since the early 2000s.

In addition, every country has its own domains:

|  |  |  |
| --- | --- | --- |
| *us* | – | United States |
| *uk* | – | United Kingdom |
| *fr* | – | France |
| *...* | – | ... |

Countries, particularly those other than the US often use their own subdomains to distinguish their organizations. The UK, for example, has domains *co.uk* and *ac.uk*, which correspond to *com* and *edu* respectively.

**DNS queries:** The Internet DNS is primarily used for simple host name resolution andfor looking up electronic mail hosts, as follows:

***Host name resolution*:**In general, applications use the DNS to resolve host names into IPaddresses. For example, when a web browser is given a URL containing the domain name *www.dcs.qmul.ac.uk*, it makes a DNS enquiry and obtains the corresponding IP address.

***Mail host location*:**Electronic mail software uses the DNS to resolve domain names intothe IP addresses of mail hosts – i.e., computers that will accept mail for those domains. For example, when the address *tom@dcs.rnx.ac.uk* is to be resolved, the DNS is queried with the address *dcs.rnx.ac.uk* and the type designation ‗mail‘. It returns a list of domain names of hosts that can accept mail for *dcs.rnx.ac.uk*, if such exist (and, optionally, the corresponding IP addresses). The DNS may return more than one domain name so that the mail software can try alternatives if the main mail host is unreachable for some reason.

In principle, the DNS can be used to store arbitrary attributes. A query is specified by a domain name, class and type. For domain names in the Internet, the class is IN. The type of query specifies whether an IP address, a mail host, a name server or some other type of information is required.

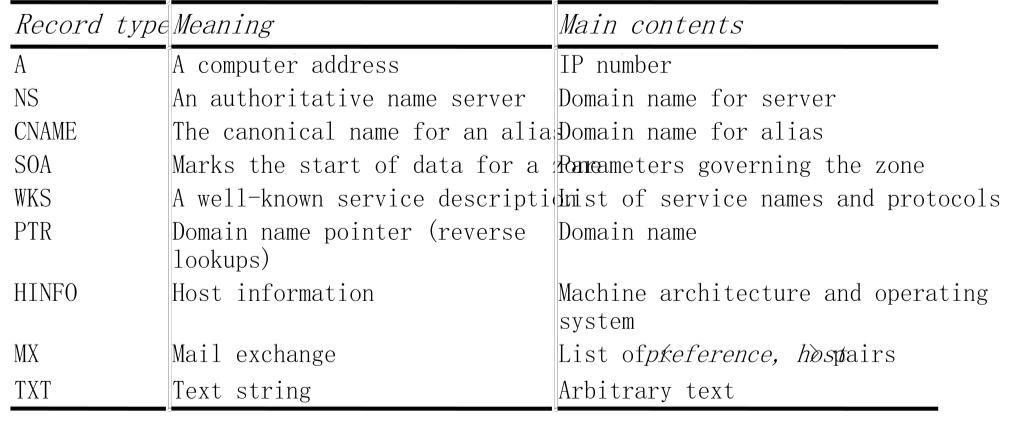
**DNS name servers:** The problems of scale are treated by a combination of partitioningthe naming database and replicating and caching parts of it close to the points of need. The DNS database is distributed across a logical network of servers. Each server holds part of the naming database – primarily data for the local domain. Queries concerning computers in the local domain are satisfied by servers within that domain. However, each server records the domain names and addresses of other name servers, so that queries pertaining to objects outside the domain can be satisfied.

The DNS naming data are divided into *zones*. A zone contains the following data:

1. Attribute data for names in a domain, less any subdomains administered by lower level authorities.
2. The names and addresses of at least two name servers that provide *authoritative* data for the zone. These are versions of zone data that can be relied upon as being reasonably up-to-date.
3. The names of name servers that hold authoritative data for delegated subdomains; and ‗glue‘ data giving the IP addresses of these servers.
4. Zone-management parameters, such as those governing the caching and replication of zone data.

A server may hold authoritative data for zero or more zones. So that naming data are available even when a single server fails, the DNS architecture specifies that each zone must be replicated authoritatively in at least two servers.

**Navigation and query processing:** A DNS client is called a*resolver*. It is normallyimplemented as library software. It accepts queries, formats them into messages in the form expected under the DNS protocol and communicates with one or more name servers in order to satisfy the queries. A simple request-reply protocol is used, typically using UDP packets on the Internet (DNS servers use a well-known port number). The resolver times out and resends its query if necessary. The resolver can be configured to contact a list of initial name servers in order of preference in case one or more are unavailable.



**Figure 14.2.2:** DNS resource records

The DNS architecture allows for recursive navigation as well as iterative navigation. The resolver specifies which type of navigation is required when contacting a name server. However, name servers are not bound to implement recursive navigation. In order to save on network communication, the DNS protocol allows for multiple queries to be packed into the same request message and for name servers correspondingly to send multiple replies in their response messages.

**Resource records:** Zone data are stored by name servers in files in one of several fixedtypes of resource record. For the Internet database, these include the types given in Figure 4.2.3. Each record refers to a domain name, which is not shown. The data for a zone starts with an *SOA*-type record, which contains the zone parameters that specify, for example, the version number and how often secondaries should refresh their copies. This is followed by a list of records of type *NS* specifying the name servers for the domain and a list of records of type *MX* giving the domain names of mail hosts, each prefixed by a number expressing its preference.

Further records of type *A* later in the database give the IP addresses for the two name servers *dns0* and *dns1*. The IP addresses of the mail hosts and the third name server are given in the databases corresponding to their domains.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *domain* | *name time to live* | *class* | *type* | *value* |
| *dcs.qmul.ac.uk* | *1D* | *IN* | *NS* | *dns0* |
| *dcs.qmul.ac.uk* | *1D* | *IN* | *NS* | *dns1* |
| *dcs.qmul.ac.uk* | *1D* | *IN* | *MX* | *1 mail1.qmul.ac.uk* |
| *dcs.qmul.ac.uk* | *1D* | *IN* | *MX* | *2 mail2.qmul.ac.uk* |

**Figure 14.2.3: DNS zone data records**

The majority of the remainder of the records in a lower-level zone like *dcs.qmul.ac.uk* will be of type *A* and map the domain name of a computer onto its IP address. They may contain some aliases for the well-known services,

for example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *domain name* | *time to live* | *class* | *Type* | *value* |
| *www* | *1D* | *IN* | *CNAME* | traffic |
| *traffic* | *1D* | *IN* | *A* | 138.37.95.150 |

If the domain has any sub-domains, there will be further records of type *NS* specifying their name servers, which will also have individual *A* entries. For example, at one point the database for *qmul.ac.uk* contained the following records for the name servers in its sub-domain *dcs.qmul.ac.uk*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *domain name* | *time to live* | *class* | *type* | *value* |
| *dcs* | *1D* | *IN* | *NS* | *dns0.dcs* |
| *dns0.dcs* | *1D* | *IN* | *A* | *138.37.88.249* |
| *dcs* | *1D* | *IN* | *NS* | *dns1.dcs* |
| *dns1.dcs* | *1D* | *IN* | *A* | *138.37.94.248* |

**Load sharing by name servers:** At some sites, heavily used services such as the Weband FTP are supported by a group of computers on the same network. In this case, the same domain name is used for each member of the group. When a domain name is shared by several computers, there is one record for each computer in the group, giving its IP address. By default, the name server responds to queries for which multiple records match the requested name by returning the IP addresses according to a round-robin schedule. Successive clients are given access to different servers so that the servers can share the workload. Caching has a potential for spoiling this scheme, for once a non-authoritative name server or a client has the server‘s address in its cache it will continue to use it. To counteract this effect, the records are given a short time to live.

**The BIND implementation of the DNS:** The Berkeley Internet Name Domain (BIND)is an implementation of the DNS for computers running UNIX. Client programs link in library software as the resolver. DNS name server computers run the named daemon.

BIND allows for three categories of name server: primary servers, secondary servers and caching-only servers. The named program implements just one of these types, according to the contents of a configuration file. The first two categories are described as above. Caching-only servers read in from a configuration file sufficient names and addresses of authoritative servers to resolve any name. Thereafter, they only store this data and data that they learn by resolving names for clients.

