DISTRIBUTED SYSTEMs

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

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

Centralized 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.

Common Characteristics

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
  - accommodate 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

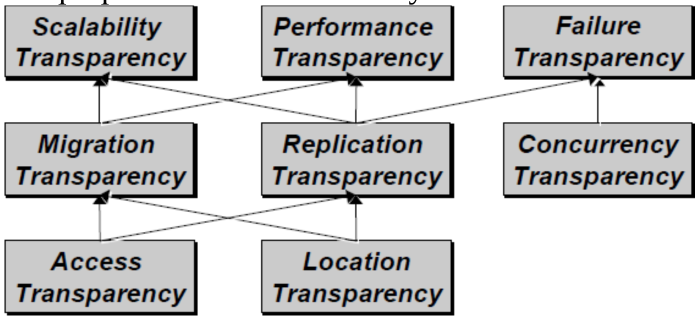
Fault Tolerance
Hardware, software and networks fail!

- ▪ 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
This 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
This 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
This 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

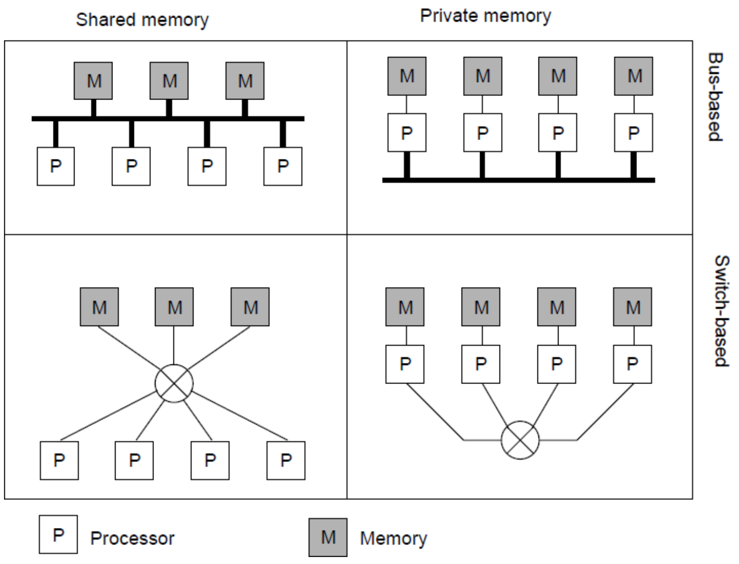
Migration Transparency
This allows the movement of information objects within a system without affecting the operations of users or application programs
• Example: NFS
• Example: Web Pages

Performance Transparency
This allows the system to be reconfigured to improve performance as loads vary.
• Example: Distributed make.

Scaling Transparency
This allows the system and applications to expand in scale without change to the system structure or the application algorithms.
• Example: World-Wide-Web
• Example: Distributed Database

Distributed Systems Hardware Concepts of the following are considered**.**
• Multiprocessors
• Multi-computers Networks of Computers

Multiprocessors and Multi-computers Distinguishing features includes:
• Private versus shared memory
• Bus versus switched interconnection



Networks of Computers

High degree of node heterogeneity features:

• High-performance parallel systems (multiprocessors as well as multi-computers)

