

Unit 3: Processes

we take a closer look at how the different types of processes plays crucial role in distributed systems.

Outlines

Background

- Thread
- Virtualization
- Clients
- Servers
- Code Migration

Background

- The concept of a process originates from the field of operating systems.
- Process → Program in execution
- EG. To efficiently organize client-server systems, it is often convenient to make use of multithreading technique.
- A main contribution of threads in distributed systems is that they results in a high level of performance of multithreading in distributed system.

Multithreading

- The ability of an OS to support multiple, concurrent paths of execution within a single process.

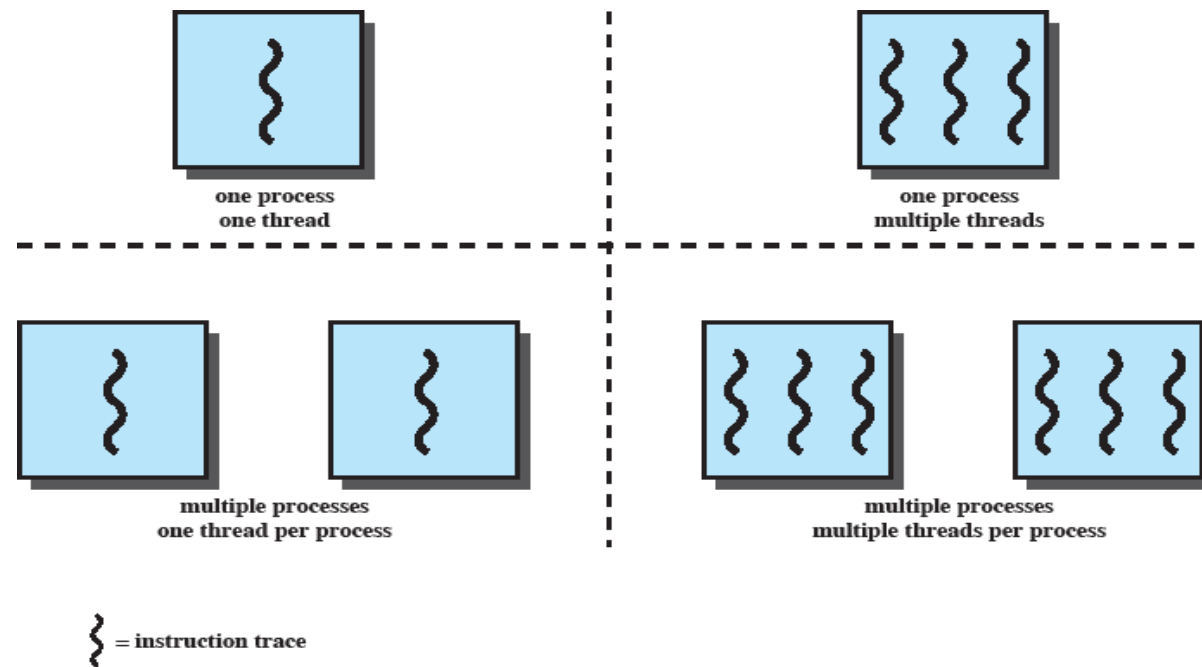


Figure 4.1 Threads and Processes [ANDE97]

Threads vs. processes

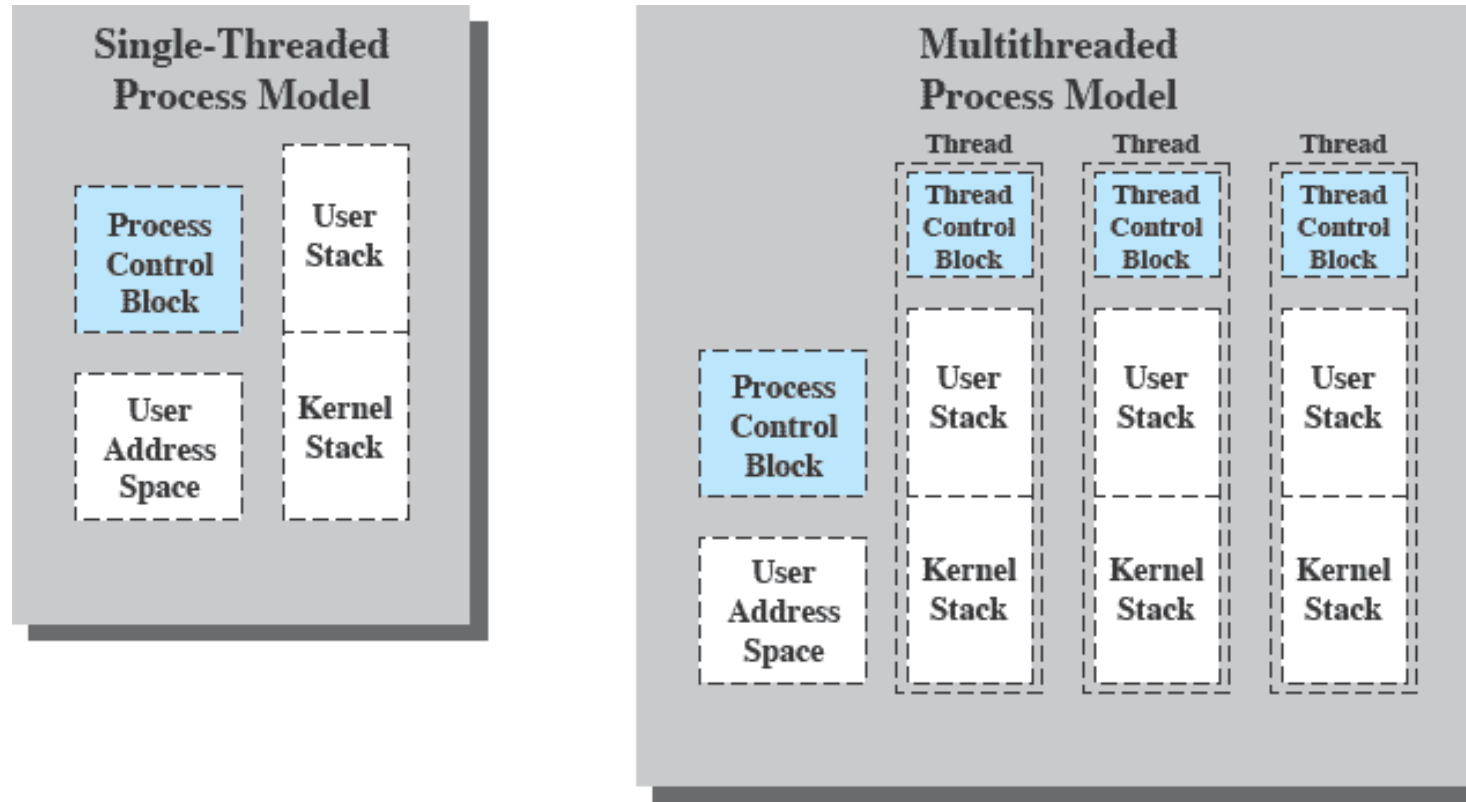


Figure 4.2 Single Threaded and Multithreaded Process Models

Introduction to Processes and Threads

- To understand the role of threads in distributed systems, it is important to understand what a process is, and how processes and threads relate.
- To execute a program, an operating system creates a number of **virtual processors**, each one for running a different program.
- To keep track of these virtual processors, the operating system has a process table, containing entries to store CPU register values, memory maps, open files, accounting information, privileges, etc.
- A process is often defined **as a program in execution**, that is, a program that is currently being executed on one of the operating system's virtual processors.
- In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent.

- Each time a process is created, the operating system must create a complete **independent address space**.
- Allocation can mean initializing memory segments(for example, a data segment) by copying the associated program into a code segment, and setting up a stack segment for temporary data.
- Likewise, **switching the CPU between two processes** may be relatively **expensive** as well.
- Apart from saving the CPU context (which consists of register values, program counter, stack pointer, etc.), the operating system will also have to modify registers of the memory management unit (MMU) and invalidate address translation caches such as in the translation look-aside buffer (TLB).
- In addition, if the operating system supports more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.
- Like a process, a thread executes its own piece of code, independently from other threads.

Two implications of multi-thread systems

- There are two important implications of this approach.:
 - First of all, the performance of a multithreaded application is not worse than that of its single-threaded counterpart. In fact, in many cases, multithreading leads to a performance gain.
 - Second, because threads are not automatically protected against each other, development of multithreaded applications requires additional intellectual effort.

Matrix Multiplication

Note that each *element* of the resultant matrix can be computed independently, that is to say by a different thread.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} =$$

$$\begin{pmatrix} a_{11}.b_{11} + a_{12}.b_{21} + a_{13}.b_{31} & a_{11}.b_{12} + a_{12}.b_{22} + a_{13}.b_{32} & a_{11}.b_{13} + a_{12}.b_{23} + a_{13}.b_{33} \\ a_{21}.b_{11} + a_{22}.b_{21} + a_{23}.b_{31} & a_{21}.b_{12} + a_{22}.b_{22} + a_{23}.b_{32} & a_{21}.b_{13} + a_{22}.b_{23} + a_{23}.b_{33} \\ a_{31}.b_{11} + a_{32}.b_{21} + a_{33}.b_{31} & a_{31}.b_{12} + a_{32}.b_{22} + a_{33}.b_{32} & a_{31}.b_{13} + a_{32}.b_{23} + a_{33}.b_{33} \end{pmatrix}$$

PROCESS VERSUS THREAD

PROCESS	THREAD
An instance of a computer program that is being executed	A component of a process which is the smallest execution unit
Heavyweight	Lightweight
Switching requires interacting with the operating system	Switching does not require interacting with the operating system
Each has its own memory space	Use the memory of the process they belong to
Requires more resources	Requires minimum resources
Difficult to create a process	Easier to create
Inter-process communication is slow because each process has a different memory address	Inter-thread communication is fast because the threads share the same memory address of the process they belong to
In a multi-processing environment, each process executes independently	A thread can read, write or modify data of another thread

Visit www.PEDIAA.com

Difference	Process	Thread
Resource Allocation	Allocate new resources each time we run a program.	Share resources of process.
Resource Sharing	In general, resources are not shared. The code may be shared for the same program.	Share code, heap, data area except stack.
Address	Have a separate address space	Share address space
Communication	Communicate through IPC.	Communicate freely with modifying shared variables.
Context Switching	Generally slower than thread.	Generally faster than process.

