Distributed Systems

LECTURE NOTES

B.TECH IV YEAR – VIII SEM

DEPARTMENT OF INFORMATION TECHNOLOGY

DISTRIBUTED SYSTEMS

UNIT I

1. **Distributed Systems**
2. **Features of distributed systems**
3. **Model of distributed systems**
4. **nodes of a distributed system**
5. **Types of Operating systems**
   - **5.1 Distributed Operating Systems**
   - **5.2 Network Operating Systems**
   - **5.3 Centralized Operating System**
6. **Design issues in distributed operating systems**
7. **Distributed computation paradigms**
8. **Trends in distributed systems**
9. **Systems Concepts and Architectures**
   - **9.1 Goals, Transparency, Services, Architecture Models**
10. **Distributed Computing Environment**
11. **Theoretical issues in distributed systems**
    - **11.1 Notions of time states**
      - **1.11.1 States and events in a distributed system**
      - **1.11.1 The Berkeley algorithm**
      - **1.11.3 Network Time Protocol (NTP)**

DISTRIBUTED SYSTEMS

UNIT I

1.1 Distributed Systems

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

1.1.1 Distributed systems Principles

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

Centralised System Characteristics

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

Distributed System Characteristics

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

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

- telecommunication networks:
- telephone networks and cellular networks,
- computer networks such as the Internet,
- wireless sensor networks,
- routing algorithms;

- network applications:
- World wide web and peer-to-peer networks,
- massively multiplayer online games and virtual reality communities,
- distributed databases and distributed database management systems,

- network file systems,
- distributed information processing systems such as banking systems and airline reservation systems;
- real-time process control:
- aircraft control systems,
- industrial control systems;
- parallel computation:
- scientific computing, including cluster computing and grid computing and various volunteer computing projects (see the list of distributed computing projects),
- distributed rendering in computer graphics.

1.1.2 Features of distributed systems

Certain common characteristics can be used to assess distributed systems

- Resource Sharing
- Openness
- Concurrency
- Scalability
- Fault Tolerance
- Transparency

Resource Sharing

- Ability to use any hardware, software or data anywhere in the system.
- Resource manager controls access, provides naming scheme and controls concurrency.
- Resource sharing model (e.g. client/server or object-based) describing how

    - resources are provided,
    - they are used and
    - provider and user interact with each other.

Openness

- Openness is concerned with extensions and improvements of distributed systems.
- Detailed interfaces of components need to be published.
- New components have to be integrated with existing components.
- Differences in data representation of interface types on different processors (of different vendors) have to be resolved.

Concurrency

Components in distributed systems are executed in concurrent processes.

- Components access and update shared resources (e.g. variables, databases, device drivers).
- Integrity of the system may be violated if concurrent updates are not coordinated.
    - Lost updates
    - Inconsistent analysis

Scalability

- Adaption of distributed systems to
    - accomodate more users
    - respond faster (this is the hard one)
- Usually done by adding more and/or faster processors.
- Components should not need to be changed when scale of a system increases.
- Design components to be scalable

Fault Tolerance

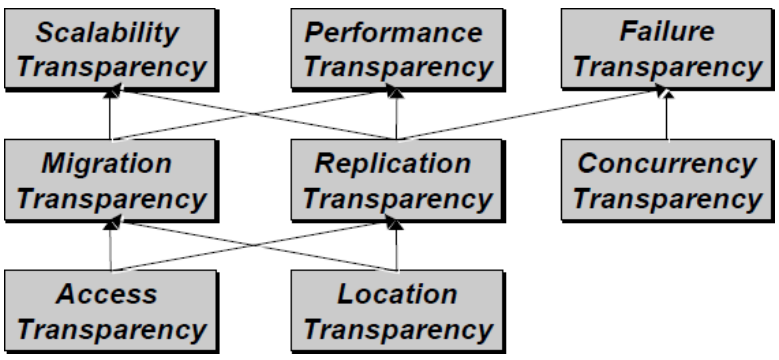
Hardware, software and networks fail!

- Distributed systems must maintain availability even at low levels of hardware/software/network reliability.
- Fault tolerance is achieved by
    - recovery
    - redundancy

Transparency

Distributed systems should be perceived by users and application programmers as a whole rather than as a collection of cooperating components.

- Transparency has different dimensions that were identified by ANSA.
- These represent various properties that distributed systems should have.



Access Transparency

Enables local and remote information objects to be accessed using identical operations.

- Example: File system operations in NFS.
- Example: Navigation in the Web.
- Example: SQL Queries

Location Transparency
Enables information objects to be accessed without knowledge of their location.

- Example: File system operations in NFS
- Example: Pages in the Web
- Example: Tables in distributed databases

Concurrency Transparency

Enables several processes to operate concurrently using shared information objects without interference between them.

- Example: NFS
- Example: Automatic teller machine network
- Example: Database management system

Replication Transparency

Enables multiple instances of information objects to be used to increase reliability and performance without knowledge of the replicas by users or application programs

- Example: Distributed DBMS
- Example: Mirroring Web Pages.

Failure Transparency

- Enables the concealment of faults
- Allows users and applications to complete their tasks despite the failure of other components.
- Example: Database Management System

Migration Transparency

Allows the movement of information objects within a system without affecting the operations of users or application programs

- Example: NFS
- Example: Web Pages

Performance Transparency
Allows the system to be reconfigured to improve performance as loads vary.

- Example: Distributed make.

Scaling Transparency

Allows the system and applications to expand in scale without change to the system structure or the application algortithms.

- Example: World-Wide-Web
- Example: Distributed Database
-

1.2 Model of distributed systems

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

Each type of model is intended to provide an abstract, simplified but consistent description of a relevant aspect of distributed system design:
Physical models are the most explicit way in which to describe a system; they capture the hardware composition of a system in terms of the computers (and other devices, such as mobile phones) and their interconnecting networks.

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

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

Architectural models

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

Software layers

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

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

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

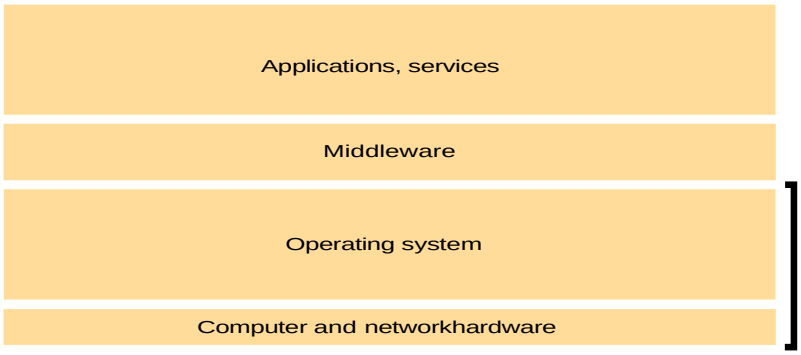
A platform for distributed systems and applications consists of the lowest-level hardware and software layers. These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates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

– Remote Procedure Calls – Client programs call procedures in server programs
– Remote Method Invocation – Objects invoke methods of objects on distributed hosts
– Event-based Programming Model – Objects receive notice of events in other objects in which they have interest

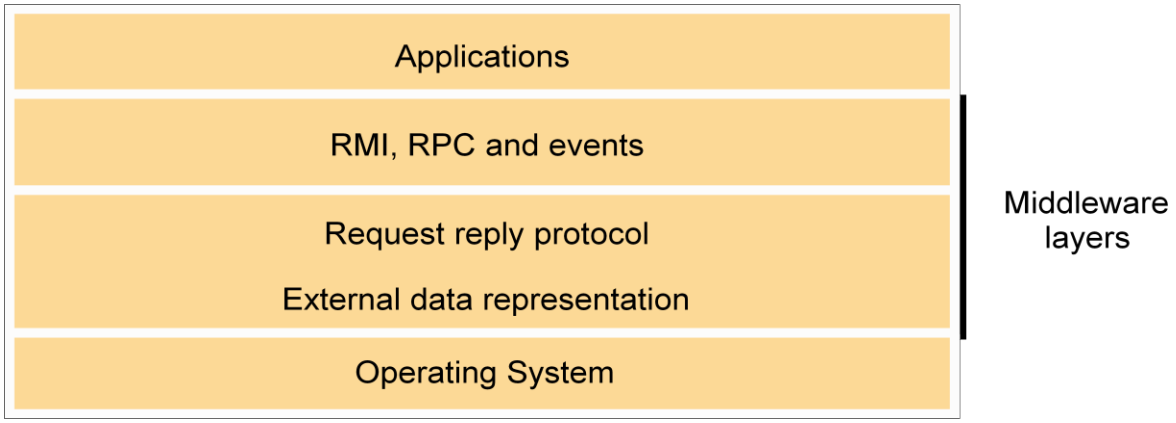
Middleware

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

passing
– Uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events
– Supports location transparency
– Usually uses an interface definition language (IDL) to define interfaces



-



Interfaces in Programming Languages

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

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

• Interfaces in Distributed Systems

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

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

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

• Object Interfaces

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

A class may implement an interface by specifying behavior for each method in the interface.

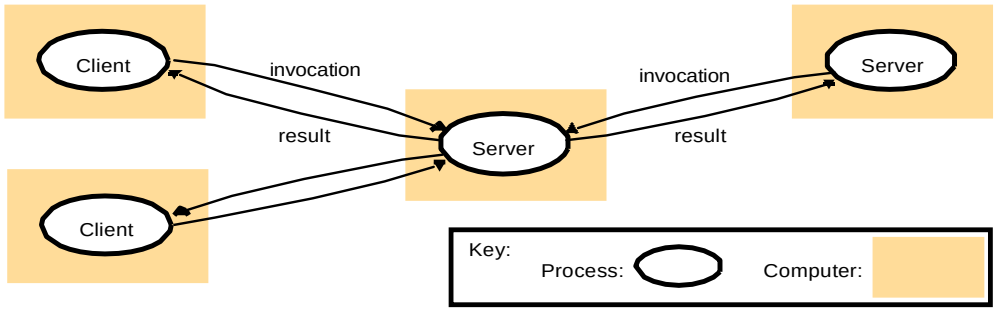
Interfaces do not have constructors.

System architectures

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

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

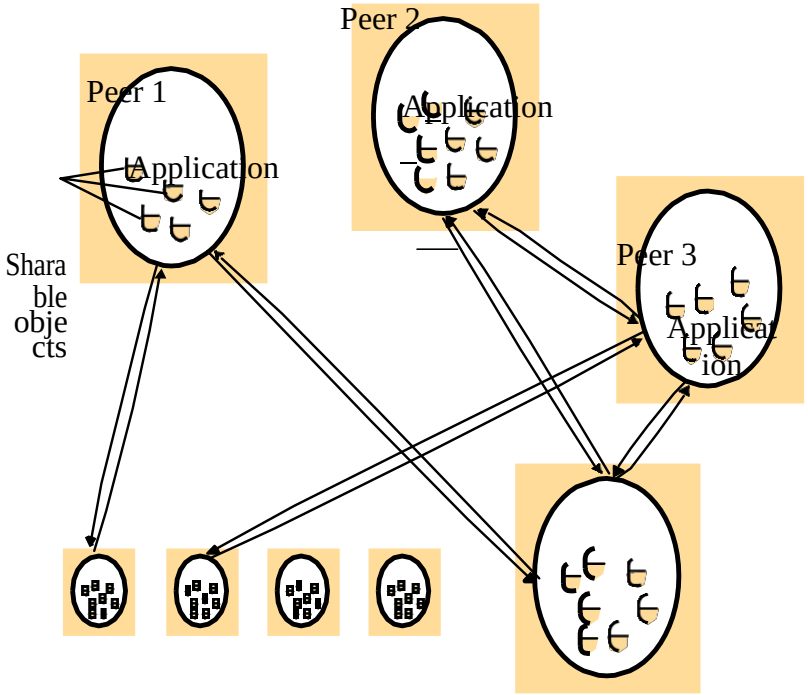
Clients invoke individual servers



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

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

A distributed application based on peer processes



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

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

About their characteristics and the failures and security risks they might exhibit. In general, such a fundamental model should contain only the essential ingredients that need to consider in order to understand and reason about some aspects of a system's behaviour. The purpose of such a model is:

• To make explicit all the relevant assumptions about the systems we are modelling.
• To make generalizations concerning what is possible or impossible, given those assumptions. The generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed. The guarantees are
dependent on logical analysis and, where appropriate, mathematical proof.

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

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

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

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

1.2.4 Interaction model

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

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

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

Two significant factors affecting interacting processes in a distributed system:
• Communication performance is often a limiting characteristic.
• It is impossible to maintain a single global notion of time.

1.2.5 Failure model

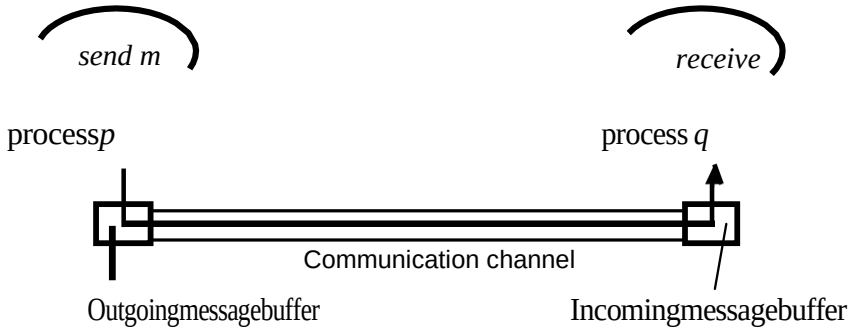
In a distributed system both processes and communication channels may fail – that is, they may depart from what is considered to be correct or desirable behaviour. The failure model defines the ways in which failure may occur in order to provide an understanding of the effects of failures. Hadzilacos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

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

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

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

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



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

takes unintended processing steps. Arbitrary failures in processes cannot be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply.