• 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

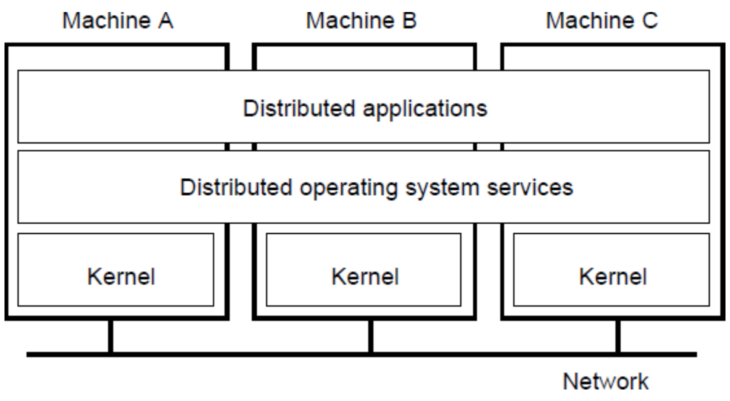
Distributed operating system

_ Network operating system

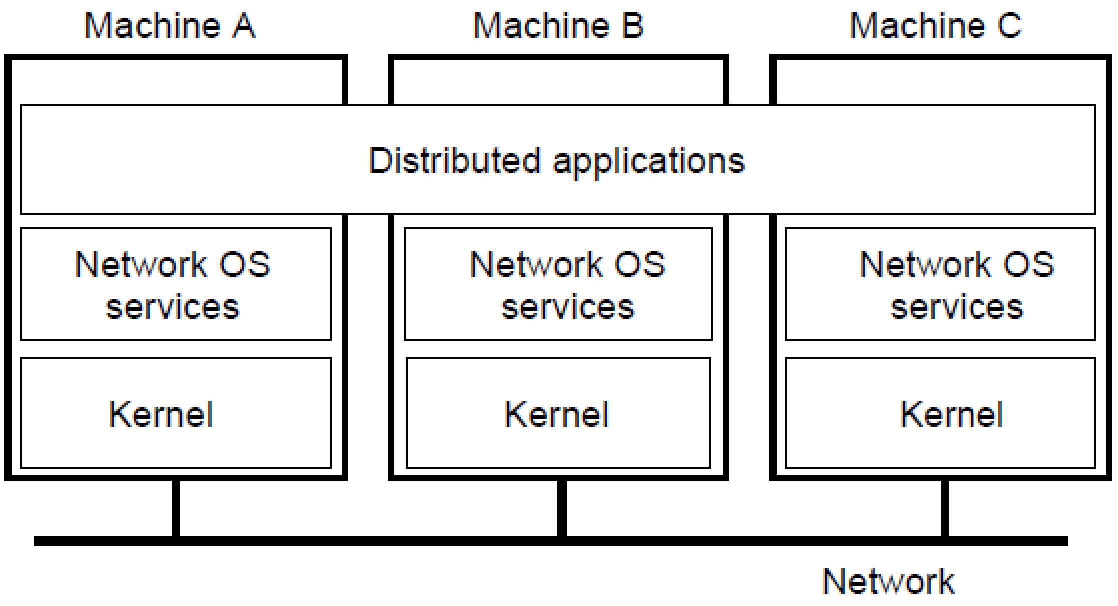
_ Middleware

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

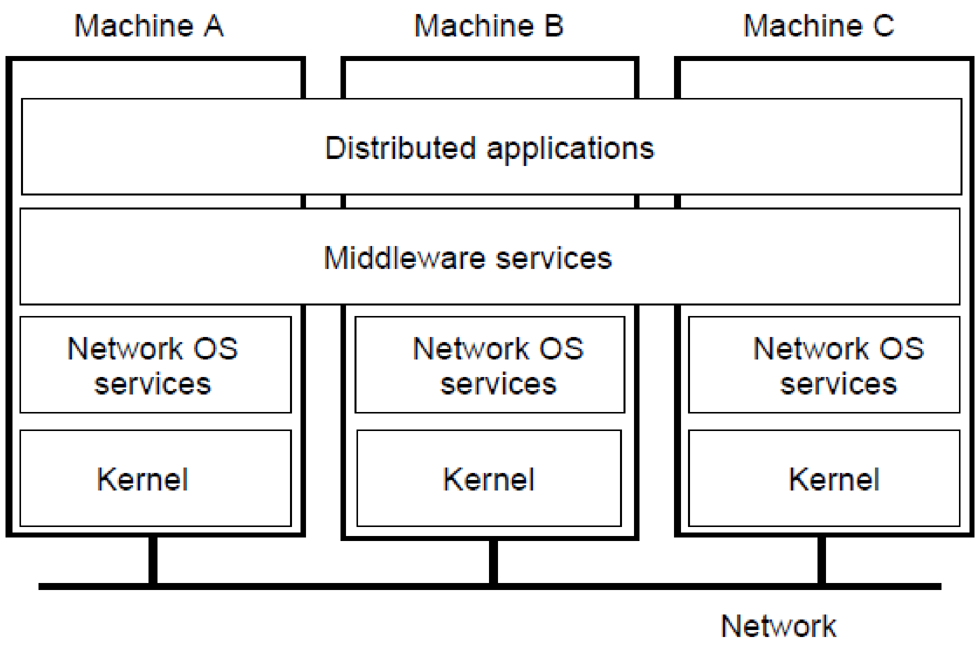
Some characteristics of Distributed Operating System:
_ OS on each computer knows about the other computers
_ OS on different computers generally the same
_ Services are generally (transparently) distributed across computers



Some characteristics of Network Operating Systems:
_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)
_They have a highly file oriented (basically, processors share *only* files)



Some characteristics of Distributed System (Middleware):
_ 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

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*.

Distributed systems have 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.

TRENDS IN DISTRIBUTED SYSTEMS

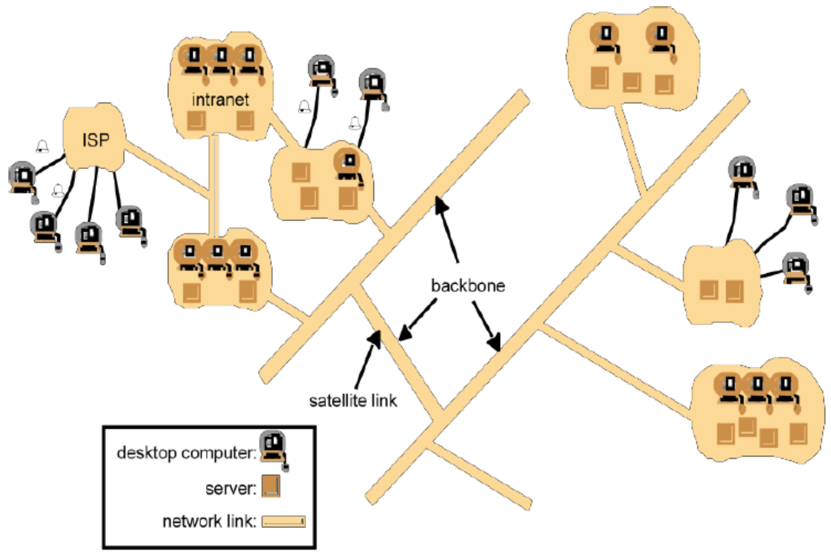
Distributed systems are undergoing a period of significant change and this can be traced back to a number of influential trends:

- 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 – sub networks 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

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.

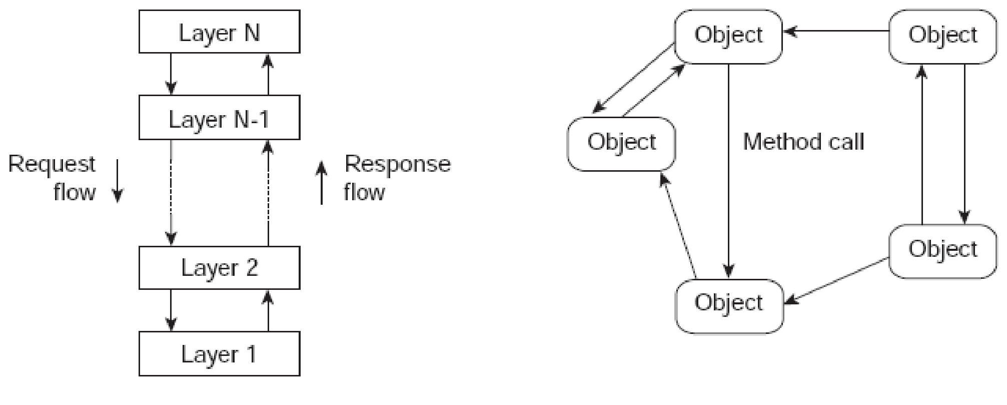
Heterogeneity and mobile code • The term *mobile code* is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable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

DISTRIBUTED SYSTEMS ARCHITECTURE

Architecture Styles

- Formulated in terms of components, and the way they are connected:
  - A **component** is a modular unit with well-defined interfaces; replaceable; reusable
  - A **connector** is a communication link between modules; mediates coordination or cooperation among components
- Four styles that are most important:
  1. Layered architecture
  2. Object-based architecture
  3. Data-centered architecture -- processes communicate through a common repository (passive or active).
  4. Event-based architecture -- processes communicate through the propagation of events

Organize into logically different components, and distribute those components over the various machines.

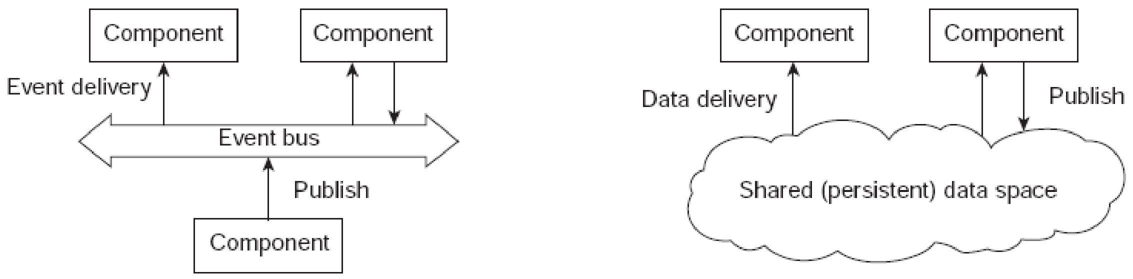


(a) (b)

- (a) Layered style is used for client-server system

  (b) Object-based style for distributed object systems.

  - less structured
  - component = object
  - connector = RPC or RMI
- Decoupling processes in space (anonymous) and also time (asynchronous) has led to alternative styles.



(a) (b)

(a) Publish/subscribe [decoupled in space] (event-based architecture)

Event-based architecture supports several communication styles:

- Publish-subscribe
- Broadcast
- Point-to-point

➢ Decouples sender & receiver; asynchronous communication

➢ Event-driven architecture (EDA) promotes the production, detection, consumption of, and reaction to events.