Thread usage in traditional, non-distributed systems

- 1 • Spreadsheet: It maintains the inter-dependencies between different cells, often from different spreadsheets

whenever a cell is modified, all dependent cells are automatically updated. When a user changes the value in a single cell, such a modification can trigger a large series of computations. If there is only a single thread of control, computation cannot proceed while the program is waiting for input. Likewise, it is not easy to provide input while dependencies are being calculated. The easy solution is to have at least two threads of control: **one for handling interaction** with the user and **one for updating the spreadsheet**. In the mean time, a third thread could be used for **backing up the spreadsheet** to disk while the other two are doing their work.

2. possible to exploit parallelism

each thread is assigned to a different CPU while shared data are stored in shared main memory. When properly designed, such parallelism can be transparent: the process will run equally well on a uniprocessor system, a bit slower.

Multithreading for parallelism is becoming increasingly important with the availability of relatively cheap multiprocessor workstations. Such computer systems are typically used for running servers in client-server applications.

• useful in the context of large applications

Such applications are often developed as a collection of cooperating programs, each to be executed by a separate process. This approach is typical for a UNIX environment. Cooperation between programs is implemented by means of inter-process communication (IPC) mechanisms.

The major drawback of all IPC mechanisms is that communication often requires extensive context switching, shown at three different points in Figure.

Thread Usage in Non-distributed Systems: IPC and System Call Costs

Because IPC requires kernel intervention, a process will generally:

- First have to switch from user mode to kernel mode. This requires changing the memory map in the MMU, as well as flushing the TLB.

- Within the kernel, a process context switch takes place, after which the other party can be activated by switching from kernel mode to user mode again.

- The latter switch again requires changing the MMU map and flushing the TLB.

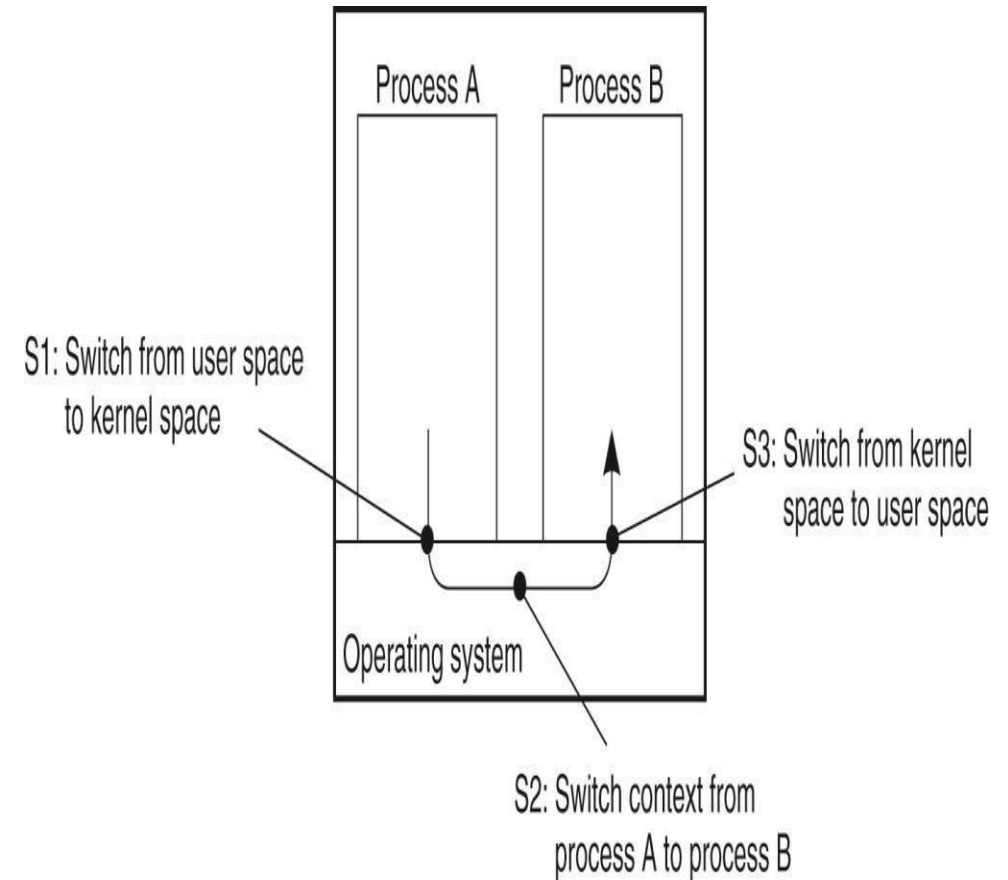


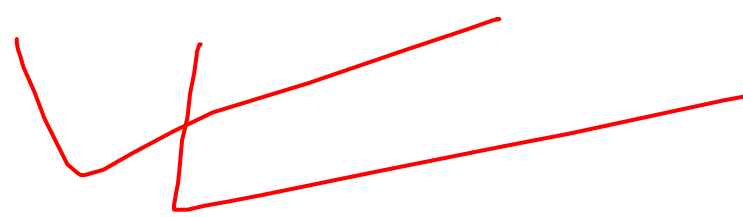
Figure. Context switching as the result of IPC.

Thread Usage in Non-distributed Systems: IPC and System Call Costs

- Instead of using processes, an application can also be constructed such that different parts are executed by separate threads.
- Communication between those parts is entirely dealt with by using shared data.
- Thread switching can sometimes be done entirely in user space, although in other implementations, the kernel is aware of threads and schedules them. The effect can be a dramatic improvement in performance.

- 
- a pure software engineering reason

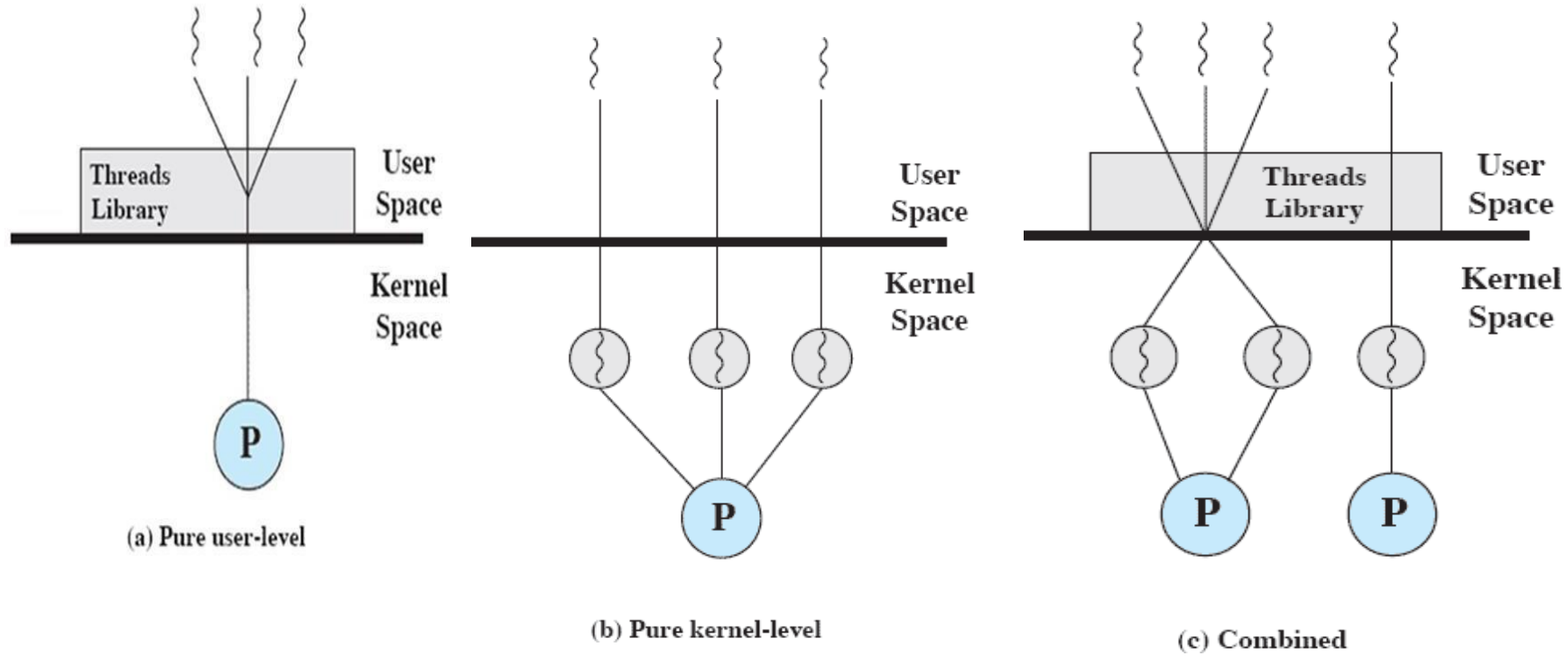
For example, in the case of a word processor, separate threads can be used for handling user input, spelling and grammar checking, document layout, index generation, etc.