Communication channels can suffer from arbitrary failures; for example, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once. Arbitrary failures of communication channels are rare because the communication software is able to recognize them and reject the faulty
messages. For example, checksums are used to detect corrupted messages, and message sequence numbers can be used to detect nonexistent and duplicated messages.

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

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

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

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

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

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

Validity: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.
Integrity: The message received is identical to one sent, and no messages are delivered twice.

The threats to integrity come from two independent sources:
• Any protocol that retransmits messages but does not reject a message that arrives twice. Protocols can attach sequence numbers to messages so as to detect those that are delivered twice.

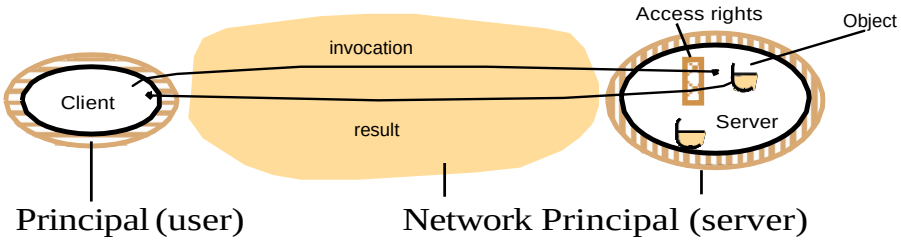
• Malicious users that may inject spurious messages, replay old messages or tamper with messages. Security measures can be taken to maintain the integrity property in the face of such attacks.

Security model

The sharing of resources as a motivating factor for distributed systems, and in Section 2.3 we described their architecture in terms of processes, potentially encapsulating higher-level abstractions such as objects, components or services, and providing access to them through interactions with other processes. That architectural model provides the basis for our security model:
the security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

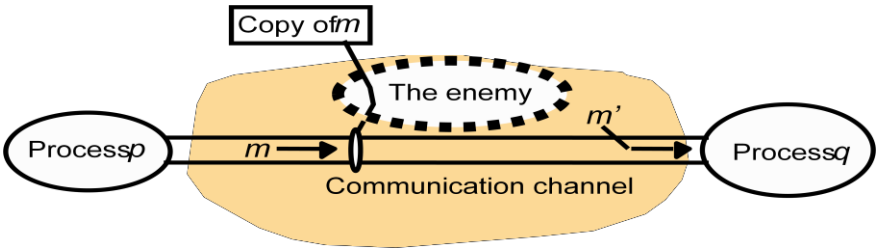
Protection is described in terms of objects, although the concepts apply equally well to resources of all types

Protecting objects :
Server that manages a collection of objects on behalf of some users. The users can run client programs that send invocations to the server to perform operations on the objects. The server carries out the operation specified in each invocation and sends the result to the client.
Objects are intended to be used in different ways by different users. For example, some objects may hold a user's private data, such as their mailbox, and other objects may hold shared data such as web pages. To support this, *access rights* specify who is allowed to perform the operations of an object – for example, who is allowed to read or to write its state.



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

The enemy • To model security threats, we postulate an enemy (sometimes also known as the adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes, as shown in the following figure. Such attacks can be made simply by using a computer connected to a network to run a program that reads network messages addressed to other computers on the network, or a program that generates messages that make false requests to services, purporting to come from authorized users. The attack may come from a computer that is legitimately connected to the network or from one that is connected in an unauthorized manner. The threats from a potential enemy include *threats to processes* and *threats to communication channels*.



Defeating security threats

Cryptography and shared secrets: Suppose that a pair of processes (for example, a particular client and a particular server) share a secret; that is, they both know the secret but no other process in the distributed system knows it. Then if a message exchanged by that pair of processes includes information that proves the sender's knowledge of the shared secret, the recipient knows for sure that the sender was the other process in the pair. Of course, care must be taken to ensure that the shared secret is not revealed to an enemy.

Cryptography is the science of keeping messages secure, and *encryption* is the process of scrambling a message in such a way as to hide its contents. Modern cryptography is based on encryption algorithms that use secret keys – large numbers that are difficult to guess – to transform data in a manner that can only be reversed with knowledge of the corresponding decryption key.

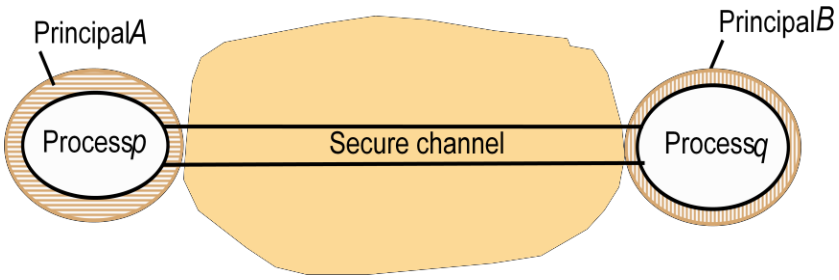
Authentication: The use of shared secrets and encryption provides the basis for the *authentication* of messages – proving the identities supplied by their senders. The basic authentication technique is to include in a message an encrypted portion that contains enough of the contents of the message to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

a secret key shared between the file server and the requesting process. The server would decrypt this and check that it corresponds to the unencrypted details specified in the request.

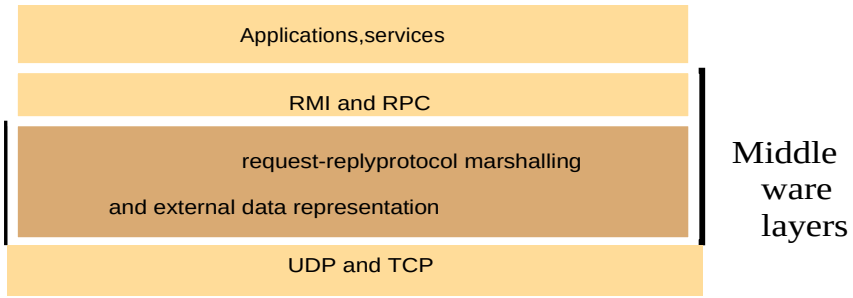
Secure channels: Encryption and authentication are used to build secure channels as a service layer on top of existing communication services. A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal, as shown in the following figure. A secure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

• Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing. Therefore if a client and server communicate via a secure channel, the server knows the identity of the principal behind the invocations and can check their access rights before performing an operation. This enables the server to protect its objects correctly and allows the client to be sure that it is receiving results from a *bona fide* server.

• A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it.
• Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered.



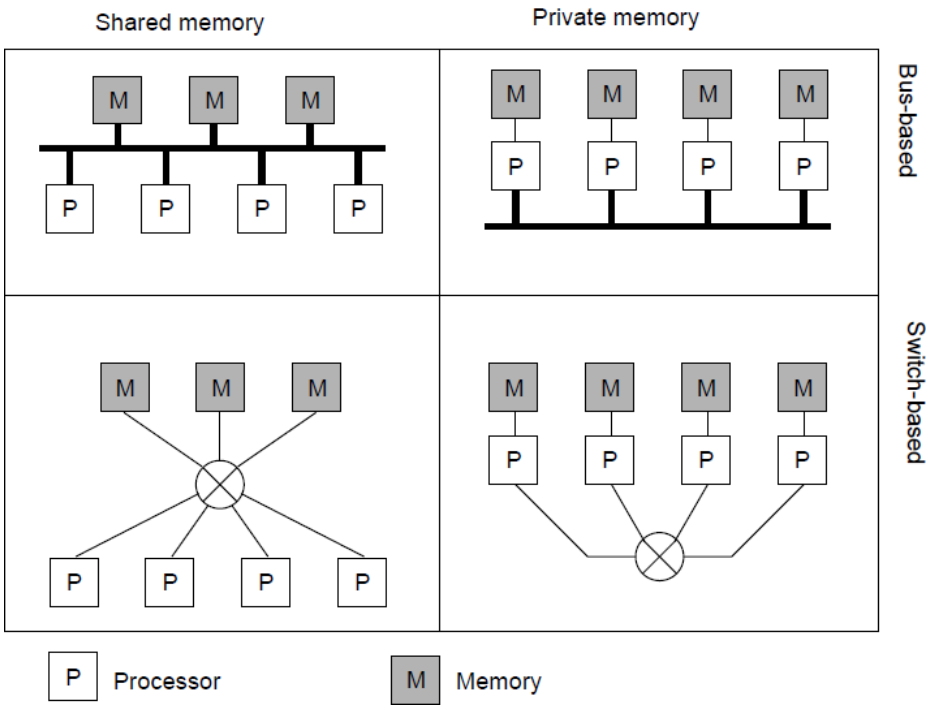
Communication aspects of middleware, although the principles discussed are more widely applicable. This one is concerned with the design of the components shown in the darker layer in the following figure.



1.3 Nodes of a distributed system

Multiprocessors and Multicomputers
Distinguishing features:

- Private versus shared memory
- Bus versus switched interconnection



Networks of Computers

High degree of node heterogeneity:

- High-performance parallel systems (multiprocessors as well as multicomputers)
- High-end PCs and workstations (servers)
- Simple network computers (offer users only network access)
- Mobile computers (palmtops, laptops)
- Multimedia workstations

High degree of network heterogeneity:

- Local-area gigabit networks
- Wireless connections
- Long-haul, high-latency connections
- Wide-area switched megabit connections

Distributed Systems: Software Concepts

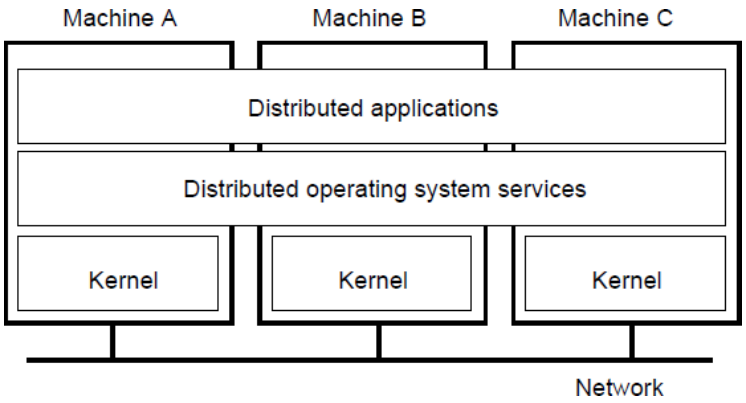
Distributed operating system
_ Network operating system
_ Middleware

System	Description	Main goal
DOS	Tightly-coupled OS for multiprocessors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled OS for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middle-ware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

1.4 Distributed Operating System

Some characteristics:

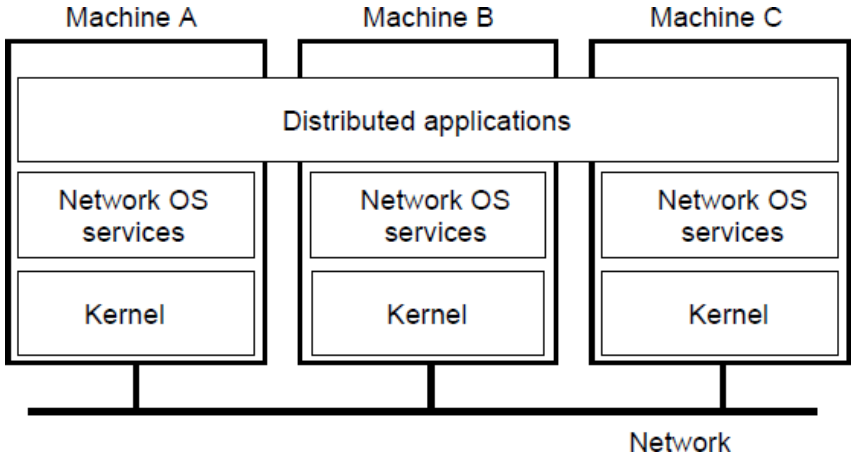
_ OS on each computer knows about the other computers
_ OS on different computers generally the same
_ Services are generally (transparently) distributed across computers



1.4.1. Network Operating System

Some characteristics:

_ Each computer has its own operating system with networking facilities
_ Computers work independently (i.e., they may even have different operating systems)
_ Services are tied to individual nodes (ftp, telnet, WWW)
_ Highly file oriented (basically, processors share *only* files)

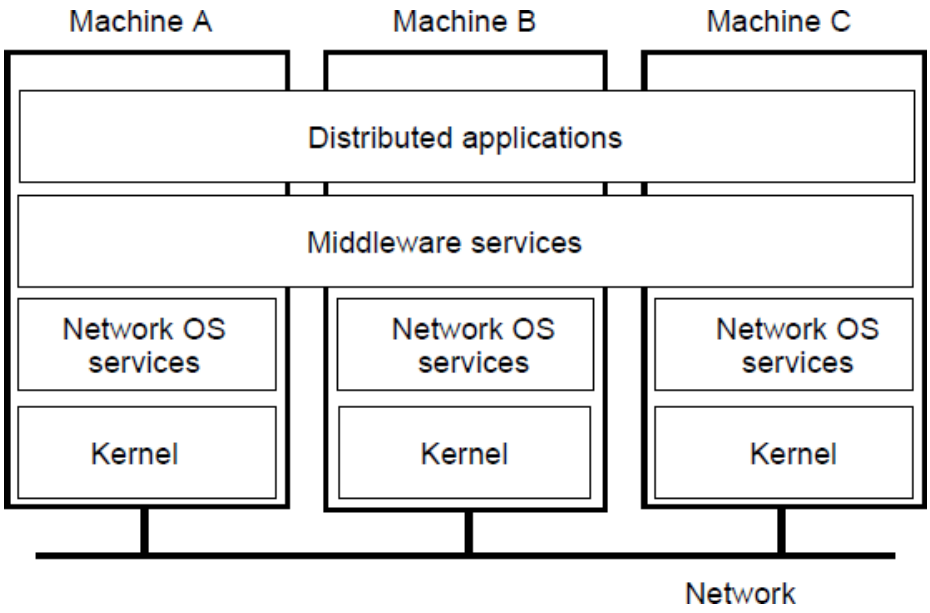


1.4.2 Distributed System (Middleware)

Some characteristics:

_ OS on each computer need not know about the other computers
_ OS on different computers need not generally be the same
_ Services are generally (transparently) distributed across computers

-



Need for Middleware
Motivation: Too many networked applications were hard or difficult to integrate:
_ Departments are running different NOSs
_ Integration and interoperability only at level of primitive NOS services
_ Need for federated information systems:
– Combining different databases, but providing a single view to applications
– Setting up enterprise-wide Internet services, making use of existing information systems
– Allow transactions across different databases
– Allow extensibility for future services (e.g., mobility, teleworking, collaborative applications)
_ Constraint: use the existing operating systems, and treat them as the underlying environment (they provided the basic functionality anyway)

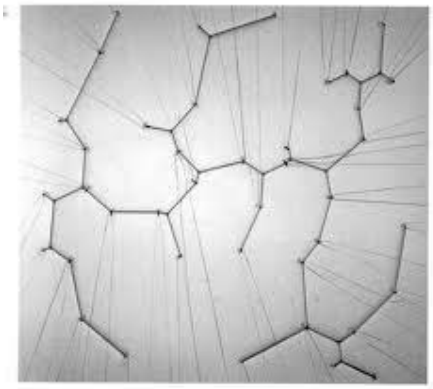
Communication services: Abandon primitive socket based message passing in favor of:
_ Procedure calls across networks
_ Remote-object method invocation
_ Message-queuing systems
_ Advanced communication streams
_ Event notification service
Information system services: Services that help manage data in a distributed system:
_ Large-scale, system wide naming services
_ Advanced directory services (search engines)
_ Location services for tracking mobile objects
_ Persistent storage facilities
_ Data caching and replication

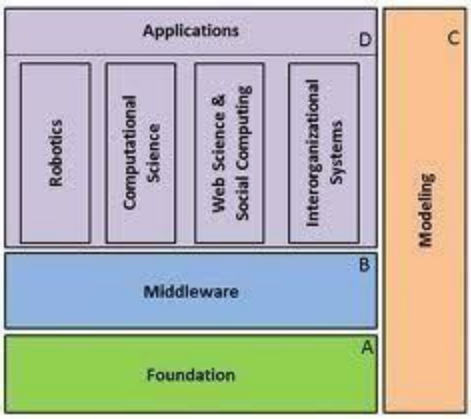
Control services: Services giving applications control over when, where, and how they access data:

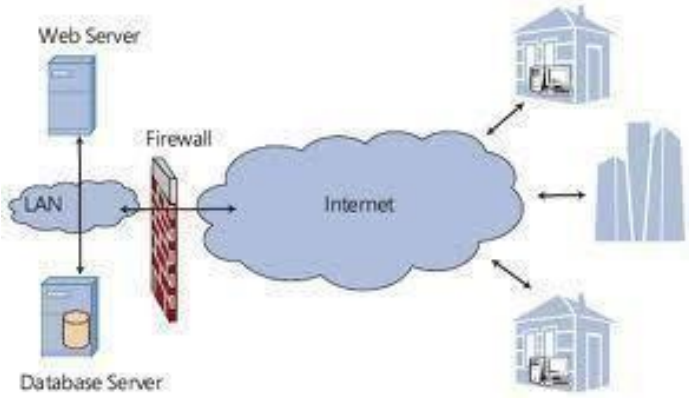
_ Distributed transaction processing
_ Code migration
Security services: Services for secure processing and communication:
_ Authentication and authorization services
_ Simple encryption services
_ Auditing service

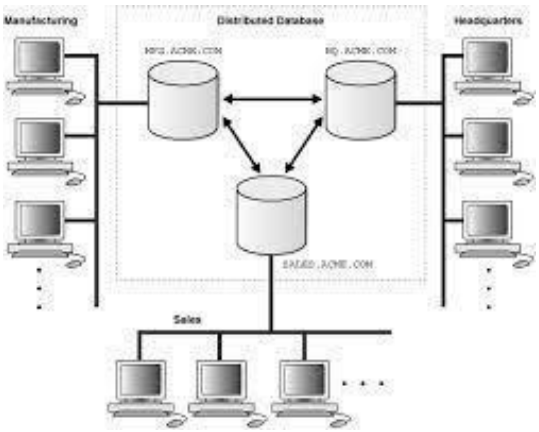
1.4.3 Comparison of DOS, NOS, and Middleware

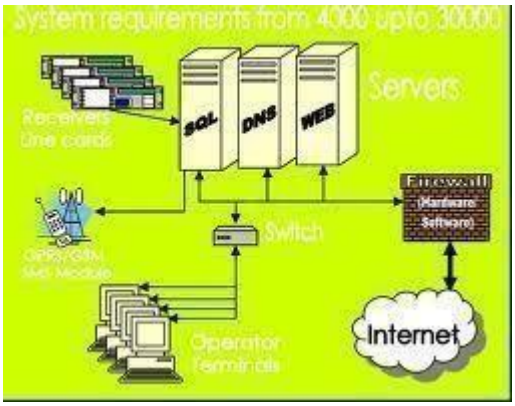
Item	Distributed OS		Network OS	Middle-ware DS
	multiproc.	multicomp.		
1	Very High	High	Low	High
2	Yes	Yes	No	No
3	1	N	N	N
4	Shared memory	Messages	Files	Model specific
5	Global, central	Global, distributed	Per node	Per node
6	No	Moderately	Yes	Varies
7	Closed	Closed	Open	Open



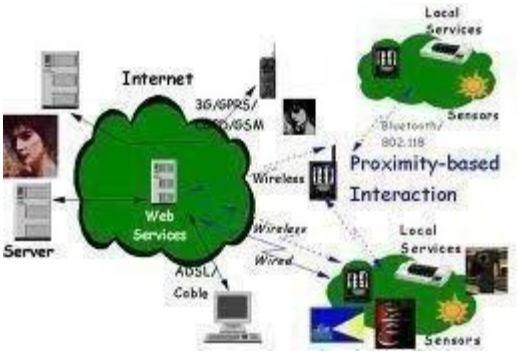

Applications
D
C
Robotics
Computational Science
Web Science & Social Computing
Interorganizational Systems
Modeling
B
Middleware
A
Foundation


Web Server
Firewall
LAN
Internet
Database Server


Manufacturing
Distributed Database
Headquarters
Sales


Servers
SQL
DNS
WEB
Switch
Firewall
(Hardware/
Software)
Internet
Operator
Terminals

-

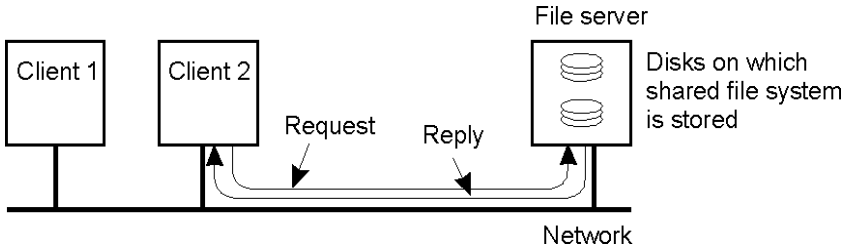


Network , Distributed and Middleware Operating System

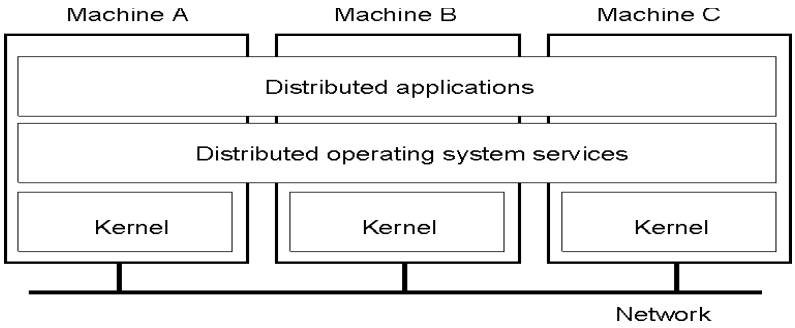
Networks of computers are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks – all of these, both separately and in combination, share the essential characteristics that make them relevant subjects for study under the heading *distributed systems*.

The operating system's middleware support role, it is useful to gain some historical perspective by examining two operating system concepts that have come about during the development of distributed systems: network operating systems and distributed operating systems.

Both UNIX and Windows are examples of *network operating systems*. They have a networking capability built into them and so can be used to access remote resources. Access is network-transparent for some – not all – types of resource. For example, through a distributed file system such as NFS, users have network-transparent access to files. That is, many of the files that users access are stored remotely, on a server, and this is largely transparent to their applications.



An operating system that produces a single system image like this for all the resources in a distributed system is called a *distributed operating system*



Middleware and the Operating System

What is a distributed OS?

- Presents users (and applications) with an integrated computing platform that hides the individual computers.
- Has control over all of the nodes (computers) in the network and allocates their resources to tasks without user involvement.
    - In a distributed OS, the user doesn't know (or care) where his programs are running.
- Examples:
    - Cluster computer systems
    - V system, Sprite

- In fact, there are no distributed operating systems in general use, only network operating systems such as UNIX, Mac OS and Windows.
- to remain the case, for two main reasons.

The first is that users have much invested in their application software, which often meets their current problem-solving needs; they will not adopt a new operating system that will not run their applications, whatever efficiency advantages it offers.

The second reason against the adoption of distributed operating systems is that users tend to prefer to have a degree of autonomy for their machines, even in a closely knit organization.

Combination of middleware and network OS

- No distributed OS in general use
    - Users have much invested in their application software
    - Users tend to prefer to have a degree of autonomy for their machines
- Network OS provides autonomy
- Middleware provides network-transparent access resource

The relationship between OS and Middleware

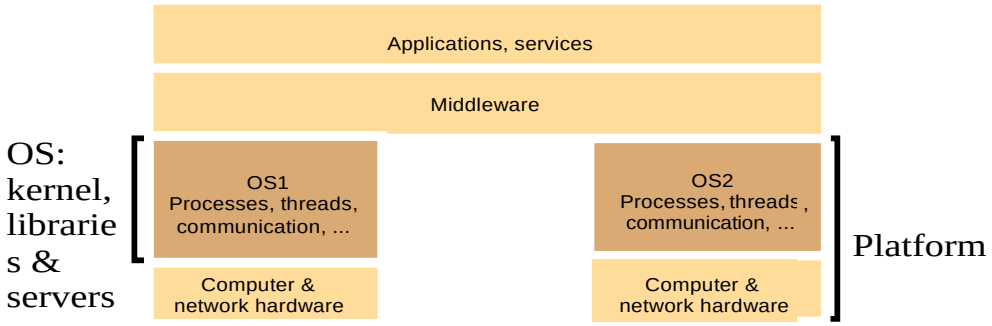
- Operating System
    - Tasks: processing, storage and communication
    - Components: kernel, library, user-level services
- Middleware

- runs on a variety of OS-hardware combinations

– remote invocations

Functions that OS should provide for middleware

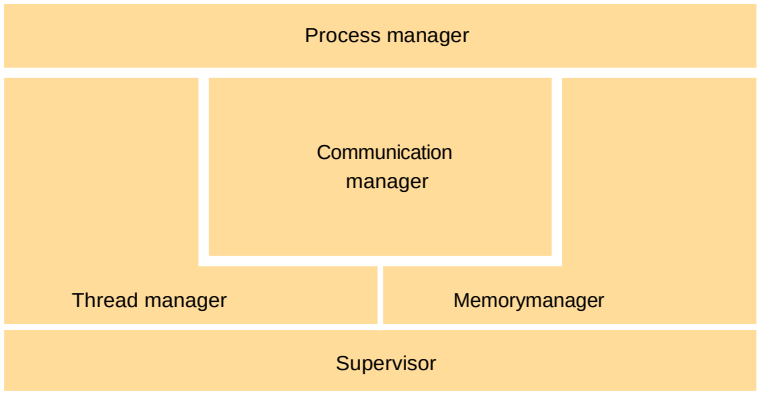
The following figure shows how the operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.



Node 1 Node 2

Encapsulation: They should provide a useful service interface to their resources – that is, a set of operations that meet their clients' needs. Details such as management of memory and devices used to implement resources should be hidden from clients.
Protection: Resources require protection from illegitimate accesses – for example, files are protected from being read by users without read permissions, and device registers are protected from application processes.
Concurrent processing: Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency.
Communication: Operation parameters and results have to be passed to and from resource managers, over a network or within a computer.
Scheduling: When an operation is invoked, its processing must be scheduled within the kernel or server.

The core OS components

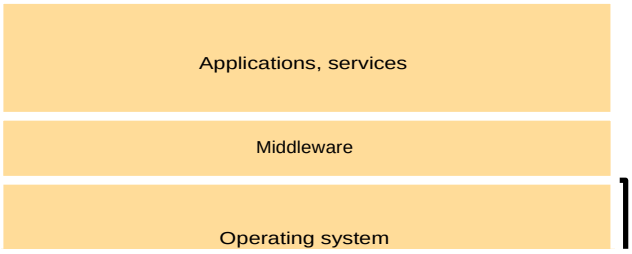


- **Process manager**
  - Handles the creation of and operations upon processes.

