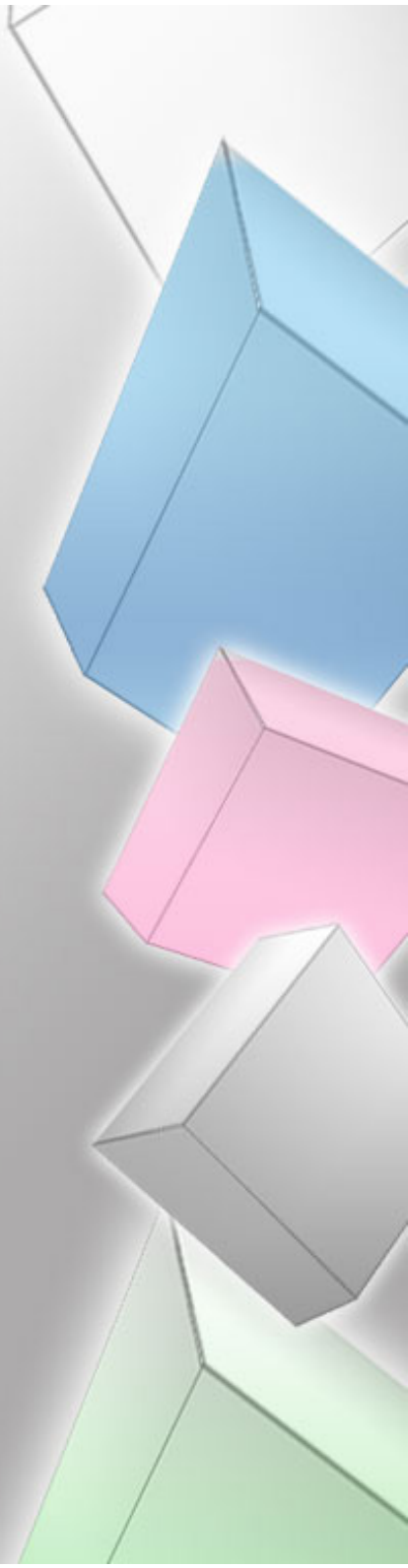
Advanced Topics in Operating Systems

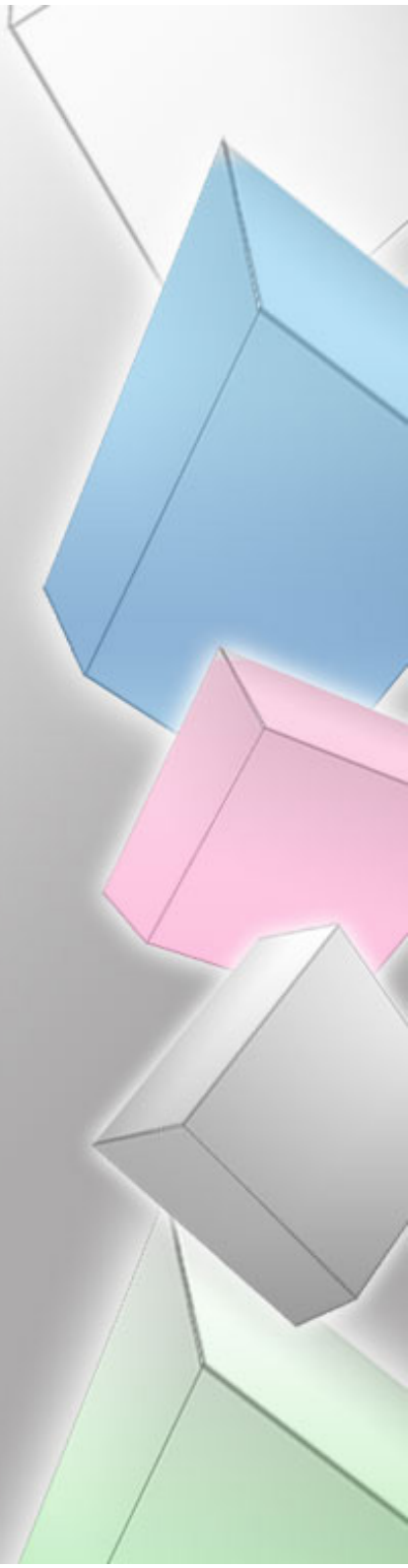
MSc in Computer Science
UNYT-UoG

Dr. Marenglen Biba
18-19-20 December 2009



Lesson 3

01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems



Processes

- The concept of a process is fundamental in the field of operating systems where it is generally defined as a program in execution.
- From an operating-system perspective, the management and scheduling of processes are perhaps the most important issues to deal with.
- However, when it comes to distributed systems, other issues turn out to be equally or more important.

Process granularity

- Although processes form a building block in distributed systems, practice indicates that the granularity of processes as provided by the operating systems on which distributed systems are built is not sufficient.

- Instead, it turns out that having a finer granularity in the form of multiple threads of control per process makes it much easier to build distributed applications and to attain better performance.

- In this section, we take a closer look at the role of threads in distributed systems and explain why they are so important.

Processes and threads

- We build virtual processors in software, on top of physical processors:
  - Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.
  - Thread: A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.
  - Process: A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

Concurrency transparency

- An important issue is that the operating system takes great care to ensure that independent processes cannot maliciously or inadvertently affect the correctness of each other's behavior.

- In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent.

Context switching

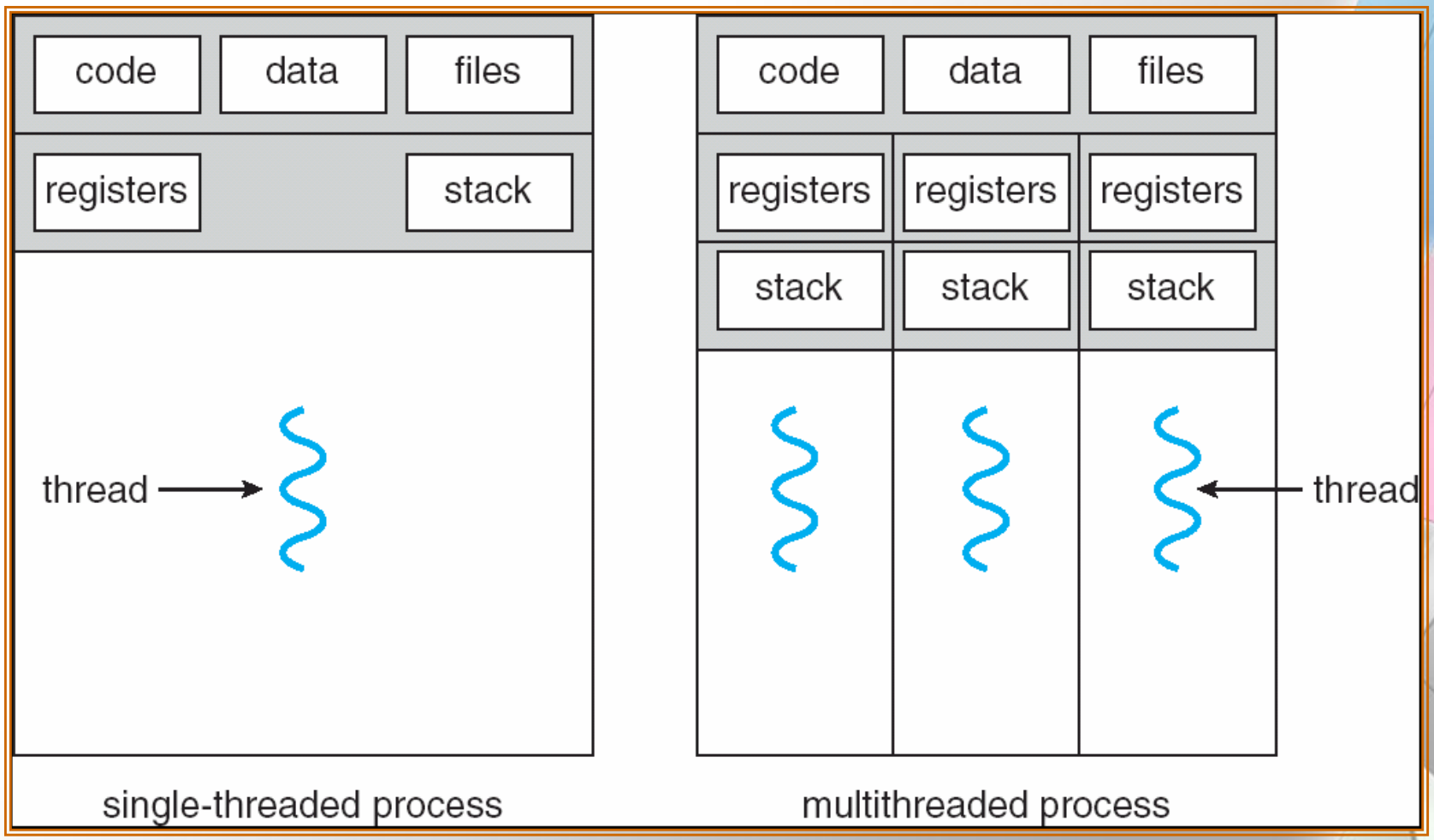
- The concurrency transparency comes at a relatively high price. For example, each time a process is created, the operating system must create a complete independent address space.
  - Allocation can mean initializing memory segments by, for example, zeroing a data segment, copying the associated program into a text segment, and setting up a stack 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 lookaside buffer (TLB).

Threads

- A thread is the basic unit of CPU utilization
- A thread is a flow of control within a process.
- A thread comprises
  - A thread ID
  - Program counter
  - Register set
  - Stack
- Shares with the other threads belonging to the same process
  - Code section
  - Data section
  - Other operating systems resources: files and signals

Single and Multithreaded Processes



Benefits of threads

- **Responsiveness**
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
  - For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.
- **Resource Sharing**
  - By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Benefits

- **Economy**
  - Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
  - Empirically finding the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads.
  - In Solaris, for example, creating a process is about **thirty times slower** than is creating a thread, and context switching is about **five** times slower.