➢ An event can be defined as "a significant change in state". For example, when a consumer purchases a car, the car's state changes from "for sale" to "sold". A car dealer's system architecture may treat this state change as an event to be produced, published, detected and consumed by various applications within the architecture.

The main advantage of this architecture is that they are **loosely coupled**; they need not explicitly refer to each other. For example, if we have an alarm system that records information when the front door opens, the door itself doesn't know that the alarm system will add information when the door opens, just that the door has been opened.

(b) Shared data-space [decoupled in space and time] (data-centered + event-based)

Access and update of data store is the main purpose of data-centered system. Processes communicate/exchange info primarily by reading and modifying data in some shared repository (e.g database, distributed file system)

e.g. processes of many web-based distributed systems communicate through the use of shared Web-based data services.

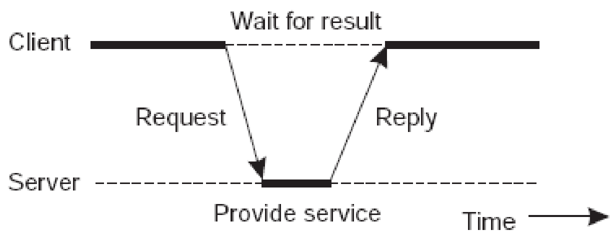
Combination of data-centered and event based architectures: This allows the Processes to communicate asynchronously.

Centralized Architectures

Basic Client Server Model
Characteristics:

- there are processes offering services (servers)
- there are processes that use services (clients)
- Clients and servers can be on different machines
- Clients follow request/reply model when using services
- Synchronous communication: request-reply protocol
- In LANs, often implemented with a connectionless protocol (unreliable)
- In WANs, communication is typically connection-oriented TCP/IP (reliable)

- High likelihood of communication failures



An operation is said to be **idempotent** if it can be repeated multiple times without harm.

Idempotency

- ✓ Typical response to lost request in connectionless communication: re-transmission
- ✓ Consider effect of re-sending a message such as "Increment X by 1000" If first message was acted on, now the operation has been performed twice
- ✓ Idempotent operations: can be performed multiple times without harm e.g., "Return current value of X"; check on availability of a product
- ✓ Non-idempotent: "increment X", order a product

<u>Application Layering</u>

Traditional three-layered view

- User-interface layer contains units for an application's user interface
- Processing layer contains the functions of an application, i.e. without specific data
- Data layer contains the data that a client wants to manipulate through the application components
  - o Data usually is persistent; exists even if no client is accessing it
  - o File or database system

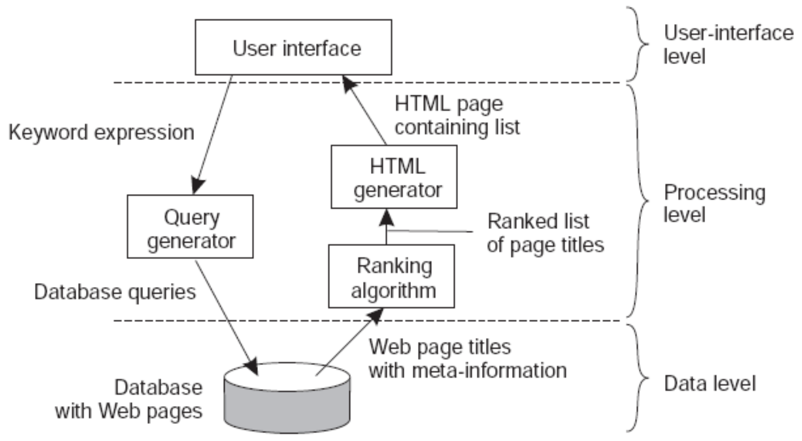
Observation

This layering is found in many distributed information systems, using traditional database technology and accompanying applications.

Examples

Web search engine

Interface: type in a keyword string
Processing level: processes to generate DB queries, rank replies, format response
Data level: database of web pages



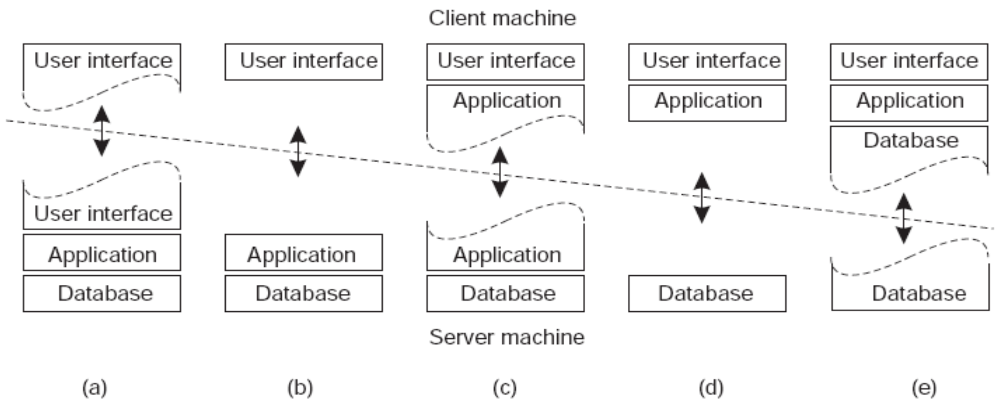
- ▪ Stock broker's decision support system
  - o Interface: likely more complex than simple search
    Processing: programs to analyze data; rely on statistics, AI perhaps, may require large simulations
    Data level: DB of financial information
- ▪ Desktop "office suites"

Interface: access to various documents, data,
Processing: word processing, database queries, spreadsheets,.
Data: file systems and/or databases

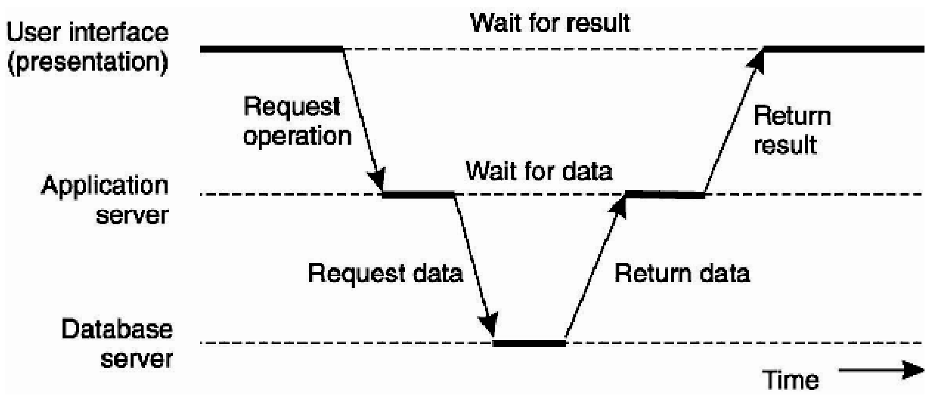
Multi-Tiered Architectures

- Single-tiered: dumb terminal/mainframe configuration
- Two-tiered: client/single server configuration
- Three-tiered: each layer on separate machine

Traditional two-tiered configurations:



A server may act as a client, leading to three-**tier architecture**:



Decentralized Architectures

In the last couple of years we have been seeing a tremendous growth in peer-to-peer systems

