1.1 Introduction

1.1.1 Definition – A d*istributed system* consists of a number of components, which are themselves computer systems. These components are connected by some communication medium, usually a sophisticated network. Applications execute by using a number of processes in different component systems. These processes communicate and interact to achieve productive work within the application.

A d*istributed system* in this context is simply a collection of autonomous computers connected by a computer network to enable resource sharing and co-operation between applications to achieve a given task.

A state-of-the-art d*istributed system* is one that combines the accessibility, coherence and manageability advantages of centralized systems with the sharing, growth, cost and autonomy advantages of networked systems and the added advantage of security, availability and reliability.

A *Distributed System* is one that runs a collection of machines that do not have shared memory, yet it looks to its users as a single computer.

A fundamental design goal of distributed systems technology is that, where possible, distribution should be concealed, giving the users the illusion that all available resources are located at the user's workstation.

1.1.2 Why are Computing Systems Distributed?

Distributed systems have been around since the early 1970's and have been made possible by advances in computer and communications technology. Systems are distributed for either or both of two main reasons:

1. Organizations today are expanding far beyond their traditional geographic boundaries in search for new business, new customers, new markets and improved financial and organizational viability. Consequently, an organization and its information systems may be inherently distributed and in connecting its systems into a seamless whole, a distributed system appears.

2. An organization may take inherently centralized information processing systems and distribute them to achieve higher reliability, availability, safety or performance, or all of the above.

1.1.3 Characteristics of Distributed Computing Systems

1. Multiple autonomous processing elements – A distributed system is composed of several independent components each with processing ability. There is no master-slave relationship between processing elements. Thus, it excludes traditional centralized mainframe based systems.

2. Information exchange over a network – the network connects autonomous processing elements.

3. Processes interact via non-shared local memory – multiple processor computer systems can be classified into those that share memory (multiprocessor computers) and those without shared memory (multi-computers). A hybrid configuration involves separate computers with a distributed shared memory.

4. Transparency – A distributed system is designed to conceal from the users the fact that they are operating over a wide spread geographical area and provide the illusion of a single desktop environment. It should allow every part of the system to be viewed the same way regardless of the system size and provide services the same way to every part of the system. Some aspects of transparency include:

- Global names – the same name works everywhere. Machines, users, files, control groups and services have full names that mean the same thing regardless of where in the system the name is used.
- Global access – the same functions are usable everywhere with reasonable performance. A program can run anywhere and get the same results. All the services and objects required by a program to run are available to the program regardless of where in the system it is executing.
- Global security – same user authentication and control access work everywhere e.g. same mechanism to let the same person next door and someone at another site read ones files. Authentication to any computer in the system.
- Global management – The same person can administrate system components anywhere. System management tools perform the same actions e.g. configuration of workstations.

1.1.4 Components of a Distributed System

A distributed System has three major components:

1. Data – this is concerned with the structures and functions for information retention and manipulation.

2. Processing – this is concerned with processing data objects.

3. Presentation – this is processing directly concerned with making data visible to users and handling user interaction.

1.1.5 Advantages of Distributed Systems

A distributed system has a number of advantages over a single computer system:

- It can be more fault-tolerant. It can be designed so that if one component of the system fails then the others will continue to work. Such a system will provide useful work in the face of quite a large number of failures in individual component systems.

- It is more flexible - A distributed system can be made up from a number of different components. Some of these components may be specialized for a specific task while others may be general purpose. Components can be added, upgraded, moved and removed without impacting upon other components.

- It is easier to extend - More processing, storage or other power can be obtained by increasing the number of components.
- It is easier to upgrade - When a single large computer system becomes obsolete all of it has to be replaced in a costly and disruptive operation. A distributed system may be upgraded in increments by replacing individual components without a major disruption, or a large cash injection.
- Local autonomy – by allowing domains of control to be defined where decisions are made relating to purchasing, ownership, operating priorities, IS development and management, etc. Each domain decides where resources under its control are located.
- Increased Reliability and Availability – In a centralized system, a component failure can mean that the whole system is down, stopping all users from getting services. In a distributed system, multiple components of the same type can be configured to fail independently. This aspect of replication of components improves the fault tolerance in distributed systems, consequently, the reliability and availability of the system is enhanced.

- Improved Performance – Large centralized systems can be slow performers due to the sheer volume of data and transactions being handled. A service that is partitioned over many server computers each supporting a smaller set of and users access to local data and resources results in faster access. Another performance advantage is the support for parallel access to distributed data across the organization.
- Security breaches are localized – In distributed systems with multiple security control domains, a security breach in one domain does not compromise the whole system. Each security domain has varying degree of security authentication, access control and auditing.

1.1.6 Disadvantages of Distributed Systems

- It's more difficult to manage and secure – Centralized systems are inherently easier to secure and easier to manage because control is done from a single point. Distributed systems require more complex procedures for security, administration, maintenance and user support due to greater levels of co-ordination and control required.
- Lack of skilled support and development Staff – Since the equipment and software in a DS can be sourced from different vendors, unlike in traditional systems where everything is sourced from the same vendor, its difficult to find personnel with a wide range of skills to offer comprehensive support.
- They are significantly more complex.
- They introduce problems of synchronization between processes.
- They introduce problems of maintaining consistency of data.

1.2. Distributed Systems and Object Oriented Models

Object technology offers a different programming model from traditional structured programming, which is based on the separation of data and processing through functions and procedures. An object is a computational entity that encapsulates both private data describing its state and a set of associated operations. An object's state is visible only within the object and is completely protected and hidden from other objects as a way of hiding complexity and as a protection from misuse. The only way to communicate with an object is by sending a message to the object through its public interface.

Distributed Systems can be built from any desired language if required, but O-O models are appropriate because they naturally support the concepts of Distributed Computing. The concepts of interfaces, inheritance, refinement and encapsulation can be applied in distributed systems. Object technology is suited for distributed systems because:

- Implementation detail is hidden and use of objects emphasized through the definition of interfaces.
- Reusability of objects makes it possible to integrate and re-package existing functionality
- The message passing nature of inter-object communication maps easily to a distributed environment. A Message passing, RPC or remote IPC mechanism can be used to implement the communication channels through which messages can be passed between objects.

Interfaces – An interface is like a procedure definition that has one or several implementations. An Object supports a number of interfaces with which it can communicate with other objects. In order to communicate with another object, the only requirement is that it has to know the interface the object supports and nothing more. The implementation detail is hidden and can vary depending on a number of factors.

Encapsulation – Objects provide us with an effective way of encapsulating things so that we use them in other parts of the model

Inheritance and Refinement – Inheritance and refinement make objects more reusable. Reusability of objects makes systems more open since they support diversity, but permit comprehension of systems into one.

1.3 Distributed Operating Systems

Examples – Amoeba, Argus, Cronas and V-System

Hardware for distributed systems is important, but it is the software that largely determines what a distributed system looks like. DOS are very much like traditional operating systems; they act as resource managers for the underlying hardware. Most important, they attempt to hide the intricacies and heterogeneous nature of the underlying hardware by providing a virtual machine on which applications can be easily executed.

Distributed applications can run multiple processes in multiple computers linked by communications channel. The application programmer will be supported by a programming environment and run-time system that will make many aspects of distribution in the system transparent. For instance the programmer may not have to worry about where the parts of the application are running, this can all be taken care of, if required; this is called *location transparency*.

In a distributed operating system this interface is enhanced so that a program may be run on any computer in the distributed system and access data on any other computer. The operating system provides data, execution and location transparency, often through an extended naming scheme. Distributed Operating Systems extend the notion of a *virtual machine* over a number of interconnected computers or hosts. Note that the user/programmer still has the illusion of working on a single system. All the issues of concurrency and distribution are completely hidden by the virtual machine, and the user/programmer is not at liberty to exploit them (nor should they be hindered by them!). Distributed Operating Systems are often broadly classified into two extremes of a spectrum:

- Loosely Coupled Systems – (Network Operating Systems) Components are Workstations, LAN, and Servers e.g. V-System, BSD Unix... they can be thought of as a collection of computers each running their own O/S. However these OS work together to make their services and resources available to others. To actually come to a distributed system, enhancements to the services of NOS are needed such that a better support for distribution transparency is provided. These enhancements lead to what is known as ***middleware***.

- Tightly Coupled Systems - Components are Processors, Memory, Bus, I/O e.g. Meiko Compute Surface. The operating system tries to maintain a single global view of the resources it manages.

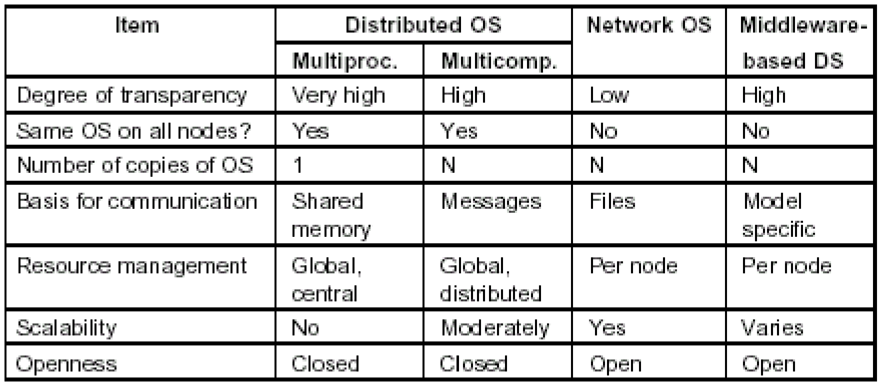
Often this classification is really a reflection of the reliability and performance of the communications sub-system. Frequently, shared memory systems are regarded as more tightly coupled than message passing systems. Another way of looking at these classifications is to think of tightly coupled systems as being *dependent*, and loosely coupled systems as *independent*, where the dependency is in terms of system availability in the face of failure of some single host. In tightly coupled systems, it is reasonable to consider shared memory (or at least hierarchical cache mechanisms) as a communications mechanism. In loosely coupled systems, only message passing can be considered.

The advantage of a distributed operating system is that is uses an interface below that of the application program. This means the existing programming environments may be used, the programmer may use the system with little or no extra training, and in some cases existing software may be used. Essentially, the Distributed Operating System dictates the policies of distribution for all aspects of programming. This means that the programmer is not able to use the distributed functionality in an application specific way to optimize a solution.

Another major disadvantage is that the distributed system is tied to a style of operating system interface. There are lots of different operating systems today, to meet different requirements (real or imaginary); there is no reason why future distributed systems will not need different operating system interfaces. Consequently it is not possible to build a truly heterogeneous open distributed system by building it on top of a homogeneous distributed operating system.

1.4 A Comparison between Systems

A brief comparison between distributed operating systems, network operating
Systems, and (middleware-based) distributed systems are given in Fig. 1-24.



Item	Distributed OS		Network OS	Middleware-
	Multiproc.	Multicomp.		based DS
Degree of transparency	Very high	High	Low	High
Same OS on all nodes?	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

Figure 1-24. A comparison between multiprocessor operating systems, multicomputer operating systems, network operating systems, and middleware-based distributed systems.

With respect to transparency, it is clear that distributed operating systems do a better job than network operating systems. In multiprocessor systems we have to hide only that there are more processors, which is relatively easy. The hard part is also hiding that memory is physically distributed, which is why building multi-computer operating systems that support full distribution transparency is so difficult. Distributed systems often improve transparency by adopting a specific model for distribution and communication. For example, distributed file systems are generally good at hiding the location and access to files. However, they lose some generality as users are forced to express everything in terms of that specific model, which may be sometimes inconvenient for a specific application. Distributed operating systems are homogeneous, implying that each node runs the same operating system (kernel). In multiprocessor systems, no copies of tables and such are needed, as they can all be shared through main memory. In this case, all communication also happens through main memory, whereas in multi-computer operating systems messages are used. In network operating systems, one could argue that communication is almost entirely file based. For example, in the Internet, a lot of communication is done by transferring files. However, high-level messaging in the form of electronic mail systems and bulletin boards is also used extensively. Communication in middleware-based distributed systems depends on the model specifically adopted by the system. Resources in network operating systems and distributed systems are managed per node, which makes such systems relatively easy to scale. However, practice shows that an implementation of the middleware layer in distributed systems often has limited scalability. Distributed operating systems have global resource management, making them harder to scale. Because of the centralized approach in multiprocessor systems (i.e., all management data is kept in main memory), these systems are often hard to scale.

Finally, network operating systems and distributed systems win when it comes to openness. In general, nodes support a standard communication protocol such as TCP/IP, making interoperability easy. However, there may be a lot of problems porting applications when many different kinds of operating systems are used. In general, distributed operating systems are not designed to be open. Instead, they are often optimized for performance, leading to many proprietary solutions that stand in the way of an open system.

1.5 Distribution Transparency

A transparency is some aspect of the distributed system that is concealed from the user, programmer or system developer. A major consideration when designing distributed systems is the extent to which the distribution of components of the system should be made transparent to the application designers and users because there are trade-offs to be made. Users demand a simple user interface to this complex world, which in turn, demands functions for concealing complexity. Most users desire a single system image where all resources are perceived to be centralized at the desktop computer. The term **distribution transparency** is used to describe the visibility of distributed components within a distributed system. There are two major transparency choices:

1. **Full Distribution Transparency** – Complexities introduced by distribution are completely concealed. It simplifies application development and improves usability. In addition, it accommodates the incorporation of existing systems based on centralized systems. For example, a database application designed to execute on a single host machine is able to make use of distributed resources without knowledge of whether the resource is local or remote and with no change to the application.
2. **Partial or selective Distribution Transparency –** The application designer may need to reveal that some DS components are distributed and therefore requires a development environment that gives some freedom to take account of this. Full distribution does not allow applications the .opportunity to exploit decentralization at the level of application process. It’s the designer who chooses the extent to which users are made aware of distributed components by selecting a level of distribution transparency appropriate to the application.

v The different transparencies are:

- **Access transparency** – hiding the use of communications to access remote resources like p frogram files, data, printers, etc. so that the user is under the illusion that all resources are local. Remote resources are accessed using exactly the same mechanism for accessing local resources. From a programmer’s point of view, the access method to a remote object may be identical to access a local object of the same class. This transparency has two parts:

    1. Keeping a syntactical or mechanical consistency between distributed and non-distributed access

    2. Keeping the same semantics. Because the semantics of remote access are more complex, particularly failure modes, this means the local access should be a subset.

- **Location Transparency** - The details of the topology of the system should be of no concern to the user. The location of an object in the system may not be visible to the user or programmer. This differs from access transparency in that both the naming and access methods may be the same. Names may give no hint as to location.

- **Concurrency Transparency -** Users and Applications should be able to access shared data or objects without interference between each other. This requires very complex mechanisms in a distributed system, since there exists true concurrency rather than the simulated concurrency of a central system. For example, a distributed printing service must provide the same atomic access per file as a central system so that printout is not randomly interleaved.

- **Replication Transparency** – Hiding differences between replicated and non-replicated resoggurces. If the system provides replication (for availability or performance reasons) it should not concern the user.

- **Fault Transparency -** If software or hardware failures occur, these should be hidden from the user. This can be difficult to provide in a distributed system, since partial failure of the communications subsystem is possible, and this may not be reported. As far as possible, fault transparency will be provided by mechanisms that relate to access transparency. However, when the faults are inherent in the distributed nature of the system, then access transparency may not be maintained. The mechanisms that allow a system to hide faults may result in changes to access mechanisms (exx.g. access to reliable objects may be different from access to simple objects). In a software system, especially a networked one, it is often hard to tell the difference between a failed and a slow running process or processor. This distinction is hidden or made visible here.

- **Migration Transparency -** If objects (processes or data) migrate (to provide better performance, or reliability, or to hide differences between hosts), this should be hidden from the user. This means that resources can be relocated dynamically without the user being aware of configurations.

- **Performance Transparency** – Minimizing performance overheads in using remote resources, so that the response time and through put are comparable with cases when all resources are local. The configuration of the system should not be apparent to the user in terms of performance. This may require complex resource management mechanisms. It may not be possible at all in cases where resources are only accessible via low performance networks.

- **Scaling Transparency -** A system should be able to grow without affecting application algorithms. Graceful growth and evolution is an important requirement for most enterprises. A system should also be capable of scaling down to small environments where required, and be space and/or time efficient as required.

If all of the above transparency functions are built into the IT infrastructure (usually the software components) then full distribution transparency can be achieved. The degree of distribution transparency is a measure of the distributed nature of the IT infrastructure.

1.6 Issues in Design of Distributed Computing systems

There are key designs issues that people building distributed systems must deal with, with a goal to ensure that they are attained. These are:

1. Transparency
2. Fault tolerance
3. Openness
4. Concurrency
5. Scalability
6. Performance

1. **Transparency –** This is one of the key design issues in distributed systems. To design a system that achieves a single system image is very challenging. The concept of transparency has been proposed to describe distributed systems that can be made to behave like their non-distributed counterparts. It is described 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 a collection of independent components." Transparency there involves hiding all the distribution from human users and application programs

   Human users – In terms of the commands issued from the terminal and the results displayed on the terminal. The distributed system can be made to look just like a single processor system.

   Programs – At the lower level, the distribution should be hidden from programs. Transparency minimizes the difference to the application developer between programming for a distributed system and programming for a single machine. In other words, the system call interface should be designed such that the existence of multi-processors is not visible. A file should be accessed the same way whether its local or remote. A system in which remote files are accessed by explicitly setting up a network connection to a remote server and then sending messages to it is not transparent because remote services are being accessed differently from local ones.

2. **Fault Tolerance** – Since failures are inevitable, a computer system can be made more reliable by making it fault tolerant. A fault tolerant system is one designed to fulfill its specified purposes despite the occurrence of component failures (machine and network). Fault tolerant systems are designed to mask component failures i.e. attempt to prevent the failure of a system in spite of the failure of some of its components. Fault tolerance can be achieved through hardware and software.

   Software – Replication of programs e.g. replication of applications

   - Backward error recovery e.g. rollbacks and save points in database systems

   Hardware – redundant hardware components e.g. multiple processors

   Redundancy is defined as those parts of the system that are not needed for the correct functioning of the system if no fault tolerance is supported. Meaning that the system will continue to work correctly without redundancy if no failure occurs. Redundancy can be exhibited in both hardware and software components.

Fault tolerance improves system availability and reliability. Fault tolerance masks faults, which means that it makes faults not visible to the users externally. The system still continues to function according to the specification. Fault tolerance can also be achieved by fault prevention i.e. trying to detect any fault in hardware and software before connecting to the system. For example program debugging.

Although fault tolerance improves the system availability and reliability, it brings some overheads in terms of:

- Cost - increased system costs
- Software development – recovery mechanisms and testing
- Performance – makes system slower in updates of replicas
- Consistency – maintaining data consistency is not trivial

3. **Concurrency** – Concurrency arises in a system when several processes run in parallel. If these processes are not controlled then inconsistencies may arise in the system. This is an issue in of distributed systems because designers have to do this carefully and keenly to control the problem of inconsistency and conflicts. The end result is to achieve a serial access illusion. Concurrency control is important to achieve proper resource sharing and co-operation of processes. Uncontrolled interleave

4. sub-operations of concurrent transactions can lead to four main types of problems:

   i. *Lost update* – transaction A reads the current balance value. Transaction B updates the balance from 500 to 600. Later, Transaction A immediately updates the balance by adding 50 to the original value, increasing it to 550. The balance now contains 550 and B's update is lost.

   ii. *Dirty Read* – A data item, holding say 500, is updated to 550 by transaction A. subsequently; transaction B reads the data item (550) as the balance. Unfortunately, transaction A aborts restoring the value 500 to the data item. Transaction B processed a dirty value.

   iii. *Non-repeatable read* - Transaction B reads the current value of a data item, say 500. Transaction A updates the data item to 550. Finally, transaction B completes its processing by re-reading the current value, but the new value 550 is returned.

   iv. *Phantom record* – Transaction B processes all debit and credit transactions for a particular customer by reading all relevant rows from a table. While processing the rows, transaction A inserts a new row corresponding to a new credit transaction. Finally, transaction A rereads all rows in the table as part of final processing. The number of rows in the initial read does not match the number returned in the final read.