Benefits

- **Utilization of multiprocessor architectures.**
  - The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.
  - A single threaded process can only run on one CPU, no matter how many are available.
  - Multithreading on a multi-CPU machine increases concurrency.

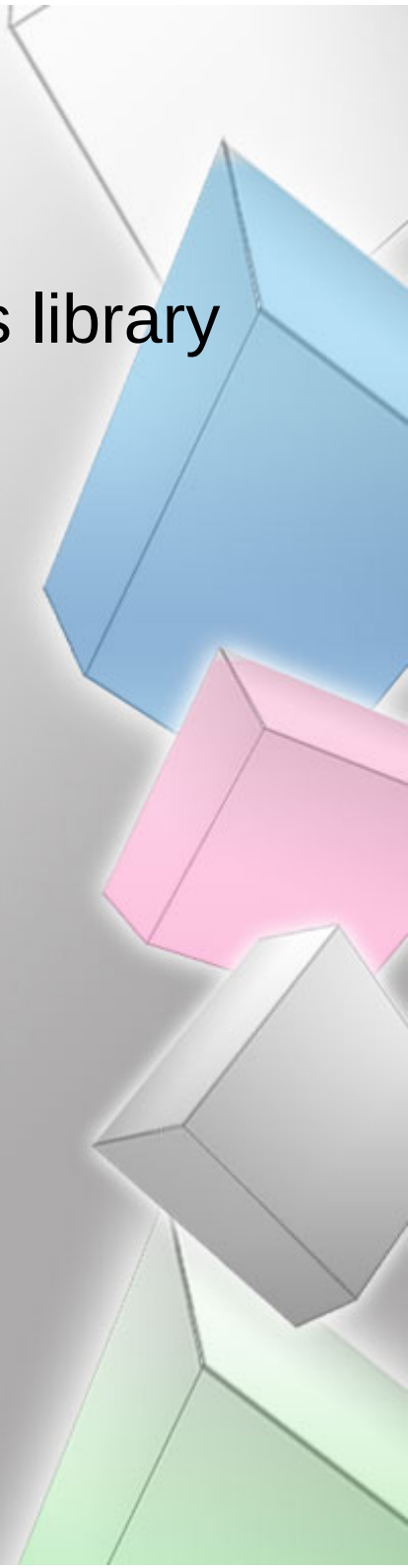
User and Kernel Threads

- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
  - User threads are supported above the kernel and are managed without kernel support,
  - Kernel threads are supported and managed directly by the operating system.
- Virtually all contemporary operating systems-including Windows XP, Linux, Mac OS x, Solaris, and Tru64 UNIX (formerly Digital UNIX)-support kernel threads.
- Ultimately, there must exist a relationship between user threads and kernel threads.

User Threads

- Thread management done by user-level threads library

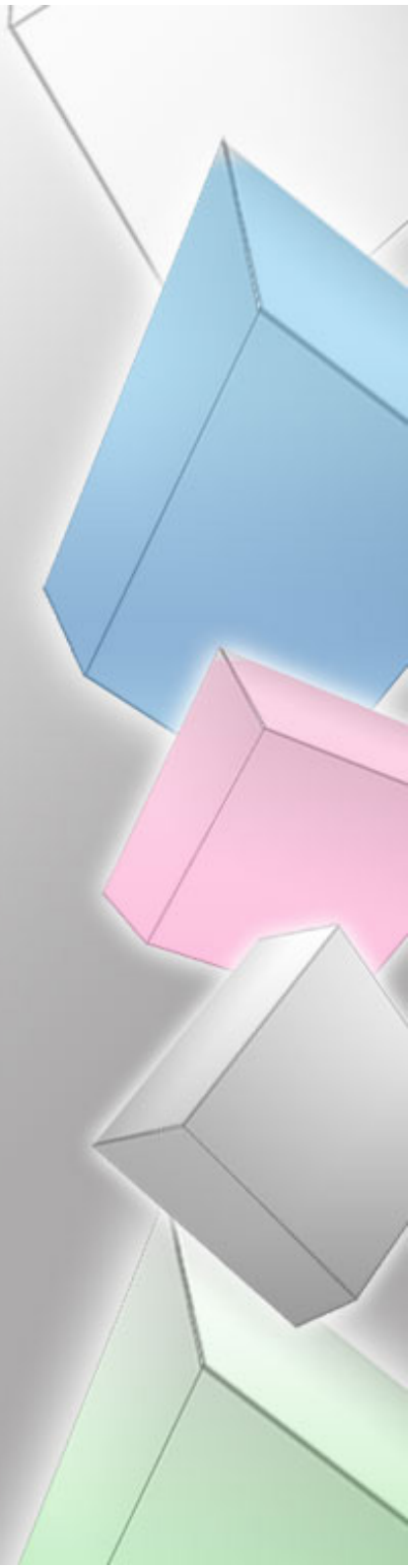
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads



Kernel Threads

- Supported by the Kernel

- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X



Threads and Context Switching

- Like a process, a thread executes its own piece of code, independently from other threads.
- However, in contrast to processes, no attempt is made to achieve a high degree of concurrency transparency if this would result in performance degradation.
- Therefore, a thread system generally maintains only the minimum information to allow a CPU to be shared by several threads. In particular, a thread context often consists of nothing more than the CPU context, along with some other information for thread management.

Threads and Context Switching

- Threads share the same address space.
  - Thread context switching can be done entirely independent of the operating system.

- Process switching is generally more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.

- Creating and destroying threads is much cheaper than doing so for processes.

Thread usage in non distributed systems

- Document Processing
  - Typing and spell-checking
- Excel
  - Updating cells functionally dependent and continuing to work or back-up
- Another advantage of multithreading is that it becomes possible to exploit parallelism when executing the program on a multiprocessor system.
  - In this case, 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.

Multithreading for large applications

- Multithreading is also 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 interprocess communication (IPC) mechanisms. For UNIX systems, these mechanisms typically include (named) pipes, message queues, and shared memory segments.
  - The major drawback of all IPC mechanisms is that communication often requires extensive context switching.

Context Switching in IPC

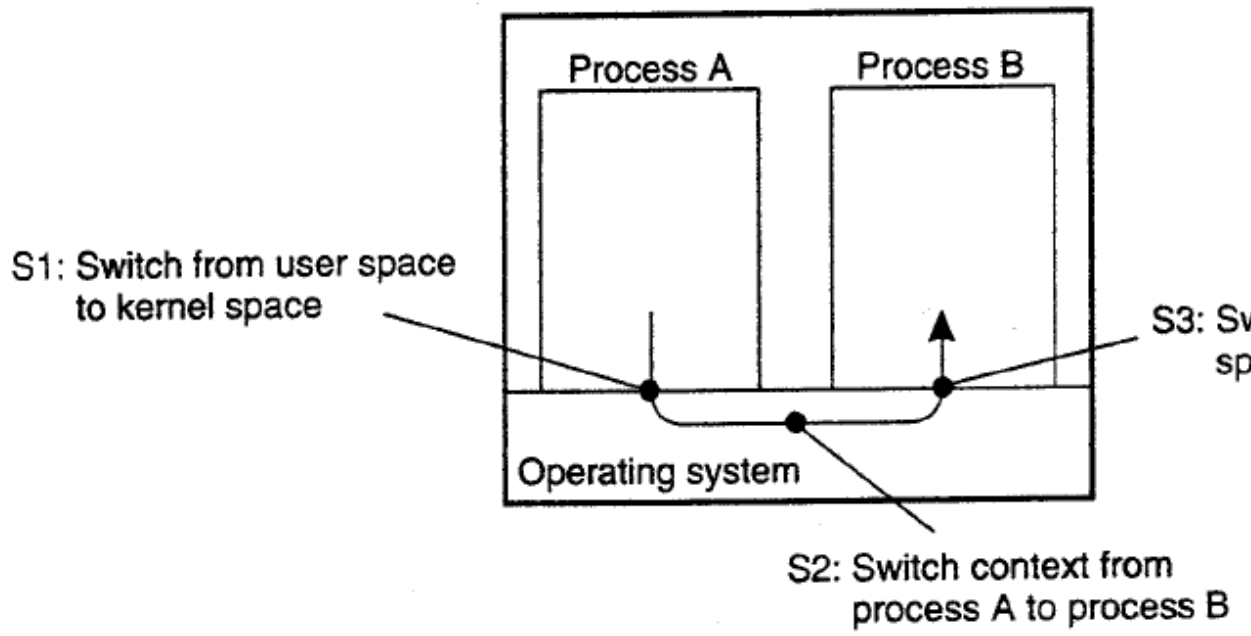


Figure 3-1. Context switching as the result of IPC.