- **Thread manager**
  - Thread creation, synchronization and scheduling
- **Communication manager**
  - Communication between threads attached to different processes on the same computer
- **Memory manager**
  - Management of physical and virtual memory
- **Supervisor**
  - Dispatching of interrupts, system call traps and other exceptions
  - control of memory management unit and hardware caches

processor and floating point unit register manipulations

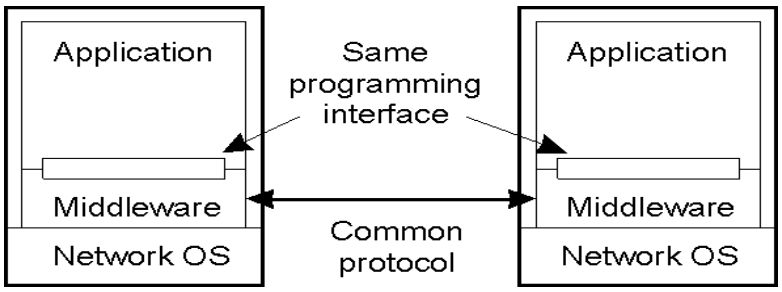
Software and hardware service layers in distributed systems



Platform

Middleware and Openness

- 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.



Typical Middleware Services

- Communication
- Naming
- Persistence
- Distributed transactions
- Security

Middleware Models

- Distributed files
  - Examples?
- Remote procedure call
  - Examples?

- Distributed objects
  - Examples?
- Distributed documents
  - Examples?
- Others?
  - Message-oriented middleware (MOM)
  - Service oriented architecture (SOA)
  - Document-oriented

Middleware and the Operating System

- Middleware implements abstractions that support network-wide programming. Examples:
  - RPC and RMI (Sun RPC, Corba, Java RMI)
  - event distribution and filtering (Corba Event Notification, Elvin)
  - resource discovery for mobile and ubiquitous computing
  - support for multimedia streaming
- Traditional OS's (e.g. early Unix, Windows 3.0)
  - simplify, protect and optimize the use of local resources
- Network OS's (e.g. Mach, modern UNIX, Windows NT)

- do the same but they also support a wide range of communication standards and enable remote processes to access (some) local resources (e.g. files).

DOS vs. NOS vs. Middleware Discussion

- What is good/bad about DOS?
  - Transparency
  - Other issues have reduced success.
  - Problems are often socio-technological.
- What is good/bad about NOS?
  - Simple.
  - Decoupled, easy to add/remove.
  - Lack of transparency.
- What is good/bad about middleware?
  - Easy to make multiplatform.
  - Easy to start something new.
    - But this can also be bad.

Types of Distributed Oss

System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

1.5 Design issues in distributed operating systems

In general, designing a distributed operating system is more difficult than designing a centralized operating system for several reasons.
In the design of a centralized operating system, it is assumed that the operating system has access to complete and accurate information about the environment in which it is functioning.

For example, a centralized operating system can request status information, being assured that the interrogated component will not charge state while awaiting a decision based on that status information, since only the single operating system asking the question may give commands. However, a distributed operating system must be designed with the assumption that complete information about the system environment will never be available.

In a distributed system, the resources are physically separated, there is no common clock among the multiple processors, delivery of messages is delayed, and messages could even be lost. Due to all these reasons, a distributed operating system does not have up-to-date, consistent knowledge about the state of the various components of the underlying distributed system. Obviously, lack of up-to-date and consistent information 15 makes many things (Such as management of resources

and synchronization of cooperating activities) much harder in the design of a distributed operating system.

1.5.1 TRANSPARENCY

A distributed system that is able to present itself to user and application as if it were only a single computer system is said to be transparent.

There are eight types of transparencies in a distributed system:

1) Access Transparency: It hides differences in data representation and how a resource is accessed by a user. Example, a distributed system may have a computer system that runs different operating systems, each having their own file naming conventions. Differences in naming conventions as well as how files can be manipulated should be hidden from the users and applications.

2) Location Transparency: Hides where exactly the resource is located physically. Example, by assigning logical names to resources like yahoo.com, one cannot get an idea of the location of the web page's main server.

3) Migration Transparency: Distributed system in which resources can be moved without affecting how the resource can be accessed are said to provide migration transparency. It hides that the resource may move from one location to another.

4) Relocation Transparency: this transparency deals with the fact that resources can be relocated while it is being accessed without the user who is using the application to know anything. Example: using a Wi-Fi system on laptops while moving from place to place without getting disconnected.

5) Replication Transparency: Hides the fact that multiple copies of a resource could exist simultaneously. To hide replication, it 16 is essential that the replicas have the same name. Consequently, as system that supports replication should also support location transparency.

6) Concurrency Transparency: It hides the fact that the resource may be shared by several competitive users. Example, two independent users may each have stored their file on the same server and may be accessing the same table in a shared database. In such cases, it is important that each user doesn't notice that the others are making use of the same resource.

7) Failure Transparency: Hides failure and recovery of the resources. It is the most difficult task of a distributed system and is even impossible when certain apparently realistic assumptions are made. Example: A user cannot distinguish between a very slow or dead resource. Same error message come when a server is down or when the network is overloaded of when the connection from the client side is lost. So here, the user is unable to understand what has to be done, either the user should wait for the network to clear up, or try again later when the server is working again.

8) Persistence Transparency: It hides if the resource is in memory or disk. Example, Object oriented database provides facilities for directly invoking methods on storage objects. First the database server copies the object states from the disk i.e. main memory performs the operation and writes the state back to the disk. The user does not know that the server is moving between primary and secondary memory.

Persistence	Hide whether a (software) resource is in memory or on disk
Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be shared by several competitive users
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

1.5.2 PERFORMANCE TRANSPARENCY

The aim of performance transparency is to allow the system to be automatically reconfigured to improve performance, as loads vary dynamically in the system. As far as practicable, a situation in which one processor of the system is overloaded with jobs while another processor is idle should not be allowed to occur. That is, the processing capability of the system should be uniformly distributed among the currently available jobs in the system. This requirements calls for the support of intelligent resource allocation and process migration facilities in distributed operating systems.

1.5.3 SCALING TRANSPARENCY:

The aim of scaling transparency is to allow the system to expand in scale without disrupting the activities of the users. This requirement calls for open-system architecture and the use of scalable algorithms for designing the distributed operating system components.

1.5.4 RELIABILITY

In general, distributed systems are expected to be more reliable than centralized systems due to the existence of multiple instances of resources. However, the existence of multiple instances of the resources alone cannot increase the system's reliability. Rather, the distributed operating system, which manages these resources must be designed properly to increase the system's reliability by taking full advantage of this characteristic feature of a distributed system.

A fault is a mechanical or algorithmic defect that may generate an error. A fault in a system causes system failure. Depending on the manner in which a failed system behaves, system failures are of two types – fail stop [Schlichting and Schneider 1983] and Byzantine [Lamport et al. 1982]. In the case of fail-step failure, the system stops functioning after changing to a state in which its failure can be detected. On the other hand, in the case of Byzantine failure, the system continues to function but produces wrong results. Undetected software bugs often cause Byzantine failure of a system. Obviously, Byzantine failures are much more difficult to deal with than fail-stop failures.

For higher reliability, the fault-handling mechanisms of a distributed operating system must be designed properly to avoid faults, to tolerate faults, and to detect and recover form faults. Commonly used methods for dealing with these issues are briefly described text.

1.5.5 FAULT AVOIDANCE:

Fault avoidance deals with designing the components of the system in such a way that the occurrence of faults in minimized. Conservative design practice such as using high reliability components are often employed for improving the system's reliability based on the idea of fault

avoidance. Although a distributed operating system often has little or no role to play in improving the fault avoidance capability of a hardware component, 19 the designers of the various software components of the distributed operating system must test them thoroughly to make these components highly reliable.

1.5.6 FAULT TOLERANCE

Fault tolerance is the ability of a system to continue functioning in the event of partial system failure. The performance of the system might be degraded due to partial failure, but otherwise the system functions properly.

Some of the important concepts that may be used to improve the fault tolerance ability of a distributed operating system are as follows :

1. Redundancy techniques : The basic idea behind redundancy techniques is to avoid single points of failure by replicating critical hardware and software components, so that if one of them fails, the others can be used to continue.

   Obviously, having two or more copies of a critical component makes it possible, at least in principle, to continue operations in spite of occasional partial failures. For example, a critical process can be simultaneously executed on two nodes so that if one of the two nodes fails, the execution of the process can be completed at the other node. Similarly, a critical file may be replicated on two or more storage devices for better reliability. Notice that with redundancy techniques additional system overhead is needed to maintain two or more copies of a replicated resource and to keep all the copies of a resource consistent.

   For example, if a file is replicated on two or more nodes of a distributed system, additional disk storage space is required and for correct functioning, it is often necessary that all the copies of the file are mutually consistent. In general, the larger is the number of copies kept, the better is the reliability but the incurred overhead involved. Therefore, a distributed operating system must be designed to maintain a proper balance between the degree of reliability and the incurred overhead. This raises an important question : How much replication is enough? For an answer to this question, note that a system is said to be k-fault tolerant if it can continue to function even in the event of the failure of k components [Cristian 1991, Nelson 1990]. Therefore, if the system is to be designed to tolerance k fail – stop failures, k + 1 replicas are needed. If k replicas are lost due to failures, the remaining one replica can be used for continued functioning of the system. On the other hand, if the system is to be designed to tolerance k Byzantine failures, a minimum of 2k + 1 replicas are needed. This is because a voting mechanism can be used to believe the 20 majority k + 1 of the replicas when k replicas behave abnormally. Another application of redundancy technique is in the design of a stable storage device, which is a virtual storage device that can even withstand transient I/O faults and decay of the storage media. The reliability of a critical file may be improved by storing it on a stable storage device.

2. Distributed control: For better reliability, many of the particular algorithms or protocols used in a distributed operating system must employ a distributed control mechanism to avoid single points of failure. For example, a highly available distributed file system should have multiple and independent file servers controlling multiple and independent storage devices. In addition to file servers, a distributed control technique could also be used for name servers, scheduling algorithms, and other executive control functions. It is important to note here that when multiple distributed servers are used in a distributed system to provide a particular type of service, the servers must be independent. That is, the design must not require simultaneous functioning of the servers; otherwise, the reliability will become worse instead of getting better. Distributed control mechanisms are described throughout this book.

FAULT DETECTION AND RECOVERY

The faulty detection and recovery method of improving reliability deals with the use of hardware and software mechanisms to determine the occurrence of a failure and then to correct the system to a state acceptable for continued operation.

Some of the commonly used techniques for implementing this method in a distributed operating system are as follows.

1. Atomic transactions : An atomic transaction (or just transaction for shore) is a computation consisting of a collection of operation 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 prevails, other processes executing concurrently cannot modify or observe intermediate states of the computation. Transactions help to preserve the consistency of a set of shared date objects (e.g. files) in the face of failures and concurrent access. They make crash recovery much easier, because transactions can only end in two states : Either all the operations of the transaction are performed or none of the operations of the transaction is performed. 21 In a system with transaction facility, if a process halts unexpectedly due to a hardware error before a transaction is completed, the system subsequently restores any data objects that were undergoing modification to their original states. Notice that if a system does not support a transaction mechanism, unexpected failure of a process during the processing of an operation may leave the data objects that were undergoing modification in an inconsistent state. Therefore, without transaction facility, it may be difficult or even impossible in some cases to roll back (recover) the data objects from their current inconsistent states to their original states.

2. Stateless servers: The client-server model is frequently used in distributed systems to service user requests. In this model, a server may be implemented by using any one of the following two service paradigms – stateful or stateless. The two paradigms are distinguished by one aspect of the client – server relationship, whether or not the history of the serviced requests between a client and a server affects the execution of the next service request. The stateful approach does depend on the history of the serviced requests, but the stateless approach does not depend on it. Stateless servers have a distinct advantage over stateful servers in the event of a failure. That is, the stateless service paradigm makes crash recovery very easy because no client state information is maintained by the server. On the other hand, the stateful service paradigm requires complex crash recovery procedures. Both the client and server need to reliably detect crashes. The server needs to detect client crashes so that it can discard any state it is holding for the client, and the client must detect server crashes so that it can perform necessary error – handling activities. Although stateful service becomes necessary in some cases, to simplify the failure detection and recovery actions, the stateless service paradigm must be used, wherever possible.

1.5.7 FLEXIBILITY

Another important issue in the design of distributed operating systems is flexibility. Flexibility is the most important features for open distributed systems. The design of a distributed operating system should be flexible due to the following reasons :

1. Ease of modification
2. Ease of enhancement

1.5.8 PERFORMANCE

If a distributed system is to be used its performance must be at least as good as a centralized system. That is, when a particular application is run on a distributed system, its overall performance should be better than or at least equal to that of running the same application on a single processor system.

However, to achieve his goal, it is important that the various components of the operating

system of a distributed system be designed properly; otherwise, the overall performance of the distributed system may turn out to be worse than a centralized system.

Some design principles considered useful for better performance are as follows :
1. Batch if possible, Batching often helps in improving performance greatly. For example, transfer of data across the network in large chunks rather than as individual pages is much more efficient. Similarly, piggybacking of acknowledgement of previous messages with the next message during a series of messages exchanged between two communicating entities also improves performance.
2. Cache whenever possible : Caching of data at clients' sites frequently improves overall system performance because it makes data available wherever it is being currently used, thus saving a large amount of computing time and network bandwidth. In addition, caching reduces contention on centralized resources.
3. Minimize copying of data : Data copying overhead (e.g. moving data in and out of buffers) involves a substantial CPU cost of many operations.
For example, while being transferred from its sender to its receiver, a message data may take the following path on the sending side :
(a) From sender's stack to its message buffer
(b) From the message buffer in the sender's address space to the message buffer in the kernel's address space
(c) Finally, from the kernel to the network interface board On the receiving side, the data probably takes a similar path in the reverse direction. Therefore, in this case, a total of six copy operations are involved in the message transfer operation. Similarly, in several systems, the data copying overhead is also large for read and write operations on block I/O devices. Therefore, for better performance, it is desirable to avoid copying of data, although this is not always simple to achieve. Making optimal use of memory management often helps in eliminating much data movement between the kernel, block I/O devices, clients, and servers.
4. Minimize network traffic : System performance may also be improved by reducing internode communication costs.
For example, accesses to remote resources require communication, possibly through intermediate nodes. Therefore, migrating a process closer to the resources it is using most heavily may be helpful in reducing network traffic in the system if the decreased cost of accessing its favorite resource offsets the possible increased post of accessing its less favored ones. Another way to reduce network traffic is to use the process migration facility to cluster two or more processes that frequently communicate with each other on the same node of the system. Avoiding the collection of global state information for making some decision also helps in reducing network traffic.
5. Take advantage of fine-grain parallelism for multiprocessing. Performance can also be improved by taking advantage of finegiam parallelism for multiprocessing.
For example, threads are often used for structuring server processes. Servers structured as a group of threads can operate efficiently, because they can simultaneously service requests from several clients. Finegrained concurrency control of simultaneous accesses by multiple processes, to a shared resource is another example of application of this principle for better performance. Throughout the book we will come across the use of these design principles in the design of the various distributed operating system components.