5. **Openness** – It's the ability of the system to accommodate different technology (hardware and software components) without changing the underlying structure of the system. For example, the ability to accommodate a 64-bit processor where a 32-bit processor was being used without changing the underlying system structure or the ability to accommodate a machine running a MAC O/S in a predominantly Windows O/S system. A distributed system should be open with respect to software vendors, developers, hardware components etc. this calls for well-defined interface so that the user just issues commands or requests without regard to the way the function is implemented so that you can have multiple independently developed copies using the same interface to provide services to the users.

6. **Scalability** – Each component of a distributed system has a finite capacity. Designing for scalability involves calculating the capacity of each of these elements and the extent to which the capacity can be increased. For, example an Ethernet network is a shared medium LAN that should never exceed around 50% utilization and capacity is not easily increased. 10 Mbps is the most common data rate, although this can be upgraded to 100 Mbps. The cost of upgrade is significant since network interface cards; hubs and possibly cabling may need to be replaced. Good distributed systems design minimizes utilization components that are not scalable. Also, the element that is weakest in terms of available capacity (and the extent to which the capacity can be increased) should be of prime importance in terms of design. There are four principle components to be considered when designing for scalability: client workstation, LAN, servers and WAN.

7. **Performance** – There are two common measures of performance for distributed systems:

   - Response time – defined as the average elapsed time from the moment the user is ready to transmit and the entire response is received. The response received depends on the nature of the user interaction.

   - Throughput – the number of requests handled per unit time.

   Satisfactory performance as perceived by the users is dependent on the nature of the task being performed. The components in the path between and application all have performance characteristics that will determine performance characteristics that will determine overall performance as perceived by the user. Servers will queue user requests until the necessary resource becomes available. In order to calculate performance in terms of response time the utilization of each component needs to be established and the effects of queuing calculated.

   An alternate approach to improving CPU performance is to implement multi-processor or multi-computer configurations where each CPU handles a portion of the total workload. Effectively, the total workload is handled by a service that is partitioned across multiple CPU server group. When workload is partitioned across multiple servers, resolution protocol is required to find the server that is able to satisfy the client request.

   Performance improvements can be made in distributed systems environment by migrating much of the processing on to a user's client workstation. This reduces the processing on the server per client request which leads to faster and more predictable response time

   Data intensive applications can improve performance by avoiding I/O operations to read from disk storage. Reading from buffer areas in memory is much faster. Applications invoking remote operations offered by remote servers can improve performance by avoiding the need to access a remote server to satisfy a request. A caching system reduces the performance cost of I/O and remote operations by storing the results of recently executed I/O or remote operations in memory and re-using the same data whenever the same operation is re-invoked and when it can be ascertained that the data is still valid. Whenever an I/O or remote operation is requested, the local cache is searched first.

   The main problem with caching is maintaining consistency between multiple caches and the server that hold authoritative state data. Cache consistency techniques range from restricting cached data to state data that does not change (read only) through to the maintenance of cached client lists by servers holding authoritative state data so that they can be informed when data is changed. When state data is updated all cache copies become invalid and each must be refreshed.

2. Middleware

2.1 Introduction

Middleware is essentially a layer of software running between client and server processes. It shields the client (and application developers) from the complexity of the underlying communications protocol, network operating systems functions and hardware configurations. A common set of application programming interfaces (APIs) is provided to application developers to enable rapid development of distributed applications and flexible access to distributed resources.

Middleware provides services such as name-to-address resolution, dynamic server process invocation, locating particular servers, load balancing, security, failure recovery, message routing, reliable message delivery etc.

A solution to achieving a truly distributed system is through an additional layer of software that is used in network operating systems to more or less hide the heterogeneity of the collection of underlying platforms but also to improve distribution transparency. Many modern distributed systems are constructed by means of such an additional layer of what is called ***middleware*.**

2.2 Positioning Middleware

Many distributed applications make direct use of the programming interface offered by network operating systems. For example, communication is often expressed through operations on sockets, which allow processes on different machines to pass each other messages (Stevens, 1998). In addition, applications often make use of interfaces to the local file system. As we explained, a problem with this approach is that distribution is hardly transparent. A solution is to place an additional layer of software between applications and the network operating system, offering a higher level of abstraction. Such a layer is accordingly called **middleware**. It sits in the middle between applications and the network operating system.

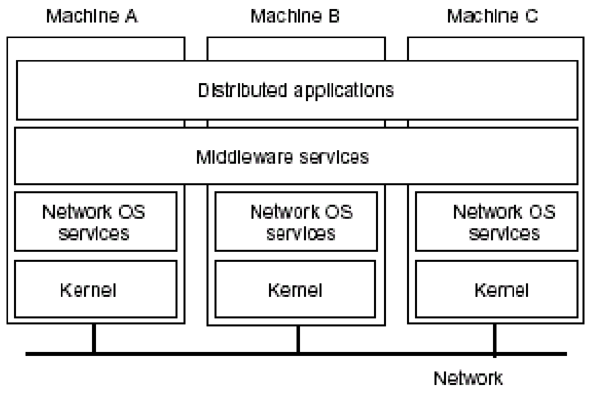


Figure 1-22. General structure of a distributed system as middleware.

Each local system forming part of the underlying network operating system is assumed to provide local resource management in addition to simple communication means to connect to other computers. In other words, middleware itself will not manage an individual node; this is left entirely to the local operating system.

An important goal is to hide heterogeneity of the underlying platforms from applications. Therefore, many middleware systems offer a more-or-less complete collection of services and discourage using anything else but their interfaces to those services. In other words, skipping the middleware layer and immediately calling services of one of the underlying operating systems is often frowned upon.

It is interesting to note that middleware was not invented as an academic exercise in achieving distribution transparency. After the introduction and widespread use of network operating systems, many organizations found themselves having lots of networked applications that could not be easily integrated into a single system. At that point, manufacturers started to build higher-level, application-independent services into their systems. Typical examples include support for distributed transactions and advanced communication facilities.

Of course, agreeing on what the right middleware should be is not easy. An approach is to set up an organization that subsequently defines a common standard for some middleware solution. At present, there are a number of such standards available. The standards are generally not compatible with each other, and even worse, products implementing the same standard but from different manufacturers rarely inter-work. Surely, it will not be long before someone offers ‘‘upperware’’ to remedy this defect.

2.3 Middleware Models

To make development and integration of distributed applications as simple as possible, most middleware is based on some model, or *paradigm*, for describing distribution and communication. A relatively simple model is that of treating everything as a file. This is the approach originally introduced in UNIX and rigorously followed in Plan 9 (Pike et al., 1995). In Plan 9, all resources, including I/O devices such as keyboard, mouse, disk, network interface, and so on, are treated as files. Essentially, whether a file is local or remote makes no difference.

An application opens a file, reads and writes bytes, and closes it again. Because files can be shared by several processes, communication reduces to simply accessing the same file.
A similar approach, but less strict than in Plan 9, is followed by middleware centered on **distributed file systems**. In many cases, such middleware is actually only one step beyond a network operating system in the sense that distribution transparency is supported only for traditional files (i.e., files that are used for merely storing data). For example, processes are often required to be started explicitly on specific machines. Middleware based on distributed file systems has proven to be reasonable scalable, which contributes to its popularity.

Another important early middleware model is that based on **Remote Procedure Calls** (**RPCs**). In this model, the emphasis is on hiding network communication by allowing a process to call a procedure of which an implementation is located on a remote machine. When calling such a procedure, parameters are transparently shipped to the remote machine where the procedure is subsequently executed, after which the results are sent back to the caller. It therefore appears as if the procedure call was executed locally: the calling process remains unaware of the fact that network communication took place, except perhaps for some loss of performance.

As object orientation came into vogue, it became apparent that if procedure calls could cross machine boundaries, it should also be possible to invoke objects residing on remote machines in a transparent fashion. This has now led to various middleware systems offering a notion of **distributed objects**. The essence of distributed objects is that each object implements an interface that hides all the internal details of the object from its users. An interface consists of the methods that the object implements, no more and no less. The only thing that a process sees of an object is its interface. Distributed objects are often implemented by having each object itself located on a single machine, and additionally making its interface available on many other machines. When a process invokes a method, the interface implementation on the process’s machine simply transforms the method invocation into a message that is sent to the object. The object executes the requested method and sends back the result. The interface implementation subsequently transforms the reply message into a return value, which is then handed over to the invoking process. As in the case of RPC, the process may be kept completely unaware of the network communication. An example is the CORBA (Common Object Request Broker Architecture) standard.

What models can do to simplify the use of networked systems is probably best illustrated by the World Wide Web. The success of the Web is mainly due to the extremely simple, yet highly effective model of **distributed documents**. In the model of the Web, information is organized into documents, with each document residing at a machine transparently located somewhere in the world. Documents contain links that refer to other documents. By following a link, the document to which that link refers is fetched from its location and displayed on the user's screen. The concept of a document need not be restricted to only text-based information. For example, the Web also supports audio and video documents, as well as all kinds of interactive graphic-based documents

2.4 Middleware Services

There are a number of services common to many middleware systems.
Communication Services - Invariably, all middleware, one way or another, attempts to implement *access transparency*, by offering high-level **communication facilities** that hide the low-level message passing through computer networks. The programming interface to the transport layer as offered by network operating system is thus entirely replaced by other facilities. How communication is supported depends very much on the model of distribution the middleware offers to users and applications. We already mentioned remote procedure calls and distributed-object invocations. In addition, many middleware systems provide facilities for transparent access to remote data, such as distributed file systems or distributed databases. Transparently fetching documents as is done in the Web is another example of high-level (one-way) communication.

Naming services -An important service common to all middleware is that of **naming**. Name services allow entities to be shared and looked up (as in directories), and are comparable to telephone books and the yellow pages. Although naming may seem simple at first thought, difficulties arise when scalability is taken into account. Problems are caused by the fact that to efficiently look up a name in a large-scale system, the location of the entity that is named must be assumed to be fixed. This assumption is made in the World Wide Web, in which each document is currently named by means of a URL. A URL contains the name of the server where the document to which the URL refers is stored. Therefore, if the document is moved to another server, its URL ceases to work.

Distributed Transaction services - In environments where data storage plays an important role, facilities are generally offered for **distributed transactions**. An important property of a transaction is that it allows multiple read and write operations to occur atomically. Atomicity means that the transaction either succeeds, so that all its write operations are actually performed, or it fails, leaving all referenced data unaffected. Distributed transactions operate on data that are possibly spread across multiple machines. Especially in the face of masking failures, which is often hard in distributed systems, it is important to offer services such as distributed transactions. Unfortunately, transactions are hard to scale across many local machines, let alone geographically dispersed machines.

Security Services - Finally, virtually all middleware systems that are used in non-experimental environments provide facilities for **security.** Compared to network operating systems, the problem with security in middleware is that it should be pervasive. In principle, the middleware layer cannot rely on the underlying local operating systems to adequately support security for the complete network. Consequently, security has to be partly implemented anew in the middleware layer itself. Combined with the need for extensibility, security has turned out to be one of the hardest services to implement in distributed systems.

2.5 Middleware and Openness

Modern distributed systems are generally constructed as middleware for a range of operating systems. In this way, applications built for a specific distributed system become operating system independent. Unfortunately, this independence is often replaced by a strong dependency on specific middleware. Problems are caused by the fact that middleware is often less open than claimed.

As we explained previously, a truly open distributed system is specified by means of interfaces that are complete. Complete means that everything that is needed for implementing the system has indeed been specified. Incompleteness of interface definitions leads to the situation in which system developers may be forced to add their own interfaces. Consequently, we may end up in a situation in which two middleware systems from different development teams adhere to the same standard, but applications written for one system cannot be easily ported to the other. Equally bad is the situation in which incompleteness leads to a situation in which two different implementations can never interoperate, despite the fact that they implement *exactly* the same set of interfaces but different underlying protocols. For example, if two different implementations rely on incompatible communication protocols as available in the underlying network operating system, there is little hope that interoperability can be easily achieved. What we need is that middleware protocols and the interfaces to the middleware are the same, as
shown below

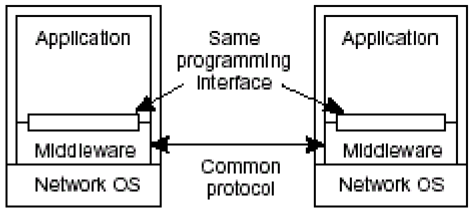


Figure 1-23. In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.

As another example, to ensure interoperability between different implementations, it is necessary that entities within the different systems are referenced in the same way. If entities in one system are referred by means of URLs, while the other system implements references using network addresses, it is clear that cross referencing is going to be a problem. In such cases, the interface definitions should have prescribed precisely what references look like.

3. Distributed Processing

3.1 Overview

Distributed processing can be loosely defined as the execution of co-operating processes which communicate by exchanging messages across an information network. It means that the infrastructure consists of distributed processors, enabling parallel execution of processes and message exchanges. Communication and data exchange can be implemented in two ways:

- Shared memory
- Message exchange: message passing and Remote Procedure Call (RPC)

3.2 Processes and Threads

A ***process*** is a logical representation of a physical processor that executes program code and has associated state and data. Sometimes described as a virtual processor. A process is the unit of resource allocation and so is defined by the resources it uses and by the location at which it's executing. A process can run either in a separate (private) address space or may share the same address space with other processes.

Processes are created either implicitly (e.g. by the operating system) or explicitly using an appropriate language construct or O/S construct such as *fork( ).* In uni-processor computer systems the illusion of many programs running at the same time is created using the *time slicing* technique, but in actual sense there is only one program utilizing the CPU at any given time. Processes are switched in and out of the CPU rapidly that each process appears to be executing continuously. Switching involves saving the state of the currently active process and setting up the state of another process, sometimes known as *context switching*.

Some operating systems allow additional 'child processes' to be created, each competing for the CPU and other resources with the other processes. All resources belonging to the 'parent process' are duplicated thus making the available to the 'child processes'. It's common for a program to create multiple processes that are required to share memory and other resources. The process may wait for a particular event to occur. Some operating systems support this situation efficiently by allowing a number of processes to share a single address space. Processes in this context are referred to as ***threads*** and the O/S is said to support *multi-threading*. Usually a processes and threads are used interchangeably.

3.3 Synchronization

There are two main reasons why there is need for synchronization mechanisms:

1. Two or more processes may need to co-operate in order to accomplish a given task. This implies that the operating mechanism must provide facilities for identifying co-operating processes and synchronizing them.
2. Two or more processes may need to compete for access to shared services or resources. The implication is that the synchronization mechanism must provide facilities for a process to wait for a resource to become available and another process to signal the release of that resource.

When processes are running on the same computer, synchronization is straightforward since all processes use the same physical clock and can share memory. This can be done using well-known techniques such

as ***semaphores*** and ***monitors*** that are used to provide mutually exclusive access to a non-sharable resource by preventing concurrent execution of the critical region of a program through which the non-sharable resource is accessed.

A Monitor is a collection of procedures, which may be executed by a collection of concurrent processes. It protects its internal data from the users, and is a mechanism for synchronizing access to the resources the procedures use. Since only the monitor can access its private data, it automatically provides mutual exclusive between customer processes. Entry to the monitor by one process excludes entry by others.

Example: Semaphore

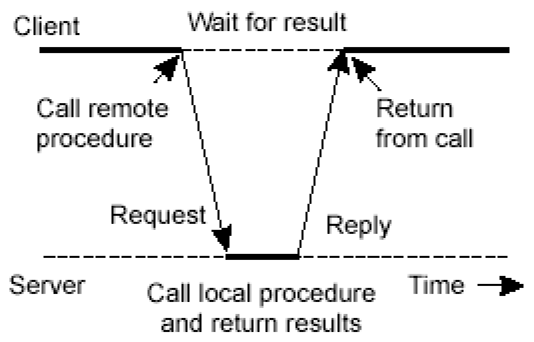
```
/* block the current process until it can acquire the mutual exclusion lock

Wait(mutex)

/* execute the critical section of code

critical_section(…………)

/* release the mutual exclusion lock

signal(mutex)

/* execute the non-critical section of code
```

Example: Monitor

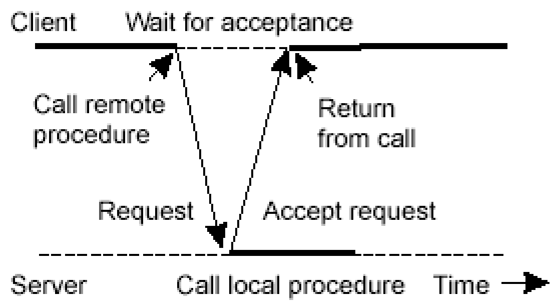
As an example, consider a simple monitor for protecting an integer variable as shown below. The monitor contains a single (private) variable count that can be accessed only by means of three (public) procedures for respectively reading its current value, incrementing it by 1, or decrementing it. The monitor construct guarantees that any process that calls one of these procedures can atomically access the private data contained in the monitor.

```
monitor Counter {
        private:
        int count = 0;
        public:
        int value( ) { return count; }
        void incr( ) { count = count + 1; }
        void decr( ) { count = count - 1; }
 }
```

Synchronization can either be ***synchronous*** (blocked) or ***asynchronous*** (non-blocking). A synchronous process is delayed until it receives a response from the destination process. A primitive is non-blocking if it’s execution never delays the invoking process. Non-blocking primitives must buffer messages to maintain synchronization. This makes programs flexible but increases their complexity. When blocking versions of message passing are used, programs are easier to write and synchronization easier to maintain. When send() operation is invoked, the invoking process blocks until the message is received. A subsequent receive() operation again blocks the invoking process until a message is actually received.



(a) Synchronous communication



(b) Asynchronous communication

3.4 Inter-process Communication (IPC)

When processes in the same local computer wish to interact they make use of an inter-process communication (IPC) mechanism that is usually provided by the O/S. The most common mode of communication is via a shared memory since the processes reside in the same address space.

A number of mechanisms are available:

Pipes / Named Pipes - perhaps the most primitive example is a synchronous filter mechanism. For example the *pipe* mechanism in UNIX

ls –l | more

The commands *ls* and *more* run as two concurrent processes, with the output of *ls* connected to the input of *more* and has the overall effect of listing the contents of the current directory one screen at a time.

File sharing - An alternative mechanism is the use of a local file. This has the advantage that it can handle large volumes of data and is well understood. This is the basis on which on-line database systems are built. The major drawback is that there are no inherent synchronization mechanisms between communicating processes to avoid state data corruption, synchronization mechanisms such as file and record locking are used to allow concurrent processes communicate while preserving data consistency. Secondly, communication is inefficient since it uses a relatively slow medium.

Shared Memory - Since all processes are local, the computer's RAM can be used to implement a shared memory facility. A common region of memory addressable by all concurrent processes is used to define shared variables which are used to pass data or for synchronization purposes. Processes must use semaphores, monitors or other techniques for synchronization purposes. A good example of a shared memory mechanism is the *clipboard* facility.

Message Queuing - A common asynchronous linkage mechanism is a message queuing mechanism that provides the ability for any process to read/write from a named queue. Synchronization is inherent in the read/write operations and the message queue which together support asynchronous communication between many different processes. Messages are identified by a unique identifier and security implemented by granting read/write permissions to processes.

IPC mechanisms can be broadly classified into:

- Reliable communication.
- Unreliable communication.