Thread Implementation

- Threads are often provided in the form of a thread package. Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes and condition variables. There are basically **three** approaches to implement a thread package.
 - **The first approach** is to construct a thread library that is executed entirely in user mode, called **User Level Threads (ULT)**.
 - **The second approach** is to have the kernel be aware of threads and schedule them, called **Kernel Level Threads (KLT)**.
 - **The third approach** is a **Hybrid Form**.

ULT Vs. KLT



User-Level Threads Advantages

- First, it is cheap to create and destroy threads. Because all thread administration is kept in the user's address space:
 - the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack.
 - Analogously, destroying a thread mainly involves freeing memory for the stack.
- A second advantage of user-level threads is that switching thread context can often be done in just a few instructions.
 - Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched. There is no need to change memory maps, flush the TLB, and do CPU operations.

User-level Threads Disadvantages

- A major drawback of user-level threads is that invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process.
- Threads are particularly useful to structure large applications into parts that could be logically executed at the same time. In that case, blocking on I/O should not prevent other parts to be executed in the meantime.

Kernel-Level Threads

- These problems can be mostly circumvented by implementing threads in the operating system's kernel. Unfortunately, there is a high price to pay:
- Every thread operation (creation, deletion, synchronization, etc.) will have to be carried out by the kernel which require a system call.
- Switching thread contexts may now become as expensive as switching process contexts. As a result, most of the performance benefits of using threads instead of processes then disappears.

Kernel-Level Threads Advantages

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Kernel-Level Threads Disadvantages

- Every thread operation (creation, deletion, synchronization, etc.) will have to be carried out by the kernel which requires a system call.
- Switching thread contexts may now become as expensive as switching process contexts.
- As a result, most of the performance benefits of using threads instead of processes then disappears.

Hybrid Approach

- A solution lies in a hybrid form of user-level and kernel-level threads, generally referred to as **lightweight processes (LWP)**. An LWP runs in the context of a single (heavy-weight) process, and there can be several LWPs per process.
- In addition to having LWPs, a system also offers a user-level thread package. Offering applications the usual operations for creating and destroying threads.
- In addition, the package provides facilities for thread synchronization such as mutexes and condition variables. The important issue is that the thread package is implemented entirely in user space. All operations on threads are carried out without intervention of the kernel.

LWP

ULP



US

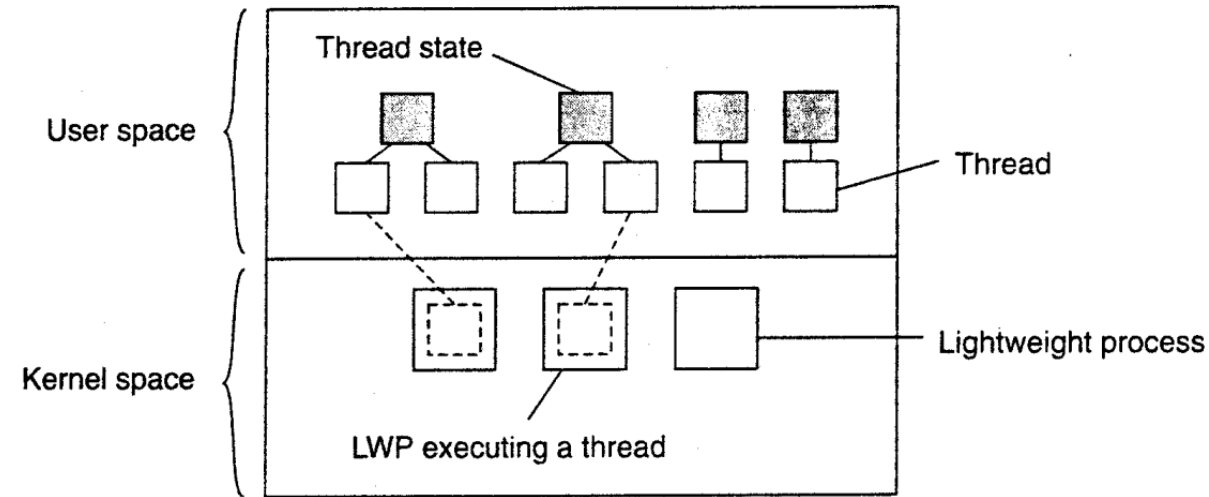


Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

Hybrid Approach Advantages

1. Creating, destroying, and synchronizing threads is relatively cheap and involves no kernel intervention at all.
2. Provided that a process has enough LWPs, a blocking system call will not suspend the entire process.
3. There is no need for an application to know about the LWPs. All it sees are user-level threads.
4. LWPs can be easily used in multiprocessing environments, by executing different LWPs on different CPUs. This multiprocessing can be hidden entirely from the application.

Hybrid Approach disadvantages

1. The only drawback of lightweight processes in combination with user-level threads is that we still **need to create and destroy LWPs, which is just as expensive as with kernel-level threads.**

However, creating and destroying LWPs needs to be done only occasionally, and is often fully controlled by the operating system.

Threads in Distributed Systems

- Multithreaded Clients → Web Browser to reduce the communication latencies
- Multithreaded Servers → Three types: Multithreaded server, Single Threaded server and Finite-State Machine Server

Multithreaded Clients

- The usual way to hide communication latencies is to initiate communication and immediately proceed with something else. A typical example where this happens is in [Web browsers](#).
- A Web browser often starts with fetching the HTML page and subsequently displays it.
- To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in.
- As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts. Each thread sets up a separate connection to the server and pulls in the data. Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process.

Multithreaded Servers

- Although there are important benefits to multithreaded clients, the main use of multithreading in distributed systems is found at the server side.
- Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uni-processor systems.
- However, now that multiprocessor computers are widely available as general-purpose workstations, multithreading for parallelism is even more useful.
- To understand the benefits of threads for writing server code, consider the organization of a file server that occasionally has to block waiting for the disk.

Three ways to construct a server

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Multithreaded Servers

- The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply.
- In Fig. , a dispatcher thread, reads incoming requests for a file operation.
- After examining the request, the server chooses an idle (i.e., blocked) worker thread and hands it the request.

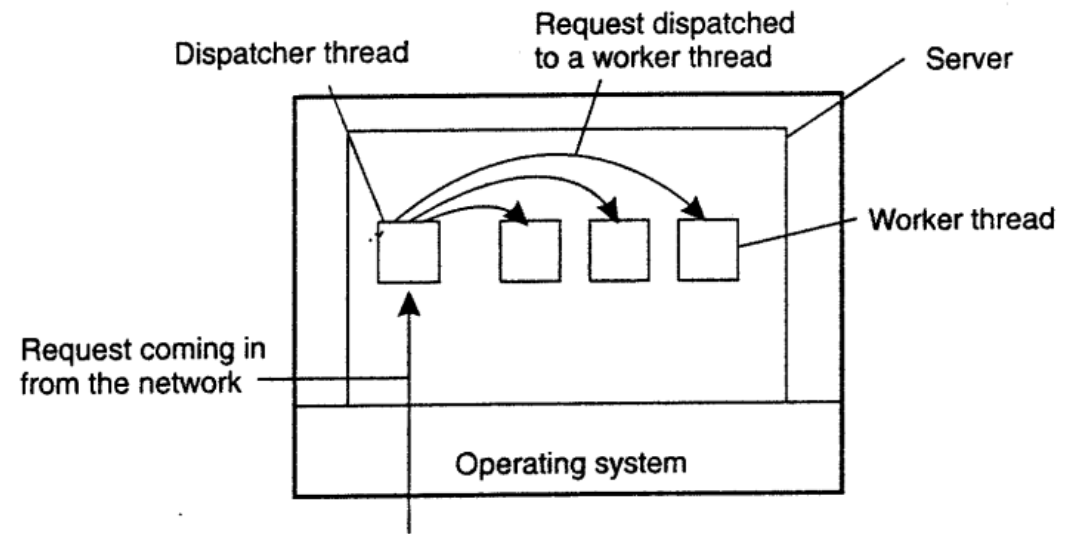


Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

Single-thread Server

One possibility is to have it operate as a single thread:

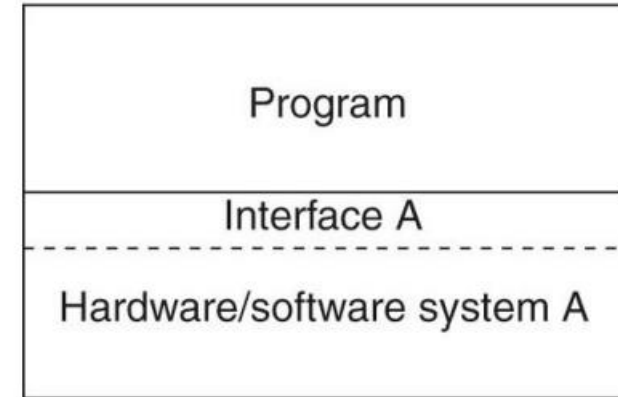
- The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one.
- While waiting for the disk, the server is idle and does not process any other requests. Consequently, requests from other clients cannot be handled.
- In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk.
- The net result is that many fewer requests/sec can be processed

Finite-state machine Server

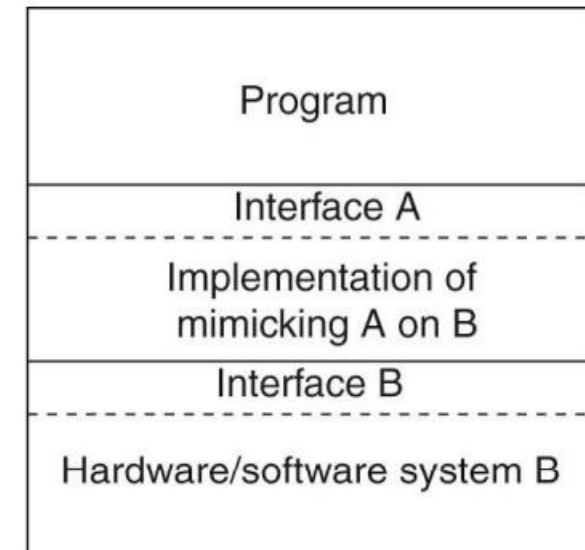
- A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.
- However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message.
- The next message may either be a request for new work or a reply from the disk about a previous operation.
- If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed and subsequently sent to the client.
- In this scheme, the server will have to make use of non-blocking calls to send and receive.

