Distributed System

Unit:3

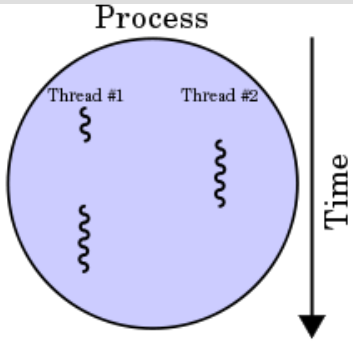
Unit 3. Processes

- 3.1 Threads
- 3.2 Virtualization
- 3.3 Clients
- 3.4 Servers
- 3.5 Code Migration

Introduction to Threads

- 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.

What is Threads?



- Thread is a light weight process created by a process.
- Thread is a single sequence of execution within a process.
- Thread has it own.
  - Program counter that keeps track of which instruction to execute next.
  - System registers which hold its current working variables.
  - Stack which contains the execution history.
- Processes are generally used to execute large, 'heavyweight' jobs such as working in word, while threads are used to carry out smaller or 'lightweight' jobs such as auto saving a word document.
- A thread shares few information with its peer threads (having same input) like code segment, data segment and open files.

Process & Thread

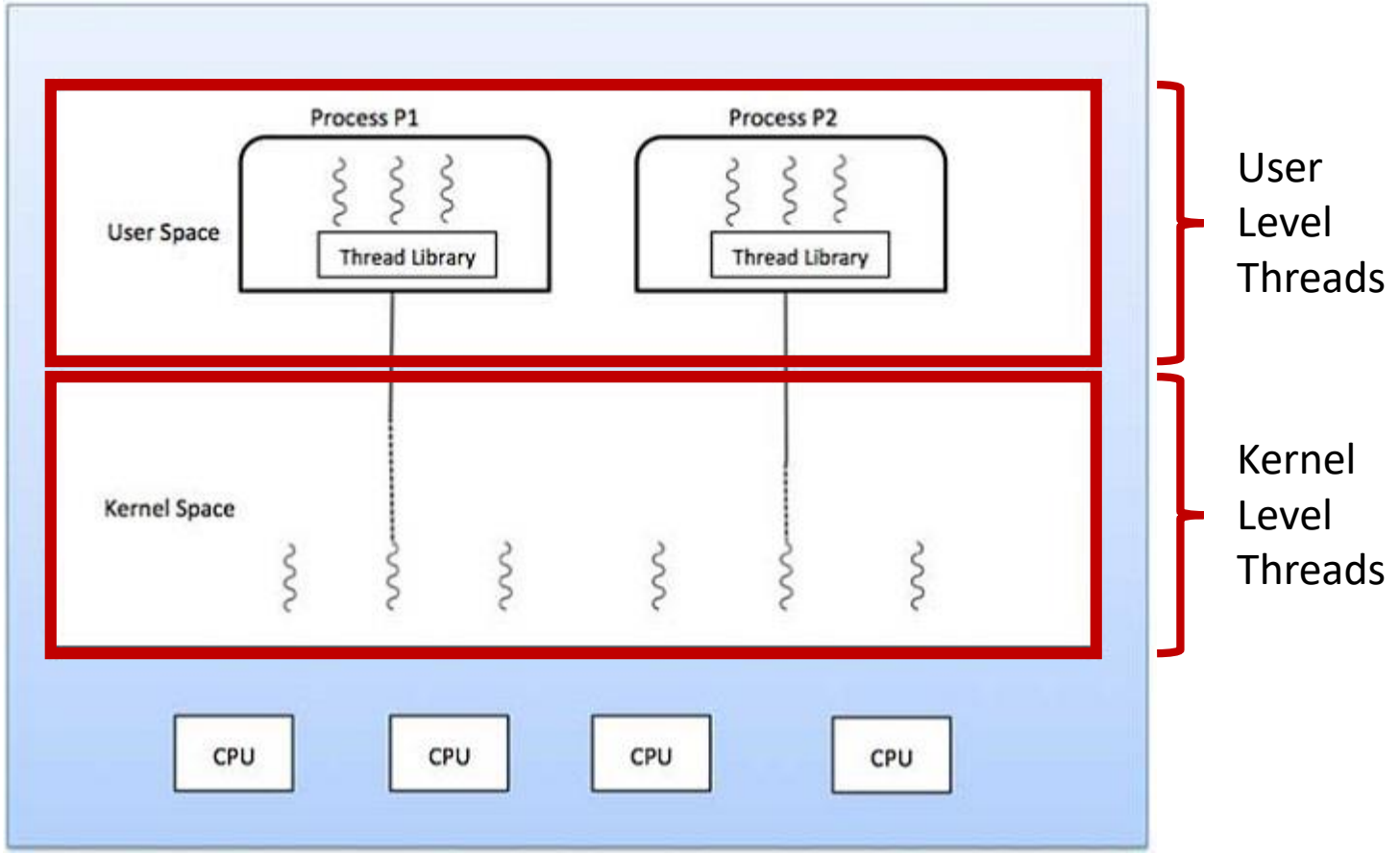
- **Similarities between Process & Thread**
  - Like processes, threads share CPU and only one thread is running at a time.
  - Like processes, threads within a process execute sequentially.
  - Like processes, thread can create children.
  - Like a traditional process, a thread can be in any one of several states: running, blocked, ready or terminated.
  - Like process, threads have Program Counter, Stack, Registers and State.

Process Vs. Thread

Process	Thread
Process means a program is in execution.	Thread means a segment of a process.
The process is not Lightweight.	Threads are Lightweight.
The process takes more time to terminate.	The thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes

Types of Threads

1. Kernel Level Thread
2. User Level Thread



User Level Thread Vs. Kernel Level Thread

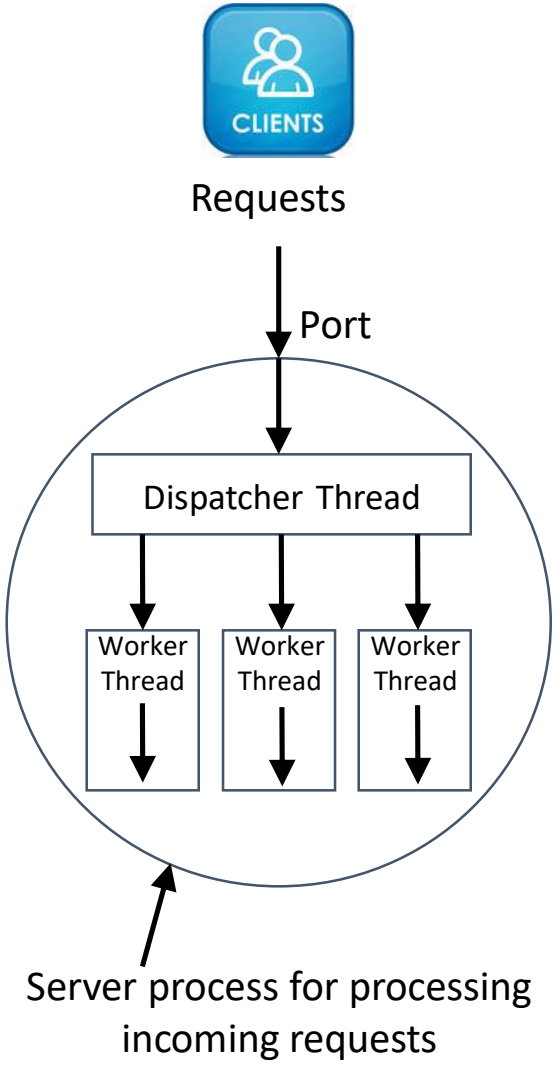
USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	Kernel threads are implemented by OS.
OS doesn’t recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complex.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Context switch requires hardware support.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread with in same process can continue execution.
Example : Java thread	Example : Window Solaris