- Structured P2P: nodes are organized following a specific distributed data structure
- Unstructured P2P: nodes have randomly selected neighbors
- Hybrid P2P: some nodes are appointed special functions in a well-organized fashion

Note

In virtually all cases, we are dealing with overlay networks: data are routed over connections setup between the nodes (application-level multicasting)

Peer to Peer

- Nodes act as both client and server; interaction is symmetric
- Each node acts as a server for part of the total system data
- Overlay networks connect nodes in the P2P system:

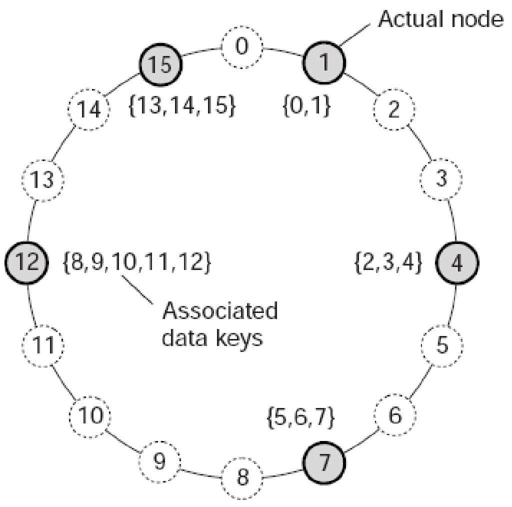
- Use their own addressing system for storing and retrieving data in the system
- Nodes can route requests to locations that may not be known by the requester.

Overlay Network

Structured P2P Systems

Basic idea

- Organize the nodes in a structured overlay network such as a logical ring, and make specific nodes.
- Responsible for services based only on their ID.
- A common approach is to use a distributed hash table (DHT) to organize the nodes.
- Traditional hash functions convert a key to a hash value, which can be used as an index into a hash table:
- Keys are unique -- each represents an object to store in the table;
- The hash function value is used to insert an object in the hash table and to retrieve it.
- In a DHT, data objects and nodes are each assigned a key which hashes to a random number from a very large identifier space (to ensure uniqueness).
- A mapping function assigns objects to nodes, based on the hash function value.
- A lookup, also based on hash function value, returns the network address of the node that stores the requested object.

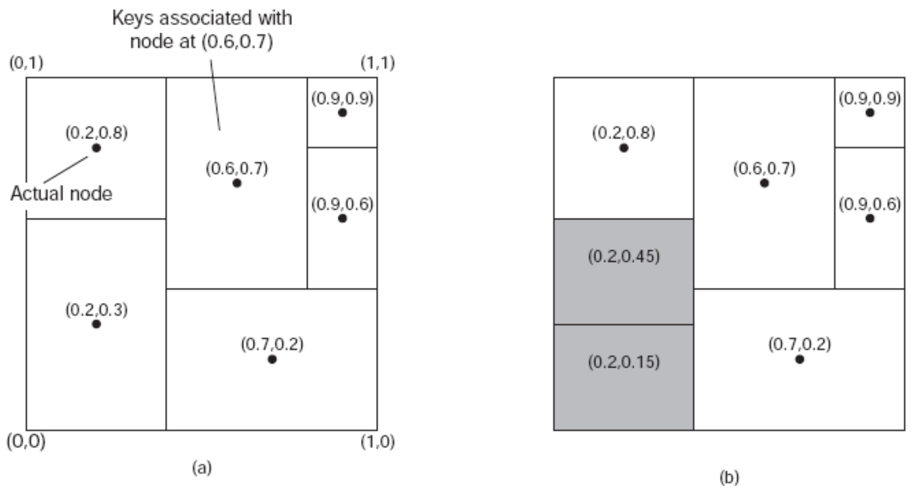


Note
The system provides an operation LOOKUP(key) that will efficiently route the lookup request to the associated node.

Achieved by organizing processes through a **distributed hash table (DHT)**

e.g. a data item with key k is mapped to the node with the smallest idenifier *id* ≥ *k*. This node is referred to as the successor of the datum with key k, and is denoted as *succ(k)*.

Other example
Organize nodes in a d-dimensional space and let every node take the responsibility for data in a specific region. When a node joins => split a
region. (e.g. d = 2 in the following figure)



a) **Mapping of data onto nodes**

- ✓ 2D space [0,1] x [0,1] is divided among 6 nodes
- ✓ Each node has an associated region
- ✓ Every data item in the newt-work (Content Addressable Network (CAN)) will be assigned a unique point in space;
  the point can be randomly picked
- ✓ A node is responsible for all data elements mapped to its region

b) Splitting a region when a node joins

- ➢ To add a new region, split the region
- ➢ To remove an existing region, neighbor will take over

<u>Unstructured P2P Systems</u>

- Rely on randomized algorithms for constructing an overlay network.
- Many unstructured P2P systems attempt to maintain a random graph.

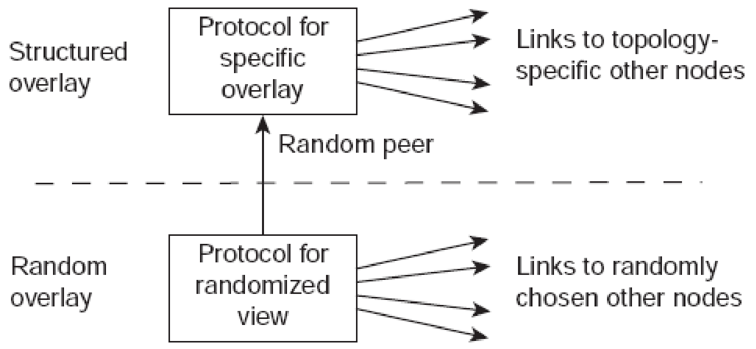
Basic principle
each node is required to contact a randomly selected other node:

- Let each peer maintain a partial view of the network, consisting of *n* other nodes
- Each node P periodically selects a node Q from its partial view
- P and Q exchange information and exchange members from their respective partial views

Note
It turns out that, depending on the exchange, randomness,
robustness of the network can be maintained.

Topology Management of Overlay Networks

- Distinguish two layers:
  (1) maintain random partial views in lower layer (unstructured peer-to-peer system);
  (2) be selective on who you keep in higher-layer partial view.


Structured overlay
Protocol for specific overlay
Links to topology-specific other nodes
Random peer
Random overlay
Protocol for randomized view
Links to randomly chosen other nodes

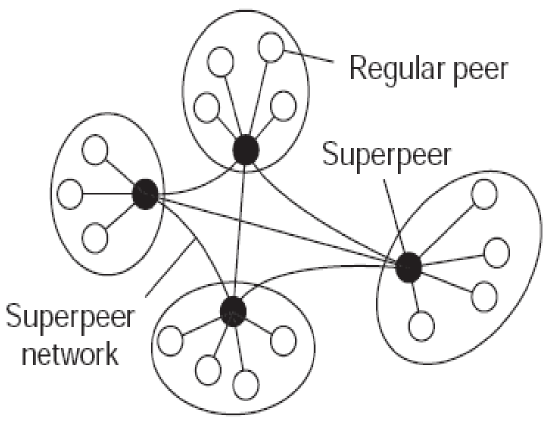
Note

Lower layer feeds upper layer the partial views; additional selection of entries takes place in upper layer, leading to a second list of neighbors.