The Role of Virtualization in Distributed Systems

- In practice, every (distributed) computer system offers a programming interface (API) to higher level software, as shown in Fig. (a).
- There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware systems.
- In its essence, virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system, as shown in Fig. (b).



(a)



(b)

Architectures of Virtual Machines (1)

Interfaces at different levels

- An interface between the hardware and software consisting of machine instructions
 - that can be invoked by any program.
- An interface between the hardware and software, consisting of machine instructions
 - that can be invoked only by privileged programs, such as an operating system.
- An interface consisting of system calls as offered by an operating system.

Architectures of Virtual Machines (2)

- An interface consisting of library calls
 - generally forming what is known as an application programming interface (API).
 - In many cases, the aforementioned system calls are hidden by an API.

Architectures of Virtual Machines (3)

- The essence of virtualization is to mimic the behavior of these interfaces.

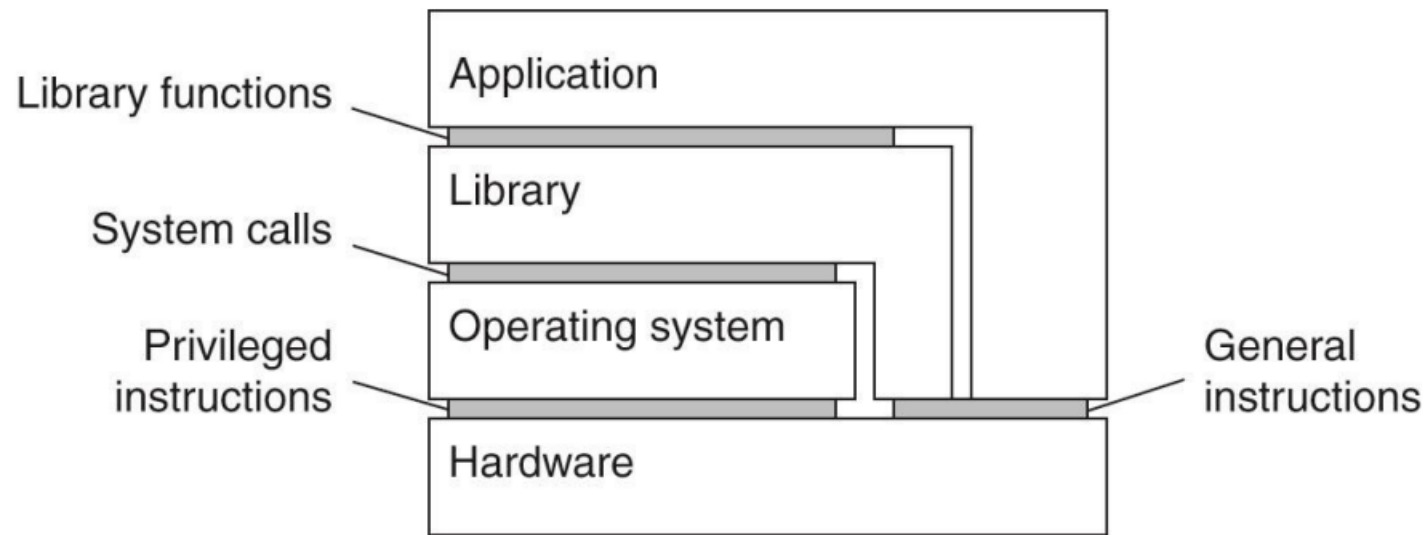


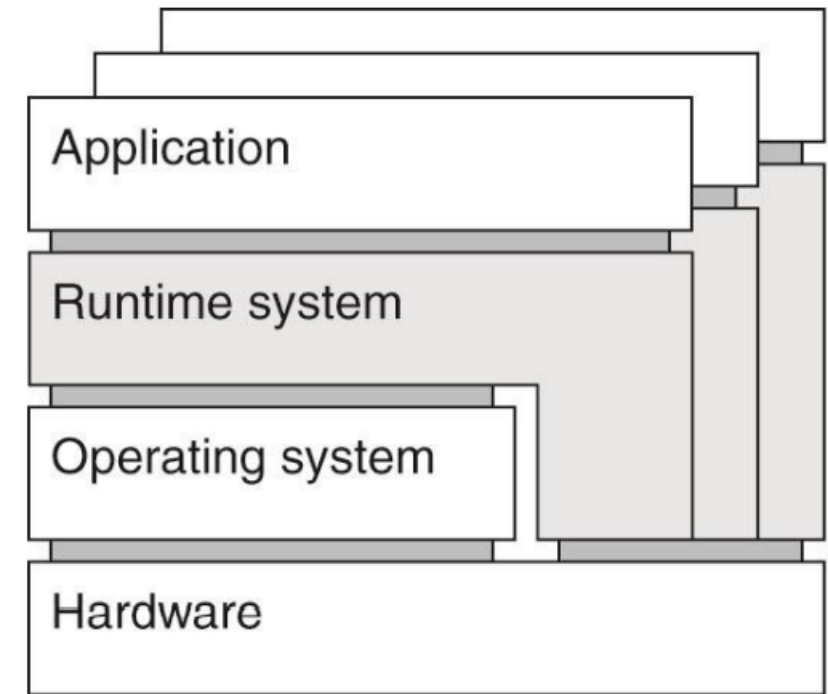
Figure . Various interfaces offered by computer systems.

Two Different Ways of Making Virtual Machines

- Virtualization can take place in two different ways:

1- First, we can build a runtime system that essentially provides an abstract instruction set that is to be used for executing applications.

Instructions can be **interpreted** (i.e. the Java runtime environment), but could also be **emulated** (i.e. running Windows applications on UNIX platforms). This type of virtualization leads to **process virtual machine**, stressing that virtualization is done essentially **only for a single process**.



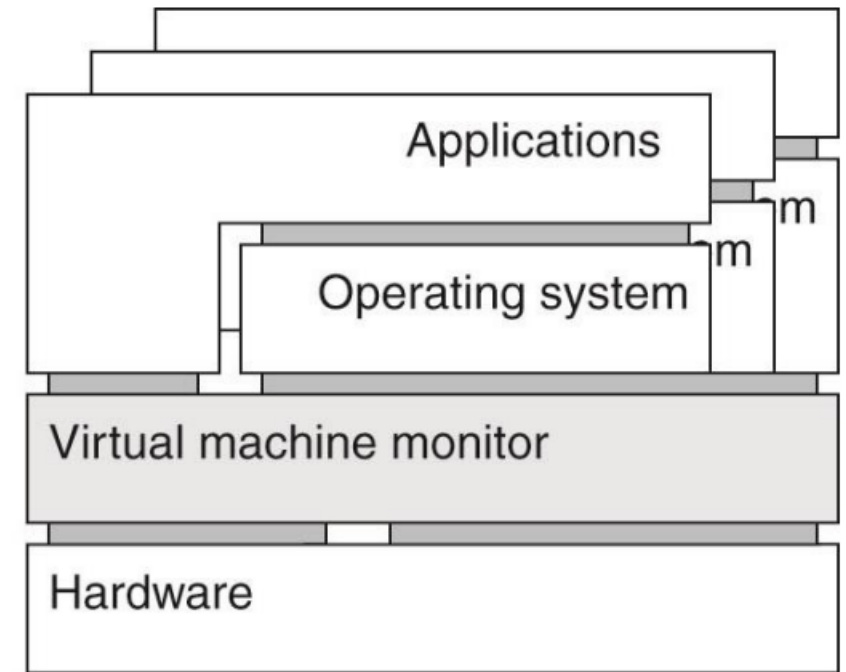
(a)

A process virtual machine, with multiple instances of (application, runtime) combinations.