1.5.9 SCALABILITY

Scalability refers to the capability of a system to adapt to increased service load. It is inevitable that a distributed system will grow with time since it is very common to add new machines or an entire subnetwork to the system to take care of increased workload or

organizational changes in a company. Therefore, a distributed operating system should be designed to easily cope with the growth of nodes and users in the system. That is, such growth should not cause serious disruption of service or significant loss of performance to users.

Some guiding principles for designing scalable distributed systems are as follows :

1. Avoid centralized entities : In the design of a distributed operating system, use of centralized entities such as a single central file server or a single database for the entire system makes the distributed system non-scalable due to the following reasons :
2. Security : In order that the users can trust the system and rely on it, the various resources of a computer system must be protected against destruction and unauthorized access. Enforcing security in a distributed system is more difficult than in a centralized system because of the lack of a single point of control and the use of insecure networks for data communication.
3. In a centralized system, all users are authenticated by the system at login time, and the system can easily check whether a user is authorized to perform the requested operation on an accessed resource. In a distributed system, however, since the client – server model is often used for requesting and providing services, when a client sends a request message to a server, the server must have some way of knowing who is the client. This is not so simple as it might appear because any client identification field in the message cannot be trusted. This is because an intruder (a person or program trying to obtain unauthorized access to system resources) may pretend to be an authorized client or may change the message contents during transmission. Therefore, as compared to a centralized system, enforcement of security in a distributed system has the following additional requirements :

   1. It should be possible for the sender of a message to know that the message was received by the intended receiver.
   2. It should be possible for the receiver of a message to know that the message was sent by the genuine sender.
   3. It should be possible for both the sender and receiver of a message to be guaranteed that the contents of the message were not changed while it was in transfer.

Distributed systems has the following significant consequences:

Concurrency: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network. We will describe ways in which this extra capacity can be usefully deployed at many points in this book. The coordination of concurrently executing programs that share resources is also an important and recurring topic.

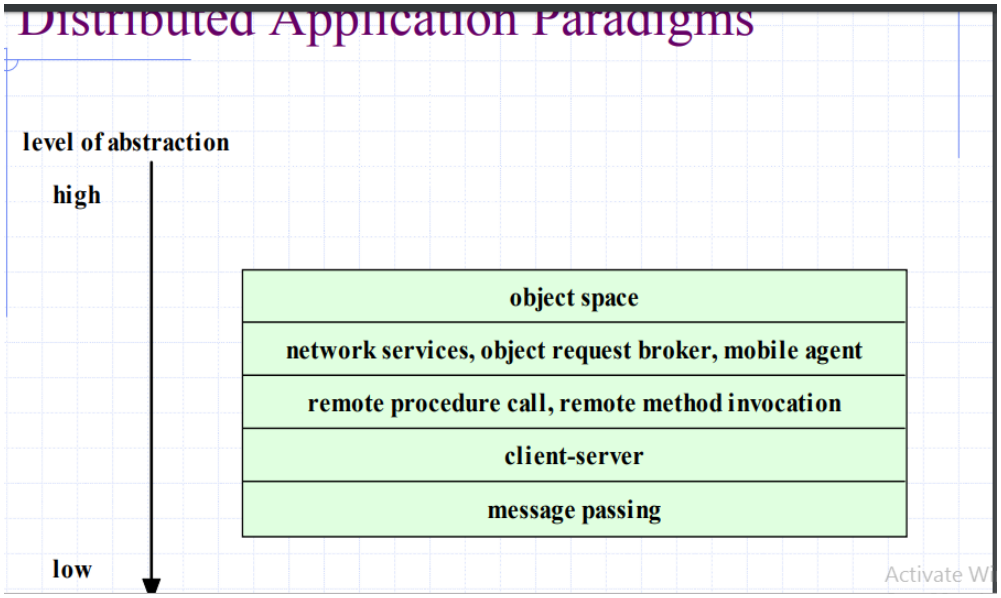
No global clock: When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network.

Independent failures: All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed

systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact, the programs
on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a *crash*), is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running.

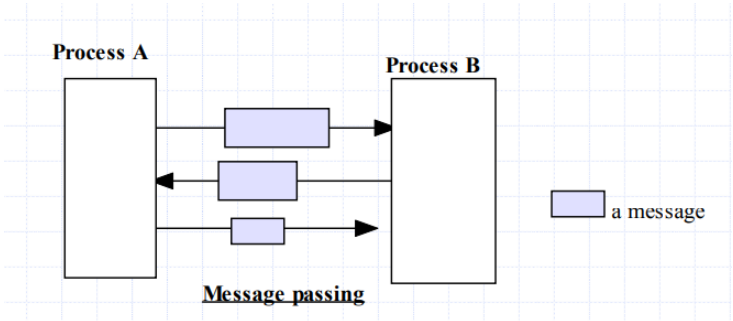
1.6 Distributed Computing Paradigms

Distributed Application Paradigms



1.6.1 The Message Passing Paradigm

Message passing is the most fundamental paradigm for distributed applications. A process sends a message representing a request. „ The message is delivered to a receiver, which processes the request, and sends a message in response. „ In turn, the reply may trigger a further request, which leads to a subsequent reply, and so forth.

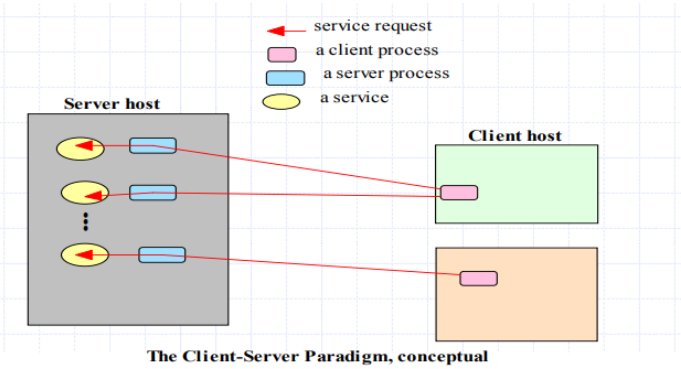


The Message Passing Paradigm

The basic operations required to support the basic message passing paradigm are send, and receive. For connection-oriented communication, the operations connect and disconnect are also required. With the abstraction provided by this model, the interconnected processes perform input and output to each other, in a manner similar to file I/O. The I/O operations encapsulate the detail of network communication at the operating-system level.

1.6.2 The Client-Server Paradigm

Perhaps the best known paradigm for network applications, the client-server model assigns asymmetric roles to two collaborating processes. One process, the server, plays the role of a service provider which waits passively for the arrival of requests. The other, the client, issues specific requests to the server and awaits its response.



The Client-Server Paradigm, conceptual

The Client-Server Paradigm –

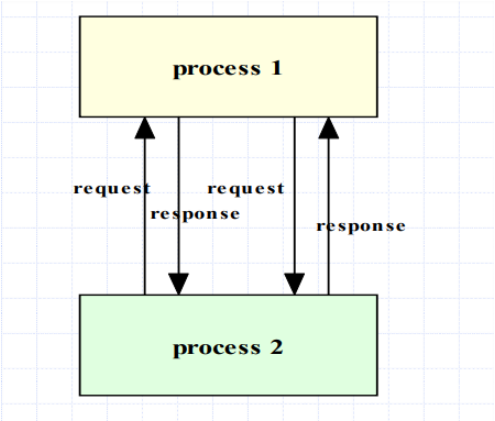
Simple in concept, the client-server model provides an efficient abstraction for the delivery of network services. Operations required include those for a server process to listen and to accept requests, and for a client process to issue requests and accept responses. By assigning asymmetric roles to the two sides, event synchronization is simplified: the server process waits for requests, and the client in turn waits for responses. Many Internet services are client-server applications. These services are often known by the protocol that the application implements. Well known Internet services include HTTP, FTP, DNS, finger, gopher, etc.

The Peer-to-Peer System Architecture:

An architecture where computer resources and services are direct exchanged between computer systems. Resources and services include the exchange of information, processing cycles, cache storage, and disk storage for files.. Computers that have traditionally been used solely as clients communicate directly among themselves and can act as both clients and servers, assuming whatever role is most efficient for the network.

1.6.3 The Peer-to-Peer Distributed Computing Paradigm

In the peer-to-peer paradigm, the participating processes play equal roles, with equivalent capabilities and responsibilities (hence the term "peer"). Each participant may issue a request to another participant and receive a response.



Peer-to-Peer distributed computing

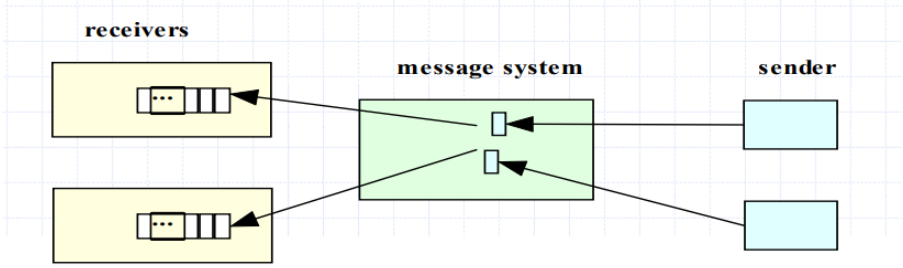
The peer-to-peer paradigm is more appropriate for applications such as instant messaging, peer-to-peer file transfers, video conferencing, and collaborative work.

„ It is also possible for an application to be based on both the client-server model and the peer-to-peer model.

A well-known example of a peer-to-peer file transfer service is Napster.com or similar sites which allow files (primarily audio files) to be transmitted among computers on the Internet. „ It makes use of a server for directory in addition to the peer-to-peer computing.

The Message System Paradigm The Message System or Message-Oriented Middleware (MOM) paradigm is an elaboration of the basic message-passing paradigm. In this paradigm, a message system serves as an intermediary among separate, independent processes.

The message system acts as a switch for messages, through which processes exchange messages asynchronously, in a decoupled manner. A sender deposits a message with the message system, which forwards it to a message queue associated with each receiver. Once a message is sent, the sender is free to move on to other tasks.



The Point-To-Point Message Model

In this model, a message system forwards a message from the sender to the receiver's message

queue.

Unlike the basic message passing model, the middleware provides a message repository, and allows the sending and the receiving to be decoupled. „

Using the middleware, a sender deposits a message in the message queue of the receiving process. A receiving process extracts the messages from its message queue, and handles each one accordingly. Compared to the basic message-passing model, this paradigm provides the additional abstraction for asynchronous operations.

The Publish/Subscribe Message Model

In this model, each message is associated with a specific topic or event. Applications interested in he occurrence of a specific event may subscribe to messages for that event. When the awaited event occurs, the process publishes a message announcing the event or topic. The middleware message system distributes the message to all its subscribers. The publish/subscribe message model offers a powerful abstraction for multicasting or group communication. The publish operation allows a process to multicast to a group of processes, and the subscribe operation allows a process to listen for such multicast.

1.7 TRENDS IN DISTRIBUTED SYSTEMS

Distributed systems are undergoing a period of significant change and this can be traced back

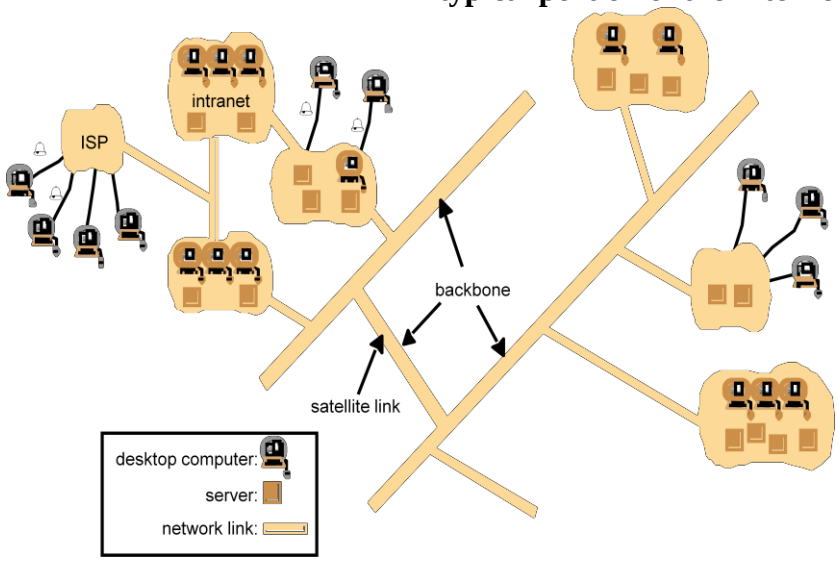
a number of influential trends:

- the emergence of pervasive networking technology;
- the emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems;
- the increasing demand for multimedia services;
- the view of distributed systems as a utility.

Internet

The modern Internet is a vast interconnected collection of computer networks of many different types, with the range of types increasing all the time and now including, for example, a wide range of wireless communication technologies such as WiFi, WiMAX, Bluetooth and third-generation mobile phone networks. The net result is that networking has become a pervasive resource and devices can be connected (if desired) at any time and in any place.