Reliable communication channels fail only with the end system (e.g. if a central computer bus fails, usually the entire machine (stable storage/memory/CPU access) fails. Unreliable channels exhibit various different types of fault. Messages may be lost, re-ordered, duplicated, changed to apparently correct but different messages and even created as if from nowhere by the channel. All of these problems may have to be overcome by the IPC mechanism.

3.5 Models of Communication in a Distributed Computing System

There are several main paradigms commonly used to structure a distributed system:

1. Master-slave model
2. Client/Server model
3. Peer-to-peer model
4. Group model
5. Distributed object model

3.5.1 Master –Slave model – It may not be an appropriate model for structuring a distributed system. In this model, a master process initiates and controls any dialogue with the slave processes. Slave processes exhibit very little intelligence, responding to commands from a single master process and exchange messages only when invited by the master process. The slave process merely complies with the dialogue rules set by the master. This is the model on which centralized systems were based and has limited applications in distributed systems because it does not make the best use of distributed resources and is a single point of failure.

3.5.2 Client Server Model – This is the most widely used paradigm for structuring distributed systems. A client requests a particular service. One or more processes called servers are responsible for the provision of services to clients. Services are accessed via a well-defined interface that is made known to the clients. On the receipt of a request the server executes the appropriate operation and sends a reply back to the client. The interaction is known as *request*/*reply* or *interrogation*. Both clients and servers are run as user processes. A single computer may run a single client or server process or may run multiple client or server processes. A server process is normally persistent (non-terminating) and provides services to more than one client process. The main distinction between master –slave and client/server models is in the fact that client and server processes are on equal footing but with distinct roles.

3.5.3 Peer-to-peer Model – This model is quite similar to the client/server model. The use of a small a small manageable number of servers (i.e. increased centralization of resources) increase system management compared to a case where potentially every computer can be configured as client and server. This model is known as a ***peer-to-peer model*** because every process has the same functionality as a peer process.

3.5.4 Group Model – In many circumstances, a set of processes need to co-operate in such a way that one process may need to send a message to all other processes in the group and receive response from one or more members. For example, in a video conferencing involving multiple participants and a whiteboard facility, when someone writes to the board, every other participant must receive the new image. In this model a set of group members are modeled conveniently to behave as a single unit called a ***group***. When a message is sent to a group interface, all the members receive it. There are different approaches to routing a 'group' message to every member:

- **Unicasting** – This involves sending a separate copy of the message to each member. An implicit assumption is that the sender knows the address of every member in the group. This may be not possible in some systems. In the absence of more sophisticated mechanisms, a system may resort to unicasting if member addresses are known. The number of network transmissions is proportional to the number of members in the group.

- **Multicasting** – In this model a single message with a group address can be used for routing purposes. When a group is first created it is assigned a unique group address. When a member is added to the group, it is instructed to listen for messages stamped with the group address as well as for its own unique address. This is an efficient mechanism since the number of network transmissions is significantly less than for unicasting.

- **Broadcasting** – Broadcast the message by sending a single message with a broadcast address. The message is sent to every possible entity on the network. Every entity must read the message and determine whether they should take action or discard it. This may be appropriate in the case where the address of members is not known since most network protocols implement broadcast facility. However, if messages are broadcasted frequently and there is no efficient network broadcast mechanism, the network becomes saturated.

In some cases, all group members or none must receive a group message at all. Group communication in this case is said to be *atomic*. Achieving atomicity in the presence of failures is difficult, resulting in many more messages being sent. Another aspect of group communication is the ordering of group messages. For example, in a computer conferencing system a user would expect to receive the original news item before any response to that item is received. This is known as *ordered multicast* and the requirement to ensure that all multicasts are received in the same order for all group members is common in distributed systems. Atomic multicasting does not guarantee that all messages will be received by group the members in the order they were sent.

3.6 Remote IPC

In a distributed system, processes interact in a logical sense by exchanging messages across a communication network. This is referred to as remote IPC. As with local processes, remote processes are either co-operating to complete a defined task or a competing for the use of a resource. Remote IPC can be implemented using the *message passing*, remote *procedure call* or the *shared memory paradigm.* Remote IPC functions are:

- Process registration for the purpose of identifying communicating processes
- Hide differences between local and remote communication
- Overcome failures
- Enforces a clean and simple interface providing a natural environment for modular structuring of distributed applications.
- Establishing communication channels between processes
- Routing messages to the destination process
- Synchronizing concurrent processes
- Shutting down communication channels

Binding

At some point, a process needs to determine the identity of the process with which it is communicating. This is known as binding. There are two major ways of binding:

- **Static binding** – destination processes are identified explicitly at program compile time.
- **Dynamic binding** – source to destination binding are created, modified and deleted at program run-time.

Static binding is the most efficient approach and is most appropriate when a client almost always binds to the same server although in some systems its often not possible to identify all potential destination processes. Dynamic binding facilitates location and migration transparency when processes are referred to indirectly (by name) and mapped to the location address at run-time. This is normally facilitated by a facility service known as a ***directory service.*** This service is used by the sender to locate the server. When a server is first activated, it exports its information to the directory service regarding the type of service being offered and where it is located. The sender imports its requirements to the directory service, which returns the address of the server that can meet its requirements. This is equivalent to finding the telephone number of a company or person in a telephone directory.

A consideration when supporting process interaction is whether to establish successful connection with the destination process before any messages are sent or whether not to. If connection is established first then the approach is known as *virtual circuit* and it minimizes the overheads of subsequent message transfer by setting up a routing path during connection in which all messages follow avoiding the need to send full address information for each subsequent message sent. Also, it is more straightforward to employ error control, flow control, sequence control and other protocols for enhanced reliability. At the end of the dialogue, the connection must be closed to release the network resources. An example is the telephone system.

However, if a relatively small number of messages are exchanged during a dialogue between co-operating processes then the protocol overhead due to connection establishment and subsequent close down is too costly. In such a case, a *data gram* (connectionless) approach is appropriate whereby no initial connection is made; instead, each message is transported as an independent unit of transfer and carries data sufficient for routing from the originating process to the destination process. A good example is the postal system. The guideline to selecting the most appropriate communication type is to use the connectionless approach when a typical dialogue consists of the exchange of a small number of messages; otherwise, the virtual circuit approach is more efficient.

3.6.1 Message Passing

A low level remote IPC in which the developer is explicitly aware of the message used in communication and the underlying message transport mechanism used in message exchange is ***message passing.*** Processes interact directly using send and receive or equivalent language primitives to initiate message transmission and reception, explicitly naming the recipient or sender, for example:

- **Send** *(message, destination_process)*
- **Receive** *(message, source_process)*

Message passing is the most flexible remote IPC mechanism that can be used to support all types of process interactions and the underlying transport protocols can be configured by the application according to the needs of the application. The above example is known as direct communication.

Another useful technique for identifying co-operating processes is known as indirect communication. Here the destination and the source identifiers are not process identifiers, instead, a *port* also known as a mailbox is specified which represents an abstract object at which messages are queued. Potentially, any process can write or read from a port. To send a message to a process, the sending process simply issues a send operation specifying a well-known port number that is associated with the destination process. To receive the message, the recipient simply issues a receive specifying the same port number. For example:

- **Send** *(message, destination_port)*
- **Receive** *(message, source_port)*

Security constraints can be introduced by allowing the owning process to specify access control rights on a port. Messages are not lost provided the queue size is adequate for the rate at which messages are being queued and de-queued.

3.6.2 Remote Procedure Call

Many distributed systems have been based on explicit message exchange between processes. However, the procedures send and receive do not conceal communication, which is important to achieve access transparency in distributed systems.
This interaction is very similar to the traditional procedure call in high-level programming languages except that the caller and the procedure to be executed are on different computers. A procedure call mechanism that allows the calling and the called procedures to be running on different computers is known as remote procedure call (RPC). When a process on machine *A* calls a procedure on machine *B*, the calling process on *A* is suspended, and execution of the called procedure takes place on *B*. Information can be transported from the sender to the recipient in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. While the basic idea sounds simple and elegant, subtle problems exist. To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications. Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. Finally, both machines can crash and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely used technique that underlies many distributed systems.

RPC is popular for developing distributed systems because it looks and behaves like a well-understood, conventional procedure call in high-level languages. A procedure call is a very effective tool for implementing abstraction since to use it all one needs to know is the name of the procedure and arguments associated with it. Packing parameters into a message is called *parameter marshaling.* RPC is a remote operation with semantics similar to a local procedure call and can provide a degree of:

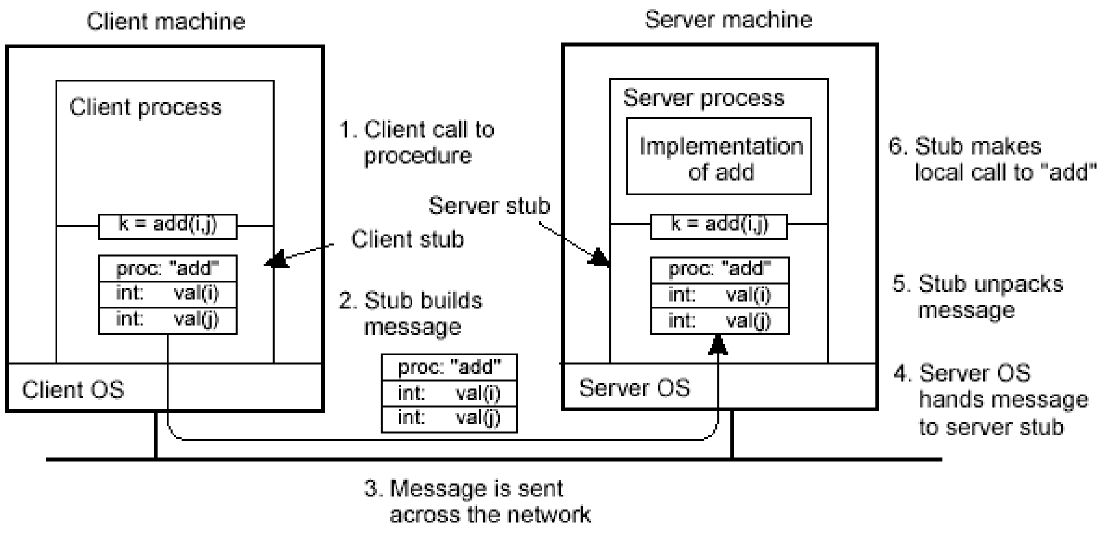
- Access transparency – since a call to a remote procedure may be similar to a local procedure.
- Location transparency – since the developer can refer to the procedure by name, unaware of where exactly the remote procedure is located.
- Synchronization – since the process invoking the RPC remains suspended (blocked) until the remote procedure is completed, just as a call to a local procedure.

A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

The net effect of all these steps is to convert the local call by the client procedure to the client stub, to a local call to the server procedure without either client or server being aware of the intermediate steps.

As a very simple example, consider a remote procedure, add(i, j), that takes two integer parameters *i* and *j* and returns their arithmetic sum as a result. (As a practical matter, one would not normally make such a simple procedure remote due to the overhead, but as an example it will do.) The call to add, is shown in the left-hand portion (in the client process) in Fig. 2-3. The client stub takes its two parameters and puts them in a message as indicated. It also puts the name or number of the procedure to be called in the message because the server might support several different calls, and it has to be told which one is required.



When the message arrives at the server, the stub examines the message to see which procedure is needed and then makes the appropriate call. If the server also supports other remote procedures, the server stub might have a switch statement in it to select the procedure to be called, depending on the first field of the message. The actual call from the stub to the server looks much like the original client call, except that the parameters are variables initialized from the incoming message. When the server has finished, the server stub gains control again. It takes the result provided by the server and packs it into a message. This message is sent back to the client stub, which unpacks it and returns the value to the client procedure.

Stub Generation **-** Once the RPC protocol has been completely defined, the client and server stubs need to be implemented. Fortunately, stubs for the same protocol but different procedures generally differ only in their interface to the applications. An interface consists of a collection of procedures that can be called by a client, and which are implemented by a server. An interface is generally available in the same programming language as the one in which the client or server is written (although this is strictly speaking, not necessary). To simplify matters, interfaces are often specified by means of an **Interface Definition Language** (**IDL**). An interface specified in such an IDL, is then subsequently compiled into a client stub and a server stub, along with the appropriate compile-time or run-time interfaces. Practice shows that using an interface definition language considerably simplifies client-server applications based on RPCs. Because it is easy to fully generate client and server stubs, all RPC-based middleware systems offer an IDL to support application development.

3.6.2.1 RPC Exceptions

The above mechanism, however, needs to cope with a wider range of exceptions than is typical of a local procedure call. For example:

- What if the parameters passed are either global variables or pointers? Many programming languages can support parameter passing using call-by-value (copy of data is passed) or call-by-reference (a pointer to the data item is passed).
- What if there are differences in the way the client and the server computers represent integers, floating point and other data types?
- What if the RPC fails?
- Is the client authorized to call the said procedure?

Marshalling is complicated by use of global variables and pointers as they only have meaning in the client's address space. Client and server processes run in different address spaces on separate machines. One solution would be to pass data values held by global variables or pointed to by the pointer. However there are cases where this will not workout, for example, when a linked list data structure is being passed to a procedure that manipulates the list.

Differences in representation of data can be overcome by use of an agreed language for representing data between client and server processes. For example, the common syntax for describing and encoding of data which known as *Abstract Syntax Notation* (ASN.1) has been defined as an international standard by the International Organization for standardization (ISO). ASN.1 is similar to the data declaration statements in a high-level programming language. Marshalling is then the process of converting the data types from the machines representation to a standard representation before transmission and converting it at the other end from the standard to the machines internal representation.

3.6.2.2 Failure Handling

RPC failures can be difficult to handle. There are four generalized types of failures that can occur when an RPC call is made:

1. The Client's request message is lost.
2. The client process fails while the server is processing the request.
3. The sever process fails while servicing the request.
4. The reply message is lost.

If the client’s message gets lost then the client will wait forever unless a time out error detection mechanism is employed. If the client process fails then, the server will carry out the remote operation unnecessarily. If the operation involves updating a data value then this can lead to a loss of data integrity. Furthermore, the server would generate a reply to client process that does not exist. This must be discarded by the client’s machine. When the client re-starts, it may be send the request again causing the server to execute more than once.

A similar situation arises when the server crashes. The server could crash just prior to the execution of the remote operation or just after execution completes but before a reply to the client is generated. In this case, clients will time-out and continually generate retries until either the server restarts or the retry limit is met.

3.6.3 RMI (Remote Method Invocation)

Remote method invocation allows applications to call object methods located remotely, sharing resources and processing load across systems. Unlike other systems for remote execution that require that only simple data types or defined structures be passed to and from methods, RMI allows any object type to be used - even if the client or server has never encountered it before. RMI allows both client and server to dynamically load new object types as required.

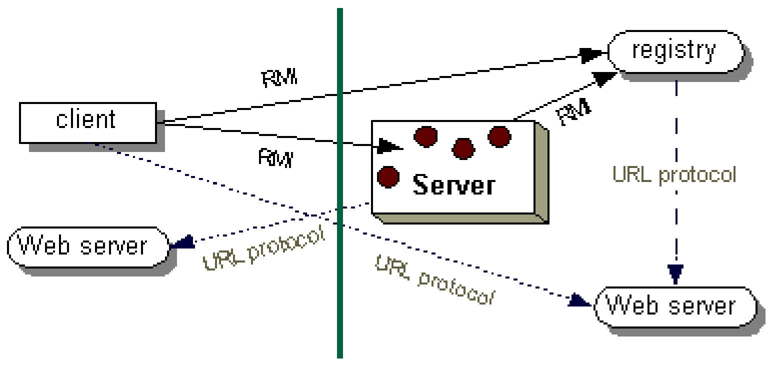
3.6.3.1 RMI Applications

RMI is the equivalent of RPC commonly used in middleware based on distributed objects model. RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.

Distributed object applications need to:

- *Locate remote objects*: Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility or the application can pass and return remote object references as part of its normal operation.
- *Communicate with remote objects*: Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard method invocation.
- *Load class bytecodes for objects that are passed around*: Because RMI allows a caller to pass objects to remote objects, RMI provides the necessary mechanisms for loading an object's code, as well as for transmitting its data.

The following illustration depicts an RMI distributed application that uses the registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing Web server to load class bytecodes, from server to client and from client to server, for objects when needed.



Advantages of Code Dynamic Loading

One of the central and unique features of RMI is its ability to download the *bytecodes* (or simply *code*) of an object's class if the class is not defined in the receiver's virtual machine. The types and the behavior of an object, previously available only in a single virtual machine, can be transmitted to another, possibly remote, virtual machine. RMI passes objects by their true type, so the behavior of those objects is not changed when they are sent to another virtual machine. This allows new types to be introduced into a remote virtual machine, thus extending the behavior of an application dynamically.

Remote Interfaces, Objects, and Methods

Like any other application, a distributed application built using RMI is made up of interfaces and classes. The interfaces define methods, and the classes implement the methods defined in the interfaces and, perhaps, define additional methods as well. In a distributed application some of the implementations are assumed to reside in different virtual machines. Objects that have methods that can be called across virtual machines are *remote objects*. An object becomes remote by implementing a *remote interface.*

RMI treats a remote object differently from a non-remote object when the object is passed from one virtual machine to another. Rather than making a copy of the implementation object in the receiving virtual machine, RMI passes a remote *stub* for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically *is*, to the caller, the remote reference. The caller invokes a method on the local stub, which is responsible for carrying out the method call on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This allows a stub to be cast to any of the interfaces that the remote object implements. However, this also means that *only* those methods defined in a remote interface are available to be called in the receiving virtual machine.

Creating Distributed Applications Using RMI

When you use RMI to develop a distributed application, you follow these general steps.

1. Design and implement the components of your distributed application.
2. Compile sources and generate stubs.
3. Make classes network accessible.
4. Start the application.

Design and Implement the Application Components

First, decide on your application architecture and determine which components are local objects and which ones should be remotely accessible. This step includes:

- *Defining the remote interfaces*: A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. Part of the design of such interfaces is the determination of any local objects that will be used as parameters and return values for these methods; if any of these interfaces or classes do not yet exist, you need to define them as well.

- *Implementing the remote objects*: Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces (either local or remote) and other methods (which are available only locally). If any local classes are to be used as parameters or return values to any of these methods, they must be implemented as well.

- *Implementing the clients*: Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Compile Sources and Generate Stubs

This is a two-step process. In the first step you use the compiler to compile the source files, which contain the implementation of the remote interfaces and implementations, the server classes, and the client classes. In the second step you use the compiler to create stubs for the remote objects. RMI uses a remote object's stub class as a proxy in clients so that clients can communicate with a particular remote object.

Make Classes Network Accessible

In this step you make everything--the class files associated with the remote interfaces, stubs, and other classes that need to be downloaded to clients--accessible via a Web server.

Start the Application

Starting the application includes running the RMI remote object registry, the server, and the client.

4. Directory Services

4.1 Introduction

A directory service is a critical generic service, binding host computer names to addresses, services to clients, and so on. Directory services differ in the range of entities about which they hold information. A *name service* specifically holds name-to-address mappings, whereas *trading service* generally holds information to assist in matching client request to servers that are able to service the request type. In general, a directory service seeks to:

- Isolate distributed components from change by placing a level of 'indirection' between references to a component (referred to by name) and its address that identifies its location. Other name-to-name mappings e.g. user name to electronic mail identifiers are also stored. A hierarchical naming structure is usually used to implement meaningful and globally unique names.
- Provide a more user-friendly view of the distributed infrastructure. This is achieved by the definition of meaningful names that increase the chances that names are predicted, remembered and understood by human users

A major problem in distributed systems environment is that its configuration is subject to change. These changes are usually as a result of:

- Systems, users, processes, objects, peripheral hardware etc., being continually added, moved and removed.
- Parts of the communication path between users being changed by introducing or reconfiguring network components.
- Changing characteristics e.g. addresses and attributes of users, systems and distributed infrastructure components.