Because IPC requires kernel intervention, a process will generally first have to switch from user mode to kernel mode

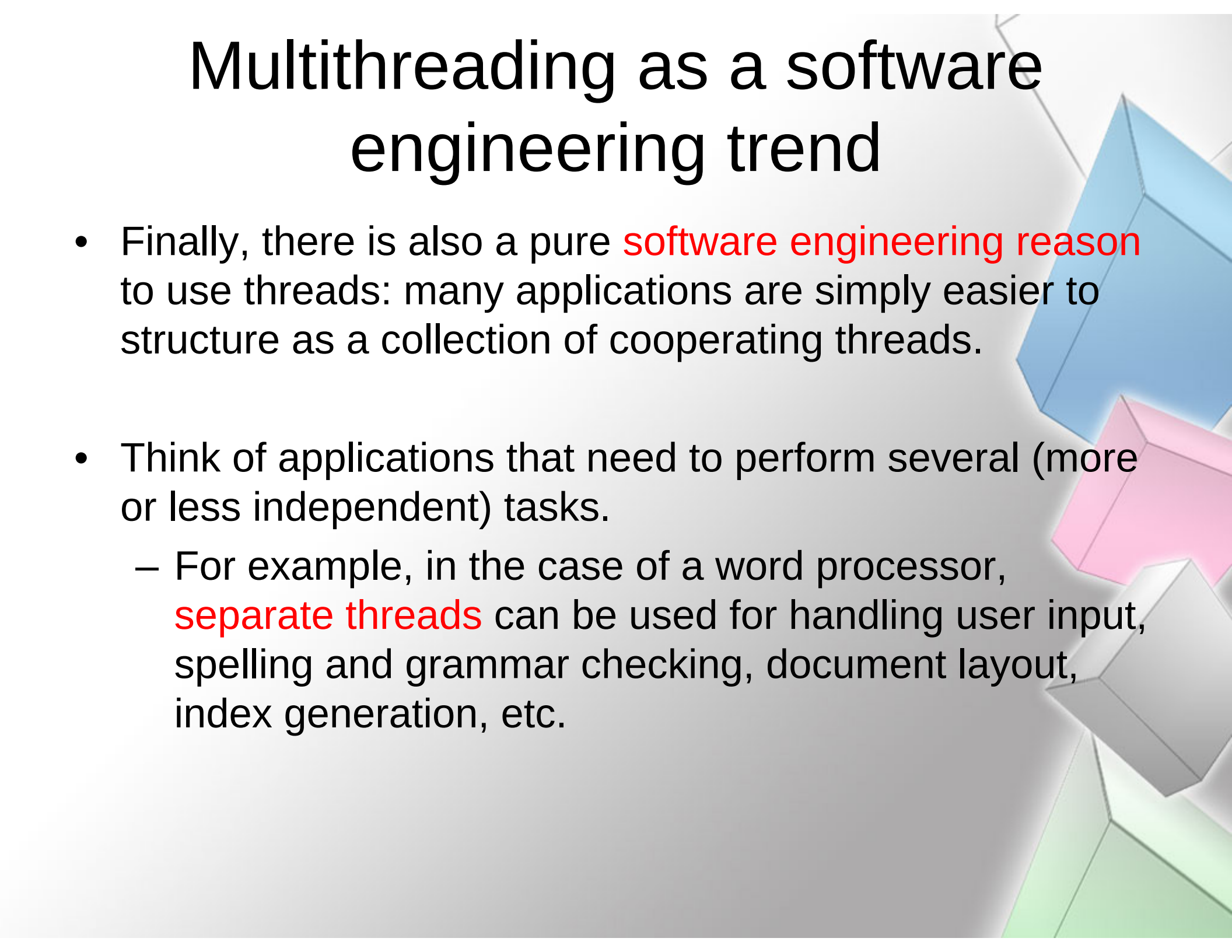
Multithreading for generic software applications

- 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.

Multithreading as a software engineering trend

- Finally, there is also a pure software engineering reason to use threads: many applications are simply easier to structure as a collection of cooperating threads.

- Think of applications that need to perform several (more or less independent) tasks.
  - 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.



Threads 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 two approaches to implement a thread package.
  - The first approach is to construct a thread library that is executed entirely in user mode.
  - The second approach is to have the kernel be aware of threads and schedule them.

User-level threads

- A user-level thread library has a number of 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.
  - 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.

User-level thread

- However, 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.

- As 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.

- For such applications, user-level threads are of no help.

- These problems can be mostly circumvented by implementing threads in the operating system's kernel.

Kernel Threads

- Unfortunately, there is a high price to pay: every thread operation (creation, deletion, synchronization, etc.), will have to be carried out by the kernel requiring 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.

Lightweight Process

- 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.
- The important issue is that the thread package is implemented entirely in user space. In other words. all operations on threads are carried out without intervention of the kernel.

Lightweight Process

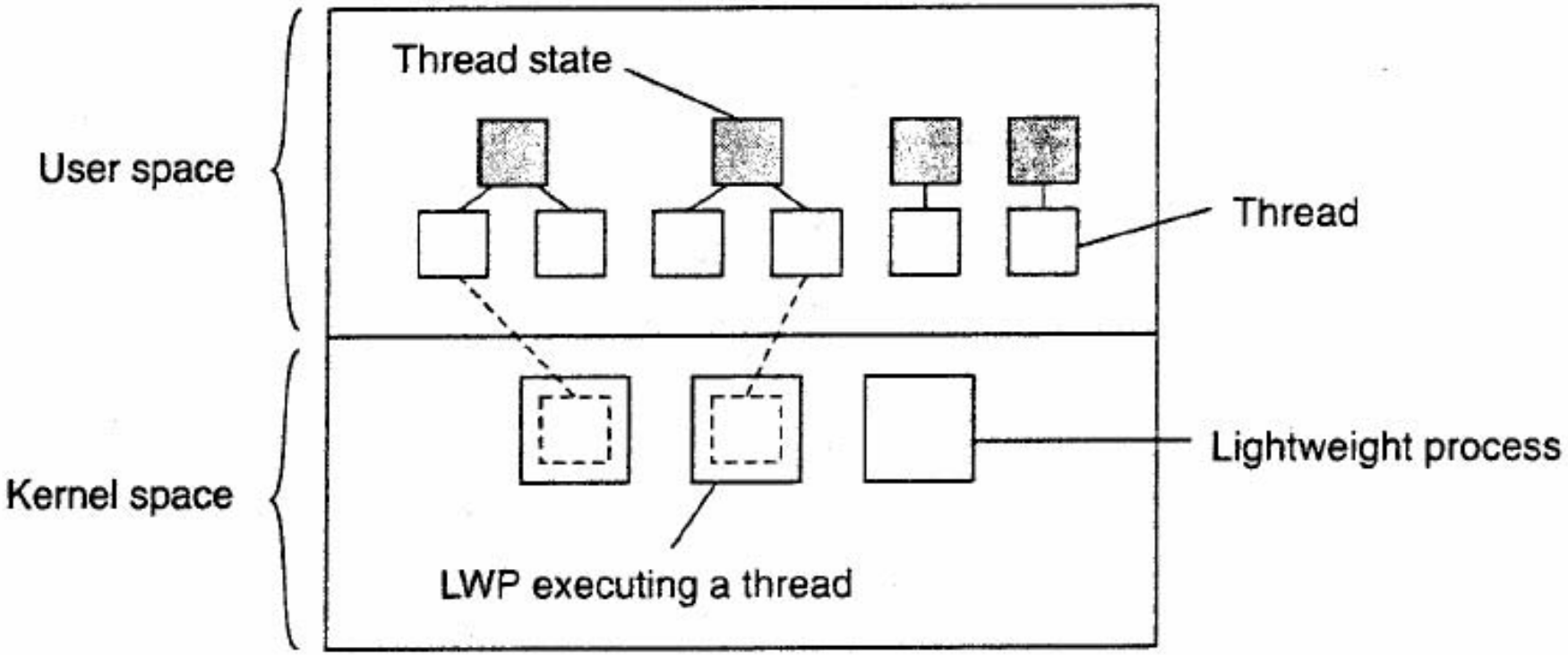


Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

Threads and Distributed Systems

- 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.

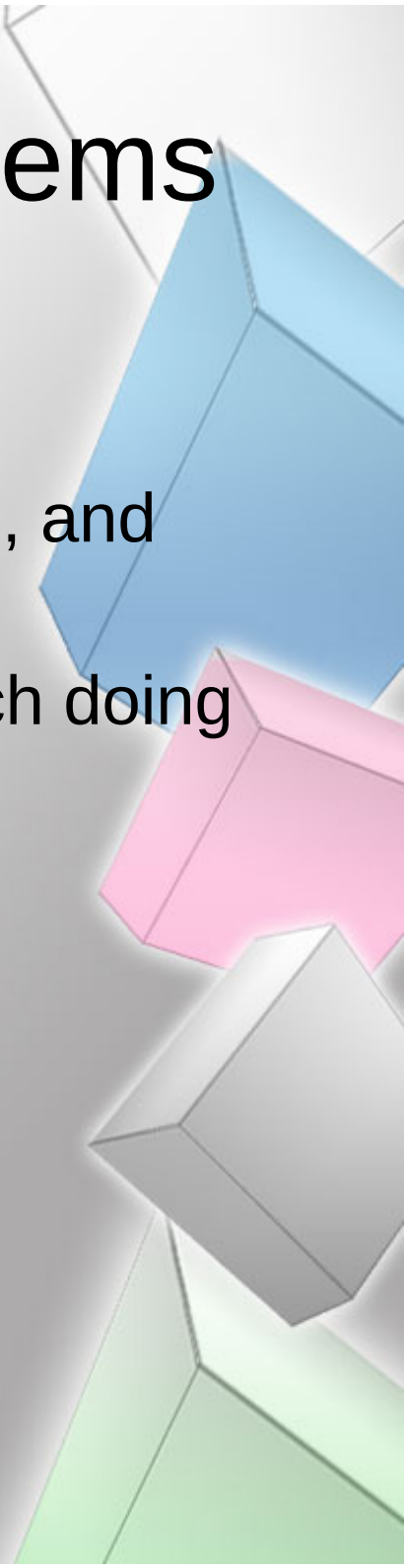
- This property makes threads particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time.

- We illustrate this point by taking a closer look at multithreaded clients and servers, respectively:

Threads and Distributed Systems

Multithreaded Web client

- Hiding network latencies:
  - Web browser scans an incoming HTML page, and finds that more files are to be fetched.
  - Each file is fetched by a separate thread, each doing a (blocking) HTTP request.
  - As files come in, the browser displays them.



Multithreaded Web browsers

- If several connections are set up to the same server and the 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.
- 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.

Multithreaded clients

Multiple request-response calls to other machines (RPC)

- A client does several calls at the same time, each one by a different thread.
- It then waits until all results have been returned.
- Note: if calls are to different servers, we may have a linear speed-up.