<u>Super-peers</u>

Sometimes it helps to select a few nodes to do specific work:

Super-peer



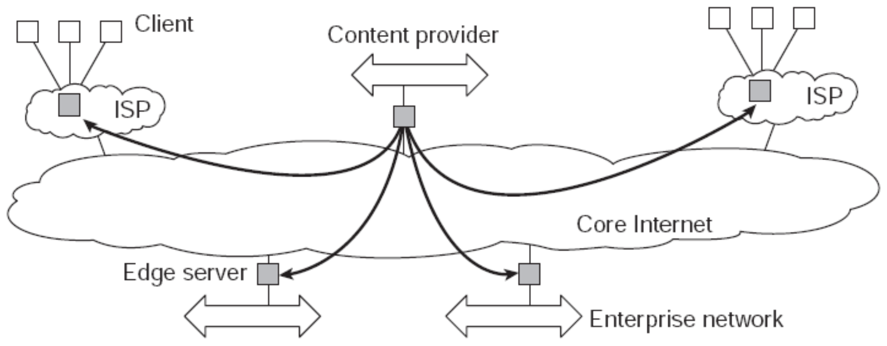
Examples

- Peers maintaining an index (for search)
- Peers monitoring the state of the network
- Peers being able to setup connections

Hybrid Architectures: Client-server combined with P2P

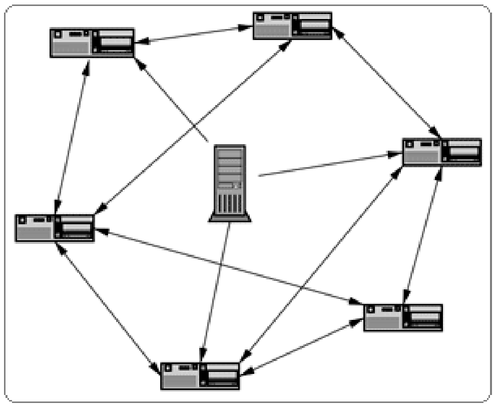
Example

Edge-server architectures, which are often used for Content Delivery Networks

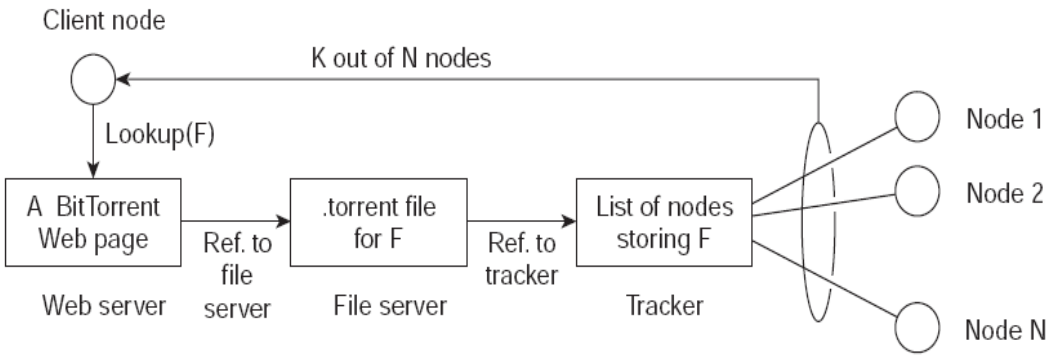


Hybrid Architectures: C/S with P2P : BitTorrent

Users cooperate in file distribution



Once a node has identified where to download a file from, it joins a swarm of downloaders who in parallel get file chunks from the source, but also distribute these chunks amongst each other.



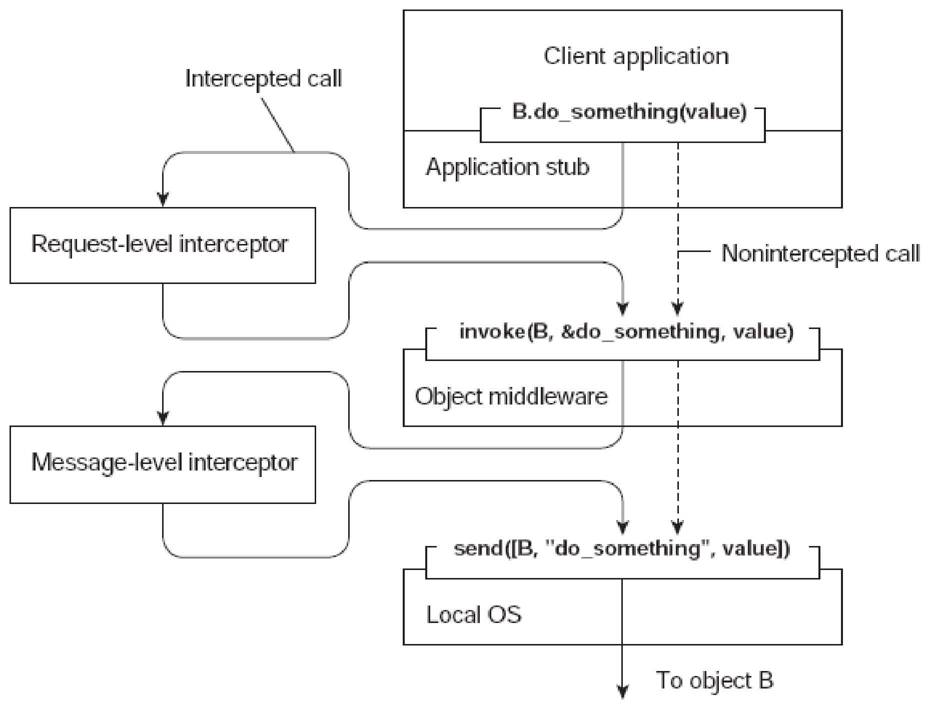
Architectures versus Middleware

Problem

In many cases, distributed systems/applications are developed according to a specific architectural style. The chosen style may not be optimal in all cases) need to (dynamically) adapt the behavior of the middleware.

Interceptors

intercept the usual flow of control when invoking a remote object.



What interceptors actually offer is a means to adapt the middleware.

Adaptive Middleware

What interceptors actually offer is a means to adapt the middleware.

Adaptive Middleware

Three basic techniques of adaptive software:

- **Separation of concerns**: Try to separate extra functionalities (e.g. reliability, performance, security etc) and later weave them together into a single implementation. Not easy.
- **Computational reflection**: Let a program inspect itself at runtime and adapt/change its settings dynamically if necessary => mostly at language level and applicability unclear.
- **Component-based design**: Organize a distributed application through components that can be dynamically replaced when needed =>highly complex, also many inter-component dependencies.