Models for Organizing Threads

- Depending on the application's needs, the threads of a process of the application can be organized in different ways.
- Threads can be organized by three different models.
  1. Dispatcher/Worker model
  2. Team model
  3. Pipeline model

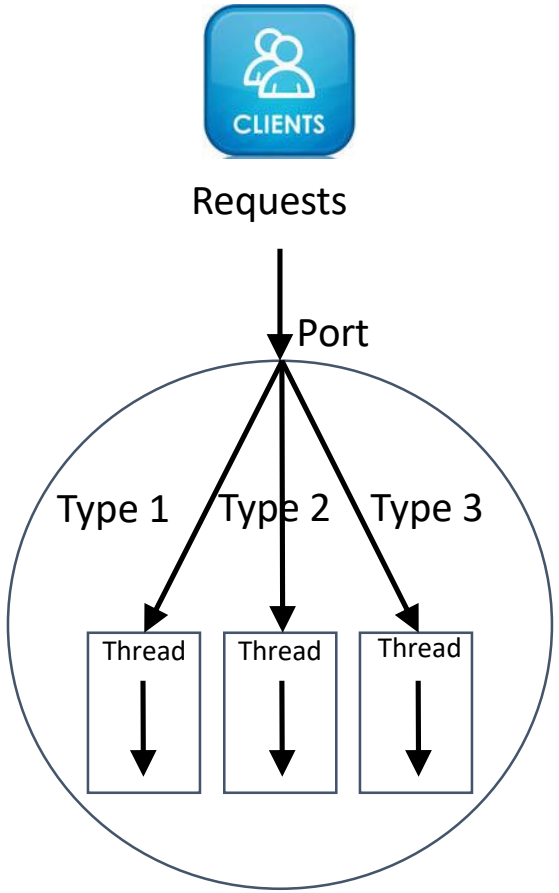
Dispatcher/Worker Model

- In this model, the process consists of a single dispatcher thread and multiple worker threads.
- The dispatcher thread:
  - Accepts requests from clients.
  - Examine the request.
  - Dispatches the request to one of the free worker threads for further processing of the request.
- Each worker thread works on a different client request.
- Therefore, multiple client requests can be processed in parallel.



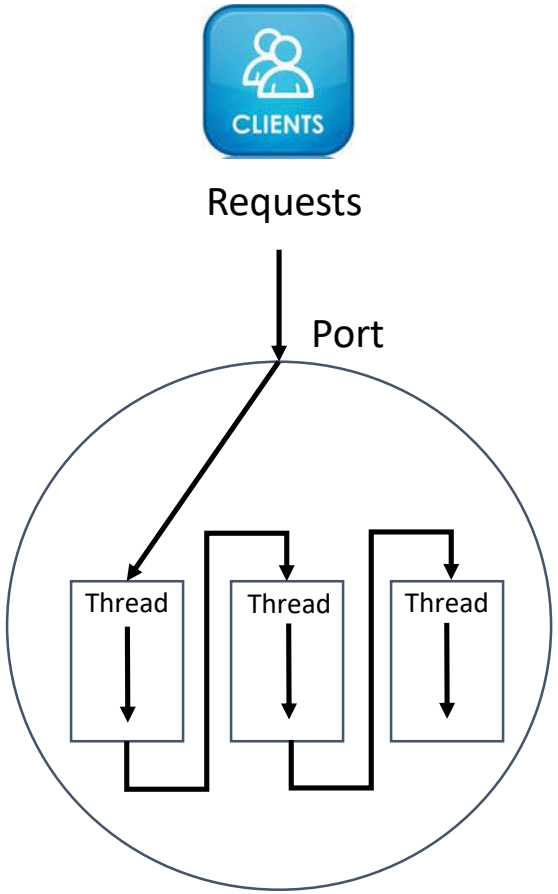
Team Model

- In this model, all threads behave as equal.
- Each thread gets and processes clients requests on its own.
- This model is often used for implementing specialized threads within a process.
- Each thread of the process is specialized in servicing a specific type of requests like copy, save, autocorrect.



Pipeline Model

- This model is useful for applications based on the producer-consumer model.
- The output data generated by one part of the application is used as input for another part of the application.
- The threads of a process are organized as a pipeline.
- The output data generated by the first thread is used for processing by the second thread, the output of the second thread is used for third thread, and so on.
- The output of the last thread in the pipeline is the final output of the process to which the threads belong.



Threads in Distributed system

- An important property of threads is that they can provide a convenient means of allowing blocking system calls without blocking the entire process in which the thread is running.
- This property makes threads particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time.
- We illustrate this point by taking a closer look at multithreaded clients and servers, respectively.
- **Multithreaded Clients**
- To establish a high degree of distribution transparency, distributed systems that operate in wide-area networks may need to conceal long interprocess message propagation times. The round-trip delay in a wide-area network can easily be in the order of hundreds of milliseconds, or sometimes even seconds.

- 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. In many cases, a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc. To fetch each element of a Web document, the browser has to set up a TCP/IP connection, read the incoming data, and pass it to a display component. Setting up a connection as well as reading incoming data are inherently blocking operations. When dealing with long-haul communication, we also have the disadvantage that the time for each operation to complete may be relatively long.
- 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. While the text is made available to the user, including the facilities for scrolling and such, the browser continues with fetching other files that make up the page, such as the images. The latter are displayed as they are brought in. The user need thus not wait until all the components of the entire page are fetched before the page is made available.

- In effect, it is seen that the Web browser is doing a number of tasks simultaneously. As it turns out, developing the browser as a multithreaded client simplifies matters considerably. 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. As is also illustrated in Stevens (1998), the code for each thread is the same and, above all, simple. Meanwhile, the user notices only delays in the display of images and such, but can otherwise browse through the document.
- There is another important benefit to using multithreaded Web browsers in which several connections can be opened simultaneously. In the previous example, several connections were set up to the same server. If that server is heavily loaded, or just plain slow, no real performance improvements will be noticed compared to pulling in the files that make up the page strictly one after the other.

- However, in many cases, Web servers have been replicated across multiple machines, where each server provides exactly the same set of Web documents. The replicated servers are located at the same site, and are known under the same name. When a request for a Web page comes in, the request is forwarded to one of the servers, often using a round-robin strategy or some other load-balancing technique (Katz et al., 1994). When using a multithreaded client, connections may be set up to different replicas, allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a nonreplicated server. This approach is possible only if the client can handle truly parallel streams of incoming data. Threads are ideal for this purpose.
- **Multithreaded Servers**
- Although there are important benefits to multithreaded clients, as we have seen, 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 uniprocessor 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. The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply. Here one thread, the dispatcher, reads incoming requests for a file operation. The requests are sent by clients to a well-known end point for this server. After examining the request, the server chooses an idle (i.e., blocked) worker thread and hands it the request.
- The worker proceeds by performing a blocking read on the local file system, which may cause the thread to be suspended until the data are fetched from disk. If the thread is suspended, another thread is selected to be executed. For example, the dispatcher may be selected to acquire more work. Alternatively, another worker thread can be selected that is now ready to run.

- Now consider how the file server might have been written in the absence of threads. 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. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.
- So far we have seen two possible designs: a multithreaded file server and a single-threaded file server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. 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 nonblocking calls to send and receive.
- In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table for every message sent and received. In effect, we are simulating threads and their stacks the hard way. The process is being operated as a finite-state machine that gets an event and then reacts to it, depending on what is in it.