Two Different Ways of Making Virtual Machines

- Virtualization can take place in two different ways:

2- An alternative approach is to provide a system that is essentially implemented as a layer completely shielding the original hardware, but offering the complete instruction set of hardware as an interface. This interface can be offered simultaneously to different programs. As a result, it is now possible to have multiple, and different operating systems run independently and concurrently on the same platform. The layer is generally referred to as a **Virtual Machine Monitor** (VMM). (i.e. VMware)



(b)

A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

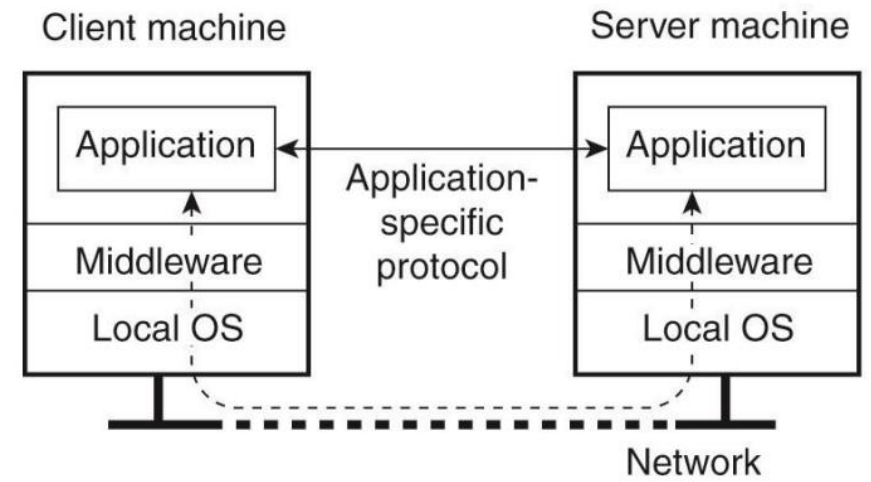
Clients and Servers

- In the previous chapters we discussed the client-server model the roles of clients and servers, and the ways they interact.
- Let us now take a closer look at the anatomy of clients and servers, respectively.
- Clients → Networked User Interfaces

Clients

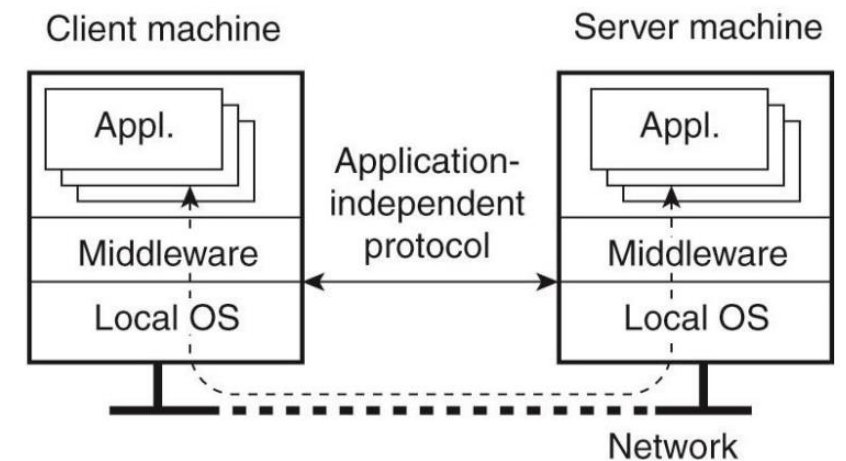
- A major task of client machines is to provide the means for users **to interact with remote servers**. There are roughly two ways in which this interaction can be supported.
- First, for each remote service the client machine will have a separate counterpart that can contact the service over the network (Fig. (a), Fig. (b)).

E.g. A typical example is an agenda running on a user's PDA that needs to synchronize with a remote, possibly shared agenda.



(a)

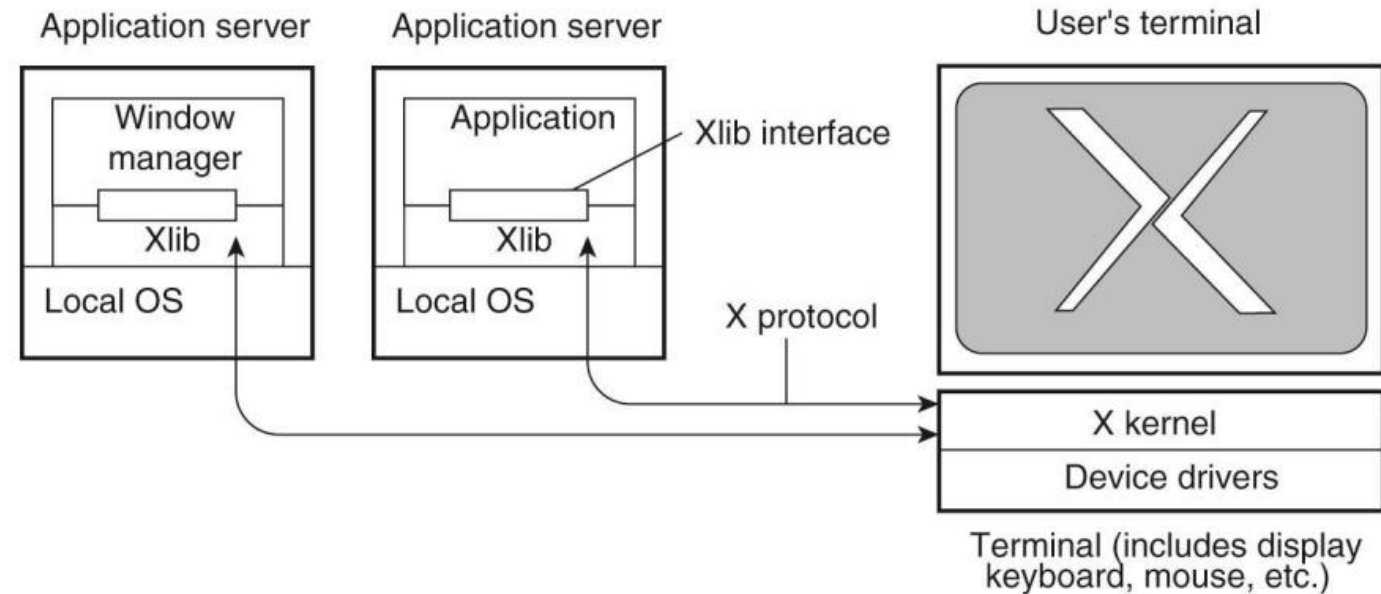
A networked application with its own protocol.



(b)

A general solution to allow access to remote applications.

- A second solution is to provide direct access to remote services by only offering a convenient user interface. Effectively, this means that the client machine is used only as a terminal with no need for local storage, leading to an application neutral solution as shown in (Fig. C)



C) The basic organization of the XWindow System

Example: The X Window System

- one of the oldest and still widely-used networked user interfaces is the X Window system.
- X can be viewed as that part of an operating system that controls the terminal.
- The heart of the system is formed by what we shall call the X kernel. It contains all the terminal-specific device drivers, and as such, is generally highly hardware dependent.
- The interesting aspect of X is that the X kernel and the X applications need not necessarily reside on the same machine.
- In particular, X provides the X protocol, which is an application-level communication protocol by which an instance of Xlib can exchange data and events with the X kernel.
- For example, Xlib can send requests to the X kernel for creating or killing a window, setting colors, and defining the type of cursor to display, among many other requests. In turn, the X kernel will react to local events such as keyboard and mouse input by sending event packets back to Xlib.

Assignment-II: February 4 (Tomorrow)

- Explain the working of Xwindows System as a Networked User Interface.
- Explain Virtualization. Describe the two Different Ways of Making Virtual Machines.

Revisit: Distribution Transparency

- Goal → to hide the fact that its processes and resources are physically distributed across multiple computers.
- In other words, it tries to make the distribution of processes and resources transparent, that is, invisible, to end users and applications.

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Client-Side Software for Distribution Transparency

- Client software comprises more than just user interfaces. In many cases, parts of the processing and data level in a client-server application are executed on the client side as well. A special class is formed by embedded client software, such as for automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, etc.
- Besides the user interface and other application-related software, client software comprises components for achieving **distribution transparency**. **Access transparency** is generally handled through the generation of a **client stub** from an **interface definition (IDL)** of what the server has to offer. There are different ways to handle **location, migration, and relocation transparency**. Using a convenient **naming system** is crucial. Many distributed systems implement **replication transparency** by means of client-side solutions

- In **failure transparency** a client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts. **Concurrency transparency** requires less support from client software. **Persistence transparency** is often completely handled at the server.

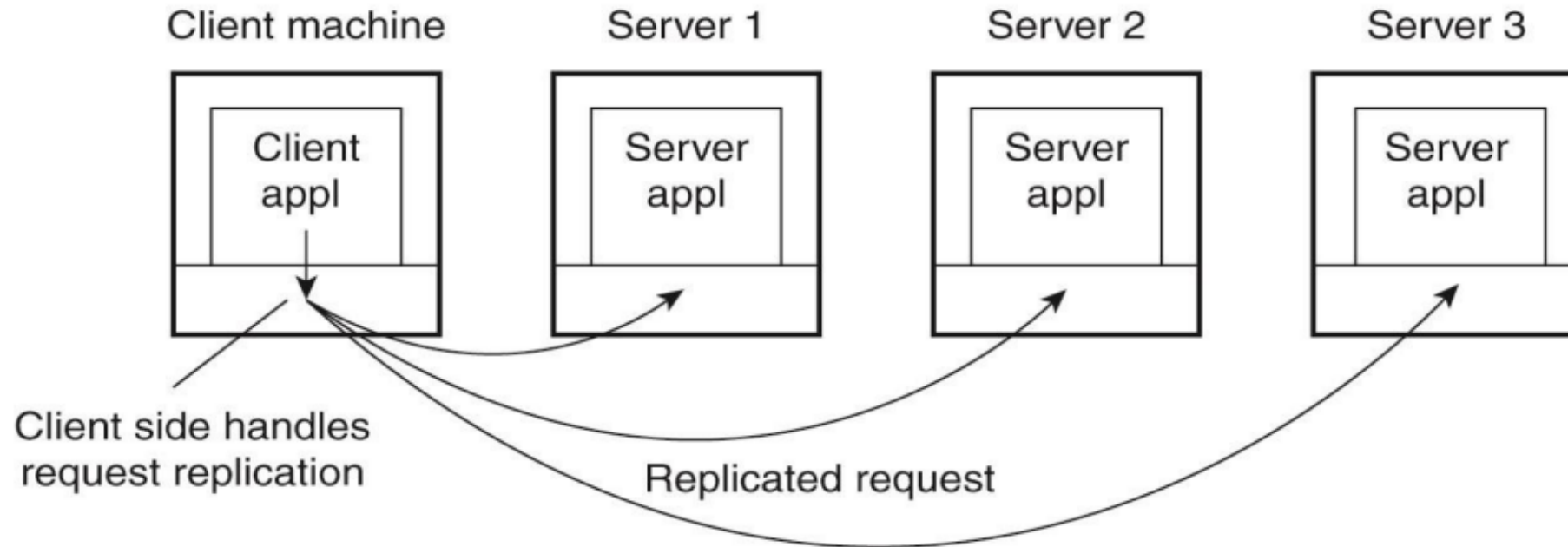


Fig: Transparent replication of a server using a client-side solution.