**DIRECTORY SERVICES**

We have described how name services store collections of <*name, attribute*> pairs, and how the attributes are looked up from a name. It is natural to consider the dual of this arrangement, in which *attributes* are used as values to be looked up. In these services, textual names can be considered to be just another attribute. Sometimes users wish to find a particular person or resource, but they do not know its name, only some of its other attributes.

A service that stores collections of bindings between names and attributes and that looks up entries that match attribute-based specifications is called a *directory service*. Examples are Microsoft‘s Active Directory Services, X.500 and its cousin LDAP, Univers and Profile.

Directory services are sometimes called *yellow pages services*, and conventional name services are correspondingly called *white pages services*, in an analogy with the traditional types of telephone directory. Directory services are also sometimes known as *attribute-based name services*.

UDDI aside, the term *discovery service* normally denotes the special case of a directory service for services provided by devices in a spontaneous networking environment.

Devices in spontaneous networks are liable to connect and disconnect unpredictably. One core difference between a discovery service and other directory services is that the address of a directory service is normally well known and preconfigured in clients, whereas a device entering a spontaneous networking environment has to resort to multicast navigation, at least the first time it accesses the local discovery service.

Attributes are clearly more powerful than names as designators of objects: programs can be written to select objects according to precise attribute specifications where names might not be known. Another advantage of attributes is that they do not expose the structure of organizations to the outside world, as do organizationally partitioned names. However, the relative simplicity of use of textual names makes them unlikely to be replaced by attribute-based naming in many applications.

**CASE STUDY: THE GLOBAL NAME SERVICE**

A Global Name Service (GNS) was designed and implemented by Lampson and colleagues at the DEC Systems Research Center to provide facilities for resource location, mail addressing and authentication. The design goals of the GNS reflect the fact that a name service for use in an internetwork must support a naming database that may extend to include the names of millions of computers and (eventually) email addresses for billions of users. The designers of the GNS also recognized that the naming database is likely to have a long lifetime and that it must continue to operate effectively while it grows from small to large scale and while the network on which it is based evolves. The structure of the name space may change during that time to reflect changes in organizational structures.

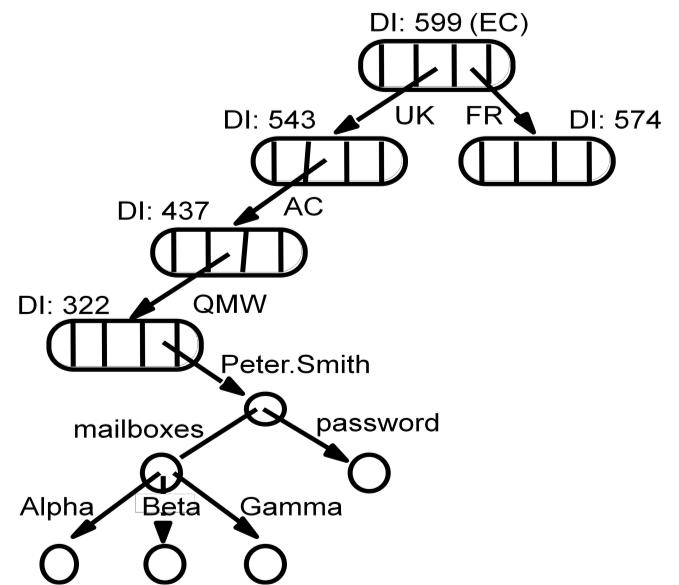
The service should accommodate changes in the names of the individuals, organizations and groups that it holds, and changes in the naming structure.

The potentially large naming database and the scale of the distributed environment in which the GNS is intended to operate make the use of caching essential and render it

extremely difficult to maintain complete consistency between all copies of a database entry. The GNS manages a naming database that is composed of a tree of directories holding names and values. Each directory is also assigned an integer, which serves as a unique *directory identifier* (DI).

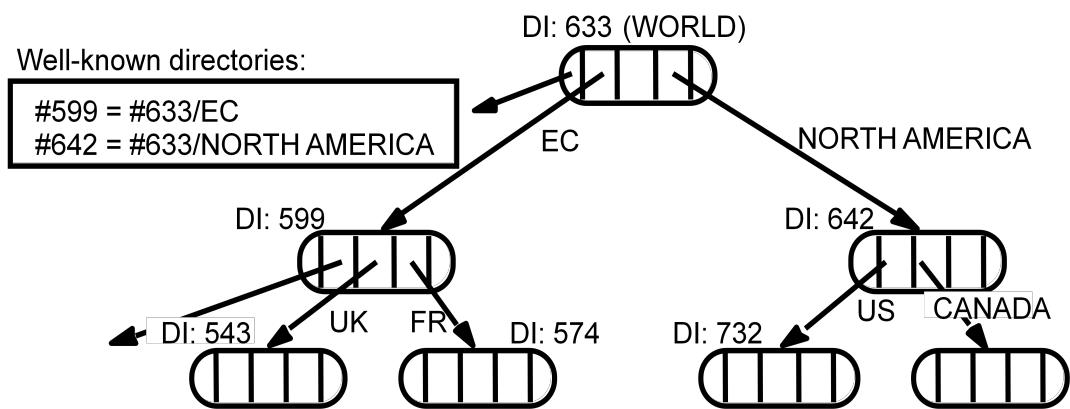
A directory contains a list of names and references. The values stored at the leaves of the directory tree are organized into *value trees*, so that the attributes associated with names can be structured values. Names in the GNS have two parts: <*directory name*, *value* *name*>. The first part identifies a directory; the second refers to a value tree, or someportion of a value tree. For example, see Figure 4.2.4, in which the DIs are illustrated as small integers

The directory tree is partitioned and stored in many servers, with each partition replicated in several servers. The consistency of the tree is maintained in the face of two or more concurrent updates – for example, two users may simultaneously attempt to create entries with the same name, and only one should succeed. Replicated directories present a second consistency problem; this is addressed by an asynchronous update distribution algorithm that ensures eventual consistency, but with no guarantee that all copies are always current.



**Figure 14.2.4: GNS directory tree and value tree for user Peter.Smith**

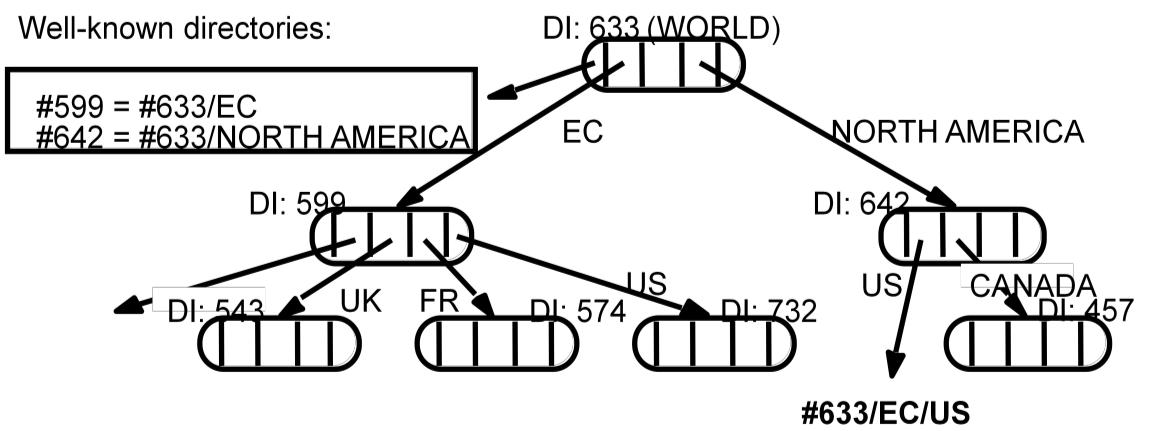
**Accommodating change:** We now turn to the aspects of the design that are concernedwith accommodating growth and change in the structure of the naming database. At the level of clients and administrators, growth is accommodated through extension of the directory tree in the usual manner. But we may wish to integrate the naming trees of two previously separate GNS services.



**Figure 14.2.5** Merging trees under a new root

The working root for each program must be identified as part of its execution environment (much as is done for a program‘s working directory).

The GNS also supports the restructuring of the database to accommodate organizational change. Figure 4.2.6 shows the new directory tree.