- It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls (e.g., an RPC to talk to the disk) and still achieve parallelism. Blocking system calls make programming easier and parallelism improves performance. The single-threaded server retains the ease and simplicity of blocking system calls, but gives up some amount of performance. The finite-state machine approach achieves high performance through parallelism, but uses nonblocking calls, thus is hard to program.

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

Figure 3-4. Three ways to construct a server.

Introduction to Virtualization

- Threads and processes can be seen as a way to do more things at the same time. In effect, they allow us build (pieces of) programs that appear to be executed simultaneously. On a single-processor computer, this simultaneous execution is, of course, an illusion. As there is only a single CPU, only an instruction from a single thread or process will be executed at a time. By rapidly switching between threads and processes, the illusion of parallelism is created.
- This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as resource virtualization. This virtualization has been applied for many decades, but has received renewed interest as (distributed) computer systems have become more commonplace and complex, leading to the situation that application software is mostly always outliving its underlying systems software and hardware. In this section, we pay some attention to the role of virtualization and discuss how it can be realized.

Virtualization

- Multiprogrammed operating systems provide the illusion of simultaneous execution through ***resource virtualization***
  - Use software to make it look like concurrent processes are executing simultaneously
- Virtual machine technology creates separate virtual machines, capable of supporting multiple instances of different operating systems.
- Virtualization is a broad term that refers to the abstraction of computer resources.
- Virtualization creates an external interface that hides an underlying implementation
- Benefits:
  - Hardware changes faster than software
  - Compromised systems (internal failure or external attack) are isolated.
  - Run multiple different operating systems at the same time

Virtualization

Common uses of the term, divided into two main categories:

- Platform virtualization
- Resource virtualization

Platform virtualization:

- It involves the simulation of virtual machines.
- Platform virtualization is performed on a given hardware platform by "host" software (a control program), which creates a simulated computer environment (a virtual machine) for its "guest" software.
- The "guest" software, which is often itself a complete operating system, runs just as if it were installed on a stand-alone hardware platform.

Resource virtualization:

- It involves the simulation of combined, fragmented, or simplified resources.
- Virtualization of specific system resources, such as storage volumes, name spaces, and network resources.

Architectures of Virtual Machines

- To understand the differences in virtualization, it is important to realize that computer systems generally offer four different types of interfaces, at four different levels:
- 1. An interface between the hardware and software, consisting of machine instructions that can be invoked by any program.
- 2. An interface between the hardware and software, consisting of machine instructions that can be invoked only by privileged programs, such as an operating system.
- 3. An interface consisting of system calls as offered by an operating system.
- 4. 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.

- These different types are shown in Fig. 3-6. The essence of virtualization is to mimic the behavior of these interfaces.

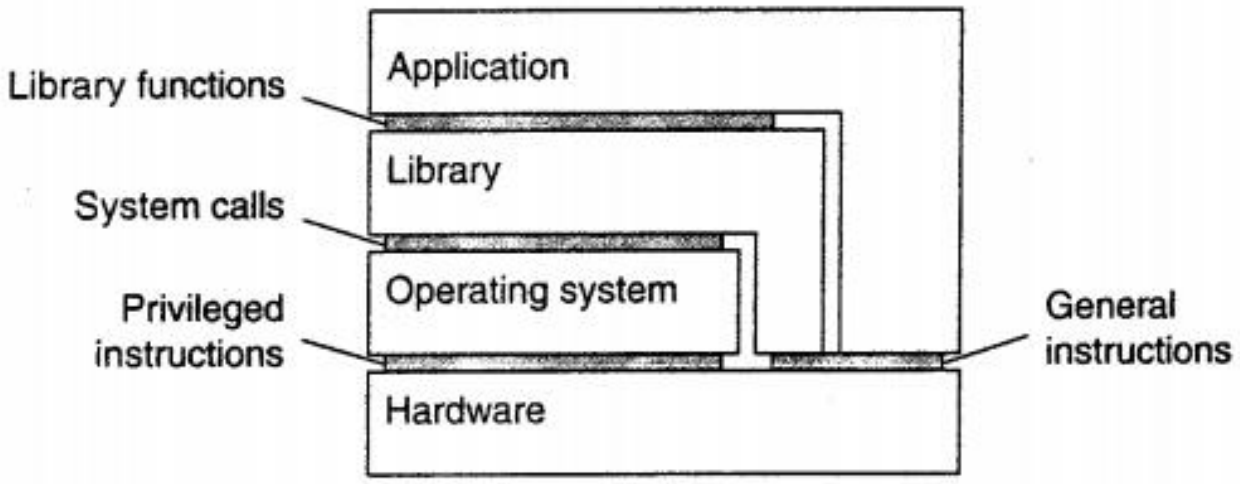
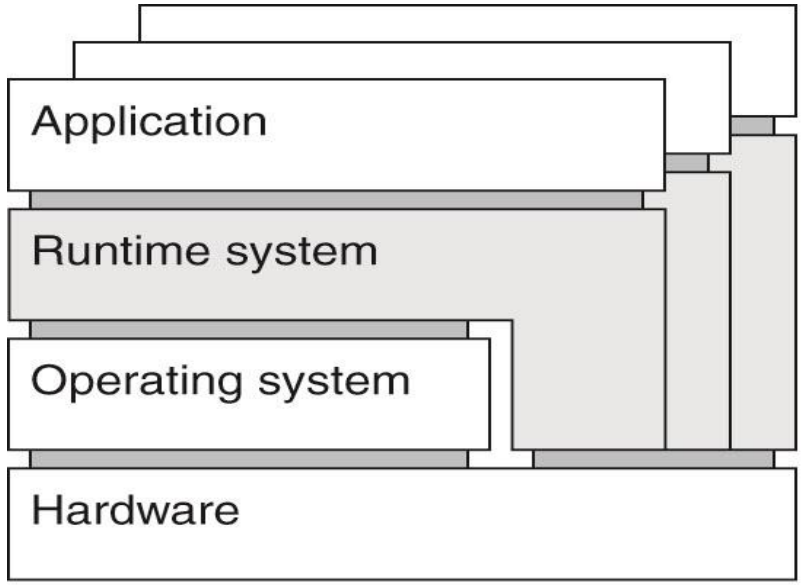


Figure 3-6. Various interfaces offered by computer systems.

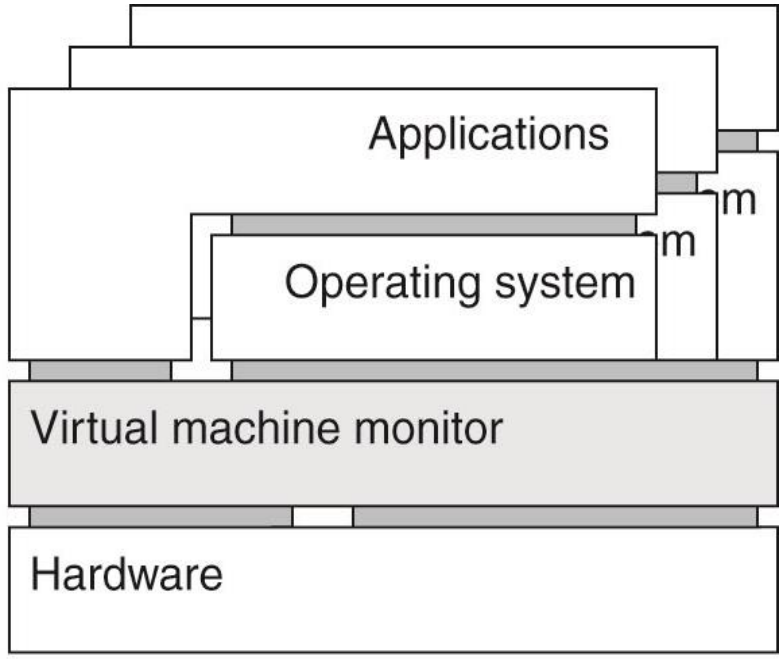
Two Ways to Virtualize

Process Virtual Machine



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

Virtual Machine Monitor



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

Clients

- A Program, Which interact with a human user or a remote servers
- Typically, the users interact with the client via a GUI
- The client is the machine (workstation or PC) running the front-end applications.
- It interacts with a user through the keyboard, display, and pointing device such as a mouse.
- The client has no direct data access responsibilities. It simply requests processes from the server and displays data managed by the server.