Middleware for distributed systems are usually bulky and complex and Adaptive software makes systems complex

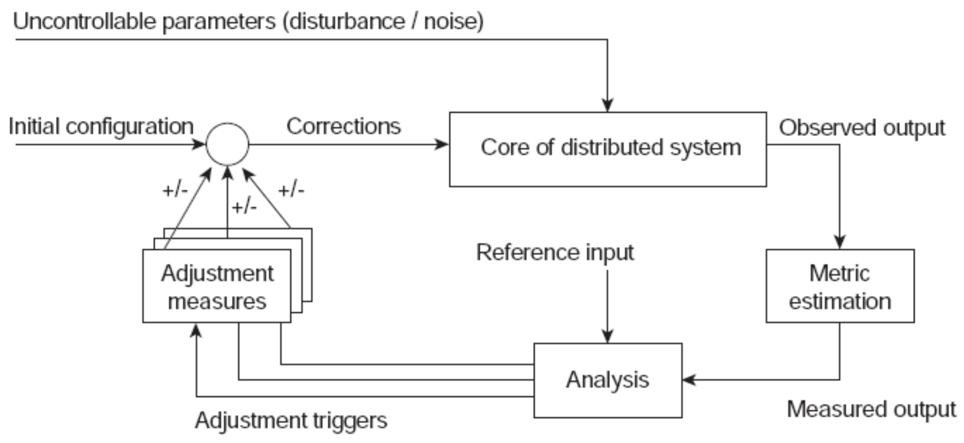
Fundamental question -- Do we need adaptive software at all, or is it the issue of adaptive systems?

The strongest argument for supporting adaptive software is that many distributed systems cannot afford to have downtime. This requires components be updated, added, deleted, or replaced on the fly.

Self-managing Distributed Systems
Distinction between system and software architectures blurs when automatic adaptivity needs to be taken into account:

- Self-configuration
- Self-managing
- Self-healing
- Self-optimizing
- Self-* (autonomic computing)

Feedback Control Model
In many cases, self-* systems are organized as a *feedback control system*.



Processes Threads

We build **virtual processors** in software, on top of physical processors:

Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.

***Thread**:* A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.

***Process**:* A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread

Context Switching:

Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).

Thread context and **Process context**

Notes:

1. Threads share the same address space. Thread context switching can be done entirely independent of the operating system.
2. Process switching is generally more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.
3. Creating and destroying threads is much cheaper than doing so for processes.

Threads and Distributed Systems

Improve performance

- Starting a thread is much cheaper than starting a new process.
- Having a single-threaded server prohibits simple scale-up to a **multiprocessor system**.
- As with clients: **hide network latency** by reacting to next request while previous one is being replied.

Better structure

- Most servers have high I/O demands. Using simple, **well-understood blocking calls** simplifies the overall structure.
- Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control.

Virtualization

Virtualization is becoming increasingly important:

- Hardware changes faster than software
- Ease of portability and code migration
- Isolation of failing or attacked components

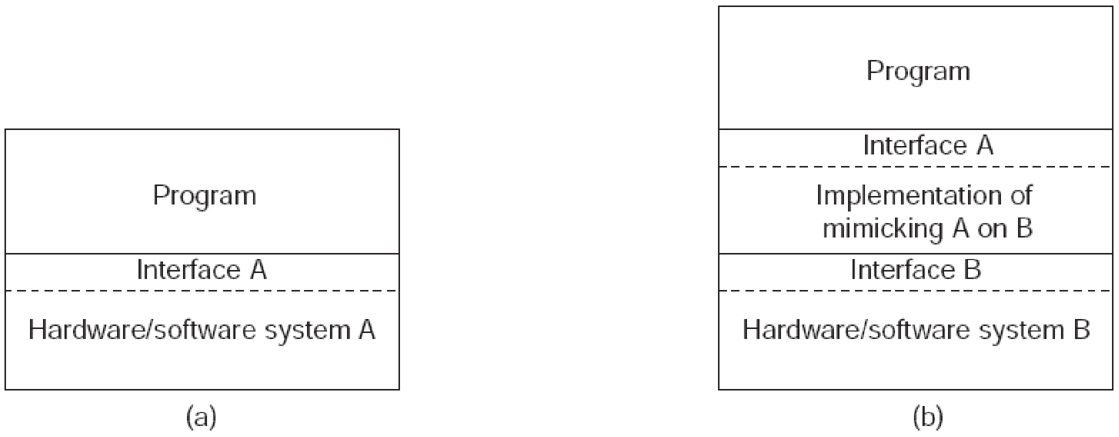

Program
Interface A
Hardware/software system A
(a)
Program
Interface A
Implementation of mimicking A on B
Interface B
Hardware/software system B
(b)

Figure:

(a) General organization between a program, interface and system.
(b)General organization of virtualizing system A on top of system B.

Architecture of VMs
Virtualization can take place at very different levels, strongly depending on the **interfaces** as offered by various systems components:

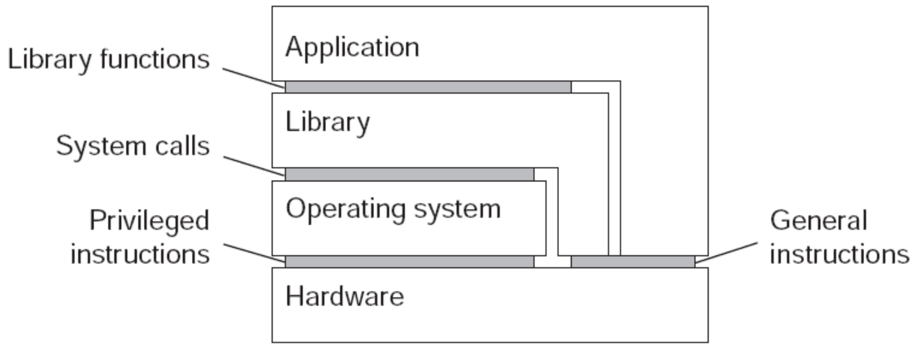


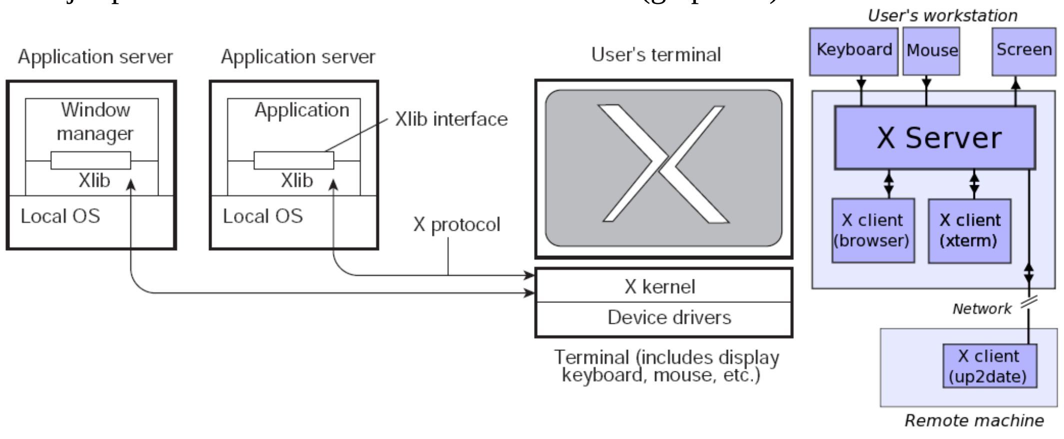
Figure: Various interfaces offered by computer systems.