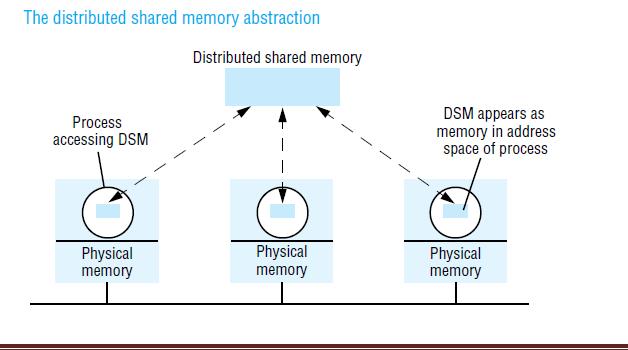
**Figure 4.2.6: Restructuring the directory**



**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).

Message passing cannot be avoided altogether in a distributed system: in the absence of 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**

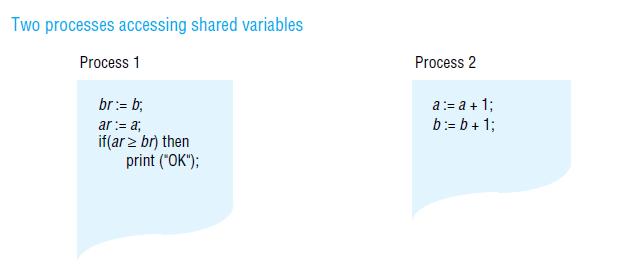
A DSM system is just such a replication system. Each application process is presented with some abstraction of a collection of objects, but in this case the ‘collection’ looks more or less like memory. That is, the objects can be addressed in some fashion or other. Different approaches to

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 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.

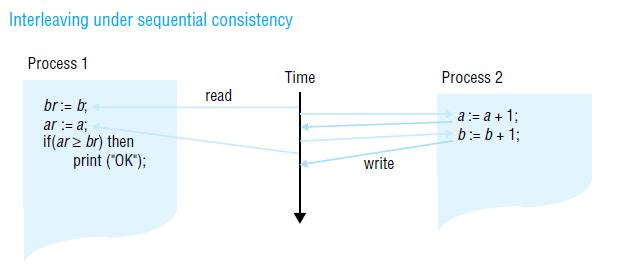
**Consistency model**

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.

**Sequential consistency**

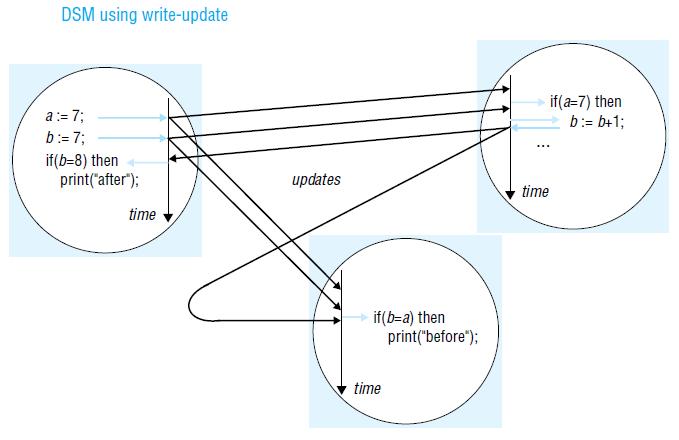
A DSM system is said to be sequentially consistent if *for any execution* there is some interleaving of the series of operations issued by all the processes that satisfies the following two criteria:

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.

**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 replicamanagers 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-writersharing. 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.

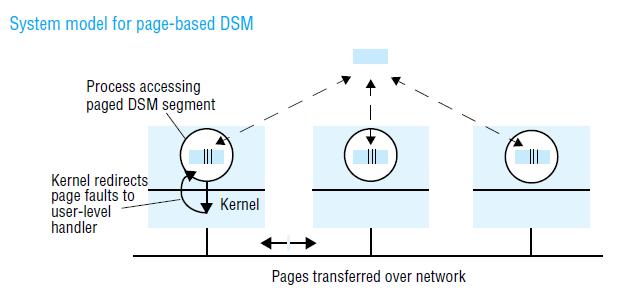
**Thrashing**

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.

**SEQUENTIAL CONSISTENCY AND IVY CASE STUDY**

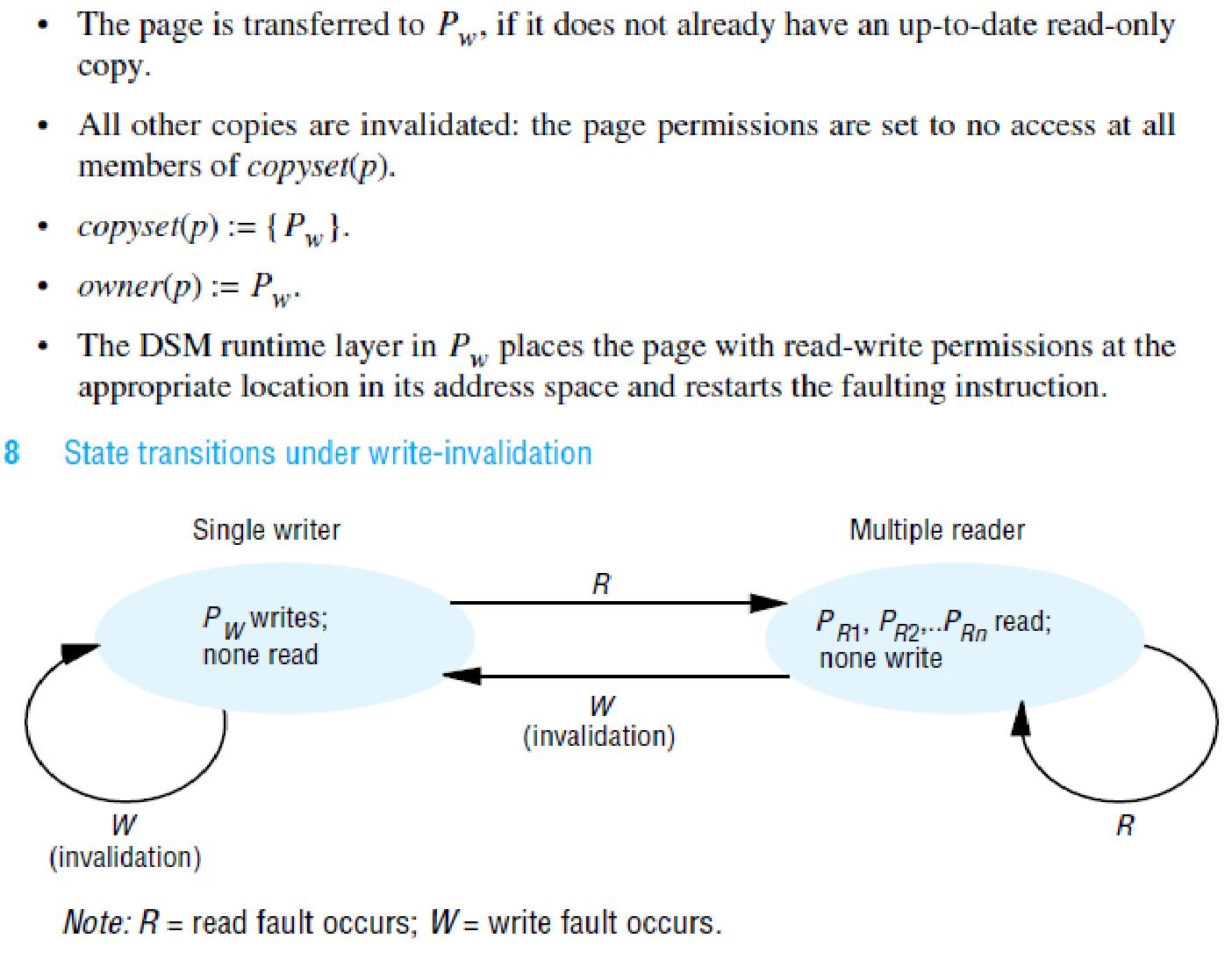
**The system model**

The basic model to be considered is one in which a collection of processes shares a segment of DSM. The segment is mapped to the same range of addresses in each process, so that meaningful pointer values can be stored in the segment. The processes execute at computers equipped with a paged memory management unit. We shall assume that there is only one process per computer that accesses the DSM segment. There may in reality be several such processes at a computer. However, these could then share DSM pages directly (the same page frame can be used in the page tables used by the different processes). The only complication would be to coordinate fetching and propagating updates to a page when two or more local processes access it. This description ignores such details.