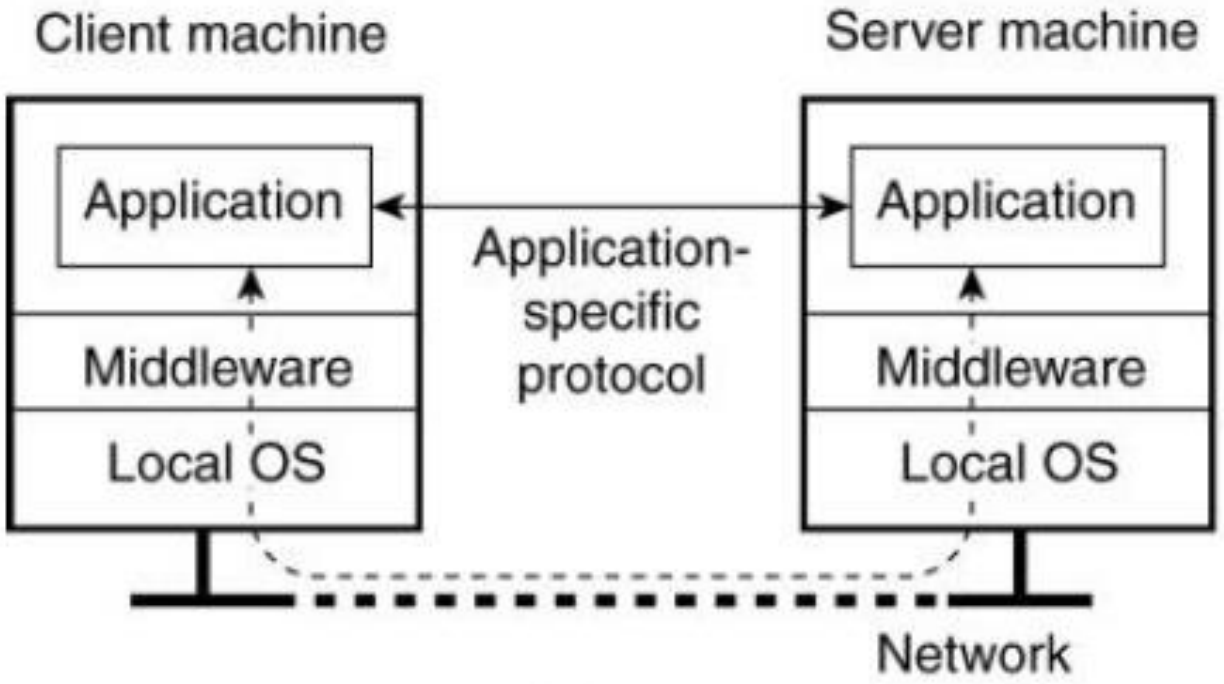
Networked User Interfaces

- A major task of client machines is to provide the means for users to interact with remote servers
- Two ways to support client-server interaction:

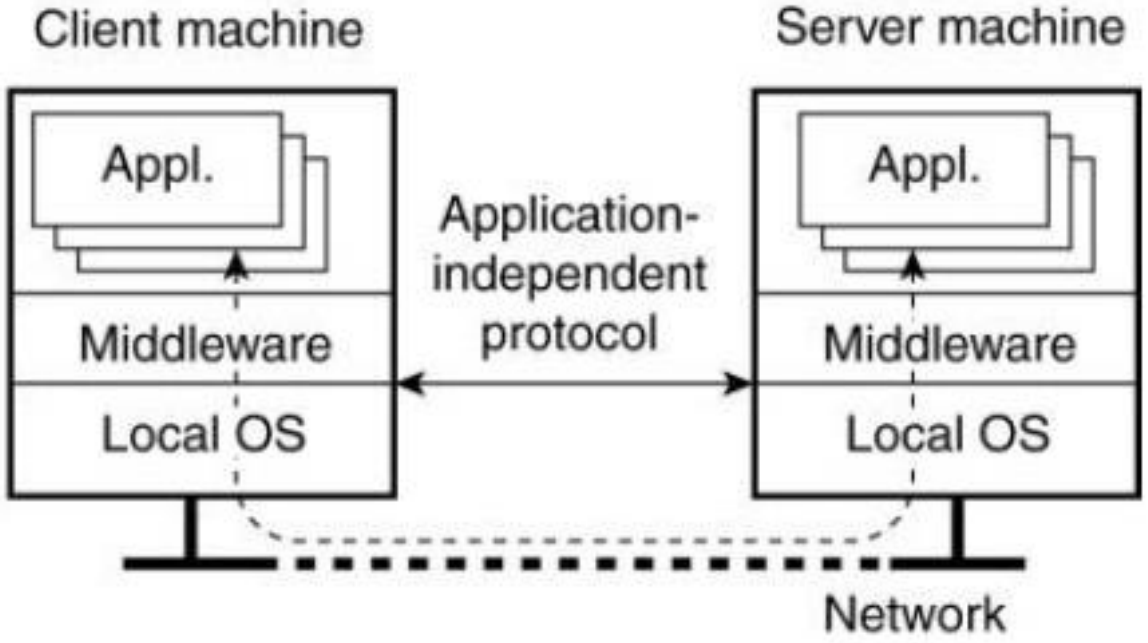
1. For each remote service - the client machine will have a separate counterpart that can contact the service over the network.
   - Example: an agenda running on a user's PDA that needs to synchronize with a remote, possibly shared agenda.
   - In this case, an application-level protocol will handle the synchronization
2. Provide direct access to remote services by only offering a convenient user interface.
   - The client machine is used only as a terminal with no need for local storage, leading to an application neutral solution.
   - In the case of networked user interfaces, everything is processed and stored at the server.
   - This thin-client approach is receiving more attention as Internet connectivity increases, and hand-held devices are becoming more sophisticated.