A typical portion of the Internet



The Internet is also a very large distributed system. It enables users, wherever they are, to make use of services such as the World Wide Web, email and file transfer. (Indeed, the Web is sometimes incorrectly equated with the Internet.) The set of services is open-ended – it can be extended by the addition of server computers and new types of service. The figure shows a collection of intranets – subnetworks operated by companies and other organizations and typically protected by firewalls. The role of a *firewall* is to protect an intranet by preventing unauthorized messages from leaving or entering. A firewall is implemented by filtering incoming and outgoing messages. Filtering might be done by source or destination, or a firewall might allow only those messages related to email and web access to pass into or out of the intranet that it protects. Internet Service Providers (ISPs) are companies that provide broadband links and other types of connection to individual users and small organizations, enabling them to access services anywhere in the Internet as well as providing local services such as email and web hosting. The intranets are linked together by backbones. A *backbone* is a network link with a high transmission capacity, employing satellite connections, fibre optic cables and other high-bandwidth circuits

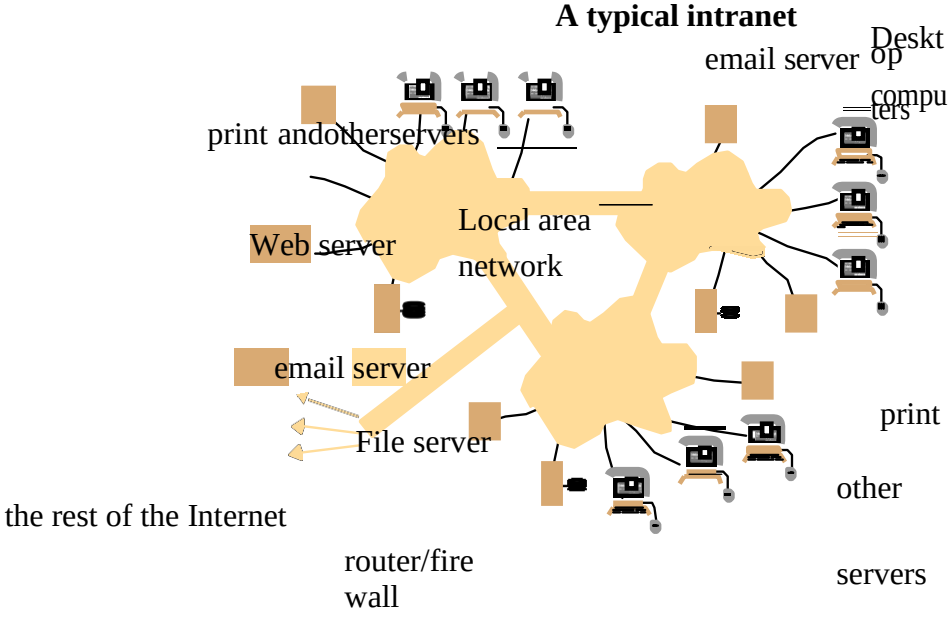
Computers vs. Web servers in the Internet

Date	*Computers*	*Web servers*	*Percentage*
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
		42,298,371	

Intranet

- A portion of the Internet that is separately administered and has a boundary that can be configured to enforce local security policies
- Composed of several LANs linked by backbone connections
- Be connected to the Internet via a router

A typical intranet



Main issues in the design of components for the use in intranet

- File services
- Firewall
- The cost of software installation and support

Mobile and ubiquitous computing

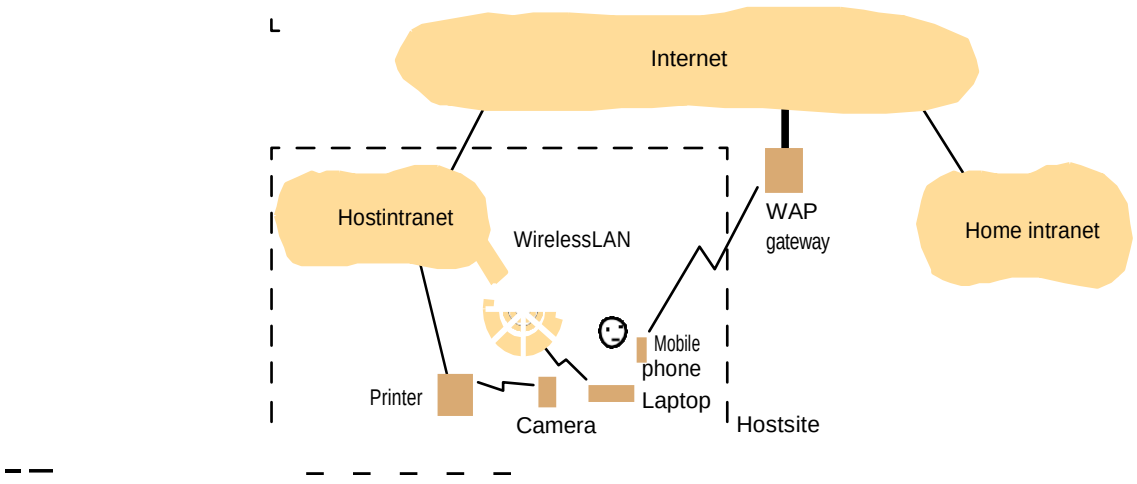
Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

- Laptop computers.
- Handheld devices, including mobile phones, smart phones, GPS-enabled devices, pagers, personal digital assistants (PDAs), video cameras and digital cameras.
- Wearable devices, such as smart watches with functionality similar to a PDA.
- Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes *mobile computing* possible. Mobile computing is the

performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their 'home' intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them. They can continue to access the Internet; they can continue to access resources in their home intranet; and there is increasing provision for users to utilize resources such as printers or even sales points that are conveniently nearby as they move around. The latter is also known as *location-aware* or *context-aware computing*. Mobility introduces a number of challenges for distributed systems, including the need to deal with variable connectivity and indeed disconnection, and the need to maintain operation in the face of device mobility.

Portable and handheld devices in a distributed system



Ubiquitous computing is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and even natural settings. The term 'ubiquitous' is intended to suggest that small computing devices will eventually

become so pervasive in everyday objects that they are scarcely noticed. That is, their computational behaviour will be transparently and intimately tied up with their physicalfunction.

The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it may be convenient for users to control their washing machine or their entertainment system from their phone or a 'universal remote control' device in the home. Equally, the washing machine could notify the user via a smart badge or phone when the washing is done.

Ubiquitous and mobile computing overlap, since the mobile user can in principle benefit from computers that are everywhere. But they are distinct, in general. Ubiquitous computing could benefit users while they remain in a single environment such as the home or a hospital. Similarly, mobile computing has advantages even if it involves only conventional, discrete computers and devices such as laptops and printers.

RESOURCE SHARING

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

A complete interaction between *client* and *server*, from the point when the *client* sends its request to when it receives the server's response

- Motivation of WWW
  - Documents sharing between physicists of CERN
  - Web is an open system: it can be extended and implemented in new ways without disturbing its existing functionality.
  - Its operation is based on communication standards and document standards
  - Respect to the types of 'resource' that can be published and shared on it.
- HyperText Markup Language
  - A language for specifying the contents and layout of pages
- Uniform Resource Locators
  - Identify documents and other resources
- A client-server architecture with HTTP
  - By with browsers and other clients fetch documents and other resources from web servers

HTML

HTML text is stored in a file of a web server.

```
<IMG SRC = http://www.cdk3.net/WebExample/Images/earth.jpg>
<P>
Welcome to Earth! Visitors may also be interested in taking a look at
the
<A HREF = "http://www.cdk3.net/WebExample/moon.html>Moon</A>.
<P>
(etcetera)
```

- A browser retrieves the contents of this file from a web server.

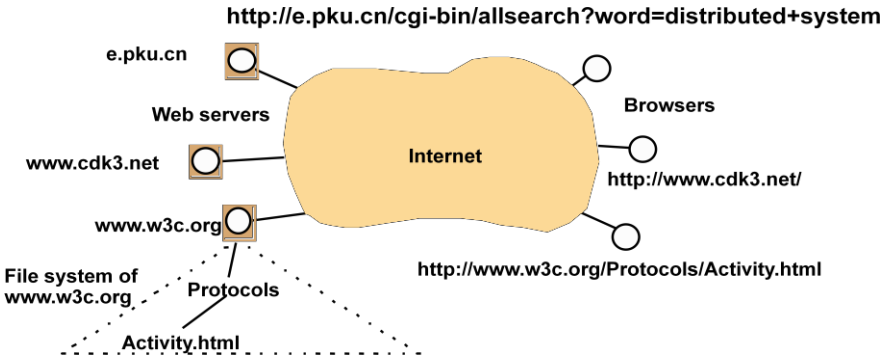
-The browser interprets the HTML text

-The server can infer the content type from the filename extension.

URL

- HTTP URLs are the most widely used
- An HTTP URL has two main jobs to do:
  - To identify which web server maintains the resource
  - To identify which of the resources at that server

Web servers and web browsers



HTTP URLs

- http://servername[:port]//pathNameOnServer][?arguments]
- e.g.

http://www.cdk3.net/
http://www.w3c.org/Protocols/Activity.html
http://e.pku.cn/cgi-bin/allsearch?word=distributed+system

Server DNS name	Pathname on server	Arguments
www.cdk3.net	(default)	(none)
www.w3c.org	Protocols/Activity.html	(none)
e.pku.cn	cgi-bin/allsearch	word=distributed+system

- Publish a resource remains unwieldy

HTTP

- Defines the ways in which browsers and any other types of client interact with web servers (RFC2616)
- Main features
  - Request-replay interaction
  - Content types. The strings that denote the type of content are called MIME (RFC2045,2046)
  - One resource per request. HTTP version 1.0
  - Simple access control

More features-services and dynamic pages

- Dynamic content
  - Common Gateway Interface: a program that web servers run to generate content for their clients
- Downloaded code
  - JavaScript
  - Applet

Discussion of Web

- Dangling: a resource is deleted or moved, but links to it may still remain
- Find information easily: e.g. Resource Description Framework which standardize the format of *metadata* about web resources
- Exchange information easily: e.g. XML – a *self describing* language
- Scalability: heavy load on popular web servers
- More applets or many images in pages increase in the download time

1.8 THE CHALLENGES IN DISTRIBUTED SYSTEM

Heterogeneity

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

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

Although the Internet consists of many different sorts of network, their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network.
Data types such as integers may be represented in different ways on different sorts of hardware – for example, there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware. Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

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

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

Heterogeneity and mobile code • The term *mobile code* is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.
The *virtual machine* approach provides a way of making code executable on a variety of host computers: the compiler for a particular language generates code for a virtual machine instead of

a particular hardware order code. For example, the Java compiler produces code for a Java virtual machine, which executes it by interpretation.
The Java virtual machine needs to be implemented once for each type of computer to enable Java programs to run.
Today, the most commonly used form of mobile code is the inclusion Javascript programs in some web pages loaded into client browsers.

Openness

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

Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures,
which are usually cumbersome and slow-moving. However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people. The designers of the Internet protocols introduced a series of documents called 'Requests For Comments', or RFCs, each of which is known by a number. The specifications of the Internet communication protocols were published in this series in the early 1980s, followed by specifications for applications that run over them, such as file transfer, email and telnet by the mid-1980s.

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

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

Security

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

(protection against alteration or corruption), and availability (protection against interference with the means to access the resources).

In a distributed system, clients send requests to access data managed by servers, which involves sending information in messages over a network. For example:
1. A doctor might request access to hospital patient data or send additions to that data.
2. In electronic commerce and banking, users send their credit card numbers across the Internet.

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

However, the following two security challenges have not yet been fully met:

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

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

Scalability

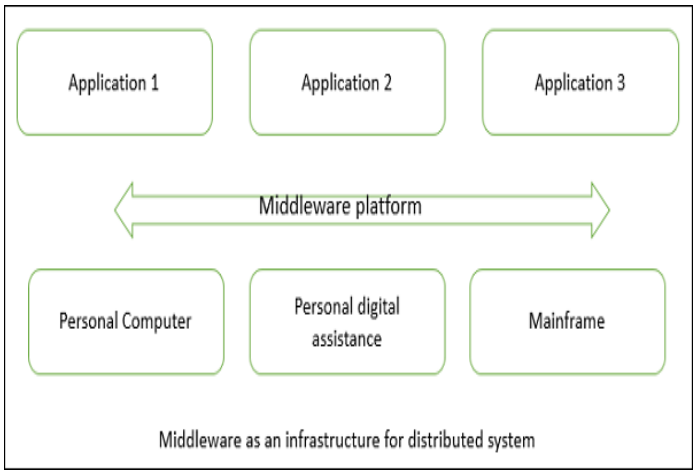
Distributed systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as *scalable* if it will remain effective when there is a significant increase in the number of resources and the number of users. The number of computers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

1.9 Distributed Architecture

Goals:In distributed architecture, components are presented on different platforms and several components can cooperate with one another over a communication network in order to achieve a specific objective or goal.

- In this architecture, information processing is not confined to a single machine rather it is distributed over several independent computers.
- A distributed system can be demonstrated by the client-server architecture which forms the base for multi-tier architectures; alternatives are the broker architecture such as CORBA, and the Service-Oriented Architecture (SOA).
- There are several technology frameworks to support distributed architectures, including .NET, J2EE, CORBA, .NET Web services, AXIS Java Web services, and Globus Grid services.
- Middleware is an infrastructure that appropriately supports the development and execution of distributed applications. It provides a buffer between the applications and the network.
- It sits in the middle of system and manages or supports the different components of a distributed system. Examples are transaction processing monitors, data convertors and communication controllers etc.

Middleware as an infrastructure for distributed system



The basis of a distributed architecture is its transparency, reliability, and availability.