Paging is transparent to the application components within processes; they can logically both read and write any data in DSM. However, the DSM runtime restricts page access permissions in order to maintain sequential consistency when processing reads and writes. Paged memory management units allow the access permissions to a data page to be set to none, read-only or read-write.

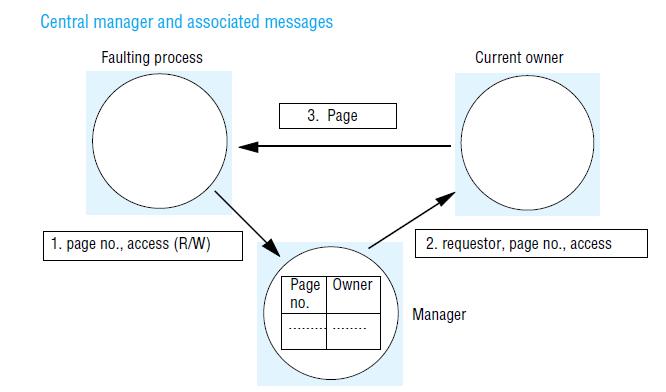
**The problem of write-update:** The previous section outlined the general implementation alternatives of write-update and write-invalidation. In practice, if the DSM is page-based, then write-update is used only if writes can be buffered. This is because standard page-fault handling is unsuited to the task of processing every single write update to a page.

**Write invalidation:** Invalidation-based algorithms use page protection to enforce consistent data sharing. When a process is updating a page, it has read and write permissions locally; all other processes have no access permissions to the page. When one or more processes are reading the page, they have read-only permission; all other processes have no access permissions (although they may acquire read permissions). No other combinations are possible.

**Invalidation protocols**

Two important problems remain to be addressed in a protocol to implement the invalidation scheme:

1. How to locate *owner*(*p*) for a given page *p*.
2. Where to store *copyset*(*p*).

For Ivy, Li and Hudak [1989] describe several architectures and protocols that take varying approaches to these problems. The simplest we shall describe is their improved centralized manager algorithm. In it, a single server called a manager is used to store the location (transport address) of *owner*(*p*) for every page *p*. The manager could be one of the processes running the application, or it could be any other process. In this algorithm, the set *copyset*(*p*) is stored at *owner*(*p*). That is, the identifiers and transport addresses of the members of *copyset*(*p*) are stored.

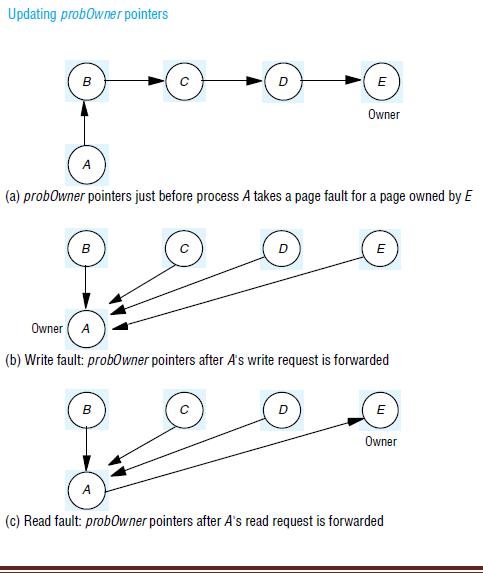
**Using multicast to locate the owner**

Multicast can be used to eliminate the manager completely. When a process faults, it multicasts its page request to all the other processes. Only the process that owns the page replies. Care must be taken to ensure correct behaviour if two clients request the same page at more or less the same time: each client must obtain the page eventually, even if its request is multicast during transfer of ownership.

**A dynamic distributed manager algorithm**

The owner of a page is located by following chains of hints that are set up as ownership of the page is transferred from computer to computer. The length of the chain – that is, the number of forwarding messages necessary to locate the owner – threatens to increase indefinitely. The algorithm overcomes this by updating the hints as more upto- date values become available. Hints are updated and requests are forwarded as follows:

* When a process transfers ownership of page *p* to another process, it updates *probOwner*(*p*) to be the recipient.
* When a process handles an invalidation request for a page *p*, it updates *probOwner*(*p*) to be the requester.
* When a process that has requested read access to a page *p* receives it, it updates *probOwner*(*p*) to be the provider.
* When a process receives a request for a page *p* that it does not own, it forwards the request to *probOwner*(*p*) and resets *probOwner*(*p*) to be the requester.

The first three updates follow simply from the protocol for transferring page ownership and providing read-only copies. The rationale for the update when forwarding requests is that, for write requests, the requester will soon be the owner, even though it is not currently. In fact, in Li and Hudak’s algorithm, assumed here, the *probOwner* update is made whether the request is for read access or write access.

**Thrashing:** It can be argued that it is the programmer’s responsibility to avoid thrashing. The programmer could annotate data items in order to assist the DSM runtime in minimizing page copying and ownership transfers. The latter approach is discussed in the next section in the context of the Munin DSM system.

**RELEASE CONSISTENCY AND MUNIN CASE STUDY:** Release consistency was introduced with the Dash multiprocessor, which implements DSM in hardware, primarily using a write-invalidation protocol [Lenoski *etal*. 1992]. Munin and Treadmarks [Keleher *et al.* 1992] have adopted a software implementation of it. Release consistency is weaker than sequential consistency and cheaper to implement, but it has reasonable semantics that are tractable to programmers.

The idea of release consistency is to reduce DSM overheads by exploiting the fact that programmers use synchronization objects such as semaphores, locks and barriers. A DSM implementation can use knowledge of accesses to these objects to allow memory to become inconsistent at certain points, while the use of synchronization objects nonetheless preserves application-level consistency.

**Memory accesses**

In order to understand release consistency – or any other memory model that takes synchronization into account – we begin by categorizing memory accesses according to their role, if any, in synchronization. Furthermore, we shall discuss how memory accesses may be performed asynchronously to gain performance and give a simple operational model of how memory accesses take effect. As we said above, DSM implementations on general-purpose distributed systems may use message passing rather than shared variables to implement synchronization, for reasons of efficiency.

***acquireLock(var int lock)*: // *lock is passed by-reference while* (*testAndSet*(*lock*) = 1)**

***skip*;**

***releaseLock(var int lock)*: // *lock is passed by-reference***

***lock* :*=* 0;**

**Types of memory access**

The main distinction is between *competing* accesses and *noncompeting* (*ordinary*) accesses. Two

accesses are competing if:

* they may occur concurrently (there is no enforced ordering between them) and
* at least one is a *write*.

So two *read* operations can never be competing; a *read* and a *write* to the same location made by two processes that synchronize between the operations (and so order them) are non-competing. We further divide competing accesses into *synchronization* and *nonsynchronization* accesses:

* synchronization accesses are *read* or *write* operations that contribute to synchronization;
* non-synchronization accesses are *read* or *write* operations that are concurrent but that do not contribute to synchronization.

**Performing asynchronous operations**

In view of the asynchronous operation that we have outlined, we distinguish between the point at which a *read* or *write* operation is issued – when the process first commences execution of the operation – and the point when the instruction is *performed* or completed.

We shall assume that our DSM is at least coherent. It means that every process agrees on the order of *write* operations to the same location. Given this assumption, we may speak unambiguously of the ordering of *write* operations to a given location.

**Release consistency**

The requirements that we wish to meet are:

* to preserve the synchronization semantics of objects such as locks and barriers;
* to gain performance, we allow a degree of asynchronicity for memory operations;
* to constrain the overlap between memory accesses in order to guarantee executions that provide the equivalent of sequential consistency.

**Munin**

The Munin DSM design [Carter *et al*. 1991] attempts to improve the efficiency of DSM by implementing the release consistency model. Furthermore, Munin allows programmers to annotate their data items according to the way in which they are shared, so that optimizations can be made in the update options selected for maintaining consistency. It is implemented upon the V

kernel [Cheriton and Zwaenepoel 1985], which was one of the first kernels to allow user-level threads to handle page faults and manipulate page tables.