Networked User Interfaces

Networked application with its own protocol

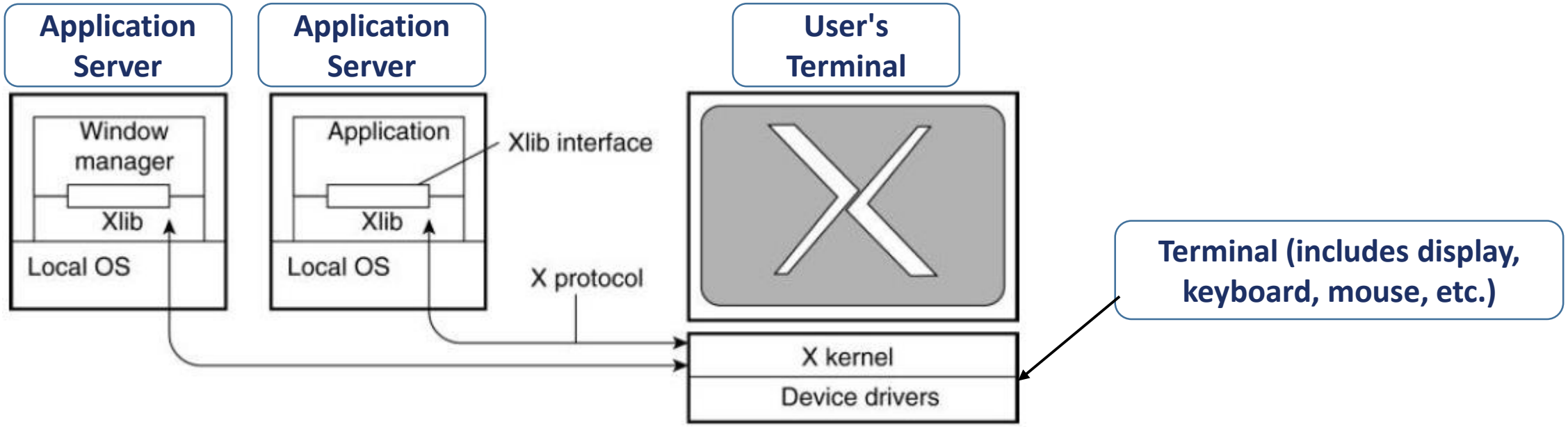


General solution to allow access to remote applications.



The X Window System

- Used to control bit-mapped terminals, which include a monitor, keyboard, and a pointing device such as a mouse.
- X kernel is heart of the system.
  - Contains all the terminal-specific device drivers - highly hardware dependent
  - X kernel offers a low-level interface for controlling the screen and for capturing events from the keyboard and mouse
  - This interface is made available to applications as a library called Xlib.



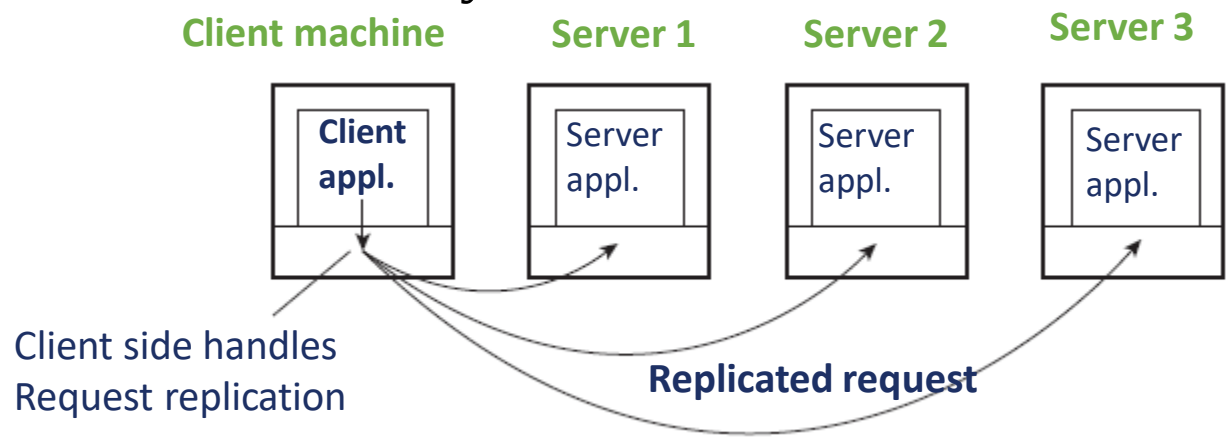
Compound Documents

- Modem user interfaces do a lot more than systems such as X or its simple applications. In particular, many user interfaces allow applications to share a single graphical window, and to use that window to exchange data through user actions. Additional actions that can be performed by the user include what are generally called drag-and-drop operations, and in-place editing, respectively.
- A typical **example** of **drag-and-drop functionality** is moving an icon representing a file A to an icon representing a trash can, resulting in the file being deleted. In this case, the user interface will need to do more than just arrange icons on the display: it will have to pass the name of the file A to the application associated with the trash can as soon as A's icon has been moved above that of the trash can application.
- **In-place editing** can best be illustrated by means of a document containing text and graphics. Imagine that the document is being displayed within a standard word processor. As soon as the user places the mouse above an image, the user interface passes that information to a drawing program to allow the user to modify the image.

- **For example**, the user may have rotated the image, which may effect the placement of the image in the document. The user interface therefore finds out what the new height and width of the image are, and passes this information to the word processor. The latter, in tum, can then automatically update the page layout of the document
- The key idea behind these user interfaces is the notion of a **compound document**, which can be defined as a collection of documents, possibly of very different kinds (like text, images, spreadsheets, etc.), which are seamlessly integrated at the user-interface level.
- A user interface that can handle compound documents hides the fact that different applications operate on different parts of the document.
- To the user, all parts are integrated in a seamless way. When changing one part affects other parts, the user interface can take appropriate measures, for example, by notifying the relevant applications.

Client-Side Software for Distribution Transparency

- Access transparency
  - Handled through client-side stubs for RPCs
  - 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 for Distribution Transparency

- Failure transparency
  - Mask server and communication failures: done through client middleware,
  - e.g. connect to another machine.
- Concurrency transparency
  - Handled through special intermediate servers, notably transaction monitors, and requires less support from client software.

Servers

- A server is a process implementing a specific service on behalf of a collection of clients.
  - Each server is organized in the same way:
  - It waits for an incoming request from a client ensures that the request is fulfilled
  - It waits for the next incoming request.
- Types of server
  1. Iterative server
  2. Concurrent server

Iterative server and Concurrent server

Iterative server

- Iterative server handles request, then returns results to the client; any new client requests must wait for previous request to complete (also useful to think of this type of server as sequential).
- Process one request at a time
- When an iterative server is handling a request, other connections to that port are blocked.
- The incoming connections must be handled one after another.
- Iterative servers support a single client at a time.
- Much easier to build, but usually much less efficient

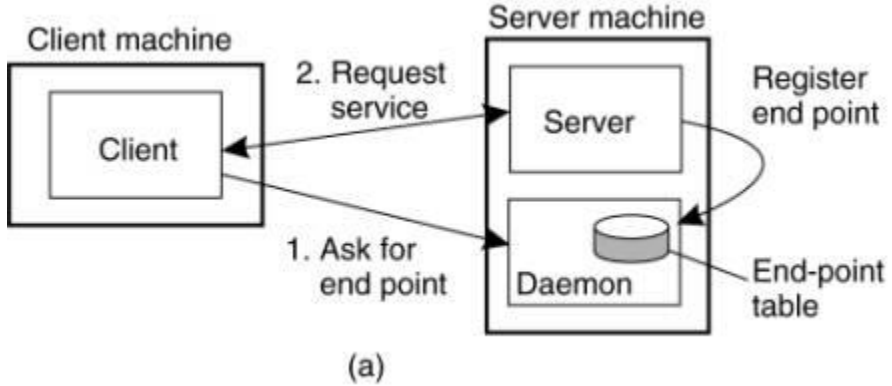
Concurrent server

- Process multiple requests simultaneously.
- Concurrent servers support multiple clients concurrently (may or may not use concurrent processes)
- Clients queue for connection, then are served concurrently. The concurrency reduces latency significantly.
- Concurrent server does not handle the request itself; a separate thread or sub-process handles the request and returns any results to the client; the server is then free to immediately service the next client (i.e., there's no waiting, as service requests are processed in parallel).
- A multithreaded server is an example of a concurrent server.