The following table lists the different forms of transparency in a distributed system −

Sr.No.	Transparency & Description
1	**Access** Hides the way in which resources are accessed and the differences in data platform.
2	**Location** Hides where resources are located.

3	**Technology** Hides different technologies such as programming language and OS from user.
4	**Migration / Relocation** Hide resources that may be moved to another location which are in use.
5	**Replication** Hide resources that may be copied at several location.
6	**Concurrency** Hide resources that may be shared with other users.
7	**Failure** Hides failure and recovery of resources from user.
8	**Persistence** Hides whether a resource (software) is in memory or disk.

Advantages

- **Resource sharing** − Sharing of hardware and software resources.
- **Openness** − Flexibility of using hardware and software of different vendors.
- **Concurrency** − Concurrent processing to enhance performance.
- **Scalability** − Increased throughput by adding new resources.
- **Fault tolerance** − The ability to continue in operation after a fault has occurred.

Disadvantages

- **Complexity** − They are more complex than centralized systems.
- **Security** − More susceptible to external attack.
- **Manageability** − More effort required for system management.
- **Unpredictability** − Unpredictable responses depending on the system organization and network load.

Centralized System vs. Distributed System

Criteria	Centralized system	Distributed System
Economics	Low	High

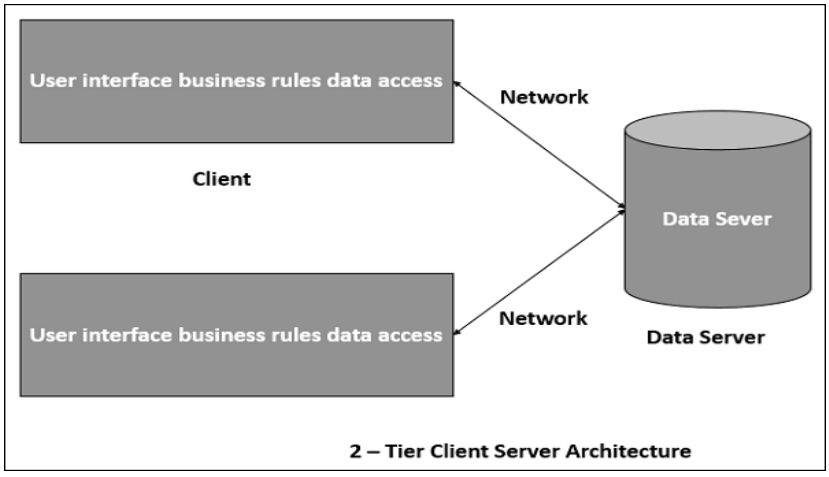
Availability	Low	High
Complexity	Low	High
Consistency	Simple	High
Scalability	Poor	Good
Technology	Homogeneous	Heterogeneous
Security	High	Low

1.9.1 Client-Server Architecture

The client-server architecture is the most common distributed system architecture which decomposes the system into two major subsystems or logical processes −

- **Client** − This is the first process that issues a request to the second process i.e. the server.
- **Server** − This is the second process that receives the request, carries it out, and sends a reply to the client.

In this architecture, the application is modelled as a set of services that are provided by servers and a set of clients that use these services. The servers need not know about clients, but the clients must know the identity of servers, and the mapping of processors to processes is not necessarily



Client-server Architecture can be classified into two models based on the functionality of the client −

Thin-client model

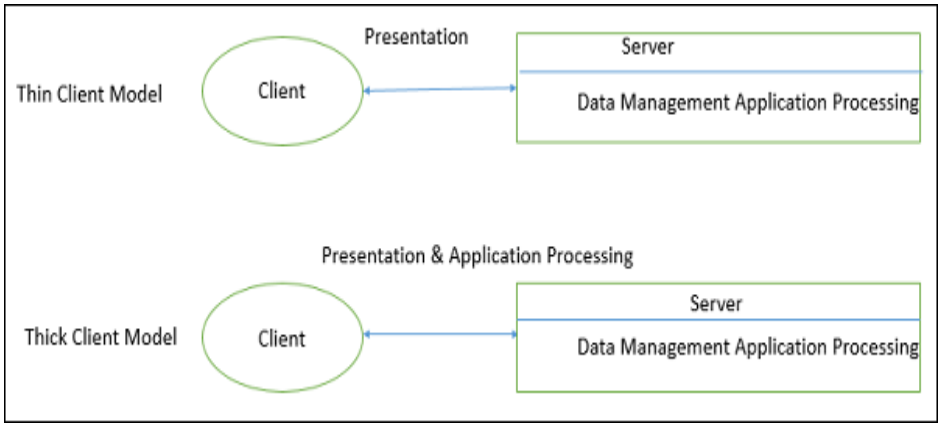
In thin-client model, all the application processing and data management is carried by the server. The client is simply responsible for running the presentation software.

- Used when legacy systems are migrated to client server architectures in which legacy system acts as a server in its own right with a graphical interface implemented on a client
- A major disadvantage is that it places a heavy processing load on both the server and the network.

Thick/Fat-client model

In thick-client model, the server is only in charge for data management. The software on the client implements the application logic and the interactions with the system user.

- Most appropriate for new C/S systems where the capabilities of the client system are known in advance
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.



Advantages

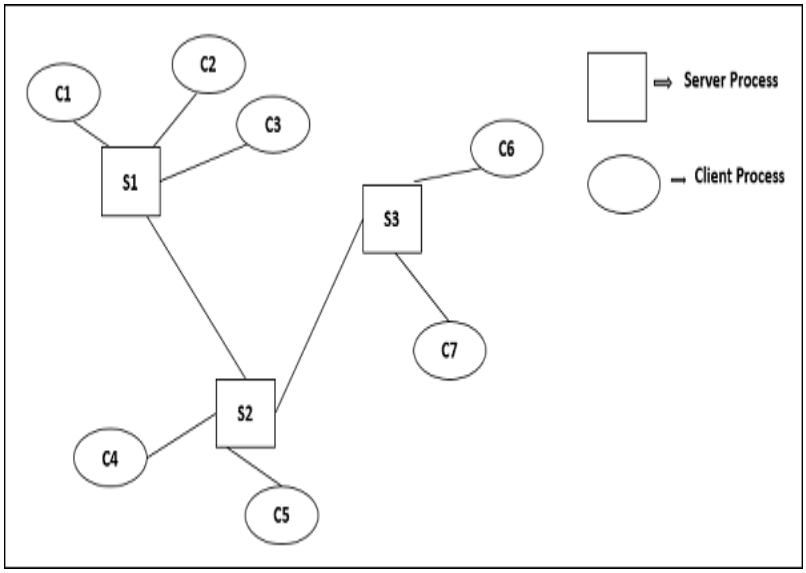
- Separation of responsibilities such as user interface presentation and business logic processing.
- Reusability of server components and potential for concurrency
- Simplifies the design and the development of distributed applications
- It makes it easy to migrate or integrate existing applications into a distributed environment.
- It also makes effective use of resources when a large number of clients are accessing a high-performance server.

Disadvantages

- Lack of heterogeneous infrastructure to deal with the requirement changes.
- Security complications.
- Limited server availability and reliability.
- Limited testability and scalability.
- Fat clients with presentation and business logic together.

Multi-Tier Architecture (n-tier Architecture)

Multi-tier architecture is a client–server architecture in which the functions such as presentation, application processing, and data management are physically separated. By separating an application into tiers, developers obtain the option of changing or adding a specific layer, instead of reworking the entire application. It provides a model by which developers can create flexible and reusable applications.



The most general use of multi-tier architecture is the three-tier architecture. A three-tier architecture is typically composed of a presentation tier, an application tier, and a data storage tier and may execute on a separate processor.

Presentation Tier

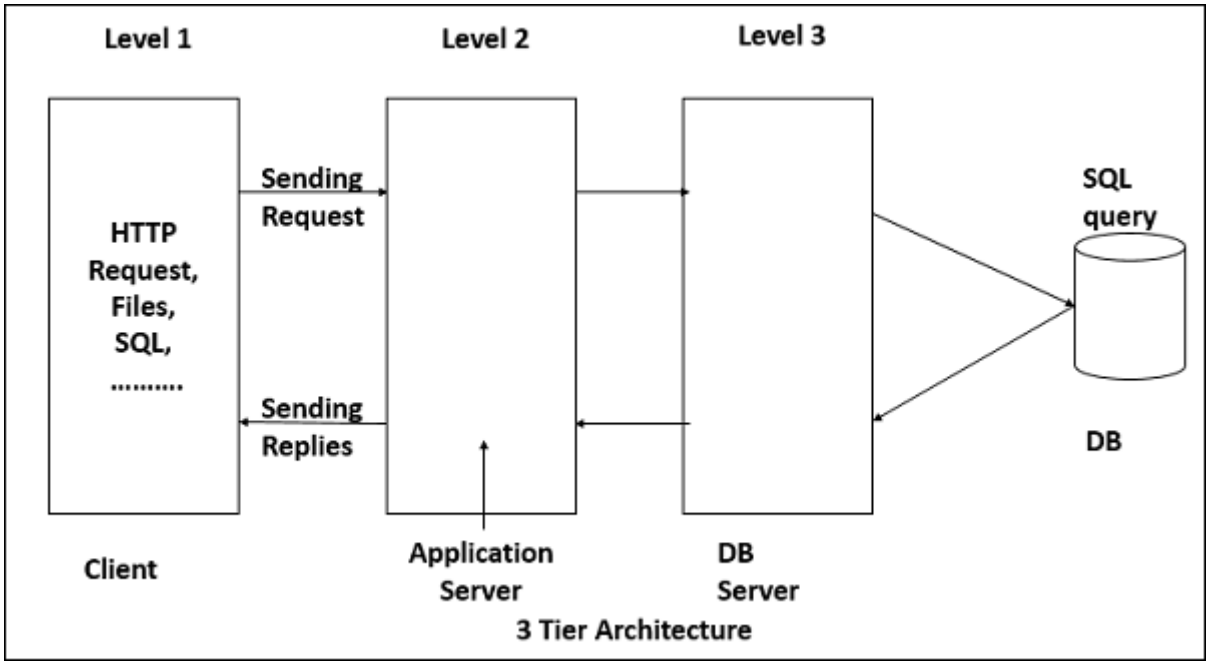
Presentation layer is the topmost level of the application by which users can access directly such as webpage or Operating System GUI (Graphical User interface). The primary function of this layer is to translate the tasks and results to something that user can understand. It communicates with other tiers so that it places the results to the browser/client tier and all other tiers in the network.

Application Tier (Business Logic, Logic Tier, or Middle Tier)

Application tier coordinates the application, processes the commands, makes logical decisions, evaluation, and performs calculations. It controls an application's functionality by performing detailed processing. It also moves and processes data between the two surrounding layers.

Data Tier

In this layer, information is stored and retrieved from the database or file system. The information is then passed back for processing and then back to the user. It includes the data persistence mechanisms (database servers, file shares, etc.) and provides API (Application Programming Interface) to the application tier which provides methods of managing the stored data.



Advantages

- Better performance than a thin-client approach and is simpler to manage than a thick-client approach.
- Enhances the reusability and scalability − as demands increase, extra servers can be added.
- Provides multi-threading support and also reduces network traffic.
- Provides maintainability and flexibility

Disadvantages

- Unsatisfactory Testability due to lack of testing tools.
- More critical server reliability and availability.

1.9.2 DISTRIBUTED COMPUTING SYSTEM MODELS

They are briefly described below.

Minicomputer Model : The minicomputer model is a simple extension of the centralized time sharing system.

A distributed computing system based on this model consists of a few minicomputers (they may be large supercomputers as well) interconnected by a communication network. Each minicomputer usually has multiple users simultaneously logged on to it. For this, several interactive terminals are connected to each minicomputer.

Each user is logged on to one specific minicomputer, with remote access to other minicomputers. The network allows a user to access remote resources that are available on some machine other than the one on to which the user is currently logged.

1. The minicomputer model may be used when resource sharing (Such as sharing of information databases of different types, with each type of database located on a different machine) with remote users is desired.
2. The early ARPAnet is an example of a distributed computing system based on the minicomputer model.

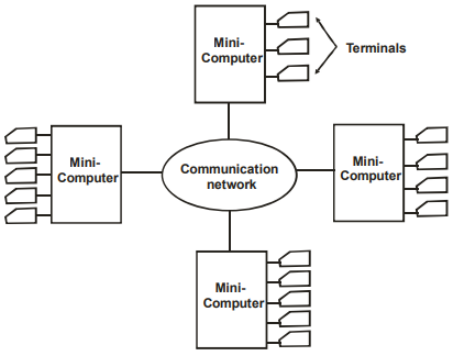


Fig. 1.2 : A distributed computing system based on the minicomputer model

1.9.3 Workstation Model

A distributed computing system based on the workstation model consists of several workstations interconnected by a communication network. A company's office or a university department may have several workstations scattered throughout a building or campus, each workstation equipped with its own disk and serving as a single-user computer.

1. It has been often found that in such an environment, at any one time (especially at night), a significant proportion of the workstations are idle (not being used), resulting in the waste of large amounts of CPU time. Therefore, the idea of the workstation model is to interconnect all these workstations by a high speed LAN so that idle workstations may be used to process jobs of users who are logged onto other workstations and do not have sufficient processing power at their own workstations to get their jobs processed efficiently.
2. In this model, a user logs onto one of the workstations called his or her "home" workstation and submits jobs for execution. When the system finds that the user's workstation does not have sufficient processing power for executing the processes of the submitted jobs efficiently, it transfers one or more of the process from the user's workstation to some other workstation that is currently idle and gets the process executed there, and finally the result of execution is returned to the user's workstation.