The following points apply to Munin’s implementation of release consistency:

* Munin sends update or invalidation information as soon as a lock is released.
* The programmer can make annotations that associate a lock with particular data items. In this case, the DSM runtime can propagate relevant updates in the same message that transfers the lock to a waiting process – ensuring that the lock’s recipient has copies of the data it needs before it accesses them.

**Sharing annotations**

Munin implements a variety of consistency protocols, which are applied at the granularity of individual data items. The protocols are parameterized according to the following options:

* whether to use a write-update or write-invalidate protocol;
* whether several replicas of a modifiable data item may exist simultaneously;
* whether or not to delay updates or invalidations (for example, under release consistency);
* whether the item has a fixed owner, to which all updates must be sent;
* whether the same data item may be modified concurrently by several writers;
* whether the data item is shared by a fixed set of processes;
* whether the data item may be modified.

*Read-only*: No updates may be made after initialization and the item may be freely copied.

*Migratory*: Processes typically take turns in making several accesses to the item, at least one ofwhich is an update. For example, the item might be accessed within a critical section. Munin always gives both read and write access together to such an object, even when a process takes a read fault. This saves subsequent write-fault processing.

*Write-shared*: Several processes update the same data item (for example, an array) concurrently,but this annotation is a declaration from the programmer that the processes do not update the same parts of it. This means that Munin can avoid false sharing but must propagate only those words in the data item that are actually updated at each process. To do this, Munin makes a copy of a page (inside a write-fault handler) just before it is updated locally. Only the differences between the two versions are sent in an update.

*Producer-consumer*: The data object is shared by a fixed set of processes, only one of whichupdates it. As we explained when discussing thrashing above, a writeupdate protocol is most suitable here. Moreover, updates may be delayed under the model of release consistency, assuming that the processes use locks to synchronize their accesses.

*Reduction:* The data item is always modified by being locked, read, updated and unlocked. Anexample of this is a global minimum in a parallel computation, which must be fetched and modified atomically if it is greater than the local minimum. These items are stored at a fixed owner. Updates are sent to the owner, which propagates them.

*Result:* Several processes update different words within the data item; a single process reads thewhole item. For example, different ‘worker’ processes might fill in different elements of an array, which is then processed by a ‘master’ process. The point here is that the updates need only be propagated to the master and not to the workers (as would occur under the ‘write-shared’ annotation just described).

*Conventional:* The data item is managed under an invalidation protocol similar to that describedin the previous section. No process may therefore read a stale version of the data item.

**OTHER 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 . Thisis 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.

*Pipelined RAM*: All processors agree on the order of writes issued by any given processorIn addition to release consistency, hybrid models include:

*Entry consistency*: Entry consistency was proposed for the Midway DSM system. In this model,every shared variable is bound to a synchronization object such as a lock, which governs access to that variable. Any process that first acquires the lock is guaranteed to read the latest value of the variable. A process wishing to write the variable must first obtain the corresponding lock in

‘exclusive’ mode – making it the only process able to access the variable.

Several processes may read the variable concurrently by holding the lock in nonexclusive mode. Midway avoids the tendency to false sharing in release consistency, but at the expense of increased programming complexity.

*Scope consistency*: This memory model [Iftode *et al.* 1996] attempts to simplify theprogramming model of entry consistency. In scope consistency, variables are associated with synchronization objects largely automatically instead of relying on the programmer to associate locks with variables explicitly. For example, the system can monitor which variables are updated in a critical section.

*Weak consistency*: Weak consistency [Dubois *et al.* 1988] does not distinguish between *acquire* and *release* synchronization accesses. One of its guarantees is that all previous ordinary accesses complete before *either* type of synchronization access completes.

**Transactions & Concurrency Control**

Optimistic Concurrency Control

• Principle

– transaction proceeds without checking conflict with others and

prior to commit, *validates* its change by checking to see if data

items have changed by *committed transactions*

– each transaction has three phases

􀂊 Read phase

􀂃 committed version of data items for read - *read set*

􀂃 tentative version of data items for write - *write set*

􀂊 Validation phase

􀂃 starts with EndTransaction request

􀂃 validate its change by checking to see if data items have changed by

other transactions

􀂃 if no conflicts, commit; otherwise, abort

􀂊 Write phase

􀂃 make changes permanent

Distributed Systems - Transactions & Concurrency Control (2/2)

• Validation test rule

– T*j* is *serializable* with respect to overlapping T*i* if their operations

conform to the following rules

– *transaction # is sequentially assigned when validation phase starts*

• Validation mechanisms

– backward validation

– forward validation

T*i* T*j* Rule

*Read Write* 1. T*i* must not read data items written by T*j*

*Write Read* 2. T*j* must not read data items written by T*i*

*Write Write* 3. T*i* must not write data items written by T*j* and

T*j* must not write data items written by T*i*

(Assumption: T*i* always preceeds T*j* if *i* < *j* and T*i* overlaps with T*j*)

Distributed Systems - Transactions & Concurrency Control (2/2)

• Backward validation

– algorithm

􀂊 checks transaction in validation phase with other preceding

overlapping transactions that have entered validation phase

􀂃 *Write* operations are ok since Read operations of earlier transactions are

done already (Rule1)

􀂃 check if *Read* operations have any conflict with *Write* operations of

earlier overlapping transactions (Rule 2) => if yes, abort transaction

Valid := True;

for Ti := startTn + 1 to finishTn

do

if read set of Tj intersects write set of Ti

Valid := False;

end

– no check is needed for transaction with only *Write* operations

Distributed Systems - Transactions & Concurrency Control (2/2)

Earlier committed

transactions

Working Validation Update

T1

T*v*

Transaction

being validated

T2

T3

Check if read set of T*v* conflicts with

the write sets of the preceding

overlapping transactions that have

entered validation phase

• Backward validation example

Distributed Systems - Transactions & Concurrency Control (2/2)

• Forward validation

– algorithm

􀂊 checks transaction in validation phase with other overlapping active

transactions

􀂃 *Read* operations are ok since later transactions do not write until the Tj is

done (Rule 2)

􀂃 check if *Write* operations have any conflict with *Read* operations of

overlapping active transactions (Rule 1) => if yes, abort transaction

Valid := True;

for Ti := active1 to activeN

do

if write set of Tj intersects read set of Ti

Valid := False;

*end*

– no check is needed for transaction with only *Read* operations

– other options than aborting the current transaction

􀂊 defer validation until conflicting transaction is done

􀂊 abort conflicting transaction instead

Distributed Systems - Transactions & Concurrency Control (2/2)

Optimistic Concurrency Control (cont.)

T*v*

Transaction

being validated

Later active

transactions

active1

active2

• Forward validation example

Check if write set of T*v* conflicts with

the read sets of the overlapping

active transactions

Distributed Systems - Transactions & Concurrency Control (2/2)

• Issues in optimistic concurrency control

– Overhead

􀂊 Backward validation

􀂃 if there exists long transaction, retention of old write sets of data item

may be a problem

􀂊 Forward validation

􀂃 a new transaction can start during the validation process -> increase

chances by which the current transaction is forced to abort or delay

– Starvation

􀂊 prevention of a transaction ever being able to commit

Optimistic Concurrency Control (cont.)

Distributed Systems - Transactions & Concurrency Control (2/2)

Timestamp Ordering

• Assumption

– each transaction is given a unique timestamp when it starts

– there is only one version of data item and only one transaction can

access it at a time => multiple tentative versions of data to increase

concurrency

• Rule

– *Write* operation is valid only if the data was last read and written

by earlier transaction

􀂊 *Rule1*: Tj must not write data item read by any Ti where Ti > Tj (i.e. Tj >=

max read time stamp of data item)

􀂊 *Rule 2*: Tj must not write data item written by any Ti where Ti > Tj (i.e. Tj >

max write time stamp of committed data item)

– *Read* operation is valid only if the data was last written by earlier

transaction

􀂊 *Rule 3*: Tj must not read data item written by T*i* where Ti > Tj (i.e. Tj > write

time stamp of committed data item)

Distributed Systems - Transactions & Concurrency Control (2/2)

Timestamp Ordering (cont.)

• Write operations and time stamp

if (*Tc* ≥ maximum read timestamp on *D* &&

*Tc* > write timestamp on committed version of *D*)

perform write operation on tentative version of *D* with write timestamp *Tc*

else /\* write is too late \*/

Abort transaction *Tc*

Timestamp Ordering (cont.)