These changes often occur with little or no warning to systems or users of the distributed system. However, although the rate of change may be high, the change rate of a particular component is usually low compared with the number of times it is referenced by other components.

4.2 Terminology

4.2.1 Names, Identifiers, and Addresses

A *name* in a distributed system is a string of bits or characters that is used to refer to an entity. An entity in a distributed system can be practically anything. Typical examples include resources such as hosts, printers, disks, and files. Other well-known examples of entities that are often explicitly named are processes, users, mailboxes, newsgroups, Web pages, graphical windows, messages, network connections, and so on. Entities can be operated on. For example, a resource such as a printer offers an interface containing operations for printing a document, requesting the status of a print job, and the like.

To operate on an entity, it is necessary to access it, for which we need an ***access point***. An access point is yet another, but special, kind of entity in a distributed system. The name of an access point is called an **address**. The address of an access point of an entity is also simply called an address of that entity. An entity can offer more than one access point. As a comparison, a telephone can be viewed as an access point of a person, whereas the telephone number corresponds to an address. Indeed, many people nowadays have several telephone numbers, each number corresponding to a point where they can be reached. In a distributed system, a typical example of an access point is a host running a specific server, with its address formed by the combination of, for example, an IP address and port number (i.e., the server's transport-level address).

An entity may change its access points in the course of time. For example, when a mobile computer moves to another location, it is often assigned a different IP address than the one it had before. Likewise, when a person moves to another city or country, it is often necessary to change telephone numbers as well. In a similar fashion, changing jobs or Internet Service Provider, means changing your e-mail address. An address is thus just a special kind of name: it refers to an access point of an entity. Because an access point is tightly associated with an entity, it would seem convenient to use the address of an access point as a regular name for the associated entity. Nevertheless, this is hardly ever done. There are many benefits to treating addresses as a special type of name. For example, it is not uncommon to regularly reorganize a distributed system, so that a specific server, such as the one handling FTP requests, is now running on a different host than previously. The old machine on which the server used to be running may be reassigned to a completely different server, such as a back-up server for the local file system. In other words, an entity may easily change an access point, or an access point may be reassigned to a different entity. If an address is used to refer to an entity, we will have an invalid reference the instant the access point changes or is reassigned to another entity. For example, imagine that an organization's FTP service would be known only by the address of the host running the FTP server. As soon as that server is moved to another host, the whole FTP service would become inaccessible until all its users know the new address. In this case, it would have been much better to let the FTP service be known by a separate name, independent of the address of the associated FTP server.

Likewise, if an entity offers more than one access point, it is not clear which address to use as a reference. For instance, many organizations distribute their Web service across several servers. If we would use the addresses of those servers as a reference for the Web service, it is not obvious which address should be chosen as the best one. A much better solution would be to have a single name for the Web service, independent from the addresses of the different Web servers.

These examples illustrate that a name for an entity that is independent from its addresses, is often much easier and more flexible to use. Such a name is called *location independent.* In addition to addresses, there are other types of names that deserve special treatment, such as names that are used to uniquely identify an entity. A true ***identifier*** is a name that has the following properties:

1. An identifier refers to at most one entity.
2. Each entity is referred to by at most one identifier.
3. An identifier always refers to the same entity (i.e., it is never reused).

By using identifiers, it becomes much easier to unambiguously refer to an entity. For example, assume two processes each refer to an entity by means of an identifier. To check if the processes are referring to the same entity, it is sufficient to test if the two identifiers are equal. Such a test would not be sufficient if the two processes were using regular, non-identifying names. For example ''John Smith'' cannot be taken as a unique reference to a single person. Likewise, if an address can be reassigned to a different entity, we cannot use an address as an identifier. Consider the use of telephone numbers, which are reasonably stable in the sense that a number generally refers to the same person or organization. However, using a telephone number, as an identifier will not work, as it can be reassigned in the course of time.

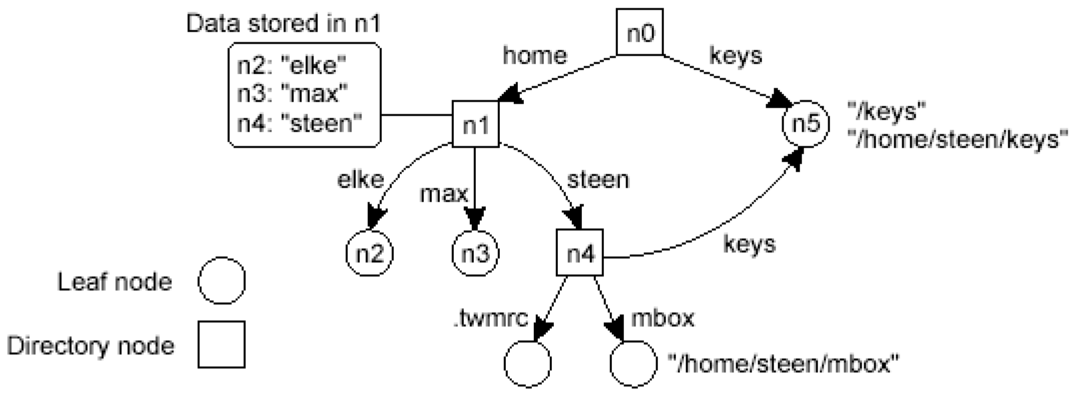
Another important type of name is that which is tailored to be used by humans, also referred to as *human-friendly names*. In contrast to addresses and identifiers, a human-friendly name is generally represented as a character string. These names appear in many different forms. For example, files in UNIX systems have character-string names that can be as long as 255 characters, and which are defined entirely by the user. Similarly, DNS names are represented as relatively simple case-insensitive character strings.

4.2.2 Name Spaces

Names in a distributed system are organized into what is commonly referred to as a *name space*. A name space can be represented as a labeled, directed graph with two types of nodes. A *leaf node* represents a named entity and has the property that it has no outgoing edges. A leaf node generally stores information on the entity it is representing—for example, its address—so that a client can access it. Alternatively, it can store the state of that entity, such as in the case of file systems in which a leaf node actually contains the complete file it is representing.

In contrast to a leaf node, a *directory node* has a number of outgoing edges, each labeled with a name, as shown below. Each node in a naming graph is considered as yet another entity in a distributed system, and, in particular, has an associated identifier. A directory node stores a table in which an outgoing edge is represented as a pair *(edge label, node identifier)*. Such a table is called a *directory table*.

The naming graph shown below has one node, namely *n0*, which has only outgoing and no incoming edges. Such a node is called the *root (node)* of the naming graph. Although it is possible for a naming graph to have several root nodes, for simplicity, many naming systems have only one. Each path in a naming graph can be referred to by the sequence of labels corresponding to the edges in that path, such as *N:<label-1, label-2, ..., label-n >* where *N* refers to the first node in the path. Such a sequence is called a ***path name.*** If the first node in a path name is the root of the naming graph, it is called an *absolute path name*. Otherwise, it is called a *relative path name.*



A Naming Graph

It is important to realize that names are always organized in a name space. As a consequence, a name is always defined relative only to a directory node. In this sense, the term absolute name is somewhat misleading. Likewise, the difference between global and local names can sometimes be confusing. A *global name* is a name that denotes the same entity, no matter where that name is used in a system.
In other words, a global name is always interpreted with respect to the same directory node. In contrast, a *local name* is a name whose interpretation depends on where that name is being used. Put differently, a local name is essentially a relative name whose directory in which it is contained is (implicitly) known. The figure shown above is an example of a *directed acyclic graph*. In such an organization, a node can have more than one incoming edge, but the graph is not permitted to have a cycle.

4.2.3 Name Resolution

Name spaces offer a convenient mechanism for storing and retrieving information about entities by means of names. More generally, given a path name, it should be possible to look up any information stored in the node referred to by that name. The process of looking up a name is called *name resolution*. To explain how name resolution works, consider a path name such as *N: <label-1, label-2... label-n>*. Resolution of this name starts at node *N* of the naming graph, where the name *label-1* is looked up in the directory table, and which returns the identifier of the node to which *label-1* refers. Resolution then continues at the identified node by looking up the name *label-2* in its directory table, and so on. Assuming that the named path actually exists, resolution stops at the last node referred to by *label-n*, by returning the content of that node. A name lookup returns the identifier of a node from where the name resolution process continues. In particular, it is necessary to access the directory table of the identified node.

4.3 Implementing a Directory Service

The directory service can vary in sophistication from a simple local ‘address book’ on a small-scale unsophisticated system, to a global service overseeing a large distributed infrastructure spanning many countries. A global directory is necessarily more sophisticated because maintaining up-to-date information on all resources can be a major management task. A scalable architecture is required based on partitioning the name space into domains, each of which is maintained by a separate administrative authority.
The major issues with respect to the implementation of a directory service are the potential size of the directory store, the frequency of clients’ request and fault tolerance. Three techniques are usually employed to implement a scalable, fault tolerant directory service:

1. Partition the service over multiple directory servers – Different parts of the directory’s namespace are assigned to different servers. The name space is usually hierarchical which allows different parts of the hierarchical tree to be assigned to different autonomous organizations, each responsible for that part of the tree. Because of the partitions, query and update requests can be handle faster than the case when using a single flat namespace. A name resolution protocol is required to determine which server is able to handle a lookup request.
2. Replicate directory servers for improved availability and reliability – When a replicated directory service is unavailable, the name resolution protocol must locate a replica. The main difficulty with replication is maintaining consistency across all replicas of directory service.
3. Cache results of directory queries to improve performance and reduce network traffic – The process of resolving names can be complex. If a name is resolved once, it will often need to be resolved again either by the same client or by a different client wanting to access the same entity. A cache in this context is a temporary database of previously resolved names. Cache inconsistency can be detected since object will not be found. Thus a cached entry will can always be the first to be used and an authoritative server used only in cases where cached entry fails.

4.3.1 Name Space Distribution

Name spaces for a large-scale, possibly worldwide distributed system, are usually organized hierarchically. As before, assume such a name space has only a single root node. To effectively implement such a name space, it is convenient to partition it into logical layers distinguishing the following three layers.

The *global layer* is formed by highest-level nodes, that is, the root node and other directory nodes logically close to the root. Nodes in the global layer are often characterized by their stability, in the sense that directory tables are rarely changed. Such nodes may represent organizations, or groups of organizations, for which names are stored in the name space.

The *administrational layer* is formed by directory nodes that together are managed within a single organization. A characteristic feature of the directory nodes in the administrational layer is that they represent groups of entities that belong to the same organization or administrational unit. For example, there may be a directory node for each department in an organization, or a directory node from which all hosts can be found. Another directory node may be used as the starting point for naming all users, and so forth. The nodes in the administrational layer are relatively stable, although changes generally occur more frequently than to nodes in the global layer.

Finally, the *managerial layer* consists of nodes that may typically change regularly. For example, nodes representing hosts in the local network belong to this layer. For the same reason, the layer includes nodes representing shared files such as those for libraries or binaries. Another important class of nodes includes those that represent user-defined directories and files. In contrast to the global and administrational layer, the nodes in the managerial layer are maintained not only by system administrators, but also by individual end users of a distributed system.

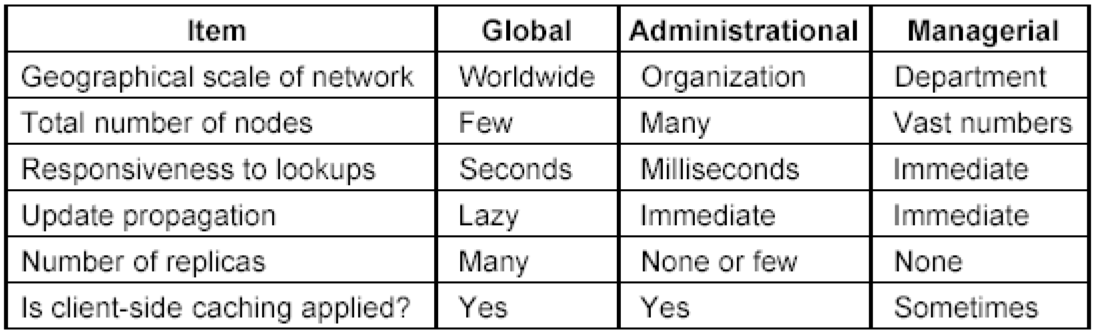
If we take a look at availability and performance, name servers in each layer have to meet different requirements. High availability is especially critical for name servers in the global layer. If a name server fails, a large part of the name space will be unreachable because name resolution cannot proceed beyond the failing server.

Performance is somewhat subtle. Due to the low rate of change of nodes in the global layer, the results of lookup operations generally remain valid for a long time. Consequently, those results can be effectively cached (i.e., stored locally) by the clients. The next time the same lookup operation is performed, the results can be retrieved from the client's cache instead of letting the name server return the results. As a result, name servers in the global layer do not have to respond quickly to a single lookup request. On the other hand, throughput may be important, especially in large-scale systems with millions of users.

The availability and performance requirements for name servers in the global layer can be met by replicating servers, in combination with client-side caching. Updates in this layer generally do not have to come into effect immediately, making it much easier to keep replicas consistent. Availability for a name server in the administrational layer is primarily important for clients in the same organization as the name server. If the name server fails, many resources within the organization become unreachable because they cannot be looked up. On the other hand, it may be less important that resources in an organization are temporarily unreachable for users outside that organization.

With respect to performance, name servers in the administrational layer have similar characteristics as those in the global layer. Because changes to nodes do not occur very often, caching lookup results can be highly effective, making performance less critical. However, in contrast to the global layer, the administrational layer should take care that lookup results are returned within a few milliseconds, either directly from the server or from the client's local cache. Likewise, updates should generally be processed quicker than those of the global layer. For example, it is unacceptable that an account for a new user takes hours to become effective. These requirements can generally be met by using high-performance machines to run name servers. In addition, client-side caching should be applied, combined with replication for increased overall availability.

Availability requirements for name servers at the managerial level are generally less demanding. In particular, it often suffices to use a single (dedicated) machine to run name servers at the risk of temporary unavailability. However, performance is crucial. Users expect operations to take place immediately. Because updates occur regularly, client-side caching is often less effective, unless special measures are taken. A comparison between name servers at different layers is shown below.



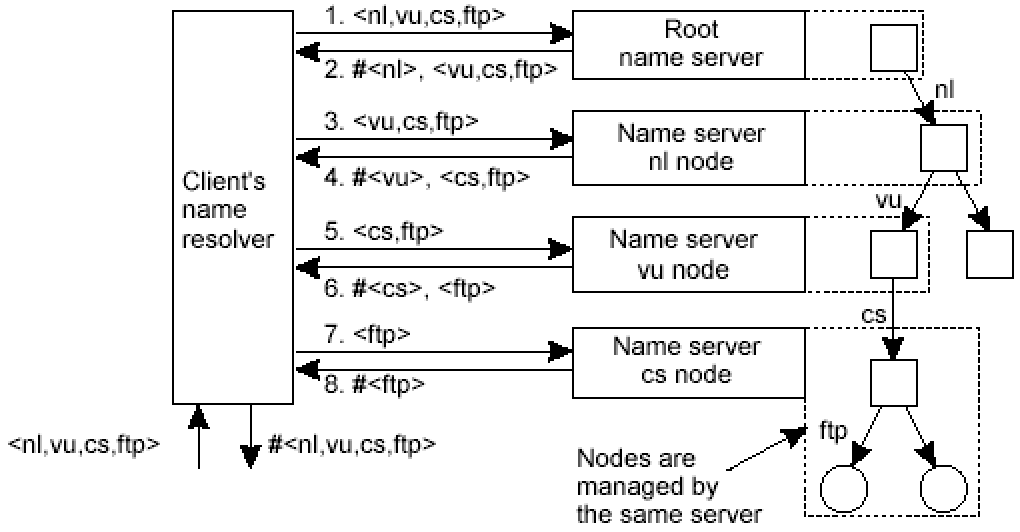
Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

4.3.2 Name Resolution

The distribution of a name space across multiple name servers affects the implementation of name resolution. Each client has access to a local ***name resolver***, which is responsible for ensuring that the name resolution process is carried out. For example, assume the (absolute) path name *root:<nl, vu, cs, ftp, pub, globe, index.txt>* is to be resolved. Using a URL notation, this path name would correspond to *ftp://ftp.cs.vu.nl/pub/globe/index.txt*. There are now two ways to implement name resolution.

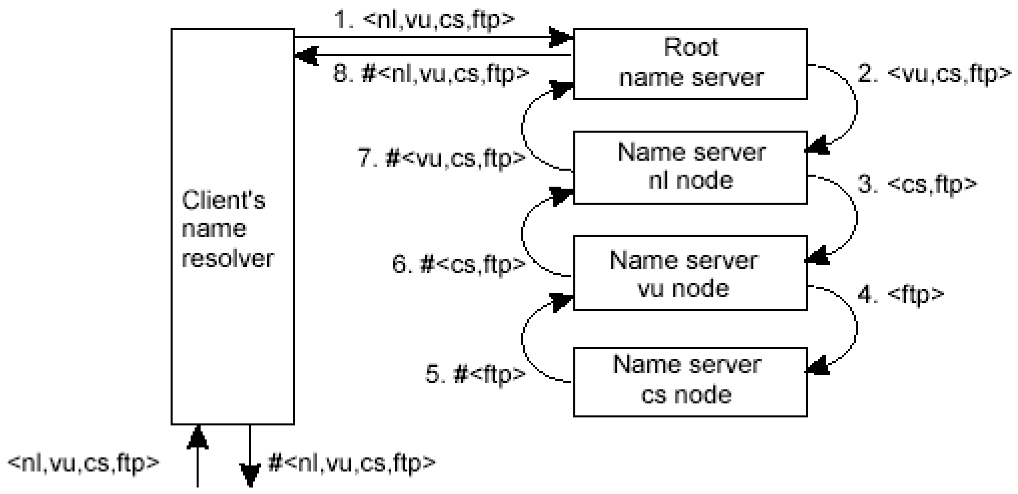
Iterative name resolution -In ***iterative name resolution***, a name resolver hands over the complete name to the root name server. It is assumed that the address where the root server can be contacted is well known. The root server will resolve the path name as far as it can, and return the result to the client. In our example, the root server can resolve only the label *nl*, for which it will return the address of the associated name server.

At that point, the client passes the remaining path name (i.e., *nl:<vu, cs, ftp, pub, globe, index.txt>*) to that name server. This server can resolve only the label *vu*, and returns the address of the associated name server, along with the remaining path name *vu:<cs, ftp, pub, globe, index.txt>*. The client's name resolver will then contact this next name server, which responds by resolving the label *cs*, and subsequently also *ftp*, returning the address of the FTP server along with the path name *ftp:<pub, globe, index.txt>*. The client then contacts the FTP server, requesting it to resolve the last part of the original path name. The FTP server will subsequently resolve the labels *pub*, *globe*, and *index.txt*, and transfer the requested file (in this case using FTP). This process of iterative name resolution is shown below. (The notation #<*cs*> is used to indicate the address of the server responsible for handling the node referred to by <*cs*>.)



The principle of iterative name resolution.

Recursive name resolution - An alternative to iterative name resolution is to use recursion during name resolution. Instead of returning each intermediate result back to the client's name resolver, with ***recursive name resolution,*** a name server passes the result to the next name server it finds. So, for example, when the root name server finds the address of the name server implementing the node named *nl*, it requests that name server to resolve the path name *nl:<vu, cs, ftp, pub, globe, index.txt>*. Using recursive name resolution as well, this next server will resolve the complete path and eventually return the file *index.txt* to the root server, which, in turn, will pass that file to the client's name resolver. As in iterative name resolution, the last resolution step, namely contacting the FTP server and asking it to transfer the indicated file, is generally carried out as a separate process by the client.