- This approach is possible only if the client can handle truly parallel streams of incoming data. Threads are ideal for this purpose.

Multithreaded Servers

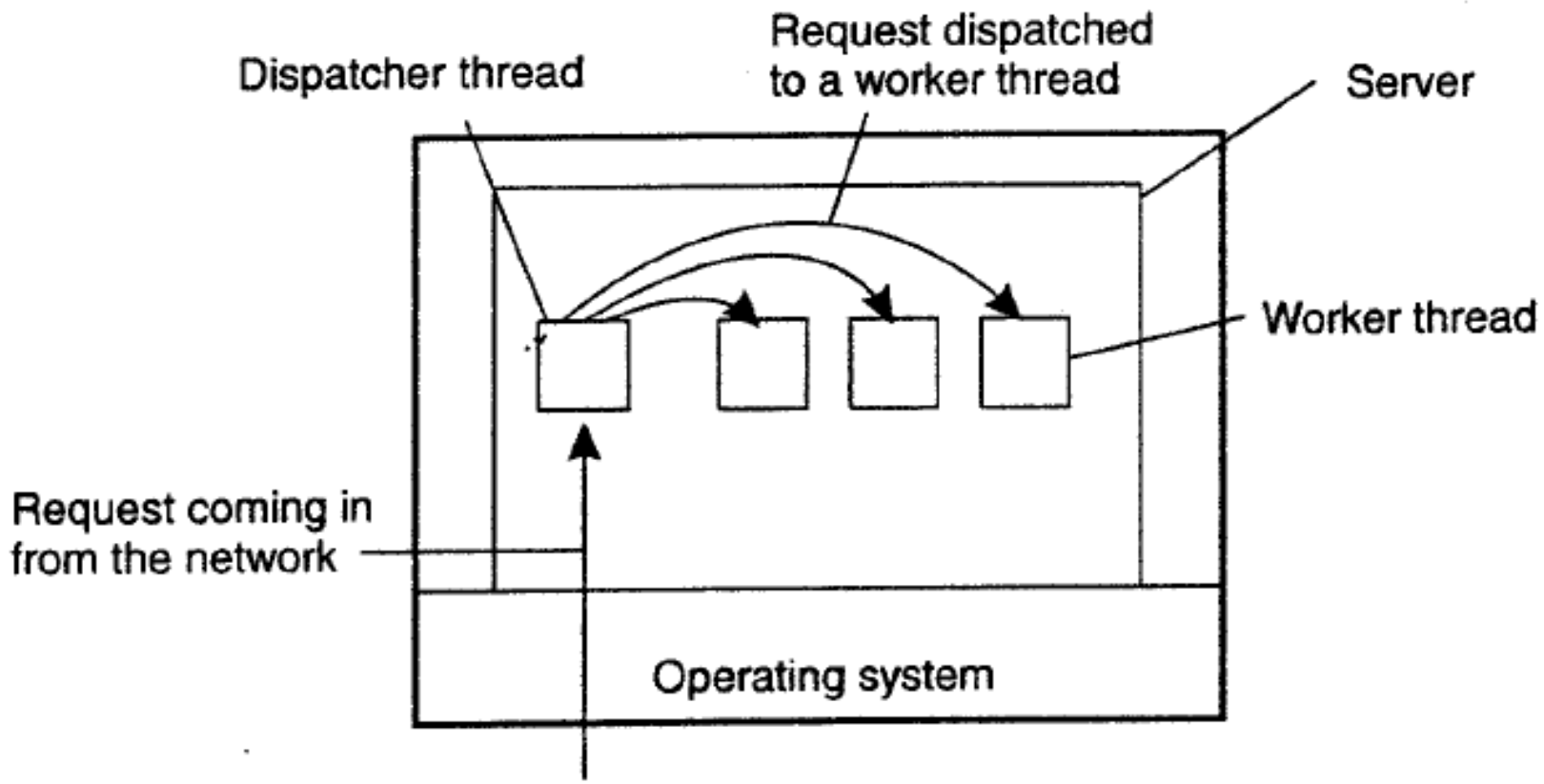


Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

Server as 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.

Threads and servers

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.

Threads 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.

Resource 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.

Virtualization

- One of the most important reasons for introducing virtualization in the 1970s, was to allow legacy software to run on expensive mainframe hardware.

- The software not only included various applications, but in fact also the operating systems they were developed for.

- This approach toward supporting legacy software has been successfully applied on the IBM 370 mainframes (and their successors) that offered a virtual machine to which different operating systems had been ported.

Virtualization

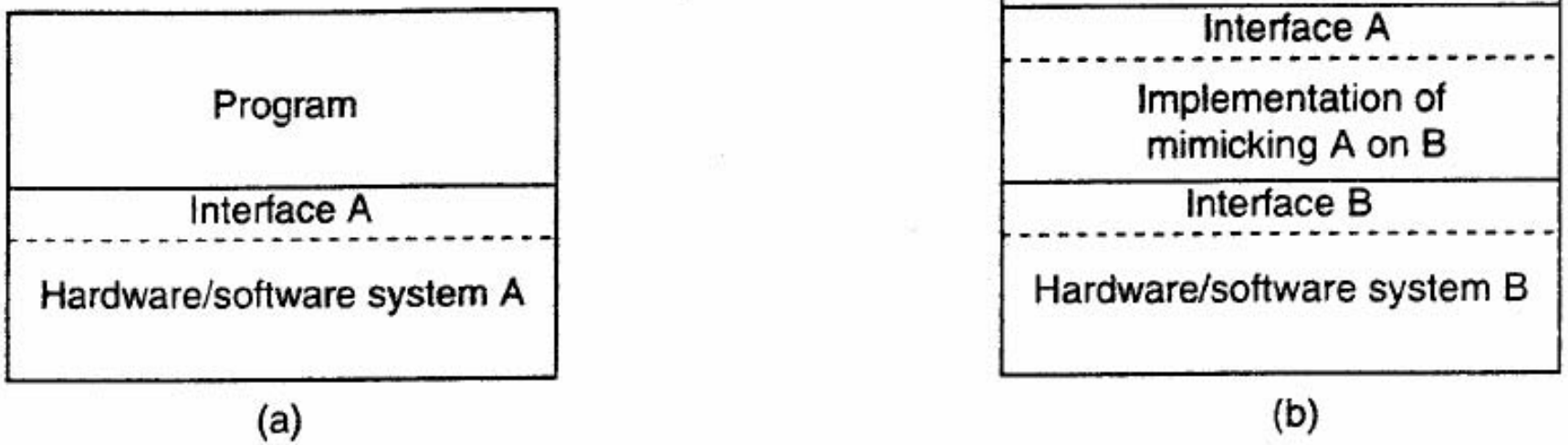


Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

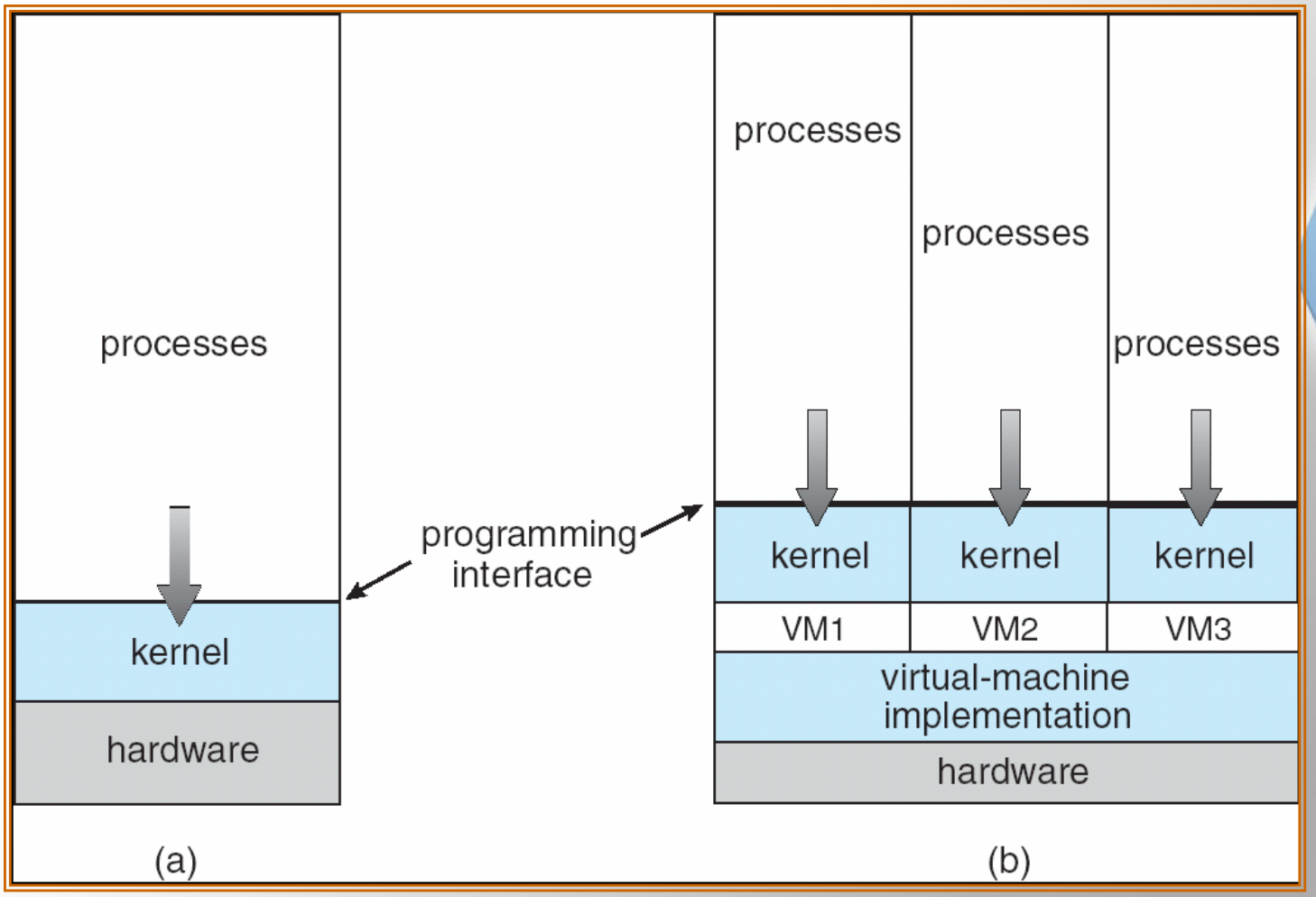
Virtual Machines 1

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface *identical* to the underlying bare hardware
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory
- A **virtual machine** (**VM**) is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine.