• Read operations and time stamp

if *( Tc* > write timestamp on committed version of *D*) {

let *D*selected be the version of *D* with the maximum write timestamp ≤ *Tc*

if (Dselected 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 *Tc*

Distributed Systems - Transactions & Concurrency Control (2/2)

• Multi-version timestamp ordering

– keep old versions of committed data as well as tentative versions

􀂊 read operation is always allowed; may need to wait for earlier

transactions to complete

􀂊 no conflict between write operations since each transaction writes its

own committed version (remove rule 2)

– write rule

􀂊 if read time stamp (most recent version) <= Tj then perform write

operation on a tentative version with write time stamp Tj

Distributed Systems - Transactions & Concurrency Control (2/2)

• Multi-version timestamp ordering - example

Time

T3 read; T3 write; T5 read; T4 write;

T2

T3 T5

T1

T3

T1 < T2 < T3 < T4 < T5

Key:

Committed Tentative

T*i* T*i* T*k* T*k*

object produced by transaction

Ti (with write timestamp Ti and

read timestamp Tk)

Distributed Systems - Transactions & Concurrency Control (2/2)

Comparison

• Locking vs. timestamp ordering

– both are pessimistic

– dynamic vs static ordering

– write-dominated vs. read-dominated

• Optimistic

– efficient when there are few conflicts

• New requirements to concurrency control

– multi-user applications

􀂊 immediate notification of change (relaxed isolation)

􀂊 need to be able to access uncommitted data item

– co-operative CAD/CAM

􀂊 co-operations of users to resolve data conflicts

Forward validation

– algorithm

􀂊 checks transaction in validation phase with other overlapping active

transactions

􀂃 *Read* operations are ok since later transactions do not write until the Tj is

done (Rule 2)

􀂃 check if *Write* operations have any conflict with *Read* operations of

overlapping active transactions (Rule 1) => if yes, abort transaction

Optimistic Concurrency Control (cont.)

Valid := True;

for Ti := active1 to activeN

do

if write set of Tj intersects read set of Ti

Valid := False;

*end*

– no check is needed for transaction with only *Read* operations

– other options than aborting the current transaction

􀂊 defer validation until conflicting transaction is done

􀂊 abort conflicting transaction instead

Issues in optimistic concurrency control

– Overhead

􀂊 Backward validation

􀂃 if there exists long transaction, retention of old write sets of data item

may be a problem

􀂊 Forward validation

􀂃 a new transaction can start during the validation process -> increase

chances by which the current transaction is forced to abort or delay

– Starvation

􀂊 prevention of a transaction ever being able to commit

Timestamp Ordering

• Assumption

– each transaction is given a unique timestamp when it starts

– there is only one version of data item and only one transaction can

access it at a time => multiple tentative versions of data to increase

concurrency

• Rule

– *Write* operation is valid only if the data was last read and written

by earlier transaction

􀂊 *Rule1*: Tj must not write data item read by any Ti where Ti > Tj (i.e. Tj >=

max read time stamp of data item)

􀂊 *Rule 2*: Tj must not write data item written by any Ti where Ti > Tj (i.e. Tj >

max write time stamp of committed data item)

– *Read* operation is valid only if the data was last written by earlier

transaction

􀂊 *Rule 3*: Tj must not read data item written by T*i* where Ti > Tj (i.e. Tj > write

time stamp of committed data item)

Write operations and time stamp

if (*Tc* ≥ maximum read timestamp on *D* &&

*Tc* > write timestamp on committed version of *D*)

perform write operation on tentative version of *D* with write timestamp *Tc*

else /\* write is too late \*/

Abort transaction *Tc*

*(a) write write*

*(c) T3 write* object produced by

transaction Ti (with

write timestamp Ti)

*T3 (b) T3*

*(d) T3 write*

T1<T2<T3<T4

Time

Before

After

T2

T2 T3

Time

Before

After

T2

T2 T3

T1

T1

Time

Before

After

T1

T1

T4

T3 T4

Time

Transaction

Read operations and time stamp

if *( Tc* > write timestamp on committed version of *D*) {

let *D*selected be the version of *D* with the maximum write timestamp ≤ *Tc*

if (Dselected 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 *Tc*

Multi-version timestamp ordering

– keep old versions of committed data as well as tentative versions

􀂊 read operation is always allowed; may need to wait for earlier

transactions to complete

􀂊 no conflict between write operations since each transaction writes its

own committed version (remove rule 2)

– write rule

􀂊 if read time stamp (most recent version) <= Tj then perform write

operation on a tentative version with write time stamp Tj

**Comparison**

• Locking vs. timestamp ordering

– both are pessimistic

– dynamic vs static ordering

– write-dominated vs. read-dominated

• Optimistic

– efficient when there are few conflicts

• New requirements to concurrency control

– multi-user applications

􀂊 immediate notification of change (relaxed isolation)

􀂊 need to be able to access uncommitted data item

– co-operative CAD/CAM

􀂊 co-operations of users to resolve data conflicts

**Distributed Transactions**

**distributed transaction** is a [database transaction](https://en.wikipedia.org/wiki/Database_transaction) in which two or more network hosts are involved. Usually, hosts provide **transactional resources**, while the **transaction manager** is responsible for creating and managing a global transaction that encompasses all operations against such resources. Distributed transactions, as any other [transactions](https://en.wikipedia.org/wiki/Database_transaction), must have all four [ACID (atomicity, consistency, isolation, durability)](https://en.wikipedia.org/wiki/ACID) properties, where atomicity guarantees all-or-nothing outcomes for the unit of work (operations bundle).

We've looked at a number of low level techniques that can be used for managing synchronization in a distributed environment: algorithms for mutual exclusion and critical section management. In addition (and we'll look at these later), we can have algorithms for deadlock resolution and crash recovery. Much as remote procedure calls allowed us to concentrate on the functionality of a program and express it in a more natural way than sends and receives, we crave a higher level of abstraction in dealing with issues of synchronization. This brings us to the topic of **atomic transactions** (also known colloquially simply as *transactions*).

In transacting business, all parties involved may have to go through a number of steps in negotiating a contract but the end result of the transaction won't be *committed* until both parties sign on the dotted line. If even one of the parties reconsiders and *aborts*, the contract will be forgotten and life goes on as before.

Consider, for example, the purchase of a house. You express your interest in purchasing a house by making an offer (and possibly putting some money down with a trusted party). At that point, you have not bought the house, but you have entered the transaction of purchasing a house. You may have things to do (such as getting a mortgage and inspection) and the seller may have things to do (such as fixing up certain flaws). If something goes wrong (you can't get a mortgage, the seller won't fix the heating system, you find the house is sitting on a fault line, the seller won't remove the black velvet wallpaper, ...), then the transaction is cancelled (*aborted*) and both parties go back to life as before: you look for another house and the seller remains in the house, possibly still trying to sell it. If, however, the transaction is not aborted and both parties sign the contract on the closing day, it is made permanent. The deed is signed over and you own the house. If the seller changes her mind at this point, she'll have to try to buy back the house. If you change your mind, you'll have to sell the house.

The concept of a transaction in the realm of computing is quite similar. One process announces that it's beginning a transaction with one or more processes. Certain actions take place. When *all* processes **commit**, the results are permanent. Until they do so, any process may **abort** (if something fails, for example). In that case, the state of computing reverts to the state before the transaction began: all side effects are gone. A transaction has an *all or nothing* property.

The origins of transactions in computing date back to the days of batch jobs scheduled to processes tapes. A days worth of "transactions" would be logged on a tape. At the end of the day, a merge job would be run with the original database tape and the transactions tape as inputs, producing a new tape with all the transactions applied. If anything went wrong, the original database tape was unharmed. If the merge succeeded, then the original tapes could be reused.

**Transaction model**

A process that wishes to use transactions must be aware of certain **primitives** associated with them. These primitives are:

1. *begin transaction* - mark the start
2. *end transaction* - mark the end; try to commit
3. *abort transaction* - kill transaction, restore old values
4. *read* data from object(file), *write* data to object(file).

In addition, ordinary statements, procedure calls, etc. are allowed in a transaction.