The principle of recursive name resolution.

5. Fault Tolerance

5.1 Basic Concepts

To understand the role of fault tolerance in distributed systems we first need to take a closer look at what it actually means for a distributed system to tolerate faults. Being fault tolerant is strongly related to what are called **dependable systems**.

Dependability is the trustworthiness of a computer system. It is very difficult to predict the behavior of a system under real workloads. Unpredictable fluctuation of workload levels, system infrastructure component utilization, software bugs and malfunctioning of hardware can result into unusual circumstances that trigger major system failure. A distributed system can be composed of a complex web of interconnected heterogeneous components that can result into the implementation of a fragile system.

The designer of a distributed system should take into consideration that services may fail in a variety of ways and seek to minimize the risk of failures by assuming that unlikely events that cause failure will occur at the worst possible times.

Dependability is a term that covers a number of useful requirements for distributed systems including the following

1. Availability
2. Reliability
3. Safety
4. Maintainability

Availability is defined as the property that a system is ready to be used immediately. In general, it refers to the probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users. In other words, a highly available system is one that will most likely be working at a given instant in time.

Reliability refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly reliable system is one that will most likely continue to work without interruption during a relatively long period of time. This is a subtle but important difference when compared to availability. If a system goes down for one millisecond every hour, it has an availability of over 99.9999 percent, but is still highly unreliable. Similarly, a system that never crashes but is shut down for two weeks every August has high reliability but only 96 percent availability. The two are not the same.

Safety refers to the situation that when a system temporarily fails to operate correctly, nothing catastrophic happens. For example, many process control systems, such as those used for controlling nuclear power plants or sending people into space, are required to provide a high degree of safety. If such control systems temporarily fail for only a very brief moment, the effects could be disastrous.

Maintainability refers to how easy a failed system can be repaired. A highly maintainable system may
Often, dependable systems are also required to provide a high degree of security, especially when it comes to issues such as integrity.

A system is said to **fail** when it cannot meet its promises. In particular, if a distributed system is designed to provide its users with a number of services, the system has failed when one or more of those services cannot be (completely) provided.

An **error** is a part of a system's state that may lead to a failure. For example, when transmitting packets across a network, it is to be expected that some packets have been damaged when they arrive at the receiver. Damaged in this context means that the receiver may incorrectly sense a bit value (e.g., reading a 1 instead of a 0), or may even be unable to detect that something has arrived.

The cause of an error is called a **fault**. Clearly, finding out what caused an error is important. For example, a wrong or bad transmission medium may easily cause packets to be damaged. In this case, it is relatively easy to remove the fault.

Building dependable systems closely relates to controlling faults. A distinction is made between preventing, removing, and forecasting faults. For our purposes the most important issue is **fault tolerance**, meaning that a system can provide its services even in the presence of faults. Faults are generally classified as transient, intermittent, or permanent.

Transient faults occur once and then disappear. If the operation is repeated, the fault goes away. A bird flying through the beam of a microwave transmitter may cause lost bits on some network (not to mention a roasted bird). If the transmission times out and is retried, it will probably work the second time.

An **intermittent fault** occurs, then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose.

A **permanent fault** is one that continues to exist until the faulty component is repaired. Burnt-out chips, software bugs, and disk head crashes are examples of permanent faults.

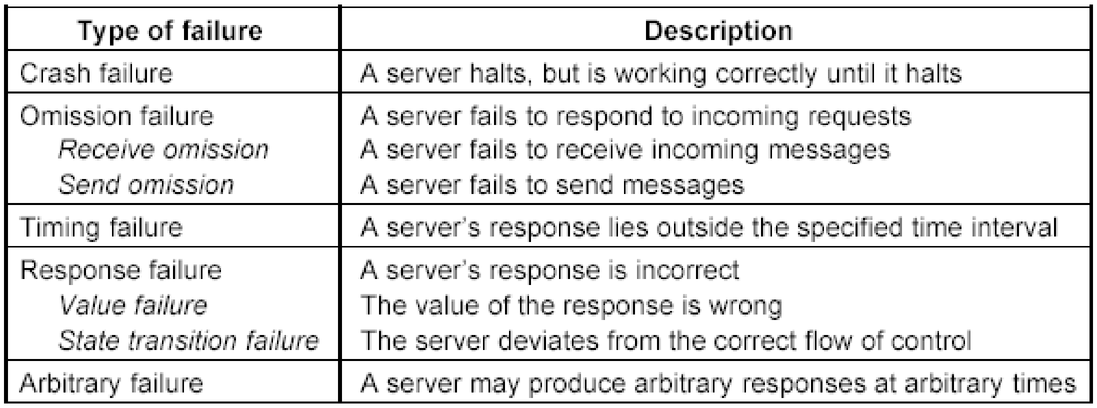
The following design approaches have been suggested for building dependable distributed systems that exhibit a high level of stability and fault tolerance.

- Interconnect for good reasons – A complex web of interconnected sub-systems should be avoided. The more complex the web, the more fragile is the overall system. Interconnect only when defined objectives are meet by doing so.
- Support only necessary services – It is tempting to implement a wide variety of services by default, some of which may not be necessary. For security reasons and in order to minimize maintenance, services should only be implemented if there is need. An enabled service may represent an unwanted sub-system that may compromise the robustness of the entire system.
- Include self-diagnosis and authentication mechanism - A distributed system should attempt to monitor itself, identify any inconsistencies and gracefully shutdown any faulty components and restart or switch to a replica. Clients should be authenticated to avoid malicious failures.
- Design for fault tolerance – By assuming that failures will occur, we are lead to a design systems that are fault tolerant to some classes of failures. A fault tolerant system is one that will continue to operate as required in the presence of faults.
- Design for scale – When designing distributed systems, take into consideration the fact that major performance problems can arise when the number of users or components grows to a large number. Scale should be a design parameter rather than a capacity planning exercise after system development.
- Avoid mechanisms that cascade failures – In systems under heavy workloads or when some failures occur, failures can cascade as other mechanisms are activated.

5.2 Failure Models

A system that fails is not adequately providing the services it was designed for. If we consider a distributed system as a collection of servers that communicate with each other and with their clients, not adequately providing services means that servers, communication channels, or possibly both, are not doing what they are supposed to do. However, a malfunctioning server itself may not always be the fault we are looking for. If such a server depends on other servers to adequately provide its services, the cause of an error may need to be searched for somewhere else. Such dependency relations appear in abundance in distributed systems. A failing disk may make life difficult for a file server that is designed to provide a highly available file system. If such a file server is part of a distributed database, the proper working of the entire database may be at stake, as only part of its data may actually be accessible.

The figure below shows a typical failure classification scheme



Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	A server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

A **crash failure** occurs when a server prematurely halts, but was working correctly until it stopped. An important aspect with crash failures is that once the server has halted, nothing is heard from it anymore. A typical example of a crash failure is an operating system that comes to a grinding halt, and for which there is only one solution: reboot. Many personal computer systems suffer from crash failures so often that people have come to expect them to be normal.

An **omission failure** occurs when a server fails to respond to a request. Several things might go wrong. In the case of a receive omission failure, the server perhaps never got the request in the first place. Note that it may well be the case that the connection between a client and a server has been correctly established, but that there was no thread listening to incoming requests. Also, a receive omission failure will generally not affect the current state of the server, as the server is unaware of any message sent to it. Likewise, a send omission failure happens when the server has done its work, but somehow fails in sending a response. Such a failure may happen, for example, when a send buffer overflows while the server was not prepared for such a situation. Note that, in contrast to a receive omission failure; the server may now be in a state reflecting that it has just completed a service for the client. As a consequence, if the sending of its response fails, the server may need to be prepared that the client will reissue its previous request. Other types of omission failures not related to communication may be caused by software errors such as infinite loops or improper memory management by which the server is said to ‘‘hang.’’

Timing failures occur when the response lies outside a specified real-time interval. Providing data too soon may easily cause trouble for a recipient if there is not enough buffer space to hold all the incoming data. More common, however, is that a server responds too late, in which case a *performance* failure is said to occur.

A serious type of failure is a **response failure**, by which the server's response is simply incorrect. Two kinds of response failures may happen. In the case of a value failure, a server provides the wrong reply to a request. For example, a search engine that systematically returns Web pages not related to any of the used search terms, has failed. A state transition failure happens when the server reacts unexpectedly to an incoming request. For example, if a server receives a message it cannot recognize, a state transition failure happens if no measures have been taken to handle such messages. In particular, a faulty server may incorrectly take default actions it should never have initiated.

The most serious are **arbitrary failures**, also known as **byzantine failures**. In effect, when arbitrary failures occur, clients should be prepared for the worst. In particular, it may happen that a server is producing output it should never have produced, but which cannot be detected as being incorrect. Worse yet a faulty server may even be maliciously working together with other servers to produce intentionally wrong answers. This situation illustrates why security is also considered an important requirement when talking about dependable systems.

Arbitrary failures are closely related to crash failures. The definition of crash failures as presented above is the most benign way for a server to halt. They are also referred to as **fail-stop failures**. In effect, a fail-stop server will simply stop producing output in such a way that its halting can be detected by other processes. For example, the server may have been so friendly to announce it is about to crash. Of course, in real life, servers halt by exhibiting omission or crash failures, and are not so friendly as to announce they are going to stop. It is up to the other processes to decide that a server has prematurely halted. However, in such **fail-silent systems**, the other process may incorrectly conclude that a server has halted. Instead, the server may just be unexpectedly slow, that is, it is exhibiting performance failures. Finally, there are also occasions in which the server is producing random output, but other processes can recognize this output as plain junk. The server is then exhibiting arbitrary failures, but in a benign way. These faults are also referred to as being **fail-safe**.

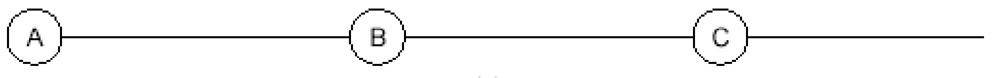
5.3 Failure Masking by Redundancy

If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes. The key technique for masking faults is to use redundancy. Three kinds are possible: information redundancy, time redundancy, and physical redundancy.

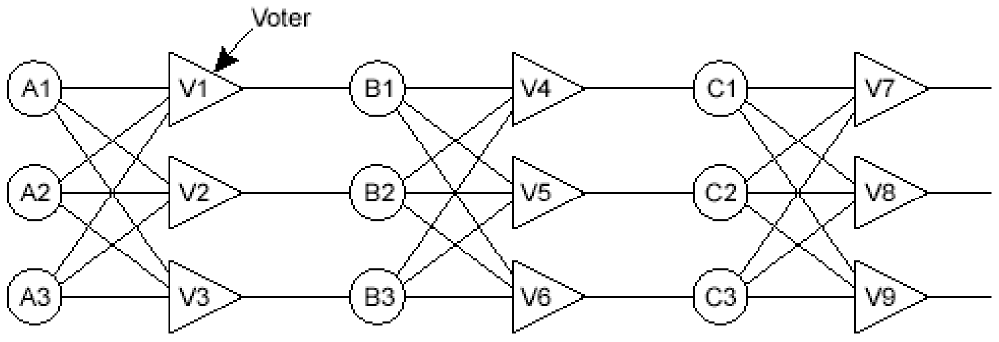
With **information redundancy**, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

With **time redundancy**, an action is performed, and then, if need be, it is performed again. Using transactions is an example of this approach. If a transaction aborts, it can be redone with no harm. Time redundancy is especially helpful when the faults are transient or intermittent.

With **physical redundancy**, extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. Physical redundancy can be done either in hardware or in software. For example, extra processes can be added to the system so that if a small number of them crash, the system can still function correctly. In other words, by replicating processes, a high degree of fault tolerance may be achieved. Physical redundancy is a well-known technique for providing fault tolerance. It is used in biology (mammals have two eyes, two ears, two lungs, etc. It has also been used for fault tolerance in electronic circuits for years; it is illustrative to see how it has been applied there. Consider, for example, the circuit of Fig. 7-2(a). Here signals pass through devices *A*, *B*, and *C*, in sequence. If one of them is faulty, the final result will probably be incorrect.



(a)



(b)

In Fig. (b), each device is replicated three times. Following each stage in the circuit is a triplicated voter. Each voter is a circuit that has three inputs and one output. If two or three of the inputs are the same, the output is equal to that input. If all three inputs are different, the output is undefined. This kind of design is known as **TMR** (**Triple Modular Redundancy**). Suppose that element *A*2 fails. Each of the voters, *V*1, *V*2, and *V*3 gets two good (identical) inputs and one rogue input, and each of them outputs the correct value to the second stage. In essence, the effect of *A*2 failing is completely masked, so that the inputs to *B*1, *B*2, and *B*3 are exactly the same, as they would have been had no fault occurred. Now consider what happens if *B*3 and *C*1 are also faulty, in addition to *A*2. These effects are also masked, so the three final outputs are still correct. At first it may not be obvious why three voters are needed at each stage. After all, one voter could also detect and pass though the majority view. However, a voter is also a component and can also be faulty. Although not all fault-tolerant distributed systems use TMR, the technique is very general, and should give a clear feeling for what a fault-tolerant system is, as opposed to a system whose individual components are reliable but whose organization is not fault tolerant. An alternative mechanism is to design a system using modular components that can be replaced easily when a fault occurs.

A technique similar to TMR, known as **N-version programming**, can be employed by software components. In a three-version method, three different versions of the same program are developed independently by three different project teams. When the program is executed in a production environment the correctness of processing is tested ad the correct output chosen either by a majority or validating the various outputs against a defined acceptance test. However, the main approach is to employ transaction processing and recovery mechanism.

6. Replication in Distributed Computing Systems

6.1 Introduction

In a distributed system environment, the system is composed of a set of *replicas* over which operations must be performed. Clients normally request for operations. A service is replicated when multiple identical instances known as replicas of the service appear on different nodes in a distributed system. Shared state consistency among replicas ensures that client request to access shared resource or information are handled in a completely consistent manner. Each replica holds an instance of a shared system state.

Communication between different system components (clients and replicas) takes place by exchanging messages. In this context, distributed systems distinguish between the *synchronous* and the *asynchronous* system model. In the synchronous model there is a known bound on the relative process speed and on the message transmission delay, while no such bounds exist in the asynchronous model. The key difference is that the synchronous system allows *correct crash detection*, while the asynchronous system does not (i.e., in an asynchronous system, when some process *p* thinks that some other process *q* has crashed, *q* might in fact not have crashed). Incorrect crash detection makes the development of replication algorithm more difficult. Fortunately, much of the complexity can be hidden in the so-called *group communication primitives*.

Databases on the other hand are not concerned by the fundamental differences between synchronous and asynchronous systems for the following reason: databases accept to live with *blocking* protocols (a protocol is said to be blocking if the crash of some process may prevent the protocol from terminating). Blocking protocols are even simpler to design than non-blocking protocols based on the synchronous model. Distributed systems usually look for non-blocking protocols.

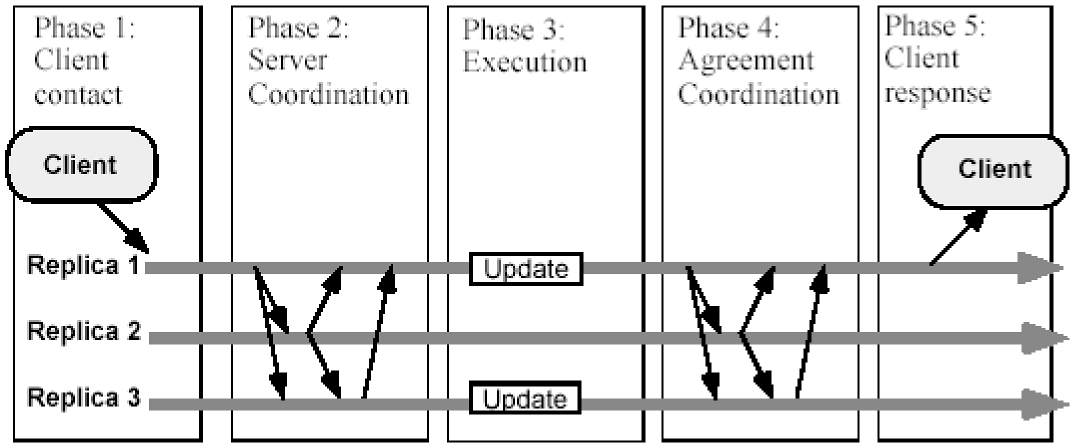
Replicas that hold the most recently written state data are said to hold **authoritative data.** Ideally, a client request can be directed to any replica if all replicas have consistency of sate; hold authoritative data. When multiple servers handle client requests, this improves availability and performance by reducing the load per replicated server. The number of servers required is determined by the level of performance and fault-tolerance required and also by the size of the client request workload.

The placement of replicas is another important issue and depends on whether replication is primarily to improve performance or to improve availability. Placing replicas in different parts of the network improves availability when the network fails by ensuring that replicas can be found in areas where client requests are concentrated and in network partitions. However, for improved performance, one or more replicas should be placed nearest to the client workstation, which contributes most server utilization in order to reduce network overhead.

6.2 Functional Model

A replication protocol can be described using five generic phases. These phases represent important steps in the protocol and will be used to characterize the different approaches. As we will later show, some replication techniques may skip some phases, order them in a different manner, iterate over some of them, or merge them into a simpler sequence. Thus, the protocols can be compared by the way they implement each one of the phases and how they combine the different phases. In this regard, an abstract replication protocol can be described as a sequence of the following five phases:

1. **Request (RE):** the client submits an operation to one or more replicas.
2. **Server coordination (SC):** the replica servers coordinate with each other to synchronize the execution of the operation.
3. **Execution (EX):** the operation is executed on the replica servers.
4. **Agreement coordination (AC):** the replica servers agree on the result of the execution.
5. **Response (END):** the outcome of the operation is transmitted back to the client.



This functional model represents the basic steps of replication:

- Submission of an operation
- Coordination among the replicas (e.g., to order concurrent operations)
- Execution of the operation
- Further coordination among the replicas (e.g., to guarantee atomicity)
- Response to the client.

The differences between protocols arise due to the different approaches used in each phase, which in some cases, obviate the need for some other phase (e.g., when messages are ordered based on an atomic broadcast primitive, the agreement coordination phase is not necessary since it is already performed as part of the process or ordering the messages).

Request Phase - During the request phase, a client submits an operation to the system. This can be done in two ways: the client can directly send the operation to all replicas or the client can send the operation to one replica which will them send the operation to all others as part of the server coordination phase.

In distributed systems, a clear distinction is made between replication techniques depending on whether the client sends the operation directly to all copies (e.g. active replication) or to one copy (e.g. passive replication). However, in distributed databases, clients never contact all replicas, they always send the operation to one copy. The reason is very simple: replication should be transparent to the client. Being able to send an operation to all replicas will imply the client has knowledge about the data location, schema, and distribution, which is not practical for any database of average size. This is knowledge intrinsically tied to the database nodes; thus, client must always submit the operation to one node, which will then send it to all others

Distributed systems distinguish between deterministic and non-deterministic replica behavior. **Deterministic** replica behavior assumes that when presented with the same operations in the same order, replicas will produce the same results. Such an assumption is very difficult to make in a database. Thus, if the different replicas have to communicate anyway in order to agree on a result, they can as well exchange the actual operation. By shifting the burden of broadcasting the request to the server, the logic necessary at the client side is greatly simplified at the price of (theoretically) reducing fault tolerance. If fault tolerance is necessary, a back up system can be used, but this is totally transparent to the client.