Virtual Machines 2

- The resources of the physical computer are shared to create the virtual machines
  - CPU scheduling can create the appearance that users have their own processor
  - Spooling and a file system can provide virtual card readers and virtual line printers
  - A normal user time-sharing terminal serves as the virtual machine operator's console
- An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine—it cannot break out of its virtual world.

Virtual Machines 3

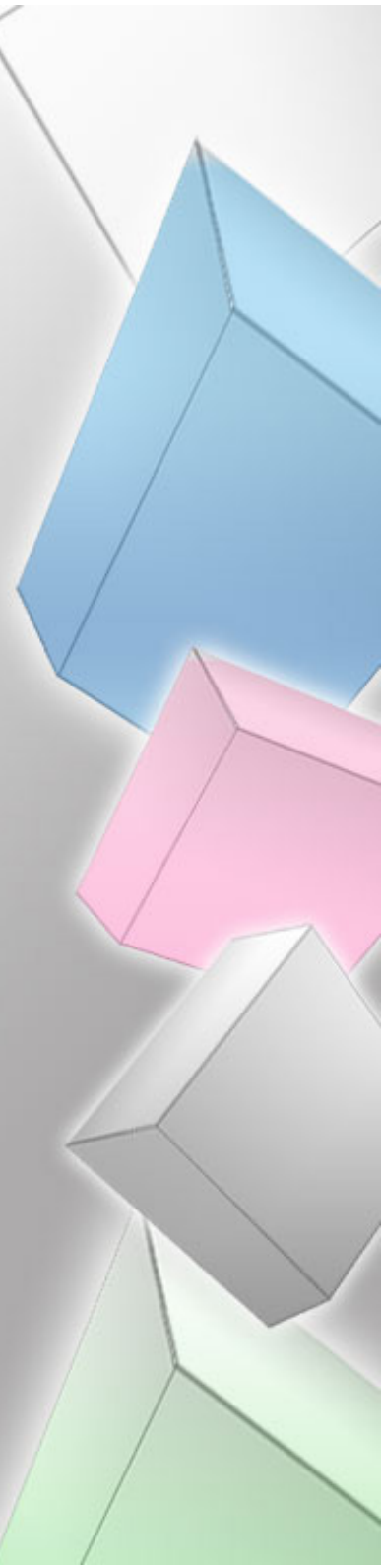


Virtual Machines 4

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine

Virtualization: Why?

- Virtualization is becoming increasingly important:
  - Hardware changes faster than software
  - Ease of portability and code migration
  - Isolation of failing or attacked components



Interfaces in a Computer System

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.

Architecture of VMs: Interfaces

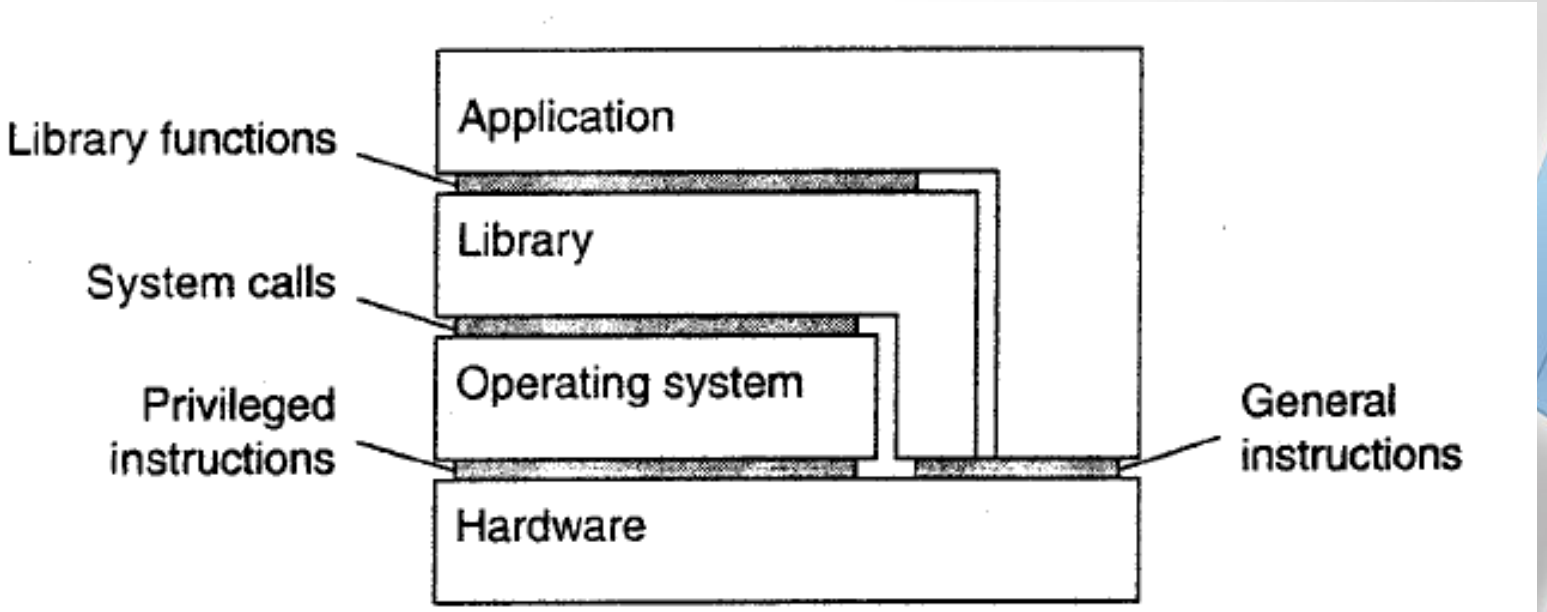


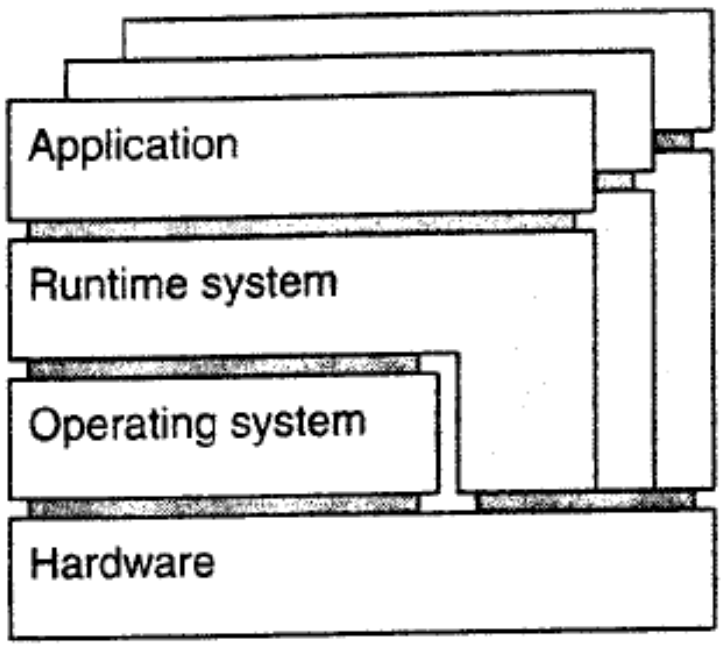
Figure 3-6. Various interfaces offered by computer systems.

Virtualization can take place at very different levels, strongly depending on the interfaces as offered by various systems components.

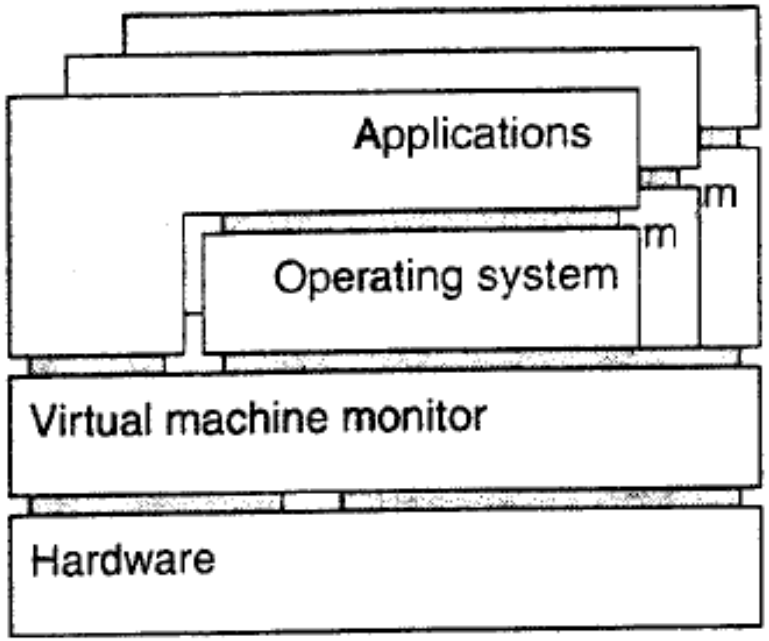
Virtual Machines 5

- A Virtual Machine Monitor (or System virtual machine) provides a complete system platform which supports the execution of a complete operating system (OS).
  - VMware
  - Virtual PC (Microsoft)
- A **process virtual machine** is designed to run a single program, which means that it supports a single process.
  - This type of VM has become popular with the Java programming language, which is implemented using the Java virtual machine.
  - Another example is the .NET Framework, which runs on a VM called the Common Language Runtime.