- **Process VM**: A program is compiled to intermediate (portable) code, which is then executed by a runtime system (Example: Java VM).
- **VM Monitor**: A separate software layer mimics the instruction set of hardware => a complete operating system and its applications can be supported (Example: VMware, Virtual Box).

VM Monitors on operating systems
we are seeing VMMs run on top of existing operating systems.

- ❖ Perform binary translation: while executing an application or operating system, translate instructions to that of the underlying machine.
- ❖ Distinguish sensitive instructions: traps to the original kernel (think of **system calls**, or **privileged instructions**).
- ❖ Sensitive instructions are replaced with calls to the VMM.

Clients: User Interfaces
A major part of client-side software is focused on (graphical) user interfaces.

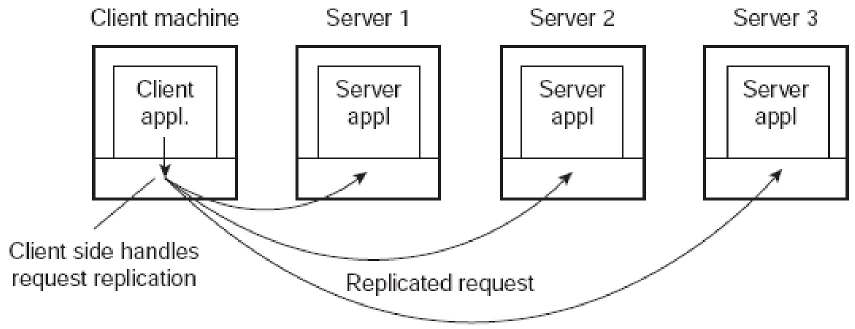


X was specifically designed to be used over network connections.
X kernel (display machine) is the server; remote application is the client.

Client-Side Software

Generally tailored for distribution transparency, A client should not know that it is communicating via network or not Distribution is often less transparent to servers than to clients

- **Access transparency**: handled through client-side stubs for RPCs: provides same interface as at the server, hides different machine architectures
- **Location/migration transparency:** let client-side software keep track of actual location (if server changes location: client rebinds to the server if necessary). Hide server locations from user.
- **Replication transparency:** multiple invocations handled by client stub:



Client-side software can collect all responses but passes only one to the client application.

- **Failure transparency:** mask server and communication failures: done through client middleware, e.g. connect to another machine. Middleware may return cached data (e.g. some Web browsers)

Servers: General organization

Basic model
A server is a process that waits for incoming service requests at a specific transport address. In practice, there is a one-to-one mapping between a port and a service.

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
	24	any private mail system
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol
sunrpc	111	SUN RPC (portmapper)
courier	530	Xerox RPC

Type of servers:
- **Iterative servers:** handles a request itself; can handle only one client at a time,
- **Concurrent servers:** Does not handle a request itself; pass it to a separate thread or another process
- **Super servers:** Servers that listen to several ports, i.e., provide several independent services. In practice, when a service request comes in, they start a sub-process to handle the request (UNIX)

Out-of-band communication
Issue
Is it possible to interrupt a server once it has accepted (or is in the process of accepting) a service request?

One approach is that the user (client) exit the application; server will tear down the connection

Solution 1
Use a separate port for urgent data:
- Server has a separate thread/process for urgent messages
- Urgent message comes in => associated request is put on hold
- Note: we require OS supports priority-based scheduling

Solution 2
Use out-of-band communication facilities of the transport layer:

- Example: TCP allows for urgent messages in same connection
- Urgent messages can be caught using OS signaling techniques

Servers and state

Stateless servers
Never keep accurate information about the status of a client after having handled a request:

- Don't record whether a file has been opened (simply close it again after access)
- Don't promise to invalidate a client's cache
- Don't keep track of your clients

Consequences

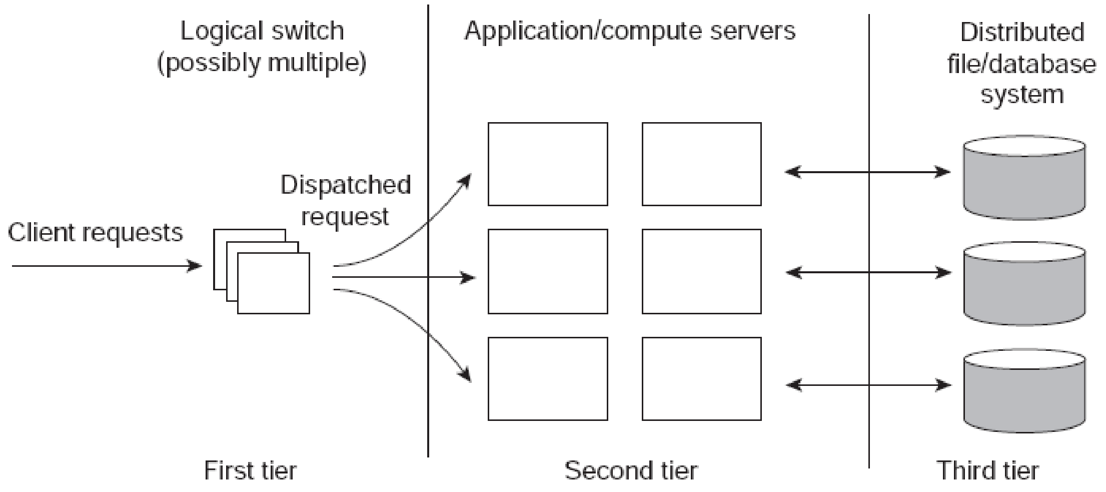
- Clients and servers are completely independent
- State inconsistencies due to client or server crashes are reduced
- Possible loss of performance because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

Stateful servers
Keeps track of the status of its clients:

- Record that a file has been opened, so that prefetching can be done
- Knows which data a client has cached, and allows clients to keep local copies of shared data

Observation
The performance of stateful servers can be extremely high, provided clients are allowed to keep local copies. As it turns out, reliability is not a major problem.

<u>Server clusters: three different tiers</u>


Logical switch (possibly multiple)
Application/compute servers
Distributed file/database system
Dispatched request
Client requests
First tier
Second tier
Third tier

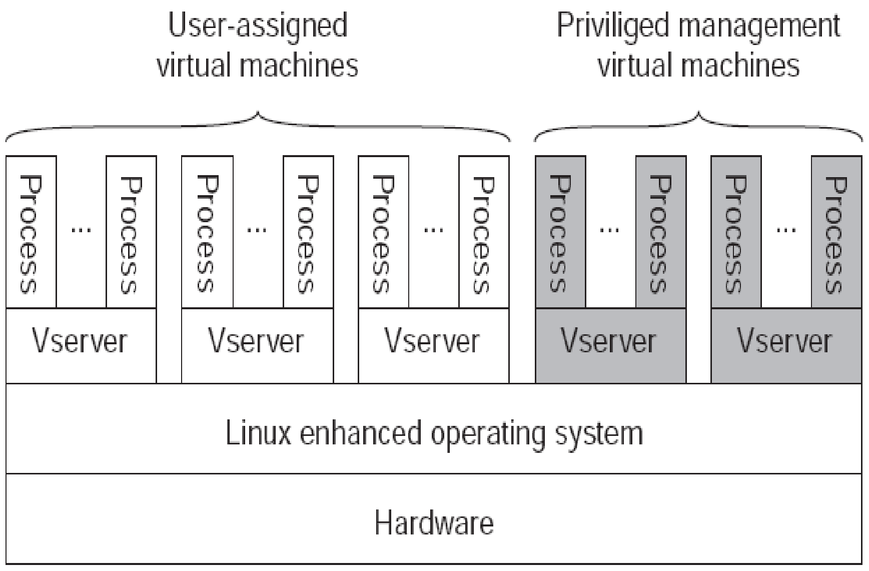
Example: Planet Lab

Essence

Different organizations contribute machines, which they subsequently **share** for various experiments.