Server Coordination Phase - During the server coordination phase, the different replicas try to find an order in which the operations need to be performed. This is the point where protocols differ the most in terms of ordering strategies, ordering mechanisms, and correctness criteria. In terms of ordering strategies, databases order operations according to data dependencies. That is, all operations must have the same data dependencies at all replicas. It is because of this reason that operation semantics play an important role in database replication: an operation that only reads a data item is not the same as an operation that modifies that data item since the data dependencies introduced are not the same in the two cases. If there are no direct or indirect dependencies between two operations, they do not need to be ordered because the order does not matter. Distributed systems, on the other hand, are commonly based on very strict notions of ordering. From causality, which is based on *potential* dependencies without looking at the operation semantics, to total order (either causal or not) in which all operations are ordered regardless of what they are.

Execution Phase - The execution phase represents the actual performing of the operation. It does not introduce many differences between protocols, but it is a good indicator of how each approach treats and distributes the operations. This phase only represents the actual execution of the operation; the applying of the update is typically done in the Agreement Coordination Phase, even though applying the update to other copies may be done by re-executing the operations.

Agreement Coordination Phase - During this phase, the different replicas make sure that they all do the same thing. This phase is interesting because it brings up some of the fundamental differences between protocols. In databases, this phase usually corresponds to a Two Phase Commit Protocol (2PC) during which it is decided whether the operation will be committed or aborted. This phase is necessary because in databases, the Server Coordination phase takes care only of ordering operations. Once the ordering has been agreed upon, the replicas need to ensure everybody agrees to actually committing the operation. Note that being able to order the operations does not necessarily mean the operation will succeed. In a database, there can be many reasons why an operation succeeds at one site and not at another (load, consistency constraints, interactions with local operations). This is a fundamental difference with distributed systems where once an operation has been successfully ordered (in the Server Coordinator phase) it will be delivered (i.e., “performed”) and there is no need to do any further checking.

Client Response Phase - The client response phase represents the moment in time when the client receives a response from the system. There are two possibilities: either the response is sent only after everything has been settled and the operation has been executed, or the response is sent right away and the propagation of changes and coordination among all replicas is done afterwards. In the case of databases, this distinction leads to two protocols:

1. Eager or synchronous (no response until everything has been done) and
2. Lazy or asynchronous (immediate response, propagation of changes is done afterwards)

In distributed systems, the response takes place only after the protocol has been executed and no discrepancies may arise. The client response phase is of increasing importance given the proliferation of applications for *mobile* users, where a copy is not always connected to the rest of the system and it does not make sense to wait until updates take place to let the user see the changes made.

6.3 Replication Control

Replica control mechanism are required to ensure that replicas continue to hold authoritative data when a request is sent to one ore more replicas with the effect of changing or updating state data. **Strict** (strong) **consistency** techniques ensure that all replicas hold the same authoritative data at any one given time. This is achieved if:

- All replicas receive every shared state data update request
- All replicas process received requests in the same sequence.

Thus both the correct reception of update requests and the order of processing by replicas are important. Correct sequence implies that the order in which they were issued by a single client. Note that it cannot be guaranteed that updates will be received in the correct sequence due to varying network delays. Correct processing can only be achieved by assigning unique sequence identifiers to update requests. However, one update request may precede or follow another, or two update requests may be unrelated

For example, if client X send update request A to server S, causing update request B also to be sent by client Y to server S, then request A should be processed before request B by the replicas of server S. The two requests are said to be *causally* related. If however, the relationship between requests A and B is *commutative,* that is, processing the requests in the order request A followed by request B, has exactly the same effect as processing in the reverse order, then the order of processing is not important.

Because of the difficulties of achieving the above requirements, strict consistency techniques do not generally scale well. The number of replicas is therefore kept to a minimum. There are two main strict consistency techniques are:

1. Active replication
2. Passive replication

1. Active replication

This is a strict consistency technique where a client request can update any replica, but the updates must also be sent to all other replicas and updates processed in the correct order. Client requests are blocked until the update has been propagated successfully to all replicas. Client read requests could be directed to any replica since every replica has the same view of state data.

Active replication, also called the *state machine* approach, is a non-centralized replication technique. Its key concept is that all replicas receive and process the same sequence of client requests. Consistency is guaranteed by assuming that, when provided with the same input in the same order, replicas will produce the same output. This assumption implies that servers process requests in a *deterministic* way.

Clients do not contact one particular server, but address servers as a group. In order for servers to receive the same input in the same order, client requests can be propagated to servers using an atomic broadcast. Weaker communication primitives can also be used if semantic information about the operation is known (e.g., two requests that commute do not have to be delivered at all servers in the same order).

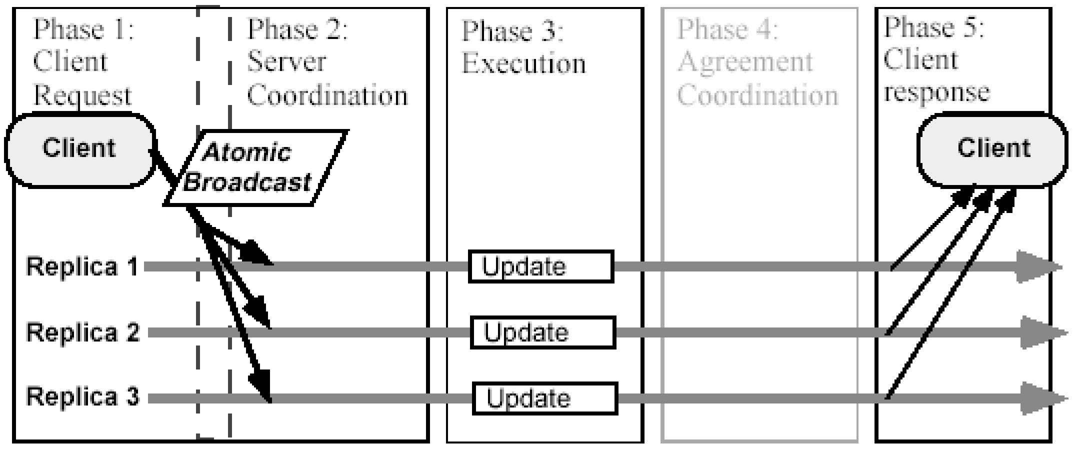
The main advantage of active replication is its simplicity (e.g., same code everywhere) and failure transparency. Failures are fully hidden from the clients, since if a replica fails, the requests are still processed by the other replicas. The determinism constraint is the major drawback of this approach. Although one might also argue that having all the processing done on all replicas consumes too much resources.

There are two major difficulties:

- Firstly, the need for synchronous update can impose a large processing and communication overhead and is generally not a scalable approach
- Secondly, maintaining replica consistency is difficult when not all of the replicas are operational. If any replica is not available at the time the update is propagated, it will be inconsistent with all other replicas. In this case, replica consistency cannot be guaranteed. The use of an additional protocol is required e.g. maintaining a directory of replica state to provide a mechanism for managing replica updates by providing a recovery mechanism when non-operational replicas become operational.

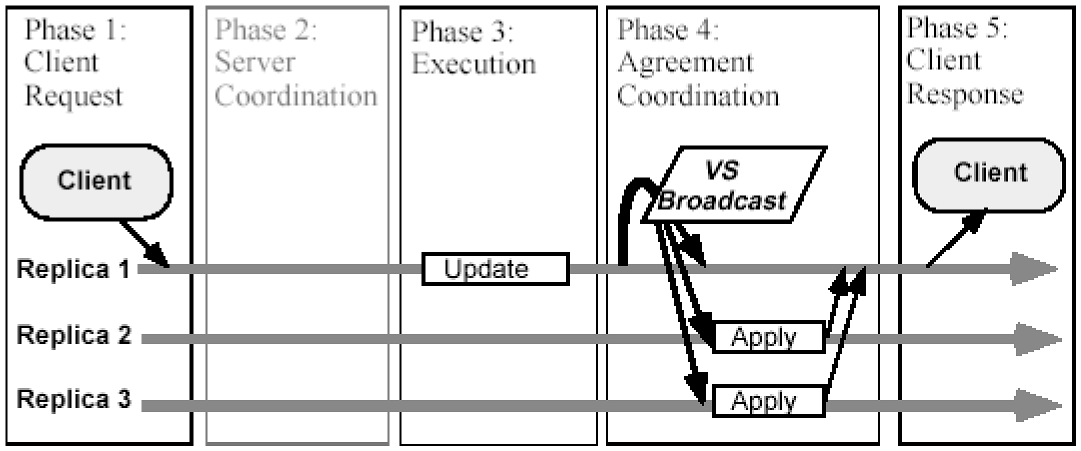
The figure below depicts the active replication technique using an atomic broadcast as communication primitive. In active replication, phases **RE** and **SC** are merged and phase **AC** is not used. The following steps are involved in the processing of an update request in the active replication, according to our functional model:

1. The client sends the request to the servers using an atomic broadcast.
2. Server coordination is given by the total order property of the Atomic Broadcast.
3. All replicas execute the request in the order they are delivered.
4. No coordination is necessary, as all replica process the same request in the same order. Because replicas are deterministic, they all produce the same results.
5. All replicas send back their result to the client, and the client typically only waits for the first answer (the others are ignored).



2. Passive (primary backup) Replication

The basic principle of passive replication, also called p*rimary backup* replication, is that clients send their requests to a primary, which executes the requests and sends update messages to the backups. The backups do not execute the invocation, but apply the changes produced by the invocation execution at the primary (i.e., updates). By doing this, no determinism constraint is necessary on the execution of invocations, the main disadvantage of active replication. Updates may be forwarded individually, or the whole database (file) may be downloaded to all replicas. In the **primary backup** method, read requests can be directed only to the primary replica because backup replicas cannot guarantee that the state data is authoritative.



The five steps of our framework are the following:

1. The client sends the request to the primary.
2. There is no initial coordination.
3. The primary executes the request.
4. The primary coordinates with the other replicas by sending the update information to the backups.
5. The primary sends the answer to the client.

An election protocol is required to designate one of the replicas as the new primary replica if the primary fails or becomes inaccessible due to network partitioning. There are two kinds of election algorithms:

- Ring algorithm and
- Bully election algorithm

In both algorithms each replica is given a unique number. In general, if the primary fails, the replica with the highest number takes over as the new primary. In the ring algorithm, each replica knows which replica succeeds it in priority order. When a replica notices that the primary has failed, it constructs an 'election' message to be sent to its successor inserting its unique number. If the successor is down, its sent to the next in line. When a successor receives an election message, it simply adds it number to the message and passes it on. Eventually, the original sender receives the message. A coordinator message is then sent informing everyone who the new primary replica is

The bully election algorithm is more contentious as the name suggests. A replica will assume the role of the new primary if two conditions are met:

- It sent an election message to all replicas with higher unique numbers
- No one responded.

If a successor received an election message, it simply responds with OK ('I am alive and well') message and the replica relinquishes control to it. The successor also needs to meet the conditions. Eventually, the replica with highest unique number will meet the conditions. Election algorithms are frequently used in distributed systems whenever any process amongst a group of peer processes needs to assume the coordinating role.

Passive replication can tolerate non-deterministic servers (e.g., multi-threaded servers) and uses little processing power when compared to other replication techniques. However, passive replication suffers from a high reconfiguration cost when the primary fails.

Communication between the primary and the backups has to guarantee that updates are processed in the same order, which is the case if primary backup communication is based on FIFO channels. However, only FIFO channels are not enough to ensure correct execution in case of failure of the primary. For example, consider that the primary fails before all backups receive the updates for a certain request, and another replica takes over as a new primary. Some mechanism has to ensure that updates sent by the new primary will be "properly" ordered with regard to the updates sent by the faulty primary. VSCAST is a mechanism that guarantees these constraints, and can usually be used to implement the primary backup replication technique

NB: View Synchronous Broadcast (VSCAST) -It is defined in the context of a group *g*, and is based on the notion of *a sequence of views* $v0(g); v1(g); : : : ; vi(g); : : :$of group *g*. Each view $vi(g)$ defines the composition of the group at same time *t*, i.e. the members of the group that are perceived as being correct at time *t*. Whenever a process *p* in some view $vi(g)$ is suspected to have crashed, or some process *q* wants to join, a new view $vi+1(g)$ is installed, which reflects the membership change. Roughly speaking, VSCAST of message *m* by some member of the group *g* currently in view $vi(g)$ ensures the following property: if one process *p* in $vi(g)$ delivers *m* before installing view $vi+1(g)$, then no process installs view $vi+1(g)$ before having first delivered *m*.

6. Clocks

6.1 Introduction

In single CPU systems, critical regions, mutual exclusion and other synchronization problems are solved using methods such as semaphores and monitors. These methods are not suitable to be used in distributed systems because they invariably rely on the existence of shared memory. For example, two processes that are using a semaphore to interact must be able to access the semaphore by having it stored in the kernel, and execute system calls to access it. Consequently, in distributed systems, simple matters such as determining whether event A happened before or after event B is difficult.

6.2 Clock Synchronization

Synchronization in distributed systems is more complicated than in centralized systems; distributed systems use distributed algorithms. It is usually not desirable to collect all the information about the system in one place. In general, distributed algorithms have the following properties:

- The relevant information is scattered among multiple machines.
- Processes make decisions based on the local information.
- A single point of failure in the system should be avoided.
- No common clock or other precise global time source exists

These points basically say that it is unacceptable to collect all information in a single place, which will result to a single point of failure. Secondly, time is unambiguous in centralized system. When a process wants to know the time, it makes a system call and the kernel tells it. If process A asks for time, and then a little later process B asks for the time, the value that B gets will be higher or equal to the value A got. In a distributed system to achieve this is not trivial since all machines run their own local clocks. The effect of having all the clocks not synchronized can be disastrous.

6.3 Logical Clocks

Nearly all computers have a circuit for keeping track of time. Despite the widespread use of the word 'clock' to refer to these devices, they are not clocks in the real sense. **Timer** is rather a better word. A computer timer is usually a precisely machined quartz crystal. A quartz crystal oscillates at a well-defined frequency depending on how it's cut, and the amount of tension. Associated with each crystal are two registers, a **counter** and a **holding register**. Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from holding register.
In this way, it's possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency. Each interrupt is called one **clock tick.** At every clock tick, the interrupt service procedure adds one to the time stored in memory. In this way, the clock is kept up-to-date.

In a single computer using a single clock, it does not matter much if this clock is off by a small amount. Since all processes on the same machine use the same clock, they will still be internally consistent. As soon as multiple CPUs are introduced, each with its own clock, the situation changes. Although the frequency at which the crystal oscillates is fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency. In practice, when a system has *n* computers, all *n* crystals will run at slightly different rates, causing the clocks gradually to get out of synch and give different values when read out. The difference in values is called **clock skew**. Consequently, programs that expect the time associated with a file, object, process or message to be correct and independent of the machine on which it was generated can fail. The issue of synchronizing these clocks to produce a single unambiguous time standard is therefore very important.

Clock synchronization need not to be absolute. If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable. Furthermore, what usually is not that all processes agree on exactly what time it is, but rather, that they agree on the order in which events occur. For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time agree with the real time. Thus for a certain class of algorithms, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time. These kinds of clocks are referred to as **logical clocks**.

When additional constraints are put that the clocks must not only be the same, but also must not deviate from the real time by more than a certain amount, the clocks are called **physical clocks**. Lamport's algorithm can be used to synchronize logical clocks. To synchronize logical clocks, Lamport defined a relation called **happens before**. The expression $a \rightarrow b$ is read "*a* happens before *b*" and means that all processes agree that first event *a* occurs, then afterwards, event *b* occurs. The happens-before relation can be observed directly in two situations:

- If *a* and *b* are events in the same process, and *a* occurs before *b*, then $a \rightarrow b$ is true
- If *a* is the event of a message being sent by one process, and *b* is the event of the message being received by another process, then $a \rightarrow b$ is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite amount of time to arrive.

Happens-before is a transitive relation, so if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. If two events *x* and *y*, happen in different processes that do not exchange messages, then $x \rightarrow y$ is not true, but neither is $y \rightarrow x$. These events are said to be **concurrent**, which simply means that nothing can be said about when they happened.

We need a way of measuring time such that for every event, *a*, we can assign a time value C (*a*) on which all processes agree. These time values must have the property that if $a \rightarrow b$, then C (*a*) < C (*b*). To rephrase the condition stated earlier, if *a* and *b* are two events within the same process and *a* occurs before *b*, then C (*a*) < C (*b*). Similarly, if *a* is the sending of a message and *b* is the reception of the message by another process, then C (*a*) and C (*b*) must be assigned in such a way that everyone agrees on the values of C (*a*) and C (*b*) with C (*a*) < C (*b*). In addition, the clock time, C, must always go forward, never backward.

Suppose that two processes in different computers are interacting and one sends a message say at time 8, which arrives at a time 6 in the other computer. This is clearly inconsistent and such a situation must be prevented. **Lamport's algorithm** provides a solution. Since the message was sent at time 8, it must arrive at time 9 or later. Each message carries the sending time, according to the sender's clock. When a message arrives and the receivers clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time. With the addition that between every two events, the clock must tick at least once. If a process sends or receives two messages in quick succession, it must advance its clock by at least one tick in between them. In some situations, an additional requirement is desirable: no two events occur at exactly the same time.

We are now able to assign time to all events in a distributed system subject to the following conditions:

1. If a happens before b in the same process, C (*a*) < C (*b*).
2. If a and b represent the sending and receiving of a message, C (*a*) < C (*b*).
3. For all events a and b C (*a*) $\neq$ C (*b*).

This algorithm gives us a way to provide a total ordering of all events in the system.

6.3 Physical Clocks

Although Lamport's algorithm gives an unambiguous event ordering, the time values assigned to events are not necessarily close to the actual times at which they occur. In some systems, for example real time systems, the actual clock time is important. For these systems, external physical clocks are required. For reasons of efficiency and redundancy, multiple physical clocks are generally considered desirable, which yields two problems:

- How do we synchronize them with real world clocks?
- How do we synchronize the clocks with each other?

The measure of real time is not simple. For a long time, it has been based on the rotation of the earth along its axis and revolution around the sun. Several quantities like **solar day**, **solar second** and **mean solar second** that takes into account the friction as a result of the rotation and turbulence deep inside the earth's crust, have been used to measure time. With the invention of the **atomic clock** in 1948, it became possible to measure time more accurately by counting transitions of cesium 133 atom. Physicists took over the job of time keeping from the astronomers and defined the second to be the time it takes the cesium 133 atom to make exactly 9, 192, 631, 770 transitions Currently, about 50 laboratories around the world have cesium 133 clocks. Periodically, each laboratory tells the Bureau International de l'Heure (BIH) in Paris how many times its clock has ticked. The BIH averages this to produce **International Atomic Time (TAI).** Although this time is fairly stable, it tends to drift from the mean solar day because the solar day is getting longer all the time. BIH solves the problem by introducing leap seconds whenever discrepancy between TAI and solar time grows to 800ms. This correction gives rise to a time system based on constant TAI seconds, but which also stays in phase with the apparent motion of the sun. It is called **Universal Coordinated Time (UTC**). It has essentially replaced the old standard, Greenwich Mean Time, which is astronomical time. To provide UTC to people who need precise time, the National Institute of Standard Time(NIST) operates a short wave radio station with call letters WWV, from Fort Collins, Colorado.