Servers: General Issues

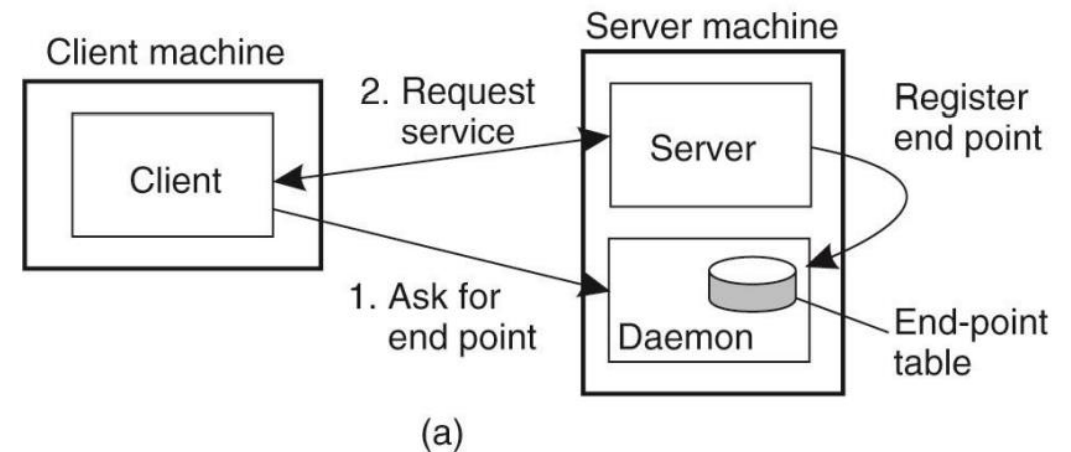
- A server is a process implementing a specific service on behalf of a collection of clients. In essence, each server is organized in the same way: it waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.
- In the case of an **iterative server**, the server itself handles the request and, if necessary, returns a response to the requesting client.
- **A concurrent server** does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request.
- **A multithreaded server** is an example of a concurrent server.
- An alternative implementation of a concurrent server is **to fork a new process for each new incoming request**.

Servers: General Issues

- An issue is where clients contact a server. In all cases, clients send requests to an **end point**, also called a **port**, at the machine where the server is running. Each server listens to a specific end point. How do clients know the end point of a service?
- The approach is to globally **assign end points for well-known services**.
- For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80.

- There are many services that do not require a pre-assigned end point (i.e. a time-of-day server). So, a client will first have to look up the end point.
- One solution is to have a special **daemon** running on each machine that runs servers. The daemon keeps track of the current end point of each service implemented by a co-located server. **The daemon itself listens to a well-known end point.** A client will first contact the daemon, request the end point, and then contact the specific server, as shown in Fig. (a).

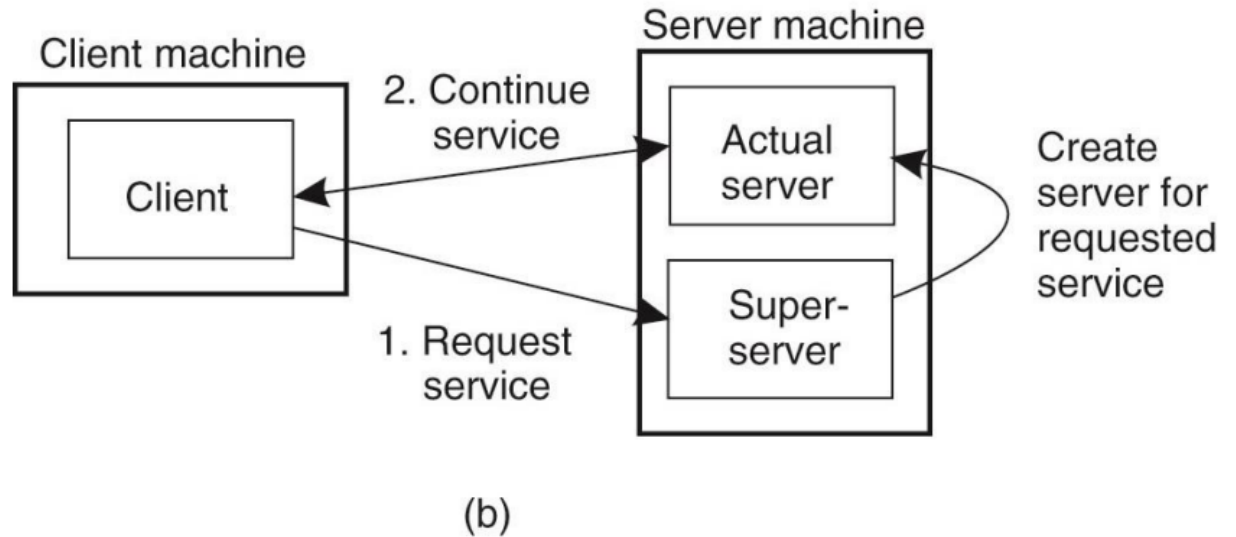
Servers General Design Issues



Client-to-server binding using a daemon.

- It is often more efficient to have a single **super-server** listening to each end point associated with a specific service, as shown in Fig. (b).
- *For example, the daemon in UNIX Listens to a number of well-known ports for Internet services. When a request comes in, the daemon forks a process to process the request. That process will exit after it is finished.*

Servers General Design Issues



Client-to-server binding using a super-server.

Servers: General Issues

- A final, important design issue, is whether or not the server is **stateless or stateful**.
- A **stateless server** does not keep information on the state of its clients, and can change its own state without having to inform any client (for example, a Web Server).
- In contrast, a **stateful server** generally maintains persistent information on its clients. This means that the information needs to be explicitly deleted by the server. A typical example is a file server that allows a client to keep a local copy of a file, even for performing update operations. Such a server would maintain a **table (state)** containing (client, file) entries

Terminology

- **Distributed** : refers to splitting a business into different sub-services and distributing them on different machines.
- **Cluster** : It means that multiple servers are grouped together to achieve the same business and can be regarded as one computer.

Server Clusters

- Simply put, a server cluster is nothing else but a collection of machines connected through a network, where each machine runs one or more servers.

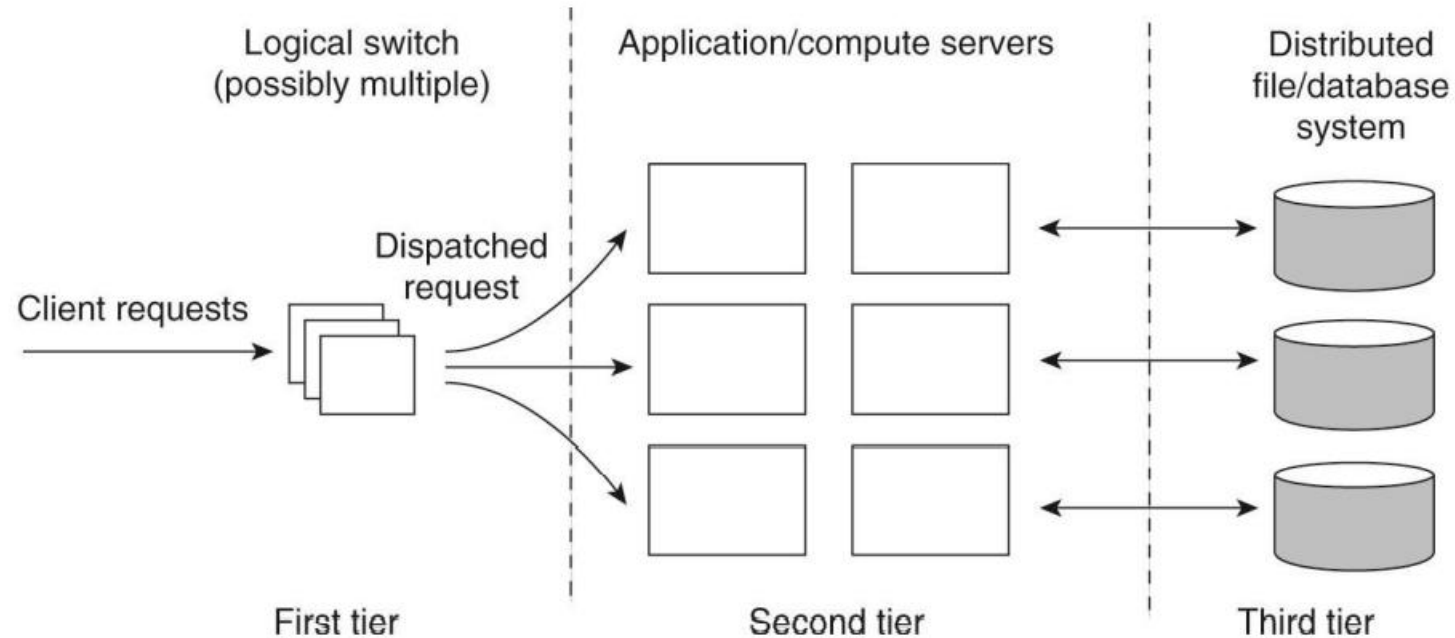


Fig: The general organization of a three-tiered server cluster.