Problem

we need to ensure that different distributed applications do not get into each other's way => **virtualization**


User-assigned virtual machines
Priviliged management virtual machines
Process
...
Process
Vserver
Linux enhanced operating system
Hardware

Vserver:

Independent and protected environment with its own libraries, server versions, and so on. Distributed applications are assigned a collection of vservers distributed across multiple machines (slice).

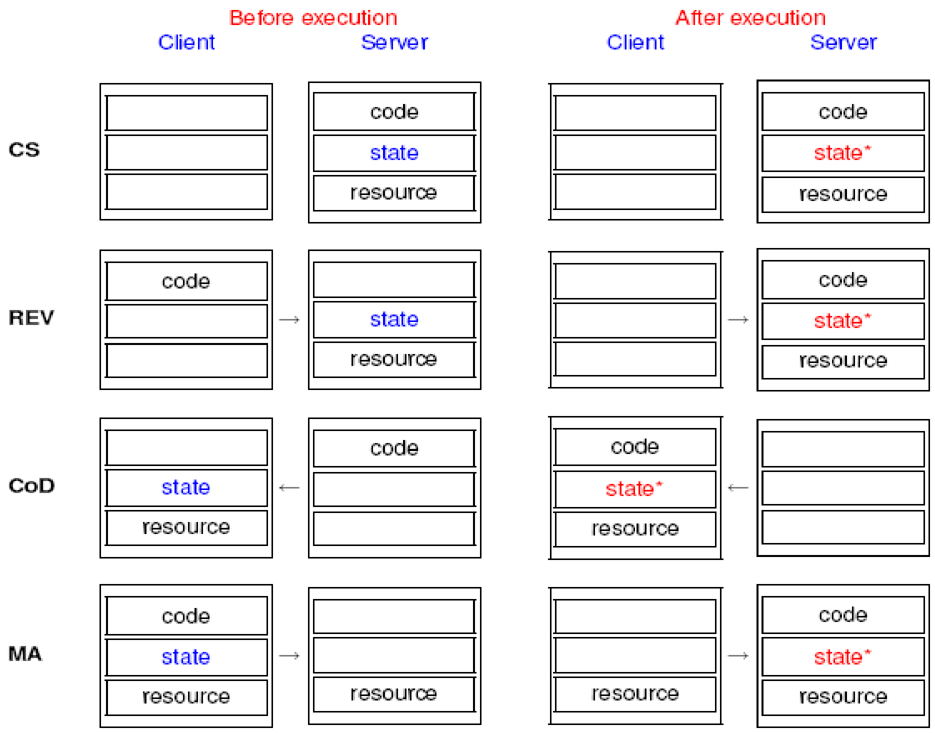
Virtual Machine Monitor (VMM):

An enhanced Linux operating system, Ensures that vservers are separated in different vservers: processes are executed concurrently and independently.

Code Migration

- Approaches to code migration
- Migration and local resources
- Migration in heterogeneous systems

Some Context



CS: Client-Server

REV: Remote evaluation
e.g.perform database operations involving large amount of data

CoD: Code-on-demand
Migrate part of server to client

MA: Mobile agents
moves from site to site

Strong and weak mobility

Object components

- ❖ **Code segment:** contains the actual code
- ❖ **Resource segment**: contains references to resources needed by the process
- ❖ **Execution segment:** stores the executation state of a process

Weak mobility

Move only code segment and perhaps some initialization data (and reboot execution):

- Relatively simple, especially if code is portable (e.g. Java applet)
- Distinguish code shipping (push) from code fetching (pull)

Strong mobility

Move components, including execution segment

- ➢ ***Migration:*** move entire object from one machine to the other. A process could be stopped, moved to another machine and resumed execution where it left off.
- ➢ ***Cloning:*** start a clone, and set it in the same execution state. Migration can be sender-initiated (e.g. uploading program), receiver-initiated (e.g. java applets) or hybrid.

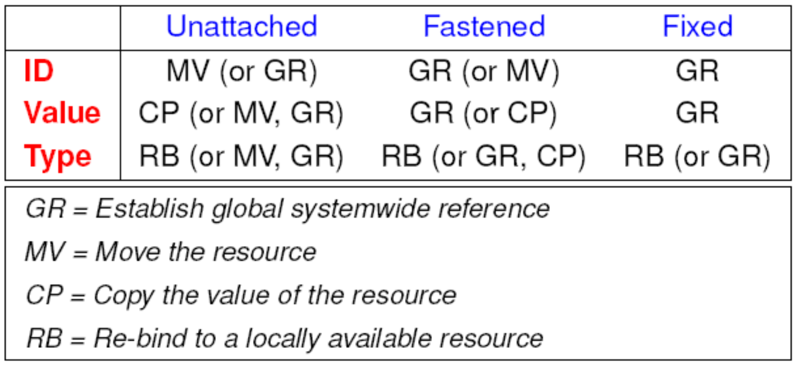
Managing local resources

Process-to-resource binding

- ***By identifier***: the object (process) requires a specific instance of a resource (strongest binding) (e.g. a specific database, a URL)
- ***By value:*** the object requires the value of a resource (e.g. a library). The execution of a process would not be affected if another resource would provide the same value
- ***By type:*** the object requires that only a type of resource is available (e.g. local devices such as a color monitor)

Also there is need to consider resource-to-machine binding:

- ***Unattached resources :*** can be easily moved between machines (e.g. files associated with programs)
- ***Fastened resources:*** costly to move or copy them from one machine to another (e.g. local databases, web sites)
- ***Fixed resources:*** bound to one machine, cannot be moved (e.g. local devices such as printers)



	Unattached	Fastened	Fixed
ID	MV (or GR)	GR (or MV)	GR
Value	CP (or MV, GR)	GR (or CP)	GR
Type	RB (or MV, GR)	RB (or GR, CP)	RB (or GR)

GR = Establish global systemwide reference
MV = Move the resource
CP = Copy the value of the resource
RB = Re-bind to a locally available resource

Migration in heterogenous systems

Main problem

❖ The target machine may not be suitable to execute the migrated code
❖ The definition of process/thread/processor context is highly dependent on local hardware, operating system and runtime system

The Only solution:-
Make use of an abstract machine that is implemented on different platforms:

✓ Interpreted languages, effectively having their own VM
✓ Using VM (as discussed previously)