6.4 Clock Synchronization Algorithms

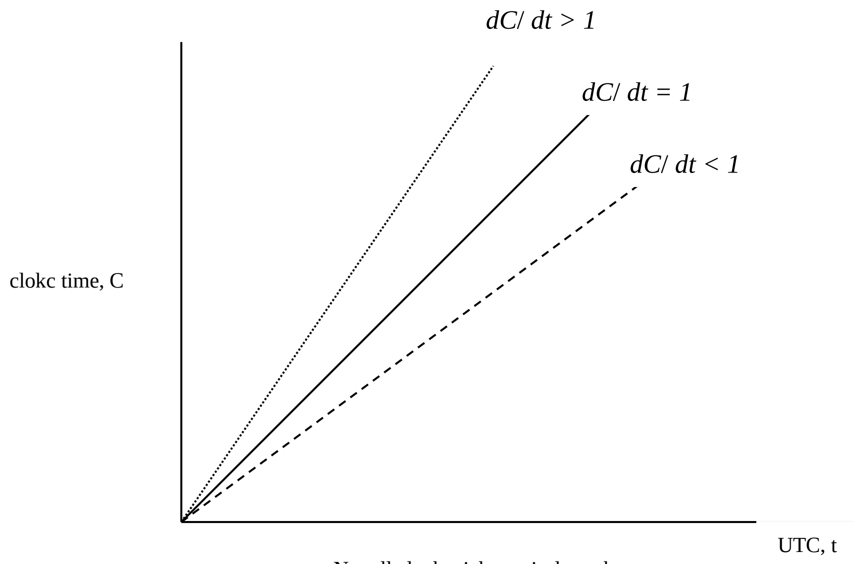
If one machine has a WWV receiver, the goal becomes keeping all the other machines synchronized to it. If no machines have WWV receivers, each machine keeps track of it own time and the goal is to keep all the machines together as well as possible. Many algorithms have been designed for doing this.

All the algorithms have the same underlying system model. Each machine is assumed to have a timer that causes an interrupt H times a second. When the timer goes off, the interrupt handler adds one to a software clock that keeps track of the number of ticks (interrupts) since some agreed upon time in the past. Lets call the value of this clock *C*. More specifically, when the UTC time is *t*, the value of the on machine *p* is $C_p(t)$. In an ideal situation we would have $C_p(t)$ = t for all *p* and all *t*.

In real life, timers do not interrupt exactly H times a second. The relative error obtainable with modern time chips is about $10^{-5.}$ If there exists some constant some constant ρ such that

$$1 - \rho <= dC/\, dt <= 1 + \rho$$

the timer can be said to be working properly within its specification. The constant ρ is specified by the manufacturer and is known as the **maximum drift rate.**



Not all clocks tick precisely at the correct rate

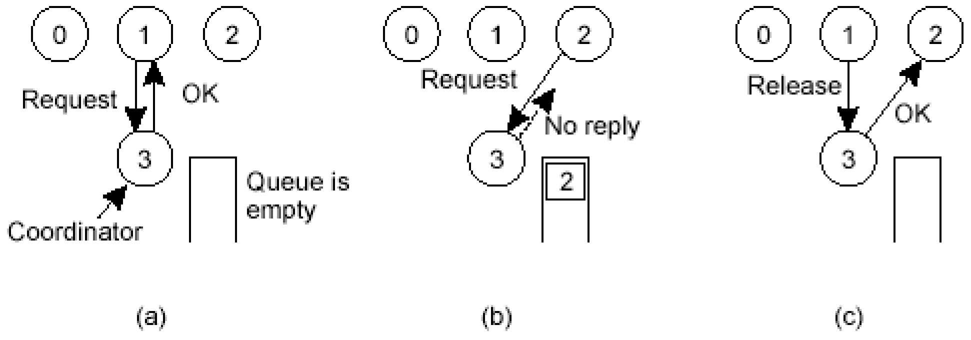
If two clocks are drifting from UTC in the opposite direction, at a time Δt after they were synchronized, they may be as much as $2\rho\Delta t$ apart. If the operating system designers want to guarantee that no two clocks ever differ by more than δ, clocks must be resynchronized in software at least every δ/2ρ seconds. There are various algorithms that differ precisely on how this resynchronization is done. Examples include Cristian’s, Berkeley and the averaging algorithms.

7. Mutual Exclusion

Systems involving multiple processes are often most easily programmed using critical regions. When a process has to read or update certain shared data structures, it first enters a critical region to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time. In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems.

7.1 A Centralized Algorithm

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g., the one running on the machine with the highest network address). Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission, as shown in Fig below. When the reply arrives, the requesting process enters the critical region.



(a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

Now suppose that another process, 2 in Fig (b), asks for permission to enter the same critical region. The coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. In Fig (b), the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying ‘‘permission denied.’’ Either way, it queues the request from 2 for the time being and waits for more messages.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Fig (c). The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. Either way, when it sees the grant, it can enter the critical region.

It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time into each critical region. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release). It can also be used for more general resource allocation rather than just managing critical regions.

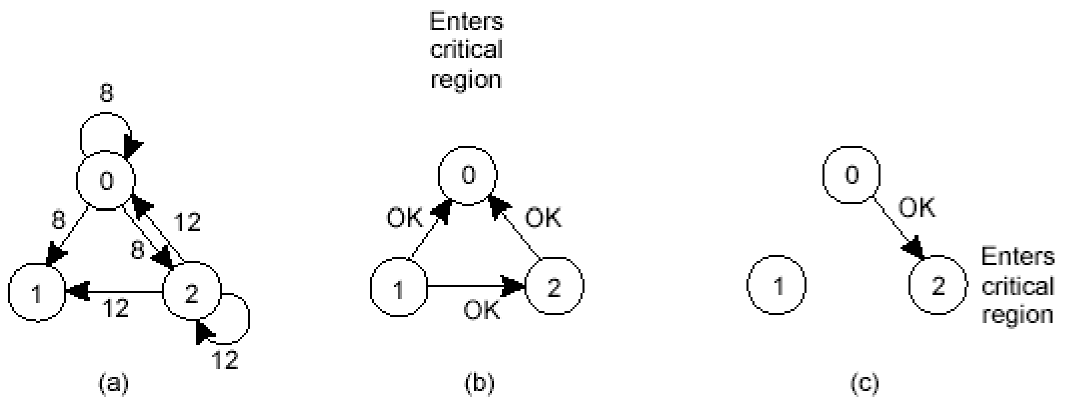
The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from ‘‘permission denied’’ since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

7.2 A Distributed Algorithm

Having a single point of failure is frequently unacceptable, so researchers have looked for distributed mutual exclusion algorithms. Lamport’s algorithm presented previously is one way to achieve this ordering and can be used to provide timestamps for distributed mutual exclusion. The algorithm works as follows. When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages. When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

1. If the receiver is not in the critical region and does not want to enter it, it sends back an *OK* message to the sender.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an *OK* message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends *OK* messages to all processes on its queue and deletes them all from the queue. Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig below.



(a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now enter the critical region.

Process 0 sends everyone a request with timestamp 8, while at the same time; process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends *OK* to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to 0 by sending *OK*. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig (b). When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to enter its critical region, as shown in Fig (c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

Note that the situation in Fig above would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and granted permission before making its own request. In this case, 2 would have noticed that it itself was in a critical region at the time of the request, and queued it instead of sending a reply.

As with the centralized algorithm discussed above, mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now $2(n - 1)$, where the total number of processes in the system is n. Best of all, no single point of failure exists. Unfortunately, the single point of failure has been replaced by n points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the n processes failing is at least n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is more than n times worse and requires much more network traffic to boot.

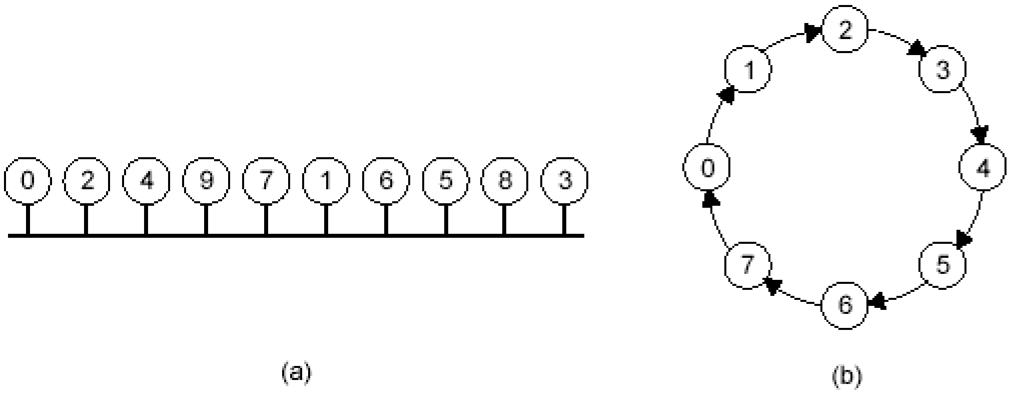
The algorithm can be patched up by the same trick that we proposed earlier. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent *OK* message. Another problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.

Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, *all* processes are involved in *all* decisions concerning entry into critical regions. If one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

Various minor improvements are possible to this algorithm. For example, getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission

7.3 A Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in Fig below Here we have a bus network, as shown in Fig. (a), (e.g., Ethernet), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig (b). The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.



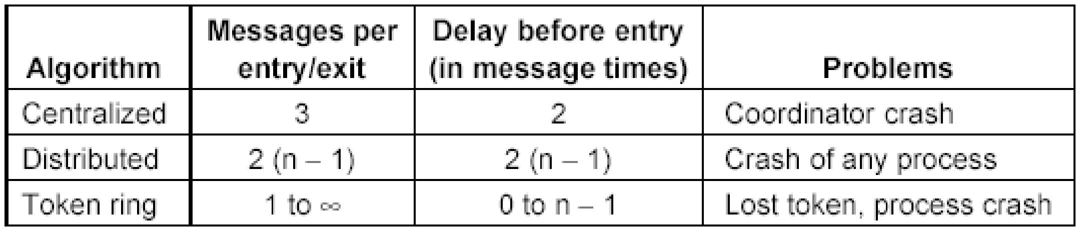
(a) An unordered group of processes on a network. (b) A logical ring constructed in software.

When the ring is initialized, process 0 is given a **token**. The token circulates around the ring. It is passed from process *k* to process *k* +1 (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token. If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring.

The correctness of this algorithm is easy to see. Only one process has the token at any instant, so only one process can actually be in a critical region. Since the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.
As usual, this algorithm has problems too. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it. The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintain the current ring configuration.

7.4 A Comparison of the Three Algorithms

A brief comparison of the three mutual exclusion algorithms we have looked at is instructive. In the figure below we have listed the algorithms and three key properties: the number of messages required for a process to enter and exit a critical region, the delay before entry can occur (assuming messages are passed sequentially over a network), and some problems associated with each algorithm.



Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

A comparison of three mutual exclusion algorithms.

The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request, a grant to enter, and a release to exit. The distributed algorithm requires $n - 1$ request messages, one to each of the other processes, and an additional $n - 1$ grant messages, for a total of $2(n - 1)$. (We assume that only point-to-point communication channels are used.) With the token ring algorithm, the number is variable. If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn. In Fig. 5-8 we show the former case. It takes only two message times to enter a critical region in the centralized case, but $2(n - 1)$ message times in the distributed case, assuming that messages are sent one after the other. For the token ring, the time varies from 0 (token just arrived) to $n - 1$ (token just departed).

Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they might do.

8. Atomic Transactions

Synchronization techniques studied so far are essentially low-level e.g. semaphores. They require the programmer to be intimately involved with all the details of mutual exclusion, critical region management, deadlock prevention and crash recovery. What we would really like is a much higher higher-level of abstraction, one that hides these technical issues and allow the programmer to concentrate on the algorithms and how the processes work together in parallel. This kind of an abstraction is widely used in distributed systems **atomic transactions**.

An atomic transaction is a computation consisting of a collection of operations that take place indivisibly in the presence of failures and concurrent computations. That is, either all of the operations are performed successfully or none of their effects prevail, and other processes executing concurrently cannot modify or observe intermediate states of the computation. Transactions help to preserve the consistency of a set of shared data objects in the face of failures or concurrent access.

8.1 Introduction to Atomic Transactions

Supposing two companies A and B have entered into a contract in which A is to supply equipment worth 20,000 shillings to B, but they have not signed the deal. Up until this point, both parties are free to terminate the negotiations, in which case the world returns to the state it was before they started negotiations. However, when both companies have signed the contract they are legally bound to finish the sale whatever the circumstances.

The computer model is similar. One transaction announces that it wants to begin a transaction with one or more other processes. They can negotiate various options, create and delete objects and perform operations for a while. Then the initiator announces that it wants all the others to commit themselves to the work done so far. If all of them agree, the results are made permanent. If one or more processes refuse (or crash before agreement), the situation is reverted to the way it was before the start. This all-or-nothing property reduces the work of the programmer.

Lets explore another example. Look at a modern banking application that updates an on-line database in place. The customer calls up the bank using a PC with a modem with the intention of withdrawing money from one account and depositing it in another. If the connection is broken after the withdrawal, but before the deposit, then the money vanishes into thin air. Being able to group these operations in an atomic transaction would solve the problem.

8.2 Transaction Primitives

Programming using transactions requires special primitives that must be supplied. Examples are:

1. BEGIN_TRANSACTION: Mark the start of a transaction
2. END_TRANSACTION: Terminate the transaction and try to commit.
3. ABORT_TRANSACTION: Kill the transaction and restore the old values
4. READ: Read data from a file
5. WRITE: Write data to a file

The exact list of primitives depends on which objects are being used in the transaction. In a mail system, there might be primitives to send, receive and forward mail. BEGIN_TRANSACTION and END_TRANSACTION are used to delimit the scope of a transaction. The operations between them form the body of the transactions. Either all of them are executed or none of them.

8.3 Properties of Transactions

Transactions have four essential properties. Transactions are:

1. Atomic: To the outside world, the transactions happen indivisibly
2. Consistent: The transaction does not violate system invariants
3. Isolated: Concurrent transactions do not interfere with each other
4. Durable: Once the transaction commits, the changes are permanent

These properties are often referred to by their initial letters, **ACID.**

Atomicity – This property ensures that each transaction either happens completely, or not at all, and if it happens, it happens in a single indivisible, instantaneous action. While a transaction is in progress, other processes cannot see any of the intermediate states. Two essential requirements for atomicity are atomicity with respect to *failures* and *concurrent* access. Failure atomicity ensures that if a transaction's work is interrupted by a failure, any partially computed results would be undone. Failure atomicity is also known as *all-or-nothing* property because a transaction is performed either completely or not at all. Concurrency atomicity is sometimes known as *consistency* property.

Consistency – Consistency ensures that while a transaction is in progress, other processes that are executing concurrently with the transaction cannot modify or observe intermediate results. Only the final state becomes visible after the transaction completes. This means that if the system has certain invariants that must always hold, if they held before the transaction, they will hold afterward too. For example, in a banking system, a key invariant is the law of conservation of money. After any internal transfer, the amount in the bank must be the same as it was before the transfer, but for a brief moment during the transaction, this invariant may be violated.

Isolation – This property also known as *serializability* property ensures that concurrently executing transactions do not interfere with each other. When transactions are isolated or serializable, if two or more transactions are running at the same time, to each of them and to other processes, the final result looks like all transactions ran sequentially in some order. That is, the concurrent execution of a set of two or more transactions is serially equivalent in the sense that the ultimate result of performing them concurrently is always the same as if they had been executed one at a time in some order.

Durability – Also known as the *permanence* property. This refers to the fact that once a transaction commits, no matter what happens, the transaction goes forward and the results become permanent. No failure after the commit can undo the results or cause them to be lost. This means that once a transaction completes successfully, the results of its operation becomes permanent and cannot be lost even if the corresponding process or the processor on which it was running crashes.

8.4 Implementation of Transactions

It should be clear that if each process executing a transaction just updates the object it uses (files, records etc) in place, transactions will not be atomic and changes will not vanish automatically if the transaction aborts. Furthermore, the results of the running multiple transactions will not be serializable either. Clearly some implementation method is required. Two methods are commonly used:

8.4.1 Private Workspace

Conceptually when a process starts a transaction it is given a private workspace containing all objects to which it has access. Until the transaction either commits or aborts, all its read and writes go to the private workspace, rather than the real object (file system).

The problem with this technique is that the cost of copying everything to a private workspace is prohibitive, but various optimizations make it feasible. The first optimization is based on the fact that when a process reads a file but does not modify it, there is no need for a private copy. Consequently, when a process starts a transaction, its sufficient to create a private workspace for it that is empty except that for a pointer back to its parent workspace. Secondly, when a file is opened for writing, instead of copying the entire file, only the file's index is copied into the private workspace. The index is the block of data associated with each file, telling where its disk blocks are. Using the private index the file can be read in the usual way. However, when a file block is first modified, a copy of the block is made and the copy of the copy inserted into the index. The block can then be updated without affecting the original. Appended block are also handled the same way. The new blocks are sometimes called **shadow blocks**.

If the transaction aborts, the private workspace is simply deleted and all the private blocks that it points to are put back on the free list. If the transaction commits, the private indices are moved into the parent's workspace atomically.

8.4.2 Two-phase Commit Protocol

The action of committing a transaction must be done atomically, that is, instantaneously and indivisibly. In a distributed system, the commit may require the cooperation of multiple processes on different machines, each of which holds some of the variables, files, and databases and other objects changed by the transaction.

The two-phase commit protocol is the most commonly used protocol to achieve atomic commit in a distributed system. One of the processes involved functions as the coordinator. Usually, this is the one executing the transaction. The commit protocol begins when the coordinator writes a log entry saying that it is starting the commit protocol, followed by sending each of other processes involved (the subordinates) a message telling them to prepare to commit. When a subordinate gets the message it checks to see if it is ready to commit, makes a log entry and sends back its decision. When the coordinator ha received all the responses it knows whether to commit or abort. If all the processes are prepared to commit, the transaction is committed. If one or more are unable to commit (or do not respond), the transaction is aborted. Either way, the coordinator writes a log entry and sends a message to each subordinate informing it of its decision. It is this write to the log that actually commits the transaction and makes it go forward no matter what happens afterward.

8.5 Concurrency Control

When multiple transactions are executing simultaneously in different processes, some mechanism is needed to keep them out of each other's way. This mechanism is called concurrency control algorithm. There are several of these algorithms

8.5.1 Locking

Locking is the most widely used concurrency control algorithm. In the simplest form, when a process needs to read or write a file as a part of a transaction, it first locks the file. Locking can be done using a single centralized manager, or with a lock manager on each machine for managing local files. In both cases the lock manager maintains a list of locked files and rejects all attempts to lock files that are already locked by another process. Since well-behaved processes do not attempt to access a file before it has been locked, keeping a lock keeps everyone else away thus ensures that it will not change during the lifetime of the transaction. Locks are normally acquired and released by the transaction system and do not require action by the programmer.

This basic scheme is overly restrictive and can be improved by distinguishing read from write locks. If a read lock is set on a file, other read locks are permitted. Read locks are used to exclude all writers. In contrast write locks are exclusive and when a file is locked for writing, no other locks of any kind are permitted.

In practice, the unit of locking need not be a file; it can be a record, page or a larger item such as a database. The issue of how large an item to lock is called the **granularity of locking**. The finer the granularity, the more precise the lock can be, and the more parallelism can be achieved (by not blocking a process that wants to use the end of a file just because another process is accessing the beginning). However, fine-grained locking requires more locks, is expensive and is more likely to lead to deadlocks.

Acquiring and releasing locks precisely at the moment they are needed or are no longer needed could lead to inconsistency or deadlocks. Consequently, most transactions that are implemented by locking use **two-phase locking**. The process first acquires all the locks it needs during the growing phase, the releases them during the shrinking phase. If a process fails to acquire all locks and refrains from doing updates until it reaches the shrinking phase, it’s easy to release all the locks and start over again.