- **Where do clients contact a server?**
- 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?
- 1. Globally assign end points for well-known services.
- Examples:
- 1. Servers that handle Internet FTP requests always listen to TCP port 21.
- 2. An HTTP server for the World Wide Web will always listen to TCP port 80.
- These end points have been assigned by the Internet Assigned Numbers Authority (IANA).
- With assigned end points, the client only needs to find the network address of the machine where the server is running.

Client-to-server binding using a daemon

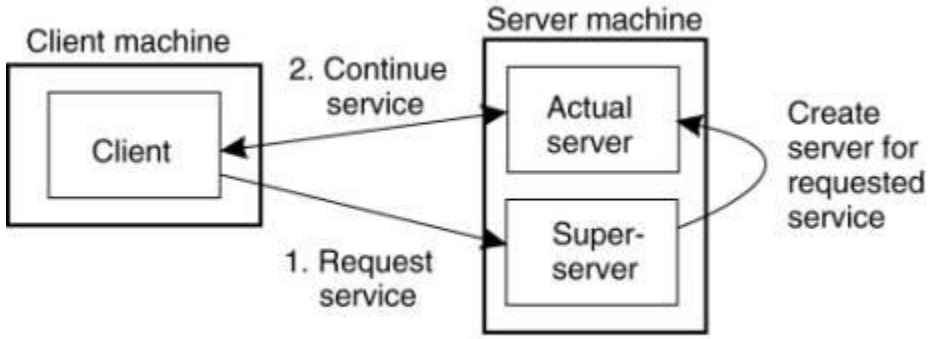
- Many services that do not require a pre-assigned end point.
- Example: A time-of-day server may use an end point that is dynamically assigned to it by its local operating system.
- A client will look need to up the end point.
- Solution: a daemon running on each machine that runs servers.
- The daemon 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 (Figure(a))



(a)

Client-to-server binding using a superserver

- Common to associate an end point with a specific service.
- Implementing each service by means of a separate server may be a waste of resources.
- Example: UNIX system
- many servers run simultaneously, with most of them passively waiting for a client request.
- Instead of having to keep track of so many passive processes, it is often more efficient to have a single superserver listening to each end point associated with a specific service, (Figure (b))



(b)

Stateless server

- A stateless server is a server that treats each request as an independent transaction that is unrelated to any previous request.
- Example: A Web server is stateless.
  - It merely responds to incoming HTTP requests, which can be either for uploading a file to the server or (most often) for fetching a file.
  - When the request has been processed, the Web server forgets the client completely.
  - The collection of files that a Web server manages (possibly in cooperation with a file server), can be changed without clients having to be informed.

Form of a stateless design - **soft state**.

- ↪ The server promises to maintain state on behalf of the client, but only for a limited time.
- ↪ After that time has expired, the server falls back to default behavior, thereby discarding any information it kept on account of the associated client.

Stateful server

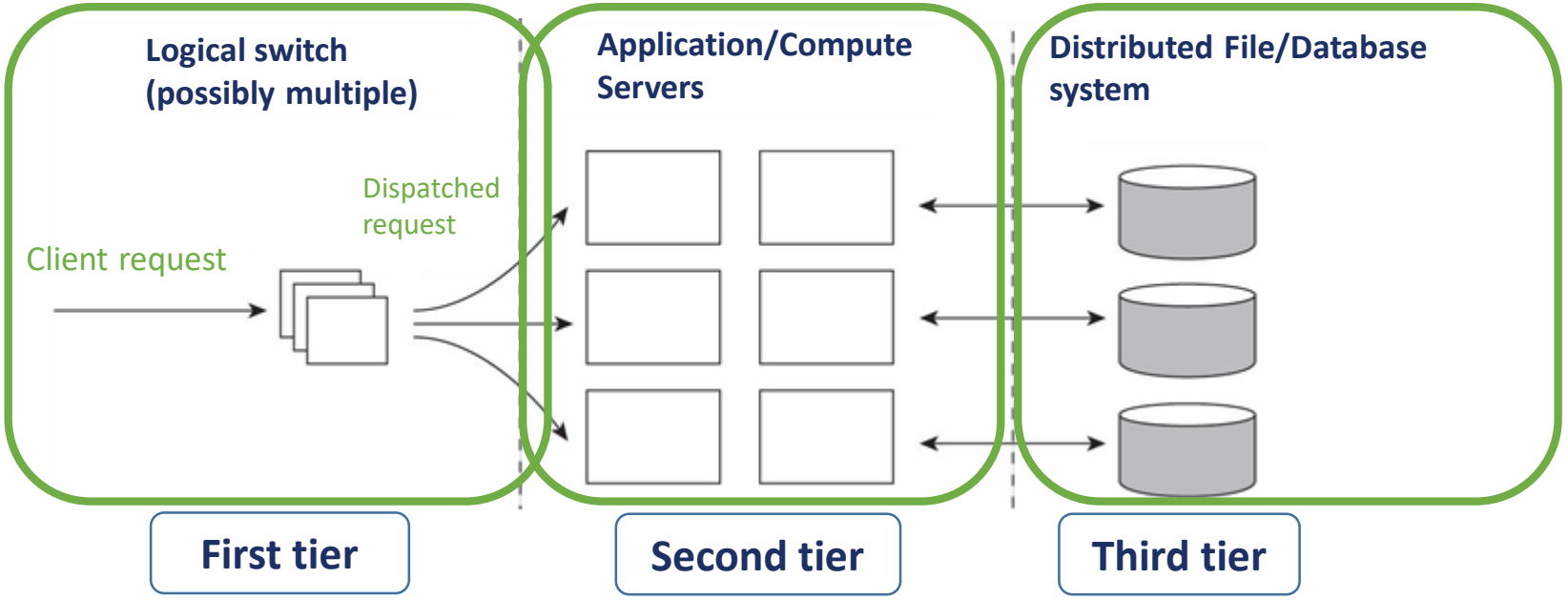
- A stateful server remembers client data (state) from one request to the next.
- Information needs to be explicitly deleted by the server.
- Example:
  - A file server that allows a client to keep a local copy of a file, even for performing update operations.
  - The server maintains a table containing (client, file) entries.
  - This table allows the server to keep track of which client currently has the update permissions on which file and the most recent version of that file
- Improves performance of read and write operations as perceived by the client

Server Clusters

- A server cluster is a collection of machines connected through a network, where each machine runs one or more servers.
- A server cluster is logically organized into three tiers
  - First tier
  - Second tier
  - Third tier

Server Clusters

- **First tier** - consists of a (logical) switch through which client requests are routed
  - The switch (access/replication transparency)
- **Second tier** - application processing
  - Cluster computing
  - Enterprise server clusters
- **Third tier** - data-processing servers - notably file and database servers
  - For other applications, the major part of the workload may be here



- **Issue**: When a server cluster offers, multiple different machines may run different application servers.
- The switch will have to be able to distinguish services or otherwise it cannot forward requests to the proper machines.
- Many second-tier machines run only a single application.
- This limitation comes from dependencies on available software and hardware, but also that different applications are often managed by different administrators.
- Consequence - certain machines are temporarily idle, while others are receiving an overload of requests.
- **Solution**: Temporarily migrate services to idle machines to balance load.
- Use virtual machines allowing a relative easy migration of code to real machines.

Distributed Servers

- The server clusters discussed so far are generally rather statically configured. In these clusters, there is often an separate administration machine that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.
- As we mentioned, most server clusters offer a single access point. When that point fails, the cluster becomes unavailable. 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.
- Having stability, like a long-living access point, is a desirable feature from a client's and a server's perspective. On the other hand, it also desirable to have a high degree of flexibility in configuring a server cluster, including the switch. This observation has lead to a design of a distributed server which effectively is nothing but a possibly dynamically changing set of machines, with also possibly varying access points, but which nevertheless appears to the outside world as a single, powerful machine.