To get a flavor for transactions, consider booking a flight from Newark, New Jersey to Ridgecrest, California. The destination requires us to land at Inyokern airport, and non-stop flights are not available:

transaction begin

1. reserve a seat for Newark to Denver (EWK→DEN)

2. reserve a seat for Denver to Los Angeles (DEN→LAX)

3. reserve a seat for Los Angeles to Inyokern (LAX→IYK)

transaction end

Suppose there are no seats available on the LAX→IYK leg of the journey. In this case, the transaction is aborted, reservations for (1) and (2) are undone, and the system reverts to the state before the reservation was made.

**Properties of transactions**

The properties of transactions are summarized with the acronym **ACID**, which stands for **A**tomic, **C**onsistent, **I**solated, and **D**urable.

**Atomic**

either an entire transaction happens completely or not at all. If the transaction does happen, it happens as a single *indivisible* action. Other processes cannot see intermediate results. For example, suppose we have a file that is 100 bytes long and a transaction begins appending to it. If other processes read the file, they only see the 100 bytes. At the end of the transaction, the file *instantly* grows to its new size.

**Consistent**

If the system has certain invariants, they must hold after the transaction (although they may be broken within the transaction). For example, in some banking application, the invariant may be that the amount of money before a transaction must equal the amount of money after the transaction. Within the transaction, this invariant may be violated but this is not visible outside the transaction.

**Isolated** (or **serializable**)

If two or more transactions are running at the same time, to each of them and to others, the *final result* looks as though all transactions ran sequentially in *some* order.

An order of running transactions is called a **schedule**. Orders may be interleaved. If no interleaving is done and the transactions are run in some sequential order, they are **serialized**.

Consider the following three (small) transactions:

|  |  |  |
| --- | --- | --- |
| begin  x=0  x=x+1  end | begin  x=0  x=x+2  end | begin  x=0  x=x+3  end |

Some possible schedules are (with time flowing from left to right):

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *schedule* | *execution order* | | | | | | final x | legal? |
| schedule 1 | x=0 | x=x+1 | x=0 | x=x+2 | x=0 | x=x+3 | 3 | yes |
| schedule 1 | x=0 | x=0 | x=x+1 | x=x+2 | x=0 | x=x+3 | 3 | yes |
| schedule 1 | x=0 | x=0 | x=x+1 | x=0 | x=x+2 | x=x+3 | 5 | NO |

**Durable**

Once a transaction commits, the results are made permanent. No failure after a commit can undo results or cause them to get lost. [Conversely, the results are *not* permanent until a transaction commits.]

**Nested transactions**

Transactions may themselves contain subtransactions (nested transactions). A top-level transaction may fork off children that run in parallel with each other. Any or all of these may execute subtransactions.

The problem with this is that the subtransactions may commit but, later in time, the parent may abort. Now we find ourselves having to undo the committed transactions. The level of nesting (and hence the level of undoing) may be arbitrarily deep. For this to work, conceptually, each subtransaction must be given a *private copy of every object* it may manipulate. On commit, the private copy displaces its parent's universe (which may be a private copy of that parent's parent).

**Implementation**

We cannot just allow a transaction to update the objects (files, DB records, et cetera) that it uses. The transactions won't be atomic in that case. One way of supporting this is by providing a **private workspace**. When a process starts a transaction, it's given a private workspace containing all the objects to which it has access. On a commit, the private workspace becomes the real workspace. Clearly this is an expensive proposition. It requires us to copy everything that the transaction may modify (every file, for example). However, it's not as bleak as it looks. A number of optimizations can make this a feasible solution.

Suppose that a process (transaction) reads a file but doesn't modify it. In that case it doesn't need a copy. The private workspace can be empty except that it contains a pointer back to the parent's workspace. How about writing a file? On an open, don't copy the file to the private workspace but just copy the index (information of where the file's data is stored; a UNIX inode, for example). The file is then read in the usual way. When a block is modified, a local copy is made and the address for the copied block is inserted into the index. New blocks (appends) work this way too. Privately allocated blocks are called **shadow blocks**.

If this transaction was to abort, the private blocks go back on the free list and the private space is cleaned up. Should the transaction commit, the private indices are moved into the parent's workspace (atomically). Any parent blocks that would be overwritten are freed.

Another mechanism for ensuring that transactions can be undone (and possibly redone) is the use of a **write-ahead log**, also known as an **intentions list**. With this system, objects are modified in place (proper locking should be observed if other processes are to access these objects). Before any data (e.g. block, memory page) is changed, a record is written to the write-ahead log in stable storage. The record identifies the transaction (with an ID number), the block or page modified, and the old and new values.

If the transaction succeeds (i.e., commits), a commit record is written to the log. If the transaction aborts, the log is used to back up to the original state (this is called a **rollback**. The write-ahead log can also be played forward for crash recovery (this becomes useful in the two-phase commit protocol, which is discussed next). A term associated with the write-ahead log was **stable storage**. This is intended to be a data repository that can survive system crashes. After a datum is written to stable storage, it is retrievable even if the system crashes immediately after the write. A disk is suitable for stable storage, but it is important that any writes are immediately flushed to the disk and not linger in the memory (unstable) buffer cache.

**The two-phase commit protocol**

(Gray, 1978)

In a distributed system, a transaction may involve multiple processes on multiple machines. Even in this environment, we still need to preserve the properties of transactions and achieve an atomic commit (either all processes involved in the transaction commit or else all of them will abort the transaction - it will be unacceptable to have some commit and some abort). A protocol that achieves this atomic commit is the **two-phase commit protocol**.

In implementing this protocol, we assume that one process will function as the coordinator and the rest as cohorts (the coordinator may be the one that initiated the transaction, but that's not necessary). We further assume that there is stable storage and a write-ahead log at each site. Furthermore, we assume that no machine involved crashes forever.

The protocol works as follows (the coordinator is ready to commit and needs to ensure that everyone else will do so as well):

|  |  |  |
| --- | --- | --- |
| *phase* | *coordinator* | *cohort* |
| 1  *request* | write *prepare to commit* message to the log | work on transaction; when done, wait for message |
| send *prepare to commit* message |  |
| wait for reply | receive message. When transaction is ready to commit, write *agree to commit* (or *abort*) to log. |
| send "agree" or "abort" reply |
| 2  *commit* | write *commit* message to the log. | *wait for commit message* |
| send *commit* (or *abort*) message | receive *commit* (or *abort*) message |
| wait for ***all*** cohorts to respond | if a *commit* was received, write "commit" to the log, release all locks & resources, update databases.  if an *abort* was received, undo all changes. |
| send *done* message. |
| clean up all state. Done. |  |

What the two phase commit protocol does is this. In phase 1, the coordinator sends a request to commit to all the cohorts and waits for a reply from *all* of them. The reply is either an agreement or an abort. Note that nobody has committed at this point. After the coordinator receives a reply from all cohorts, it knows that all transaction-relevant computation is finished so nothing more will happen to abort the transaction. The transaction can now be committed or, in the case that at lease one of the parties could not complete its transaction, aborted. The second phase is to wait for all cohorts to commit (or abort). If aborting, an abort message is sent to everyone. The coordinator waits until *every* cohort responds with an acknowledgement. If committing, a cohort receives a *commit* message, commits locally, and sends an acknowledgment back. All message deliveries are reliable (retransmits after time-out).

No formal proof will be given here of the correctness of the two-phase protocol. Inspecting for correctness, it is readily apparent that if one cohort completes the transaction, *all* cohorts will complete if eventually. If a cohort is completing a transaction, it is because it received a *commit* message, which means that we're in the commit phase and all cohorts have agreed. This information is in permanent memory in case of a crash (that's why information is written to the log before a message is sent. If any system crashes, it can replay its log to find its latest state (so it will know if it was ready to commit, for example). When the coordinator is completing, it is ensured that *every* cohort completes before the coordinator's data is erased(update).

**some vocabulary**

**abort**

transaction will not complete (commit). All changes are undone to the state before the transaction started.

**commit**

action which indicates that the transaction has successfully completed. All changes to the database, files, and objects are made permanent.

**commit protocol**

a fault-tolerant algorithm which ensures that all sides in a distributed system either commit or abort a transaction unanimously.

**log**

a record of system activity recorded in sufficient detail so that a previous state of a process can be restored.

**redo**

given a log record, redo the action specified in the log.

**stable storage**

permanent storage to which we can do atomic writes.