In many systems, the shrinking phase does not take place until the transaction has finished running and has either committed or aborted. This policy, called strict two-phase locking, has two main advantages. First, a transaction always reads a value written by a committed transaction. Second, all lock acquisitions and releases can be handled by the system without the transaction being aware of them: locks are acquired whenever a file is to be accessed and released when the transaction has finished. This policy eliminates **cascaded aborts**: having to undo a committed transaction because it saw a file that it should not have seen.

Locking, even in two-phase locking, can lead to deadlocks. If two processes each try to acquire the same pair of locks but in opposite order, a deadlock may result. The usual techniques apply here, such as acquiring some locks in some canonical order to prevent hold-and-wait cycles.

8.5.2 Optimistic Concurrency control

A second approach to handling multiple transactions at the same time is optimistic concurrency control. The idea I this approach is to let the processes go ahead and do whatever they want to do without paying attention to what everybody else is doing and if there is a problem worry about it later. In practice, conflicts are relatively rare, so most of the time it works all right.

Although conflicts are rare, they do occur. What this method does is keep track of which files have been read and written. At the point of committing, it checks all other transactions to see if any of its files have been changed since the transaction started. If so, the transaction is aborted. If not, it is committed. It fits well with the implementation of private workspaces.

The big advantages of optimistic concurrency control are that it is deadlock free and allows maximum parallelism because no process ever has to wait for the lock.

The disadvantage is that it sometimes may fail, in which case the transaction has to be run all over gain.

8.5.3 Timestamps

A completely different approach to concurrency control is to assign each transaction a timestamp at the moment it does BEGIN_TRANSACTION. Using Lamport's algorithm, we can ensure that the timestamps are unique, which is important here. Every file in the system has a read timestamp and a write timestamp associated with it, telling which committed transaction last read and wrote it, respectively. If transactions are short and widely spaced in time, it will normally occur that when a process tries to access a file, the file's read and write timestamps will be lower than the current transaction's timestamp. This ordering means that the transactions are being processed in the right order, so everything is all right.

When the ordering is incorrect, it means that the transaction that started later than the current one has managed to get in there, access the file and commit. This means that the current transaction is too late, so it is aborted. In a way this mechanism is also optimistic.

8.6 Deadlocks in Distributed Systems

Deadlocks in distributed systems are similar to deadlocks in single processor systems, only worse. They are harder to avoid, prevent or even detect and harder to cure when tracked down because all the relevant information is scattered over many machines. In some systems such as distributed databases, they can be extremely serious, so it is important to know how they differ from ordinary deadlocks and what can be done about them.

Distinction can be made between two kinds of distributed deadlocks: communication and resource deadlocks. A communication deadlock occurs, for example, when process A is trying to send a message to process B, which in turn is trying to send a message to process C, which is trying to send one to A. There are various scenarios in which this situation leads to deadlock, such as no buffers being available. A resource deadlock occurs when processes are fighting over exclusive access to I/O devices, files, locks or other resources.

Communication deadlocks are very rare, so we will not make the above-mentioned distinction further. There are four main strategies for handling deadlocks:

1. The ostrich algorithm (ignore the problem)
2. Detection (let deadlocks occur, detect them, and try to recover)
3. Prevention (statically make deadlocks structurally impossible)
4. Avoidance (avoid deadlocks by allocating resources carefully)

All four are potentially applicable to distributed systems. The ostrich algorithm is as good and as popular in distributed systems as it is in single-processor systems. In distributed systems used for programming, office automation, process control and many other applications, no system-wide mechanism is present though individual applications, such as distributed databases can implement their own if they need one.

Deadlock detection and recovery is also popular, primarily because prevention is so difficult. Deadlock avoidance is never used in distributed systems.

9. Distributed File Systems

9.1 Introduction

In a computer system, a file is a named object that comes into existence by explicit creation, is immune to temporary failures in the system and persists until explicitly destroyed. Files are for two main purposes:

- Permanent storage of information
- Sharing of information

A *file system* is a subsystem of an operating system that performs file management activities such as organization, storing and retrieval, naming, sharing and protection of files. It is designed to allow programs to use a set of operations that characterize the file abstraction and free the programmers from concerns about the details of space allocation and layout of the secondary storage device. Therefore, a file system provides an abstraction of a storage device; that is convenient mechanism for storing and retrieving information from the storage device.

A distributed file system provides similar abstraction to the user of a distributed system and makes it convenient for them to use files in a distributed environment. The design and implementation of a distributed file system, however, is more complex than a conventional file system due to the fact that the users and storage devices are physically dispersed.

In addition, a distributed file system normally supports the following:

1. **Remote information sharing** – Allows a file to be transparently accessed by processes of any node of the system irrespective of the file's location. Therefore, a process on one node can create a file that can be accessed at a later time by some other process running on another node.
2. **User mobility** – In a distributed system, user mobility implies that a user should not be forced to work on a specific node but should have the flexibility to work on different nodes at different times. This property is desirable due to reasons such as coping with node failures and suiting the need of users who may need to work at different places at different times.
3. **Availability** – For better fault tolerance, files should be available for use even in the event of temporary failure of one or more nodes of the system. To achieve this, a distributed file system normally keeps multiple copies of a file on different nodes of the system (replicas of the file). In ideal design, both the existence of multiple copies and their locations are hidden from the clients.
4. **Diskless Workstations –** Disk drives are relatively expensive compared to the cost of most other parts in a workstation. Furthermore, the heat and noise produced by them when in the vicinity of the user are annoying factors. A distributed file system with its transparent remote file-accessing capability, allows the use of diskless workstations in a system.

A distributed file system typically provides three types of services. Each can be thought of as a component of a distributed file system.

1. **Storage Service** – It deals with the allocation and management of space on a secondary storage device that is used for storage of files in the file system. It provides a logical view of the storage system by providing operations for storing and retrieving data in them. Since most systems use magnetic disks as their secondary storage device, the storage service is also known as *disk service.* Furthermore, several systems allocate disk space in units of fixed-size blocks, and hence, the storage service is also known as *block service* in these systems.

2. **True File Service** – It is concerned with the operations on individual files, such as operations for accessing and modifying data in files and creating and deleting files. To perform these primitive file operations correctly and efficiently, typical design issues of a true file service component include file-accessing mechanism, file sharing semantics, file-caching mechanism, file replication mechanism, concurrency control mechanism, data consistency and multiple copy update protocol.

3. **Name Service –** It provides a mapping between text names for file and references to files, that is, file Ids. Most file systems use directories to perform this mapping. Therefore, the name service is also known as *directory service*. The directory service is responsible for performing directory-related activities such as creation and deletion of directories, adding a new file to a directory, deleting a file from a directory, changing the name of a file, moving a file from one directory to another etc.

9.2 Features of a good Distributed File System

9.2.1 Transparency – There are four types of transparencies desirable:

- *Structure Transparency* – Although not necessary for performance, scalability and reliability reasons, a distributed file system normally uses multiple file servers. Each file server is normally a user process or sometimes a kernel process that is responsible for controlling a set of secondary storage devices on the node on which it runs. In multiple file servers, the multiplicity of the file servers should be transparent to the clients of a distributed file system. In particular, clients should not know the number or locations of file servers and the storage devices. Ideally, a distributed file system should look to its clients like a conventional file system offered by a centralized, time-sharing operating system.
- *Access Transparency* – Both local and remote files should be accessible in the same way. That is, the file system interface should not distinguish between local and remote files and the file system should automatically locate an accessed file and arrange for the transport of the data to the client's site.
- *Naming Transparency* – The name of a file should give no hint as to where the file is located. Furthermore, a file should be allowed to move from one node to another in a distributed system without having to change the name of the file.
- *Replication Transparency* – If a file is replicated on multiple nodes, both the existence of multiple copies and their locations should be hidden from the clients.

9.2.2 User Mobility – In a distributed system, a user should not be forced to work on a specific node, but should have the flexibility to work on different nodes at different times. Furthermore, the performance characteristics of the file system should not discourage users from accessing their files from workstations other than the one at which they usually work.

9.2.3 Performance – The performance of a file system is usually measured as the average amount of time needed to satisfy client requests. In centralized file systems, this time includes the time of accessing the secondary storage device on which the file is stored and the CPU processing time. In a distributed file system, however, this also includes network communication overhead when the accessed file is remote. It is desirable that the performance of a distributed file system be comparable to that of a centralized file system.

9.2.4 Simplicity and ease of use – The semantics of a distributed file system should be easy to understand. This implies that the user interface to the file system should be simple and the number of commands as small as possible. In an ideal design, the semantics of a distributed file system should be the same as that of a file system for a centralized system.

9.2.5 Scalability – It is inevitable that a distributed system will grow with time since expanding the network by adding new machines or interconnecting two networks together is commonplace. Therefore, a good distributed file system should be designed to easily cope with the growth of nodes and users in the system. Consequently, such growth should not cause serious disruption of service or significant loss of performance to users. In short, a scalable design should withstand high service load, accommodate growth of user community and enable simple integration of added resources.

9.2.6 High availability – A distributed file system should continue to function even when partial failures occur due to the failure of one or more components, such as communication link failure, a machine failure or a storage device crash. When a partial failure occurs, the file system may show degradation in performance, functionality, or both. High availability and scalability are mutually related properties. Both properties call for a design in which both control and data are distributed. This is because centralized entities introduce a single point of failure and a performance bottleneck. Therefore, a highly available and scalable distributed system should have multiple and independent file servers controlling multiple independent storage devices.

9.2.7 High reliability – In a good distributed file system, the probability of loss of stored data should be minimized as far as practicable. The users should not feel compelled to make backup copies of their files because of the unreliability of the system. Rather, the file system should automatically generate backup copies of critical files that can be used in the event of loss of the original ones.

9.2.8 Data integrity – Multiple users often share a file. For a shared file, the file system must guarantee the integrity of the data stored in it. That is, concurrent access requests from multiple users who are competing to access the file must be properly synchronized by use of some concurrency control mechanism. Atomic transactions are a high-level concurrency control mechanism often provided to the users by a file system for data integrity.

9.2.9 Security – A distributed file system should be secure so that its users can be confident of the privacy of their data. Necessary security mechanisms must be implemented to protect information stored in a file system against unauthorized access.

9.2.10 Heterogeneity – As a consequence of large scale, heterogeneity becomes inevitable in distributed systems. Heterogeneous distributed systems provide the flexibility to their users to use different computer platforms for different applications. Easy access to shared data across these diverse platforms would substantially improve usability. Therefore, a distributed file system should be designed to allow a variety of workstations to participate in the sharing of files via the distributed file system. Another heterogeneity issue is the ability to accommodate several different storage media.

9.3 File Models

Different file systems use different conceptual models of a file. The two most commonly used criteria for files modeling are structure and modifiability.

9.3.1 Unstructured and Structured Files

In the simplest model, a file is an unstructured sequence of data. In this model, the content of each file of the file system appears to the file server as an uninterrupted sequence of bytes. The operating system is not interested in the information stored in the files. It is only the application programs that interpret the data stored in the files.

Another file model that is rarely used is the structured model. In this model, a file appears to the file server as an ordered sequence of records. Records of different files of the same file system can be of different size. In this model, a record is the smallest unit of file data that can be accessed.

Most operating systems use the unstructured model because sharing of a file by different applications is easier in the structured file model. Different applications can interpret the contents of the file in different ways.

9.3.2 Mutable and Immutable Files

According to modifiability criteria, files are of two types – mutable and immutable. Most operating systems use the mutable file model. In this model, an updates performed on a file overwrites its old content to produce the new contents. In the immutable model, a file cannot be modified once it has been created except to be deleted. File versioning approach is normally used to implement updates.

Sharing only immutable files makes it easy to support consistent sharing. This feature makes it easy to support file caching and replication in distributed systems because it eliminates all the problems associated with keeping multiple copies of a file consistent.

9.4 File Replication

High availability is a desirable feature of a good distributed file system and file replication is the primary mechanism for improving file availability. A replicated file is a file that has multiple copies, with each copy located on a separate file server.

9.4.1 Advantages of File Replication

1. Increased availability – One of the most important advantages of replication is that it masks and tolerates failures in the network gracefully. By replicating critical data on servers with independent failure modes, the probability that one copy of the data will be accessible increases. Therefore, alternate copies of a replicated data can be used when the primary copy is unavailable.
2. Increased reliability – Many applications require extremely high reliability of their data stored in files. Replication is very advantageous for such applications because it allows the existence of multiple copies of their files. Due to the presence of redundancy in the system, recovery from catastrophic failures becomes possible.
3. Improved response time – Replication also helps in improving response time because it enables data to be accessed either locally or from a node to which the access time is lower than the primary access time.
4. Reduced network traffic – If a file's replica is available with a file server that resides on a client's node, the client's access request can be serviced locally, resulting in reduced network traffic.
5. Improved system throughput – Replication also enables several clients' requests for access to the same file to be serviced in parallel by different servers, resulting in improved system throughput.
6. Better scalability – As the number of users of a shared file grows, having all access requests serviced by a single file server can result in poor performance. By replicating the file over several servers, the same requests can now be serviced more efficiently by multiple servers due to workload distribution.

9.4.2 Replication Transparency

Transparency is an important issue in file replication. A replicated file service must not only function exactly like a non-replicated file service but exhibit improved performance and reliability. That is, replication of files should be designed to be transparent to the users so that multiple copies of a replicated file appear as a single logical file to its users. Two main issues to consider are naming of replicas and replication control.

- **Naming of replicas** – In systems that support object replication, a basic question is whether different replicas of an object should be assigned the same identifier or different identifiers. Replication transparency requirement calls for the assignment of a single identifier to all replicas. For immutable objects any copy of the object can be used since all copies are immutable and identical thus scaling with using a single identifier for all replicas. However, in mutable objects, different copies of the same replicated object may not be consistent at a particular instance of time. Its difficult to decide which replica is most up-to-date. This calls for consistency control mechanisms outside the kernel.
- **Replication Control** – Another transparency issue is providing replication control. This involves determining the number and location of replicas of a replicated file. In a replication transparent system, the replication control is handled entirely automatically, in a user-transparent manner. However, under certain circumstances, it is desirable to expose these details to the users to provide them with the flexibility to control replication process. Depending on whether replication control is user transparent or not, replication process is of two types:

  1. *Explicit replication* – In this type, the users are given the flexibility to control the entire replication process. That is, when a process creates a file, it specifies on which server the file should be placed. Then, if desired, the users can create additional copies of the file on other servers on explicit request.

  2. *Implicit Replication* – In this type, the entire replication process is automatically controlled by the system without the user's knowledge. The system automatically selects one server for placement of a file on creation. Later, the system automatically creates replicas of the file on other servers based on the replication policy used by the system.

10. Security

10.1 Introduction

Security requirements are different for different computer systems depending on the environment in which they are supposed to operate. For example, security requirements for systems meant to operate in a military environment are different from those systems that are meant to operate in an education environment. The security goals of a computer system are decided by its *security policies* and the methods used to achieve those goals are called *security mechanisms*. The goals of computer security are:

1. *Secrecy* – Information within a system must be accessible only to authorized users.
2. *Privacy* – Misuse of information must be prevented. That is, a piece of information given to the user should be used only for the purpose it was given.
3. *Authenticity* – When a user receives some data, the user must be able to verify its authenticity. That is, the data arrived indeed from it expected sender and not from any other source.
4. *Integrity* – Information within the system must be prevented from accidental destruction or intentional corruption by an unauthorized user.

A total approach to computer security involves both external and internal security. External security involves securing the computer system against external factors such as fire, flood, earthquakes, theft etc. For external security, the commonly used methods include maintaining adequate backup copies of stored information at places far away from the original information, using security guards, allow the access to sensitive information only to trusted users.

Internal security on the other hand deals with the following two aspects:

- *User Authentication* – Once a user is allowed physical access to the computer facility, the user's identification must be checked by the system before the user can actually use the facility.
- *Access Control* – A computer system contains many resources and several types of information. Not or resources are meant for users. Therefore once a user has passed the authentication phase, a way is needed to prohibit the user from accessing the information he/she is not authorized to access.
- *Communication Security* – In a distributed system, the communication channels that are used to connect the computers are normally exposed to attackers who may try to breach the security of the system by observing, modifying or disrupting the communications. Communication security guards against unauthorized tampering of information while it is being transmitted from one computer to another through the communication channels. Two aspects of communication security are *authenticity* of the communicating entities and the *integrity* of messages. That is, the sender of a message wants to know that the message was received by the intended receiver and the receiver wants to know that the message was sent by the genuine sender. The sender and the receiver will also want to be guaranteed that the message was not modified while in transit.

10.2 Security Threats, Policies, and Mechanisms

Security in computer systems is strongly related to the notion of dependability. Informally, a dependable computer system is one that we justifiably trust to deliver its services; dependability includes availability, reliability, safety, and maintainability. However, if we are to put our trust in a computer system, then confidentiality and integrity should also be taken into account.

Confidentiality refers to the property of a computer system whereby its information is disclosed only to authorized parties. Improper alterations in a secure computer system should be detectable and recoverable. Major assets of any computer system are its hardware, software, and data. Another way of looking at security in computer systems is that we attempt to protect the services and data it offers against **security threats**. There are four types of security threats to consider

1. Interception
2. Interruption
3. Modification
4. Fabrication

Interception refers to the situation that an unauthorized party has gained access to a service or data. A typical example of interception is where communication between two parties has been overheard by someone else. Interception also happens when data are illegally copied, for example, after breaking into a person's private directory in a file system.

An example of **interruption** is when a file is corrupted or lost. In general, interruption refers to the situation in which services or data become unavailable, unusable, destroyed, and so on. In this sense, denial of service attacks by which someone maliciously attempts to make a service inaccessible to other parties is a security threat that classifies as interruption.

Modifications involve unauthorized changing of data or tampering with a service so that it no longer adheres to its original specifications. Examples of modifications include intercepting and subsequently changing transmitted data, tampering with database entries, and changing a program so that it secretly logs the activities of its user.

Fabrication refers to the situation in which additional data or activity are generated that would normally not exist. For example, an intruder may attempt to add an entry into a password file or database. Likewise, it is sometimes possible to break into a system by replying previously sent messages. Note that interruption, modification, and fabrication can each be seen as a form of data falsification.

Simply stating that a system should be able to protect itself against all possible security threats is not the way to actually build a secure system. What is first needed is a description of security requirements, that is, a *security policy*. A **security policy** describes precisely which actions the entities in a system are allowed to take and which ones are prohibited. Entities include users, services, data, machines, and so on. Once a security policy has been laid down, it becomes possible to concentrate on the **security mechanisms** by which a policy can be enforced. Important security mechanisms are:

1. Encryption
2. Authentication
3. Authorization
4. Auditing

Encryption is fundamental to computer security. Encryption transforms data into something an attacker cannot understand. In other words, encryption provides a means to implement confidentiality. In addition, encryption allows us to check whether data have been modified. It thus also provides support for integrity checks.

Authentication is used to verify the claimed identity of a user, client, server, and so on. In the case of clients, the basic premise is that before a service will do work for a client, the service must learn the client's identity. Typically, users are authenticated by means of passwords, but there are many other ways to authenticate clients.

After a client has been authenticated, it is necessary to check whether that client is **authorized** to perform the action requested. Access to records in a medical database is a typical example. Depending on who accesses the database, permission may be granted to read records, to modify certain fields in a record, or to add or remove a record.

Auditing tools are used to trace which clients accessed what, and which way. Although auditing does not really provide any protection against security threats, audit logs can be extremely useful for the analysis of a security breach, and subsequently taking measures against intruders. For this reason, attackers are generally keen not to leave any traces that could eventually lead to exposing their identity. In this sense, logging accesses makes attacking sometimes a riskier business.