- 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.

Code Migration

- Traditionally, communication in distributed systems is concerned with exchanging data between processes.
- Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target
- Process migration in which an entire process is moved from one machine to another
- Code migration is often used for load distribution, reducing network bandwidth, dynamic customization, and mobile agents.
- Code migration increases scalability, improves performance, and provides flexibility.
- Reasons for Code Migration:
  - Performance
  - Flexibility

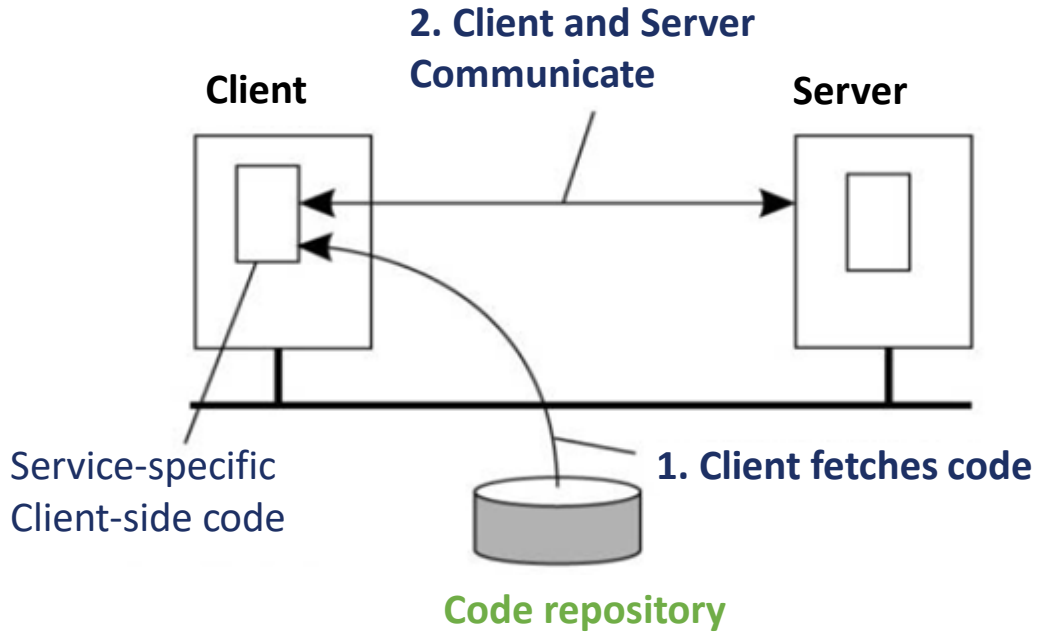
Performance in Code Migration

- Overall system performance can be improved if processes are moved from heavily-loaded to lightly loaded machines.
- How system performance is improved by code migration?
  - Using load distribution algorithms
  - Using qualitative reasoning
    - Migrating parts of the client to the server
    - Migrating parts of the server to the client

Flexibility in Code Migration

- The traditional approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed.
- For example,
  - Suppose a client program uses some proprietary APIs for doing some tasks that are rarely needed, and because of the huge size of the necessary API files, they are kept in a server.
  - If the client ever needs to use those APIs, then it can first dynamically download the APIs and then use them.

- Advantage of this model: Clients need not have all the software preinstalled to do common tasks.
- Disadvantage of this model : Security - blindly trusting that the downloaded code implements only the advertised APIs while accessing your unprotected hard disk



Models for Code Migration

- To get a better understanding of the different models for code migration, we use a framework described in Fuggetta et al. (1998).
- In this framework, a process consists of three segments.
  1. The code segment: It is the part that contains the set of instructions that make up the program that is being executed.
  2. The resource segment: It contains references to external resources needed by the process, such as files, printers, devices, other processes, and so on.
  3. The execution segment: It is used to store the current execution state of a process, consisting of private data, the stack, and, of course, the program counter.

Weak Mobility Vs. Strong Mobility

Parameters	Weak Mobility	Strong Mobility
Definition	In this model, it is possible to transfer only the code segment, along with perhaps some initialization data .	In contrast to weak mobility, in systems that support strong mobility the execution segment can be transferred as well .
Characteristic Feature	A transferred program is always started from its initial state	A running process can be stopped, subsequently moved to another machine, and then resume execution where it left off .
Example	Java applets – which always start execution from the beginning	D'Agents *https://sci-hub.hkvisa.net/10.1002/spe.449*
Benefit	Simplicity	Much more general than weak mobility

Migration Initiation

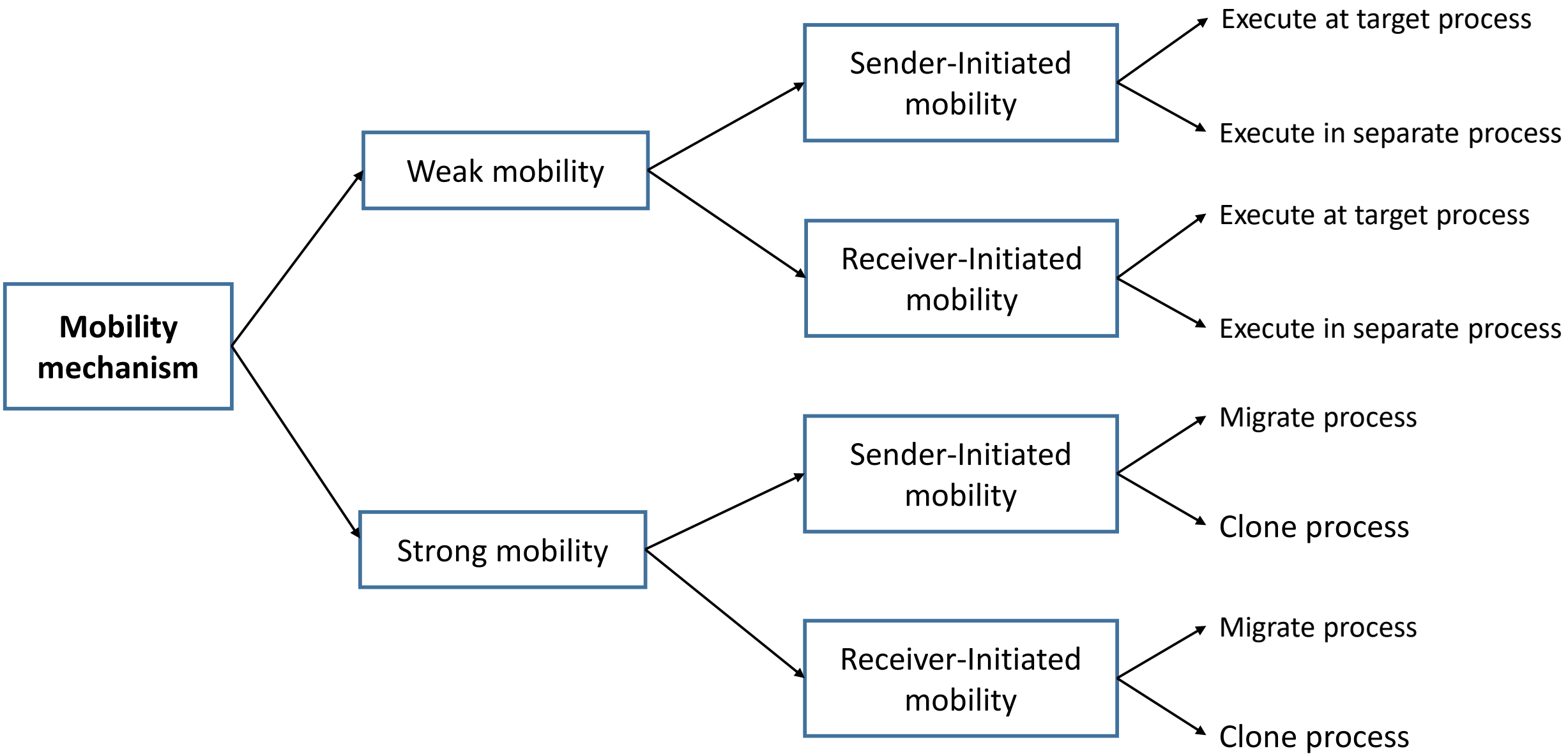
Sender-Initiated Migration

- It is initiated at the machine where the code currently resides or is being executed
- Examples:
  - Uploading programs to a compute server.
  - Sending a search program across the Internet to a web database server to perform the queries at that server.