Virtual Machines in Action



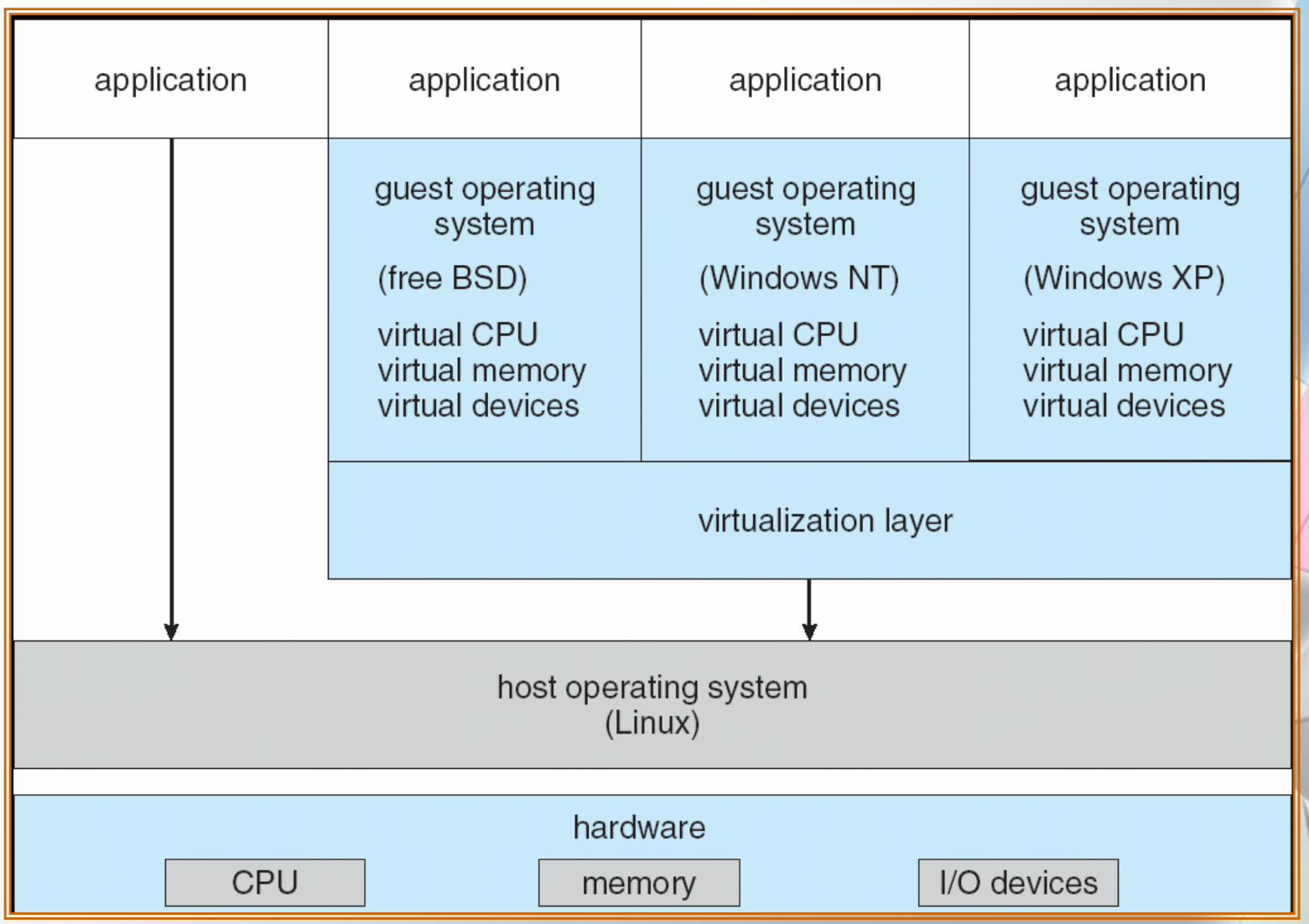
(a)



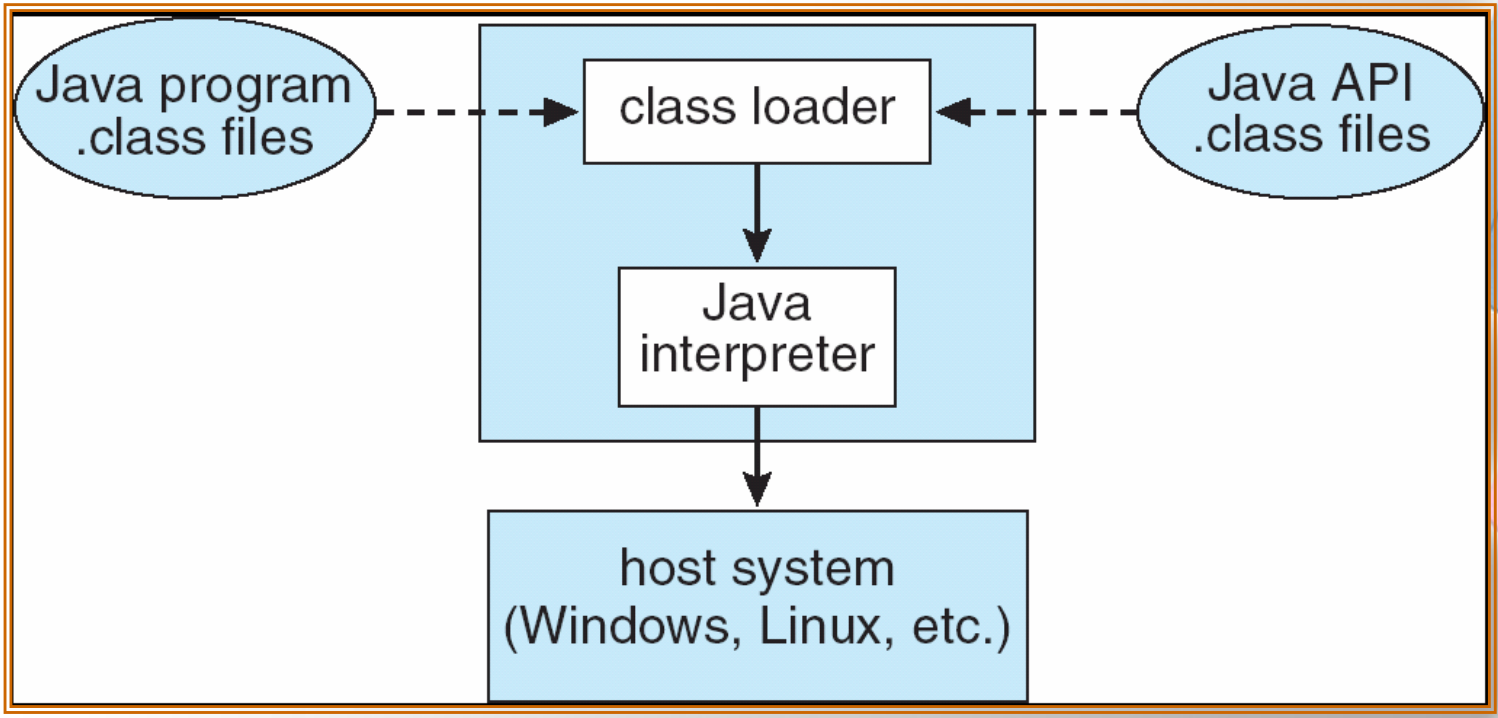
(b)

Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

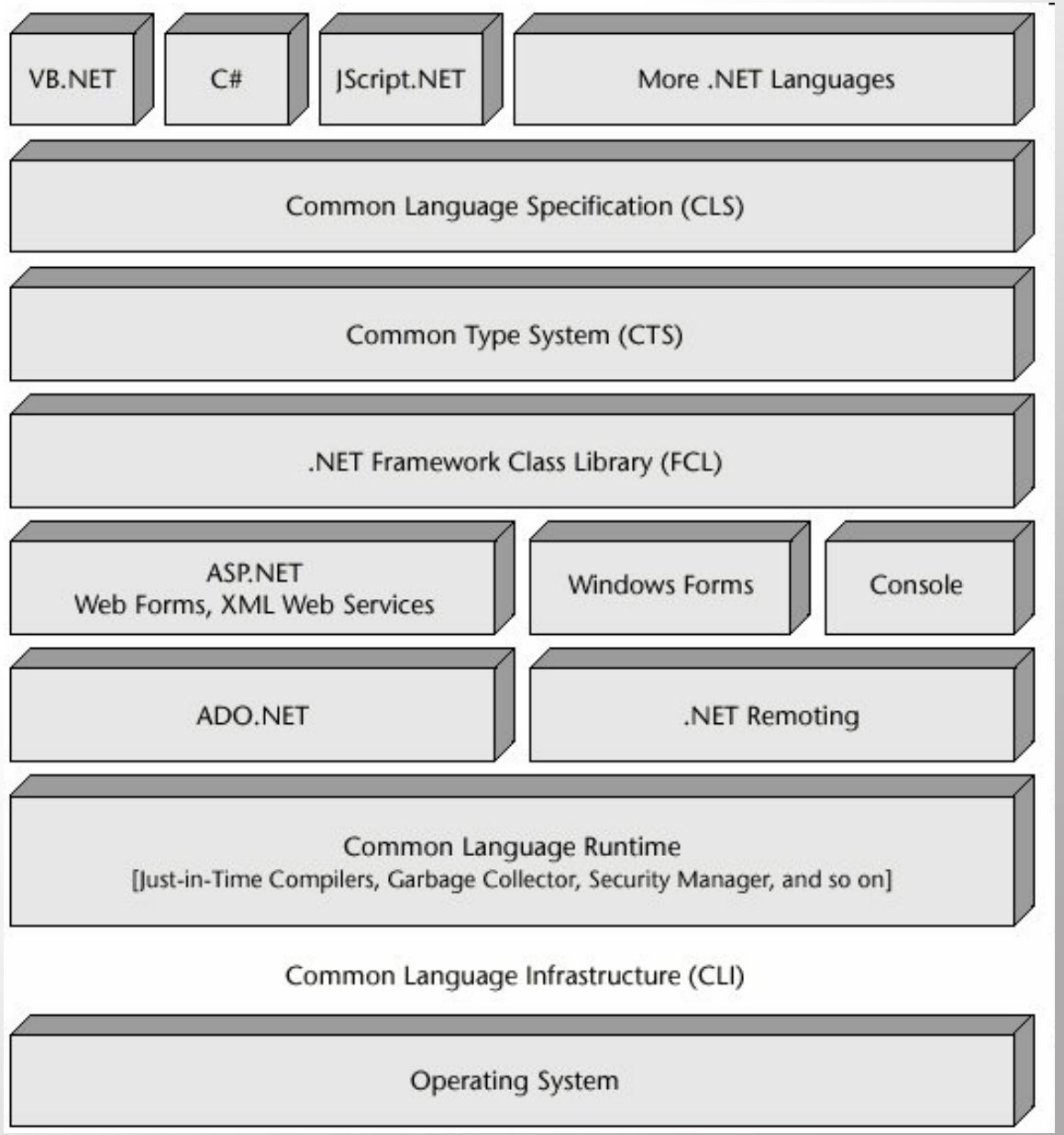
VMware Architecture



The Java Virtual Machine



.NET Architecture



System Virtual Machine Software

- ATL (A MTL Virtual Machine)
- Bochs, portable open source x86 and AMD64 PCs emulator
- CHARON-AXP, provides virtualization of AlphaServer to migrate OpenVMS or Tru64 applications to x86 hardware
- CHARON-VAX, provides virtualization of PDP-11 or VAX hardware to migrate OpenVMS or Tru64 applications to x86 or HP integrity hardware
- CoLinux Open Source Linux inside Windows
- Denali, uses paravirtualization of x86 for running para-virtualized PC operating systems.
- eVM Virtualization Platform for Windows by TenAsys
- Hercules emulator, free System/370, ESA/390, z/Mainframe
- Microsoft Virtual PC and Microsoft Virtual Server
- OKL4 from Open Kernel Labs
- Oracle VM
- SLKVM - scripts to handle kvm and vz virtual machines in a cluster environment
- Sun xVM
- VM from IBM
- VMware (ESX Server, Fusion, Virtual Server, Workstation, Player and ACE)
- vSMP Foundation (From ScaleMP)
- Xen (Opensource)
- IBM POWER SYSTEMS

Process Virtual Machine Software

- Common Language Infrastructure - C#, Visual Basic .NET, J#, C++/CLI (formerly Managed C++)
- Dalvik virtual machine - part of the Android mobile phone platform
- Java Virtual Machine - Java, Nice, NetREXX
- Juke Virtual Machine - A public domain ECMA-335 compatible virtual machine hosted at Google code.
- Low Level Virtual Machine (LLVM) - currently C, C++, Stacker
- Macromedia Flash Player - SWF
- Perl virtual machine - Perl
- CPython - Python
- Rubinius - Ruby
- SECD machine - ISWIM, Lispkit Lisp
- Sed the stream-editor can also be seen as a VM with 2 storage spaces.
- Smalltalk virtual machine - Smalltalk
- SQLite virtual machine - SQLite opcodes
- Tamarin (JavaScript engine) - ActionScript VM in Flash 9
- TrueType virtual machine - TrueType
- Valgrind - checking of memory accesses and leaks in x86/x86-64 code under Linux
- Virtual Processor (VP) from Tao Group (UK).
- Waba - Virtual machine for small devices, similar to Java
- Warren Abstract Machine - Prolog, CSC GraphTalk

Importance of VMMs

- VMMs will become increasingly important in the context of reliability and security for (distributed) systems.
- As they allow for the isolation of a complete application and its environment, a failure caused by an error or security attack need no longer affect a complete machine.
- In addition, as we also mentioned before, portability is greatly improved as VMMs provide a further decoupling between hardware and software, allowing a complete environment to be moved from one machine to another.

Clients: Networked Interfaces

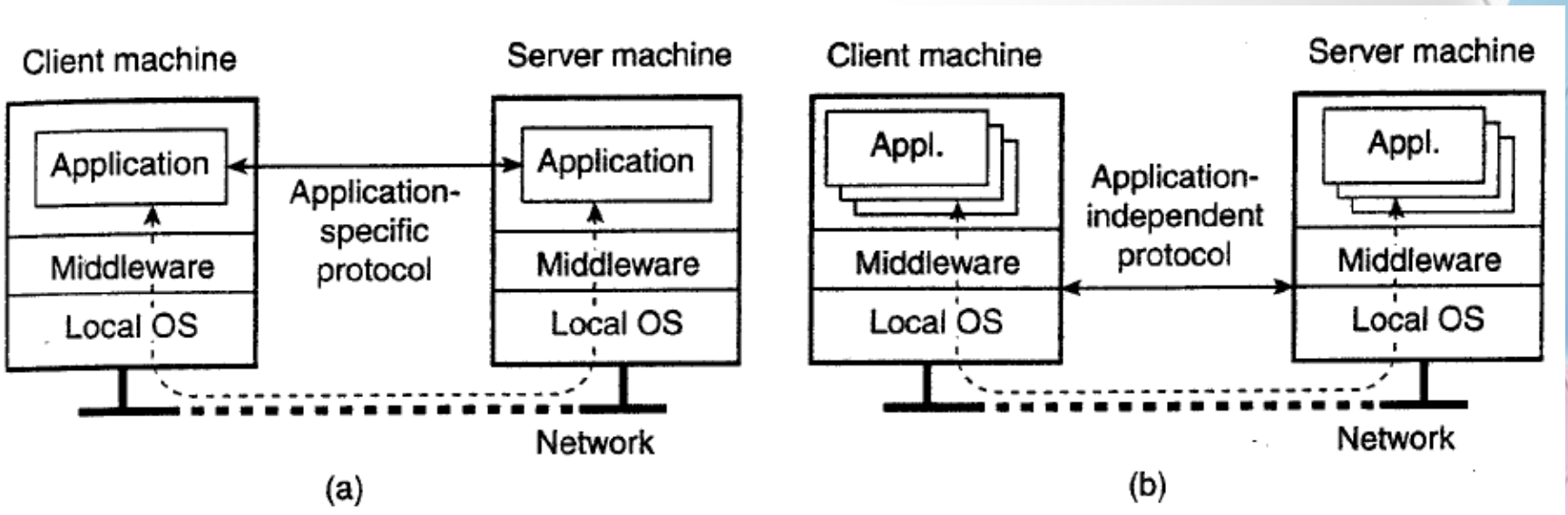


Figure 3-8. (a) A networked application with its own protocol.. (b) A general solution to allow access to remote applications.

In b) 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.

Client as terminal: X Window

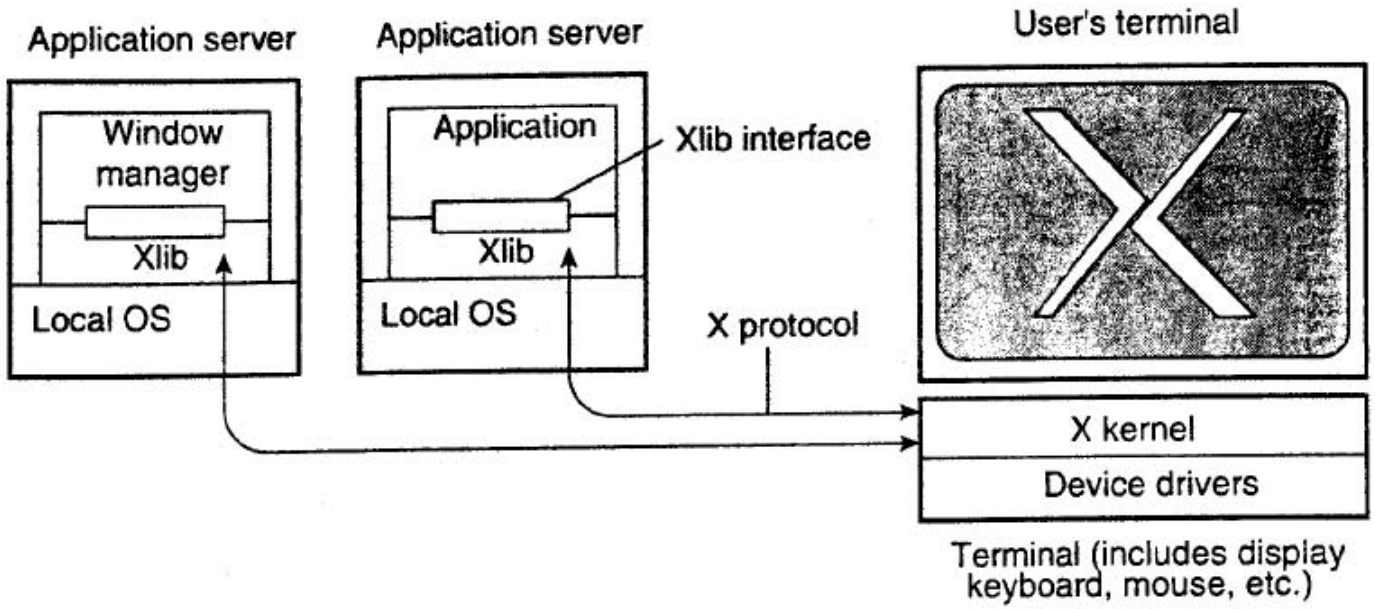


Figure 3-9. The basic organization of the X Window System.