Server Clusters

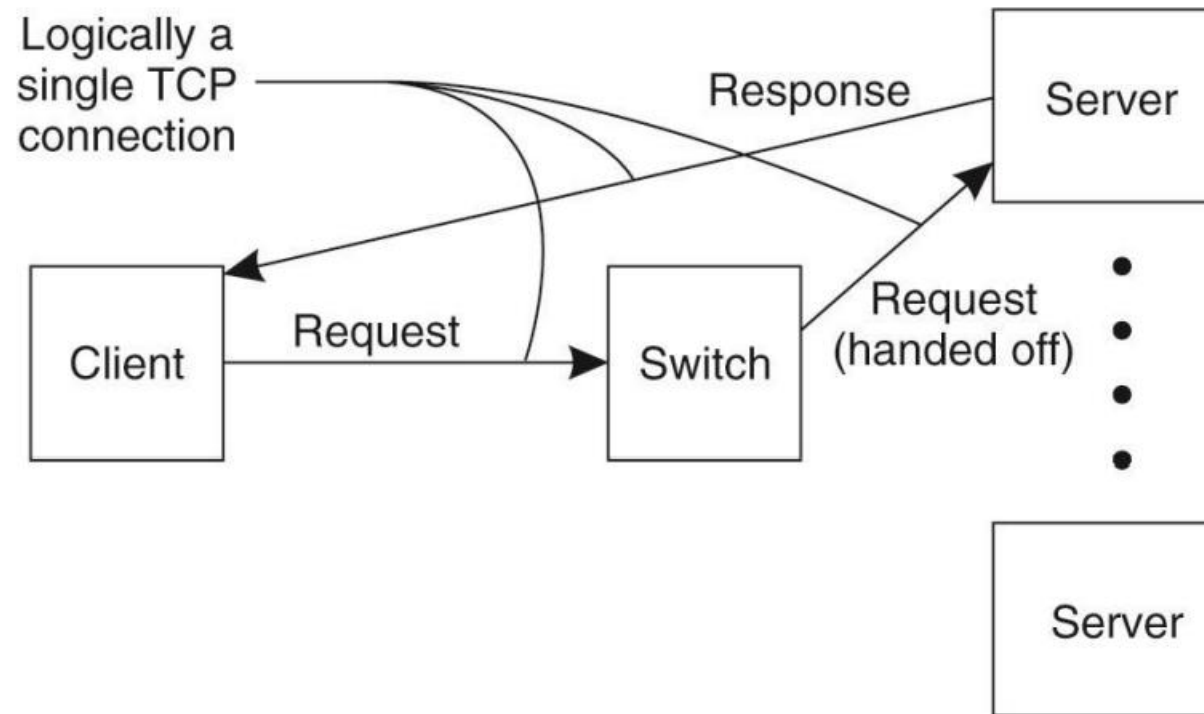


Fig: The principle of TCP handoff.

Revisit

- Server Cluster → Statically configured
→ Single Access point → Single point of Failure
- Access Point → Publicly Available

Eg. DNS

It returns multiple addresses (Same host)

If any one of the address fails, client will have to do several attempts.

-doesn't solve the problem of requiring static AP.

What we need ?

- Static, high living Access Point
- Flexibility in configuration (No more Static Configurations)
- This Lead to a design of Distributed Server.

Distributed Server

- Stable AP
- Mobile IPV6

MIPv6

- Mobile Computing
 - Communication with mobile host, WLANs
- Why we need MIPv6 ?
 - Ipv4 → local network → It works fine
 - When it left Local Network → Problem arises
- Therefore we need the concept of MIPV6.

Mobile IPv6

Basic Idea:

- Every Mobile end devices receives two IP Address simultaneously:
 - Primary IP Address
 - Home Address
 - Secondary IP Address
 - Care-of Address (temporary)

Primary IP Address: Home Address

- Stationary IP Address of the mobile host in its local network
- Always remains unchanged, even when location is changed
- Application running on the mobile host only use primary IP address.

Secondary IP Address: Care-of Address (temporary)

- Temporarily Valid
- Change with every location change and is only valid as long as the host remains in certain guest network.
- Mobile host receives second IP address upon registration in new network and communicate it promptly to a specific agent(Router in home network).

Distributed Servers

- The basic idea behind a distributed server is that clients benefit from a robust, high-performing, stable server. **These properties can often be provided by high-end mainframes, of which some have an acclaimed mean time between failure of more than 40 years.**
- However, by grouping simpler machines transparently into a cluster, and not relying on the availability of a single machine, it may be possible to achieve a better degree of stability than by each component individually. For example, such a cluster could be dynamically configured from end user machines as in the case of a collaborative distributed system

Distributed Servers

So far, **server clusters** are generally:

- 1- rather statically configured.** There is often a separate administration machine that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.
- 2- offer a single access point.** When that point fails, the cluster becomes unavailable.

Distributed Servers

Two requirements:

- 1) Having a stable and long-living access point,
- 2) High level of flexibility in configuring a server cluster.

To eliminate this potential problem, several access points can be provided, of which the addresses are made publicly available. For example, the Domain Name System (DNS) can return several addresses, all belonging to the same host name. This approach still requires clients to make several attempts if one of the addresses fails. Moreover, this does not solve the problem of requiring static access points

Distributed Servers

- Let us concentrate on how a **stable access point** can be achieved in such a system. The main idea is to make use of available networking services, notably **mobility support** for IP version 6 (MIPv6).
- In MIPv6, a mobile node is assumed to have a **home network** where it normally resides and for which it has an associated stable address, known as its **Home Address (HoA)**. This home network has a special router attached, known as the **home agent**, which will take care of traffic to the mobile node when it is away.
- To this end, when a mobile node attaches to a foreign network, it will receive a **temporary Care-of Address (CoA)** where it can be reached. This care-of address is reported to the node's home agent who will then see to it that all traffic is forwarded to the mobile node. Note that applications communicating with the mobile node **will only see the address associated with the node's home network**. They will never see the care-of address.

Assignment-IV (deadline Feb-8)

1. Distinguish between server cluster and Distributed server.
2. Explain the concept behind MIPv6 and explain its working with suitable diagram(Architecture).

Code Migration

- So far, communication is limited to passing data in distributed systems.
- However, there are situations in which passing programs, simplifies the design of a distributed system.
 - What code migration actually is.
 - Different approaches to code migration,
 - How to deal with the local resources that a migrating program uses

A particularly hard problem is migrating code in heterogeneous systems

Approaches to Code Migration

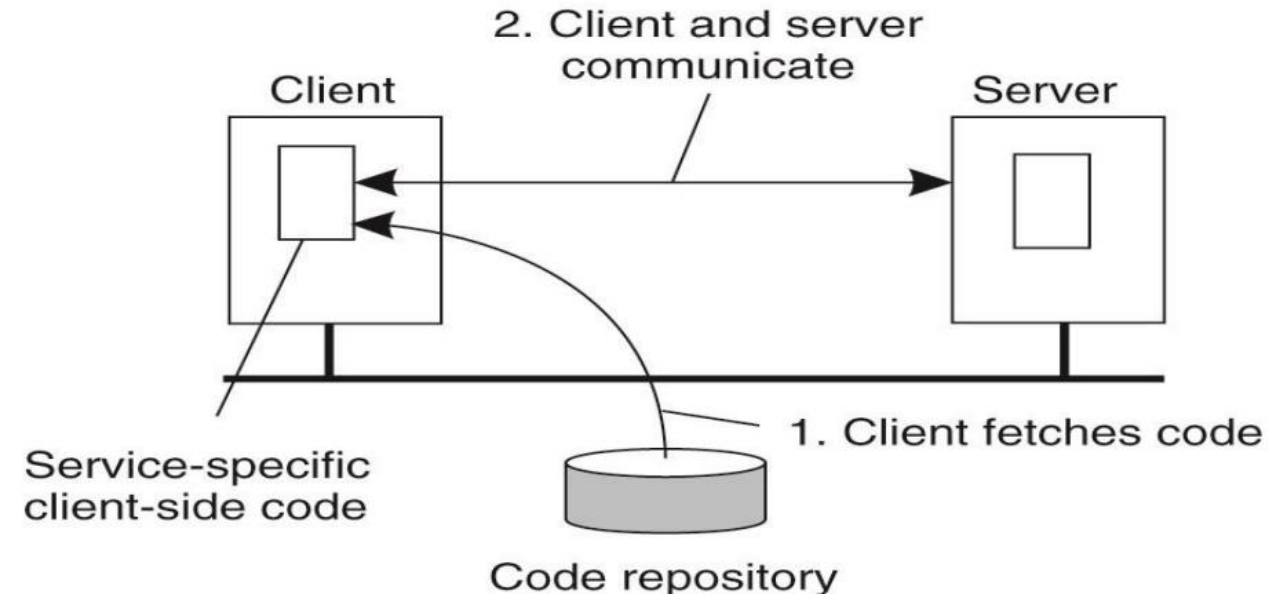
- Before taking a look at the different forms of code migration, let us first consider why it may be useful to migrate code.