Receiver-Initiated Migration

- The initiative for code migration is taken by the target machine
- Example: Java applets.

Alternatives for code migration



Execute Migrated Code for weak mobility

- In the case of weak mobility, it also makes a difference if the migrated code is executed by the target process, or whether a separate process is started.
- For example, Java applets are simply downloaded by a web browser and are executed in the browser's address space.
- Benefit for executing code at target process: There is no need to start a separate process, thereby avoiding communication at the target machine.
- Drawback for executing code at target process: The target process needs to be protected against malicious or inadvertent code executions.

Migrate or Clone Process (for strong mobility)

- Instead of moving a running process, also referred to as process migration, strong mobility can also be supported by remote cloning.
- In contrast to process migration, cloning yields an exact copy of the original process, but now running on a different machine.
- The cloned process is executed in parallel to the original process.
- Benefit of cloning process: The model closely resembles the one that is already used in many applications. The only difference is that the cloned process is executed on a different machine.
- In this sense, migration by cloning is a simple way to improve distribution transparency.

Migration and Local Resources

- 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.
- When the process moves to another location, it will have to give up the port and request a new one at the destination.
- Process-to-Resource Bindings
  1. Binding by Identifier
  2. Binding by Value
  3. Binding by Type

Process-to-Resource Bindings

Binding by Identifier

- A process refers to a resource by its identifier. In that case, the process requires precisely the referenced resource, and nothing else.
- Examples:
  - A URL to refer to a specific web site.
  - Local communication endpoints (IP, port etc.)

Binding by Value

- Only the value of a resource is needed. In that case, the execution of the process would not be affected if another resource would provide the same value.
- Example: Standard libraries for programming languages.

Binding by Type

- A process indicates it needs only a resource of a specific type.
- Example: References to local devices, such as monitors, printers and so on.

Resource Types

- When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding.
- Resource Types:

  Unattached resources :

  - They can be easily moved between different machines.
  - Example: Typically (data) files associated only with the program that is to be migrated.

  Fastened resources:

  - They may be copied or moved, but only at relatively high costs.
  - Example: Local databases and complete web sites.
  - Although such resources are, in theory, not dependent on their current machine, it is often infeasible to move them to another environment.

  Fixed resources:

  - They are intimately bound to a specific machine or environment and cannot be moved.
  - Example: Local devices, local communication end points

Managing Local Resources

- Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code

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 global system wide reference
MV Move the resource
CP Copy the value of the resource
RB Re-bind to a locally available resource

- Examples:
- A process is bound to a resource by identifier.
- i. When the resource is unattached, it is best to move it along with the migrating code.
- ii. When the resource is shared by other processes establish a global reference
- · a reference that can cross machine boundaries.
- e.g. a URL.
- iii. When the resource is fastened or fixed, the best solution is to create a global reference.

- **Problems with global reference**
- Reference may be expensive to construct.

- e.g. a program that generates high-quality images for a dedicated multimedia workstation.
- Fabricating high-quality images in real time is a compute-intensive task, for which reason the program may be moved to a high-performance compute server.
- Establishing a global reference to the multimedia workstation means setting up a communication path between the compute server and the workstation.
- Significant processing involved at both the server and the workstation to meet the bandwidth requirements of transferring the images.
- Nt result - moving the program to the compute server is not such a good idea, only because the cost of the global reference is too high.
- Migrating a process that makes use of a local communication end point.
- Here is a fixed resource to which the process is bound by the identifier.

- Two solutions:
- 1. Let the process set up a connection to the source machine after it has migrated and install a separate process at the source machine that forwards all incoming messages.
- Drawback - whenever the source machine malfunctions, communication with the migrated process may fail.
- 2. Have all processes that communicated with the migrating process, change their global reference, and send messages to the new communication end point at the target machine.
- **e.g. bindings by value.**
- A fixed resource.
- The combination of a fixed resource and binding by value occurs when a process assumes that memory can be shared between processes.
- Establishing a global reference would require a distributed form of shared memory.
- In many cases, this is not really a viable or efficient solution.

- Fastened resources.
- Fastened resources that are referred to by their value, are typically runtime libraries.
- Normally, copies of such resources are readily available on the target machine, or should otherwise be copied before code migration takes place.
- Establishing a global reference is a better alternative when huge amounts of data are to be copied, as may be the case with dictionaries and thesauruses in text processing systems.
- Unattached resources.
- The best solution is to copy (or move) the resource to the new destination, unless it is shared by a number of processes.
- In the latter case, establishing a global reference is the only option.

- **e.g. bindings by type.**
- Irrespective of the resource-to-machine binding, the solution is to rebind the process to a locally available resource of the same type.
- If the resource is not available, must copy or move the original one to the new destination, or establish a global reference.

Migration in Heterogeneous Systems

- Distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture.
- Migration requires that each platform be supported - the code segment can be executed on each platform.
- Must ensure that the execution segment can be properly represented at each platform.
- **Migrate entire computing environments**
- Compartmentalize the overall environment to provide processes in the same part their own view on their computing environment.
- Can decouple a part from the underlying system and migrate it to another machine.
- Migration would provide a form of strong mobility for processes, as they can then be moved at any point during their execution, and continue where they left off when migration completes.
- Solves bindings to local resources problem - local resources become part of the environment that is being migrated.
-

- **Why migrate entire environments?**
- It allows continuation of operation while a machine needs to be shutdown.
- e.g. in a server cluster, the systems administrator may decide to shut down or replace a machine
- can temporarily freeze an environment, move it to another machine (where it sits next to other, existing environments), and unfreeze it.
- **Example: Real-time migration of a virtualized operating system (Clark et al. 2005).**
- Useful in a cluster of servers where a tight coupling is achieved through a single, shared local-area network.
- Migration involves two major problems:
- real-time migration of a virtualized operating system
- migrating bindings to local resources.

- Real-time migration of a virtualized operating system:
- Three ways to handle migration (which can be combined):
- 1. Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.
- 2. Stopping the current virtual machine; migrate memory, and start the new virtual machine.
- 3. Letting the new virtual machine pull in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.
- Option #2 may lead to unacceptable downtime if the migrating virtual machine is running a live service.
- Option #3 extensively prolong the migration period lead to poor performance because it takes a long time before the working set of the migrated processes has been moved to the new machine.

- Clark et al. propose to use a pre-copy approach which combines the Option #1, along with a brief stop-and-copy phase as represented by the Option #2.
- This combination can lead to service downtimes of 200 ms or less.