**transaction**

an atomic action which is some computation that read and/or changes the state of one or more data objects and appears to take place indivisibly.

**write-ahead log protocol**

a method in which operations done on objects may be undone after restarting a system.

**Distributed Dead locks**

A **deadlock** is a condition in a system where a set of processes (or threads) have requests for resources that can never be satisfied. Essentially, a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs. More formally, [Coffman](http://people.cs.umass.edu/%7Emcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf) defined four conditions have to be met for a deadlock to occur in a system:

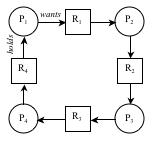


Figure 1. Deadlock

1. **Mutual exclusion** A resource can be held by at most one process.
2. **Hold and wait** Processes that already hold resources can wait for another resource.
3. **Non-preemption** A resource, once granted, cannot be taken away.
4. **Circular wait** Two or more processes are waiting for resources held by one of the other processes.

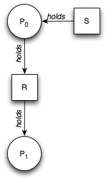
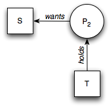
A directed graph model used to record the resource allocation state of a system. This state consists of *n* processes, P1 … Pn, and *m* resources, R1 … $m. In such a graph:

P1 → R1 means that resource R1 is allocated to process P1.

P1 ← R1 means that resource R1 is requested by process P1.

Deadlock is present when the graph has a directed cycles. An example is shown in Figure 1. Such a graph is called a **Wait-For Graph** (**WFG**).

**Deadlock in distributed systems**

Figure 2. Resource graph on A Figure 3. Resource graph on B

The same conditions for deadlock in uniprocessors apply to distributed systems. Unfortunately, as in many other aspects of distributed systems, they are harder to detect, avoid, and prevent. Four strategies can be used to handle deadlock:

1. **ignorance**: ignore the problem; assume that a deadlock will never occur. This is a surprisingly common approach.
2. **detection**: let a deadlock occur, detect it, and then deal with it by aborting and later restarting a process that causes deadlock.
3. **prevention**: make a deadlock impossible by granting requests so that one of the necessary conditions for deadlock does not hold.
4. **avoidance**: choose resource allocation carefully so that deadlock will not occur. Resource requests can be honored as long as the system remains in a safe (non-deadlock) state after resources are allocated.

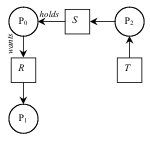
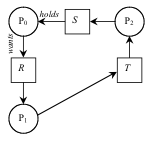
The last of these, deadlock avoidance through resource allocation is difficult and requires the ability to predict precisely the resources that will be needed and the times that they will be needed. This is difficult and not practical in real systems. The first of these is trivially simple but, of course, ineffective for actually doing anything about deadlock conditions. We will focus on the middle two approaches.

In a conventional system, the operating system is the component that is responsible for resource allocation and is the ideal entity to detect deadlock. Deadlock can be resolved by killing a process. This, of course, is not a good thing for the process. However, if processes are transactional in nature, then aborting the transaction is an anticipated operation. Transactions are designed to withstand being aborted and, as such, it is perfectly reasonable to abort one or more transactions to break a deadlock. The transaction can be restarted later at a time when, we hope, it will not create another deadlock.

**Centralized deadlock detection**

Centralized deadlock detection attempts to imitate the nondistributed algorithm through a central coordinator. Each machine is responsible for maintaining a resource graph for its processes and resources. A central coordinator maintains the resource utilization graph for the entire system: the **Global Wait-For Graph**. This graph is the union of the individual Wait-For Graphs. If the coordinator detects a cycle in the global wait-for graph, it aborts one process to break the deadlock.

In the non-distributed case, all the information on resource usage lives on one system and the graph may be constructed on that system. In the distributed case, the individual subgraphs have to be propagated to a central coordinator. A message can be sent each time an arc is added or deleted. If optimization is needed, a list of added or deleted arcs can be sent periodically to reduce the overall number of messages sent.

Figure 4. Resource graph on coordinator Figure 5. False deadlock

Here is an example (from [Tanenbaum](http://www.amazon.com/dp/0132392275/pkorg)). Suppose machine A has a process P0, which holds the resource S and wants resource R, which is held by P1. The local graph on A is shown in Figure 2. Another machine, machine B, has a process P2, which is holding resource T and wants resource S. Its local graph is shown in Figure 3. Both of these machines send their graphs to the central coordinator, which maintains the union (Figure 4).

All is well. There are no cycles and hence no deadlock. Now two events occur. Process P1 releases resource R and asks machine B for resource T. Two messages are sent to the coordinator:

message 1 (from machine A): “releasing R”

message 2 (from machine B): “waiting for T”

This should cause no problems (no deadlock). However, if message 2 arrives first, the coordinator would then construct the graph in Figure 5 and detect a deadlock. Such a condition is known as **false deadlock**. A way to fix this is to use Lamport’s algorithm to impose global time ordering on all machines. Alternatively, if the coordinator suspects deadlock, it can send a reliable message to every machine asking whether it has any release messages. Each machine will then respond with either a release message or a negative acknowledgement to acknowledge receipt of the message.

**Distributed deadlock detection**

An algorithm for detecting deadlocks in a distributed system was proposed by Chandy, Misra, and Haas in 1983. Processes request resources from the current holder of that resource. Some processes may wait for resources, which may be held either locally or remotely. Cross-machine arcs make looking for cycles, and hence detecting deadlock, difficult. This algorithm avoids the problem of constructing a Global WFG.

The **Chandy-Misra-Haas algorithm** works this way: when a process has to wait for a resource, a **probe** message is sent to the process holding that resource. The probe message contains three components: the process ID that blocked, the process ID that is sending the request, and the destination. Initially, the first two components will be the same. When a process receives the probe: if the process itself is waiting on a resource, it updates the sending and destination fields of the message and forwards it to the resource holder. If it is waiting on multiple resources, a message is sent to each process holding the resources. This process continues as long as processes are waiting for resources. If the originator gets a message and sees its own process number in the blocked field of the message, it knows that a cycle has been taken and deadlock exists. In this case, some process (transaction) will have to die. The sender may choose to commit suicide and abort itself or an election algorithm may be used to determine an alternate victim (e.g., youngest process, oldest process, …).

**Distributed deadlock prevention**

An alternative to detecting deadlocks is to design a system so that deadlock is impossible. We examined the four conditions for deadlock. If we can deny at least one of these conditions then we will not have deadlock.

Mutual exclusion

To deny this means that we will allow a resource to be held (used) by more than one process at a time. If a resource can be shared then there is no need for mutual exclusion and deadlock cannot occur. Too often, however, a process requires mutual exclusion for a resource because the resource is some object that will be modified by the process.

Hold and wait

Denying this means that processes that hold resources cannot wait for another resource. This typically implies that a process should grab all of its resources at once. This is not practical either since we cannot always predict what resources a process will need throughout its execution.

Non-preemption

A resource, once granted, cannot be taken away. In transactional systems, allowing preemption means that a transaction can come in and modify data (the resource) that is being used by another transaction. This differs from mutual exclusion since the access is not concurrent but the same problem arises of having multiple transactions modify the same resource. We can support this with optimistic concurrency control algorithms that will check for out-of-order modifications at commit time and roll back (abort) if there are potential inconsistencies.

Circular wait

Avoiding circular wait means that we ensure that a cycle of waiting on resources does not occur. We can do this by enforcing an ordering on granting resources and aborting transactions or denying requests if an ordering cannot be granted.

One way of avoiding circular wait is to obtain a globally-unique timestamp (e.g., Lamport total ordering) for every transaction so that no two transactions get the same timestamp. When one process is about to block waiting for a resource that another process is using, check which of the two processes has a younger timestamp and give priority to the older process.  
If a younger process is using the resource, then the older process (that wants the resource) waits. If an older process is holding the resource, the younger process (that wants the resource) aborts itself. This forces the resource utilization graph to be directed from older to younger processes, making cycles impossible. This algorithm is known as the **wait-die algorithm**.

An alternative, but similar, method by which resource request cycles may be avoided is to have an old process abort (kill) the younger process that holds a resource. If a younger process wants a resource that an older one is using, then it waits until the older process is done. In this case, the graph flows from young to old and cycles are again impossible. This variant is called the **wound-wait algorithm**.