Reasons for Migrating Code (imp)

- 1- Overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines.
- 2- To minimize communication. If a client application needs to perform many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. This same reason can be used for migrating parts of the server to the client.
- 3- To improve performance by exploiting parallelism. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent, that moves from site to site.
- 4- Flexibility is another reason. An approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed.
- 5- Dynamically configure distributed systems.

Dynamically configuring a client to communicate to a server

- Let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server.
- At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server.
- This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine.



The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

Dynamically configuring a client to communicate to a server

Advantage

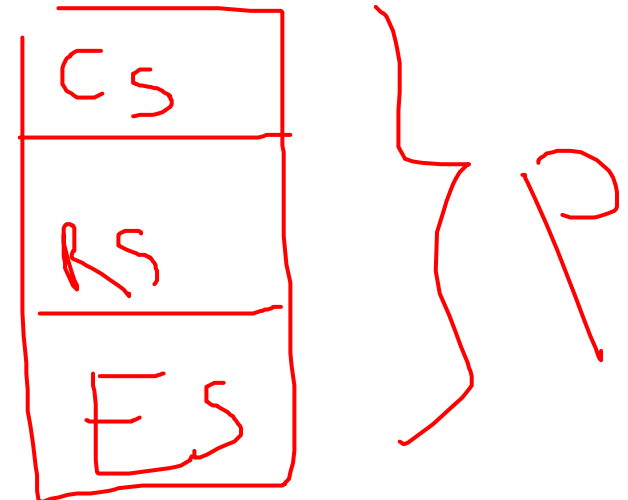
- Clients need not have all the software preinstalled to talk to servers.
- As long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on the server.

Disadvantages

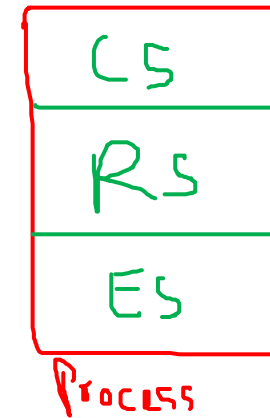
- **Security:** Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven-knows-who may not always be such a good idea.

Models for Migrating Code

- Traditionally, communication in distributed systems is concerned with exchanging data between process.
- Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target.
- A Process consists of three segments:
 - Code Segment #
 - Resources Segment #
 - Execution Segment #



Models for Migrating Code



- **Weak Migration**

only the code segment can be transferred, along with perhaps some initialization data.

Eg. with Java applets, which always start execution from the beginning. The benefit of this approach is its simplicity.

- **Strong Migration**

The code segment and execution segment can be transferred.

Features:

- ✓ a running process can be stopped, subsequently moved to another machine, and then resume execution.
- ✓ more general than weak mobility
- ✓ harder to implement

Models for Migrating Code

- **sender-initiated migration**

uploading programs to a compute server

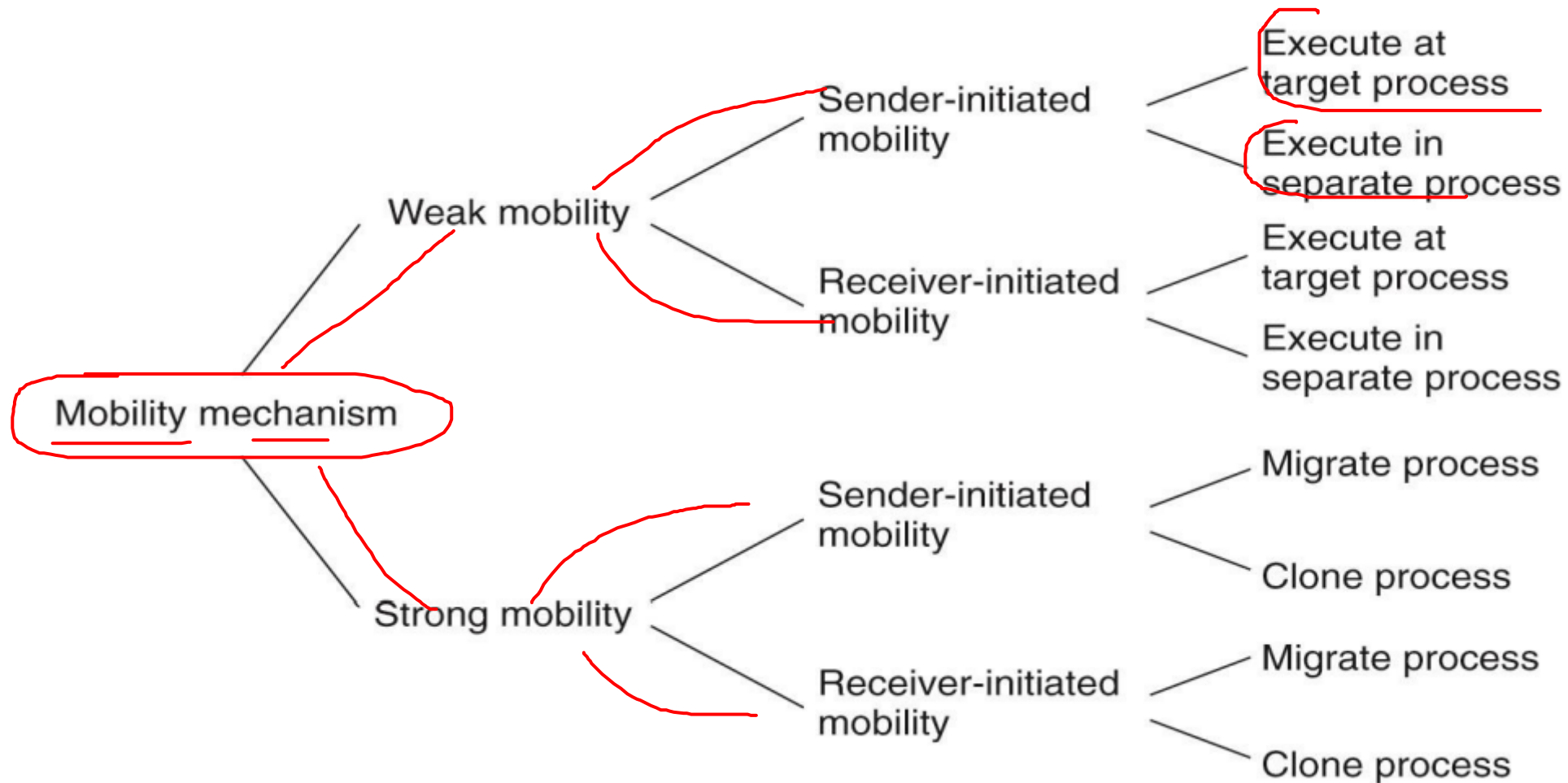
- sending a search/query program across the Internet to a Web database server to perform the queries at that server.

- **receiver-initiated migration**

Downloading code from server by a client

- Java applets are an example of this approach. (i.e. Java applets)

Models for Migrating Code



Migration and Local Resources

- So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration so difficult is that:
 - the resource segment cannot always be simply transferred along with the other segments without being changed.

Resources Migration

- Depends on type of “resources”
 - By Identifier: Specific website, ftp server
 - By Value: Java Libraries
 - By Type: Printer, Local devices
- Depends on type of “attachments”
 - Un attached to any node: Datafiles
 - Fastened resources(can be moved at high cost)
 - E.G. local databases and complete Web sites.

3 types of process-to-resource bindings

- Binding-by-identifier
 - the strongest that precisely the referenced resource, and nothing else, has to be migrated
 - E.g. when a process uses an URL
- Binding-by-value
 - weaker than BI, but only the value of the resource need be migrated
 - A program relying on a libraries (use the locally available one)
- Binding-by-type
 - nothing is migrated, but a resource of a specific type needs to be available after migration
 - E.g. local devices like monitors, printers

3 types of resource-to-machine bindings

- Unattached resources
 - a resource that can be moved easily from machine to machine (e.g. files)
- Fastened resource
 - migration is possible, but at a high cost (e.g. local databases, complete web sites)
- Fixed resources
 - a resource is bound to a specific machine or environment, and cannot be migrated.
 - (e.g. local devices, ports)

9 Possible Combinations

		Resource-to-machine binding		
Process-to-resource binding		Unattached	Fastened	Fixed
	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV,GR)	GR (or CP)	GR
	By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR	Establish a global systemwide reference
MV	Move the resource
CP	Copy the value of the resource
RB	Rebind process to locally-available resource

Actions to be taken with respect to the references to local resources when migrating code to another machine.

Migrating Code in Heterogeneous Systems

- So far, we have assumed that the migrated code can be easily executed at the target machine. This assumption is in order when dealing with homogeneous system.
- In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture. Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform.

Migrating Code in Heterogeneous Systems

- Is more complex.
- Requires code portability.
- A virtual machine approach is used.
- Weak mobility is easier
- In strong mobility it is necessary to handle the execution segment.
- A migration stack is used.