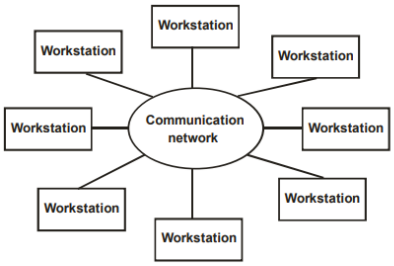


Fig. 1.3 : A distributed computing system based on the workstation model

1.9.4 Processor Pool Model

The processor – pool model is based on the observation that most of the time a user does not need any computing power but once in a while he or she may need a very large amount of computing power for a short time. (e.g., when recompiling a program consisting of a large number of files after changing a basic shared declaration). Therefore, unlike the workstation – server model in which a processor is allocated to each user, in the processor-pool model the processors are pooled together to be shared by the users as needed. The pool of processors consists of a large number of microcomputers and minicomputers attached to the network. Each processor in the pool has its own memory to load and run a system program or an application program of the distributed computing system.

In the pure processor-pool model, the processors in the pool have no terminals attached directly to them, and users access the system from terminals that are attached to the network via special devices. These terminals are either small diskless workstations or graphic terminals, such as X terminals. A special server (Called a run server) manages and allocates the processors in the

pool to different users on a demand basis. When a user submits a job for computation, an appropriate number of processors are temporarily assigned to his or her job by the run server. For example, if the user's computation job is the compilation of a program having n segments, in which each of the segments can be complied independently to produce separate relocatable object files, n processors from the pool can be allocated to this job to compile all the n segments in parallel. When the computation is completed, the processors are returned to the pool for use by other users.

1.9.5 Hybrid Model :

The workstation server model, is the most widely used model for building distributed computing systems. This is because a large number of computer users only perform simple interactive tasks such as editing jobs, sending electronic mails, and executing small programs. The workstation-server model is ideal for such simple usage. However, in a working environment that has groups of users who often perform jobs needing massive computation, the processor-pool model is more attractive and suitable.

To continue the advantages of both the workstation-server and processor-pool models, a hybrid model may be used to build a distributed computing system. The hybrid model is based on the workstation-server model but with the addition of a pool of processors. The processors in the pool can be allocated dynamically for computations that are too large for workstations or that requires several computers concurrently for efficient execution. In addition to efficient execution of computation-intensive jobs, the hybrid model gives guaranteed response to interactive jobs by allowing them to be processed on local workstations of the users. However, the hybrid model is more expensive to implement than the workstation – server model or the processor-pool model.

1.10 Distributed Computing Environment

DCE is client/server architecture, defined by the Open Software Foundation (OSF) that provides Open System platform to address the challenges of distributed computing.

DCE is middleware software sandwiched between the application and the operating system layer.

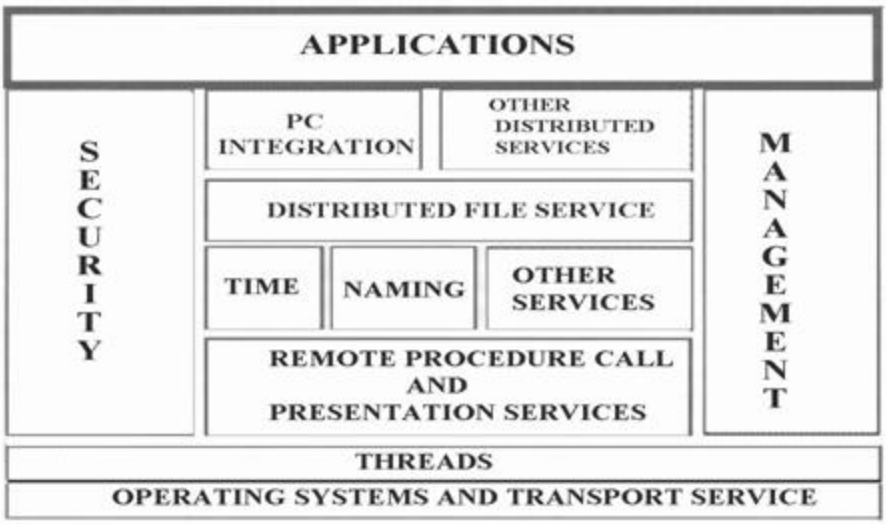
Goals of DCE:

- It should run on different computers, operating systems and networks in a distributed system.
- It should provide a coherent seamless platform for distributed applications.
- It should provide a mechanism for clock synchronization on different machines.
- It should provide tools which make it easier to write distributed applications where multiple users at multiple locations can work together.
- Provide extensive tools for authentication authorization.

1.10.1 DCE architecture

- DCE cell:

The DCE system is highly scalable.To accommodate such large systems DCE uses the concept of cells. A cell is the basic unit of operation in the DCE which breaks down a large system into smaller, manageable units.



- DCE Components: DCE uses various technologies that form the components such as:

1.Thread package

- It is a package that provides a programming model for building concurrent applications. It is a collection of user-level library procedures that supports the creation, management, and synchronization of multiple threads.
- It was designed to minimize the impact on the existing software that could convert single threaded into multithreaded one.
- Threads facility is not provided by many operating systems and DCE components require threads, hence threads package is included in DCE.

2.Remote procedure call

- A procedure call is a method of implementing the Client/Server Communication.
- The procedure call is translated into network communications by the underlying RPC mechanism.
- In DCE RPC, one or more DCE RPC interfaces are defined using the DCE interface definition language (IDL). Each interface comprises a set of associated RPC calls each with their input and output parameters.
- RPC hides communication detail and removes system and hardware dependencies. It can automatically handle data type conversions between the client and the server without considering whether they run on the same or different architecture, or have same or different byte ordering.

3.Name service

The Name service of DCE include

a. Cell Directory Service (CDS):The CDS manages a database of information about the resources in a group of machines called a DCE cell.

b. Global Directory Service (GDS):The Global Directory Service implements an international, standard directory service and provides a global namespace that connects the local DCE cells into one worldwide hierarchy.

c. Global Directory Agent (GDA): The GDA acts as a go-between for cell and global directory services.

4.Distributed file service

- Distributed File service is a worldwide distributed file system which allows users to access and share files stored on a file server anywhere on the network without knowing the physical location of the file.
- It provides characteristics such as high performance, high availability and location transparency and is able to perform tasks like replicating data; log file system data, quick recovery during crash etc.

5.Time service

- The DCE Time Service (DTS) provides synchronized time on the computers.
- It enables distributed applications on different computers to determine event sequencing, duration, and scheduling participating in a Distributed Computing Environment.
- DTS also provides services which return a time range to an application, which compare time ranges from different machines.

6.Scheduling and synchronization

- Scheduling determines how long a thread runs and which thread will run next.
- It uses three algorithms namely FIFO, Round Robin, or the Default algorithm.
- Synchronization prevents multiple threads from accessing the same resource at the same time.

7.Security service

- There are three aspects to DCE security which provide resource protection within a system against illegitimate access. They are:
- Authentication: This identifies that a DCE user or service is allowed to use a particular service.
- Secure communications: Communication over the network can be checked for tampering or encrypted for privacy.
- Authorization: The permission to access the service is given after authorization.
- These are services are provided using Registry Service, Privilege Service, Access Control List (ACL) Facility, and Login Facility etc.

Advantages of DCE

1. Supports portability and interoperability
2. Supports distributed file service

1.11 Theoretical issues in distributed systems

1.11.1 CLOCKS, EVENTS AND PROCESS STATES

Each process executes on a single processor, and the processors do not share memory (Chapter 6 briefly considered the case of processes that share memory). Each process *pi* in has a state *si* that, in general, it transforms as it executes. The process's state includes the values of all the variables within it. Its state may also include the values of any objects in its local operating system environment that it affects, such as files. We assume that processes cannot communicate with one another in any way except by sending messages through the network.

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

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

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

. In other words, when the real time in an absolute frame of reference is *t*, *Cit* is the reading on the software clock. For example,
Cit could be the 64-bit value of the number of nanoseconds that have elapsed at time *t* since a convenient reference time. In general, the clock is not completely accurate, so *Cit* will differ from

t. Nonetheless, if *Ci* behaves sufficiently well (we shall examine the notion of clock correctness shortly), we can use its value to timestamp any event at *pi* . Note that successive events will correspond to different timestamps only if the *clock resolution* – the period between updates of the clock value – is smaller than the time interval between successive events. The rate at which events occur depends on such factors as the length of the processor instruction cycle.

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

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

Coordinated Universal Time – abbreviated as UTC (from the French equivalent) – is an international standard for timekeeping. It is based on atomic time, but a so-called 'leap second' is inserted – or, more rarely, deleted – occasionally to keep it in step with astronomical time. UTC signals are synchronized and broadcast regularly from landbased
radio stations and satellites covering many parts of the world. For example, in the USA, the radio station WWV broadcasts time signals on several shortwave frequencies.
Satellite sources include the *Global Positioning System* (GPS).Receivers are available commercially. Compared with 'perfect' UTC, the signals received from land-based stations have an accuracy on the order of 0.1–10 milliseconds, depending on the station used. Signals received from GPS satellites are accurate to about 1 microsecond. Computers with receivers attached can synchronize their clocks with these timing signals.

Synchronizing physical clocks
In order to know at what time of day events occur at the processes in our distributed system – for example, for accountancy purposes – it is necessary to synchronize the processes' clocks, *Ci* , with an authoritative, external source of

time. This is *external synchronization*. And if the clocks *Ci* are synchronized with one another to a known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks, even though they are not necessarily synchronized to an external source of time. This is *internal synchronization*.We define these two modes of synchronization more closely as follows, over an interval of real time *I*:

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

Internal synchronization: For a synchronization bound *D* 0 , *Cit – Cjt D* for *i j* = 1 2*N* , and for all real times *t* in *I*. Another way of saying this is that he clocks *Ci agree* within the bound *D*. Clocks that are internally synchronized are not necessarily externally synchronized, since they may drift collectively from an external source of time even though they agree with one another. However, it follows from the definitions that if the system is externally synchronized with a bound *D* then the same system is internally synchronized with a bound of 2*D*.

Various notions of *correctness* for clocks have been suggested. It is common to define a hardware clock *H* to be correct if its drift rate falls within a known bound (a value derived from one supplied by the manufacturer, such as 10–6 seconds/second).

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

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

A clock that does not keep to whatever correctness conditions apply is defined to be *faulty*. A clock's

crash failure is said to occur when the clock stops ticking altogether;
any other clock failure is an *arbitrary failure*. A historical example of an arbitrary failure is that of a clock with the 'Y2K bug', which broke the monotonicity condition by registering the date after 31 December 1999 as 1 January 1900 instead of 2000; another example is a clock whose batteries are very low and whose drift rate suddenly becomes very large.

Clocks, Events and Process States

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

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

• **Actions** of processes*: Send, Receive,* change own state
• ***Event****:* the occurrence of a single action that a process carries out as it executes
– Events at a single process *pi*, can be placed in a total **ordering** denoted by the relation →*i*
between the events. i.e.***e*** **→*i*** ***e'*** if and only if event *e* occurs before event *e'* at process *pi*
• A history of process pi: is a series of events ordered by →*i*
history**(pi)** = hi =<e*i*0, *ei*1, *ei*2, …>

– Clocks

To timestamp events, use the computer's clock • At **real time, *t***, the OS reads the time on the computer's **hardware clock *Hi*(*t*)**
• It calculates the time on its **software clock *Ci*(*t*)**=α***Hi*(*t*)** + β

– e.g. a 64 bit value giving nanoseconds since some base time
– **Clock resolution**: period between updates of the clock value

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

Skew between computer clocks in a distributed system
Computer clocks are not generally in perfect agreement
• ***Clock skew***: the difference between the times on two clocks (at any instant)
• Computer clocks use crystal-based clocks that are subject to physical variations
– ***Clock drift***: they count time at different rates and so diverge (frequencies of oscillation differ)
– ***Clock drift rate***: the difference per unit of time from some ideal reference clock
– Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10-6 secs/sec).
High precision quartz clocks drift rate is about 10-7 or 10-8 secs/sec

1.11.2 The Berkeley algorithm

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

1.11.3 Network Time Protocol (NTP)

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

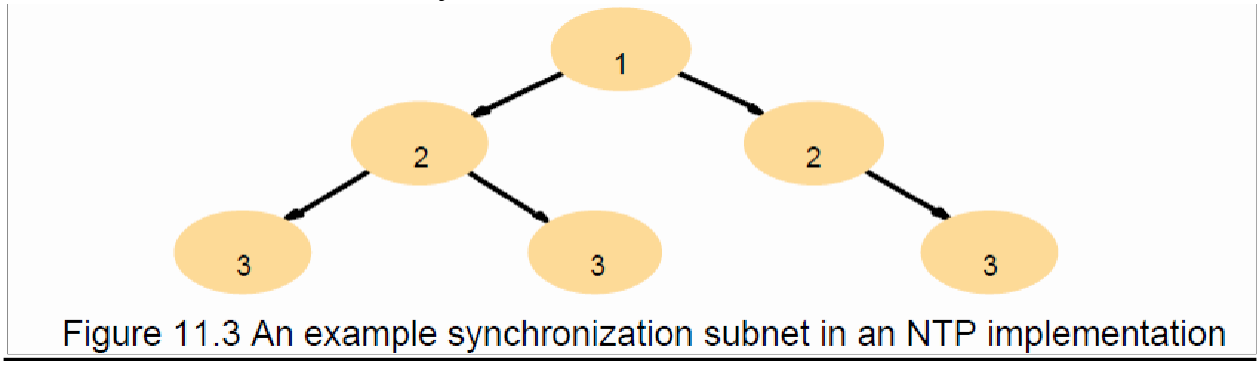


Figure 11.3 An example synchronization subnet in an NTP implementation

NTP - synchronization of servers

- The synchronization subnet can reconfigure if failuresoccur
- a primary that loses its UTC source can become a secondary
- a secondary that loses its primary can use another primary

- Modes of synchronization for NTP servers:
- Multicast
- A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not veryaccurate)
- Procedure call
- A server accepts requests from other computers (like Cristian‘s algorithm)