The X Window System, generally referred to simply as X, is used to control bit-mapped terminals, which include a monitor, keyboard, and a pointing device such as a mouse.

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.

X Window

- 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.*
- Several applications can communicate at the same time with the X kernel.
- There is one specific application that is given special rights, known as the **window manager.** This application can dictate the "look and feel" of the display as it appears to the user.

Clients: User Interfaces

Compound documents

- User interface is application-aware => inter-application communication:
  - drag-and-drop: move objects across the screen to invoke interaction with other applications (ex. Move a file to trash can).
  - in-place editing: integrate several applications at user-interface level (word processing + drawing facilities).

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. In these cases, the user interface is a relatively small part of the client software, in contrast to the local processing and communication facilities.

- Besides the user interface and other application-related software, client software comprises components for achieving distribution transparency. Ideally, a client should not be aware that it is communicating with remote processes.

- In contrast, distribution is often less transparent to servers for reasons of performance and correctness.

Client-side Software

- Access transparency: client-side stubs for RPCs
- location/migration transparency: let client-side software keep track of actual location
- replication transparency: multiple invocations handled by client stub:

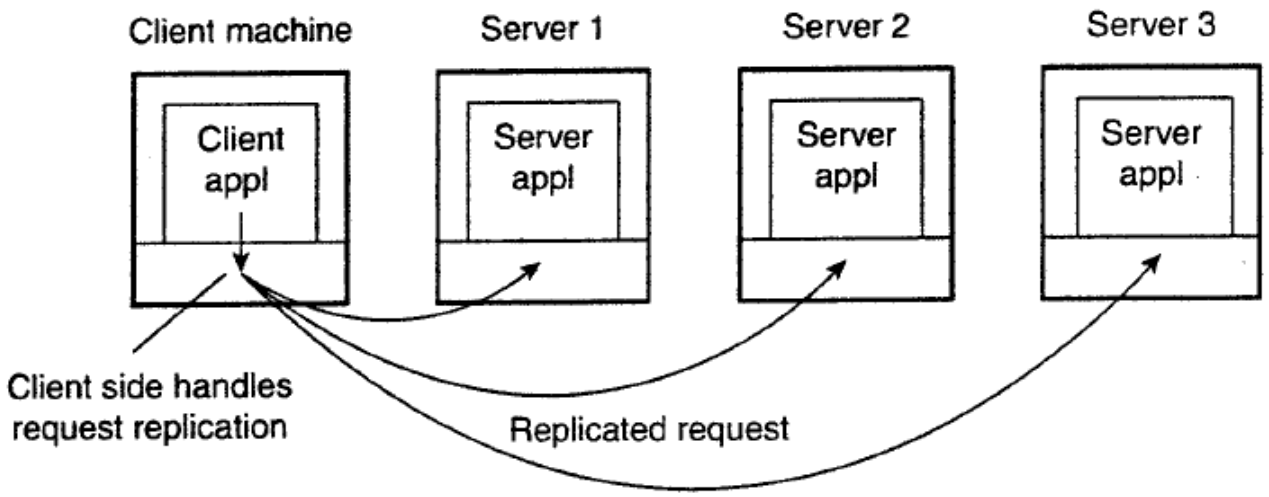
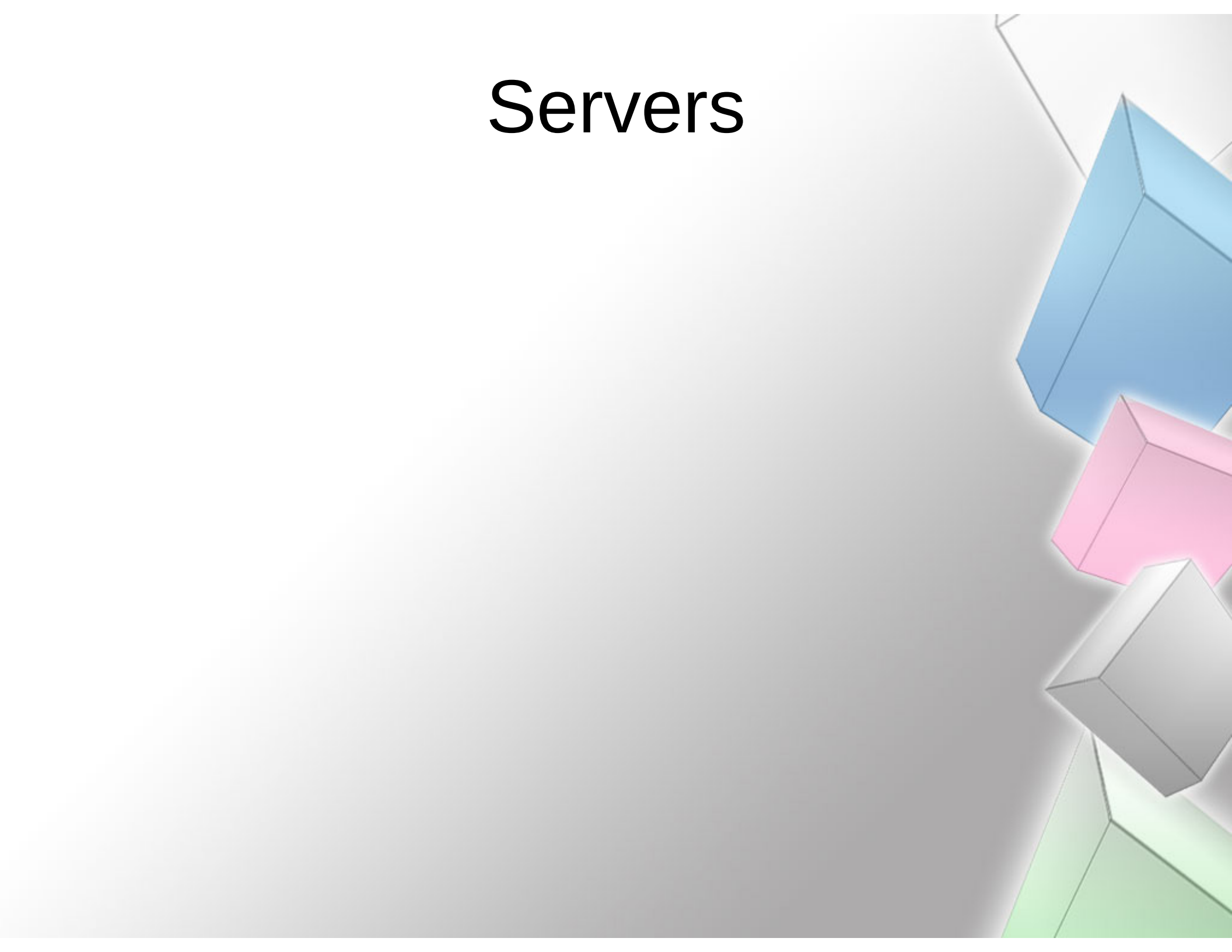


Figure 3-10. Transparent replication of a server using a client-side solution.

- failure transparency: can often be placed only at client (we're trying to mask server and communication failures).

Servers



Servers anatomy

- There are several ways to organize servers. 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. This approach is followed in man UNIX systems.

- The thread or process that handles the request is responsible for returning a response to the requesting client.

Where clients contact a server

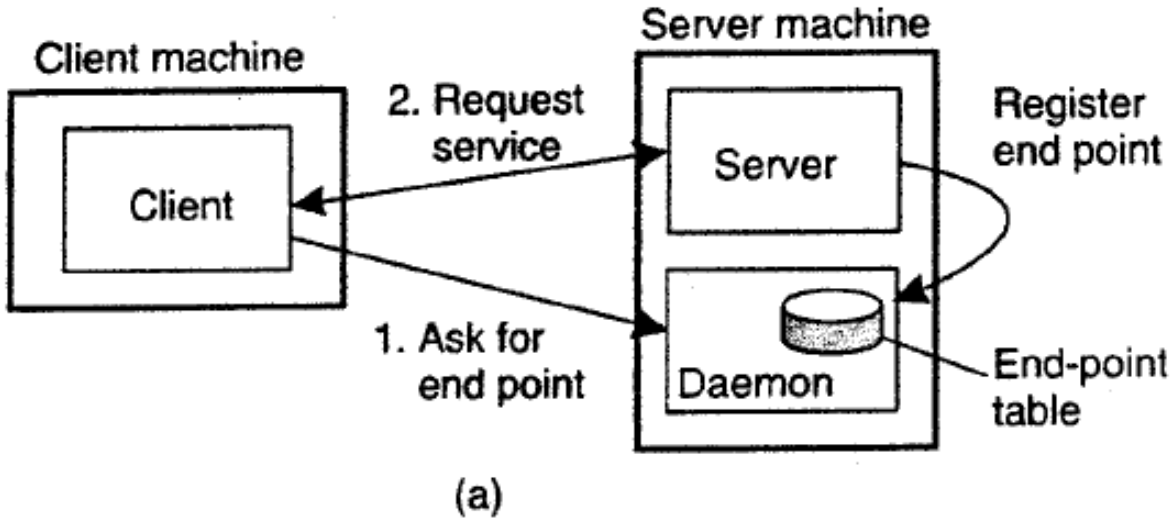
- Another issue is where clients contact a server. In all cases, clients send requests to an end point, also called a port, at the machine where the server is running. Each server listens to a specific end point.
- How do clients know the end point of a service? One approach is to globally assign end 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.
- These end points have been assigned by the Internet Assigned Numbers Authority (IAN).
- With assigned end points, the client only needs to find the network address of the machine where the server is running.
  - As we will explain alter, name services can be used for that purpose.

Ports for services

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

Other services

- There are many services that do not require a preassigned end point. For example, a time-of-day server may use an end point that is dynamically assigned to it by its local operating system.
- In that case, 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.

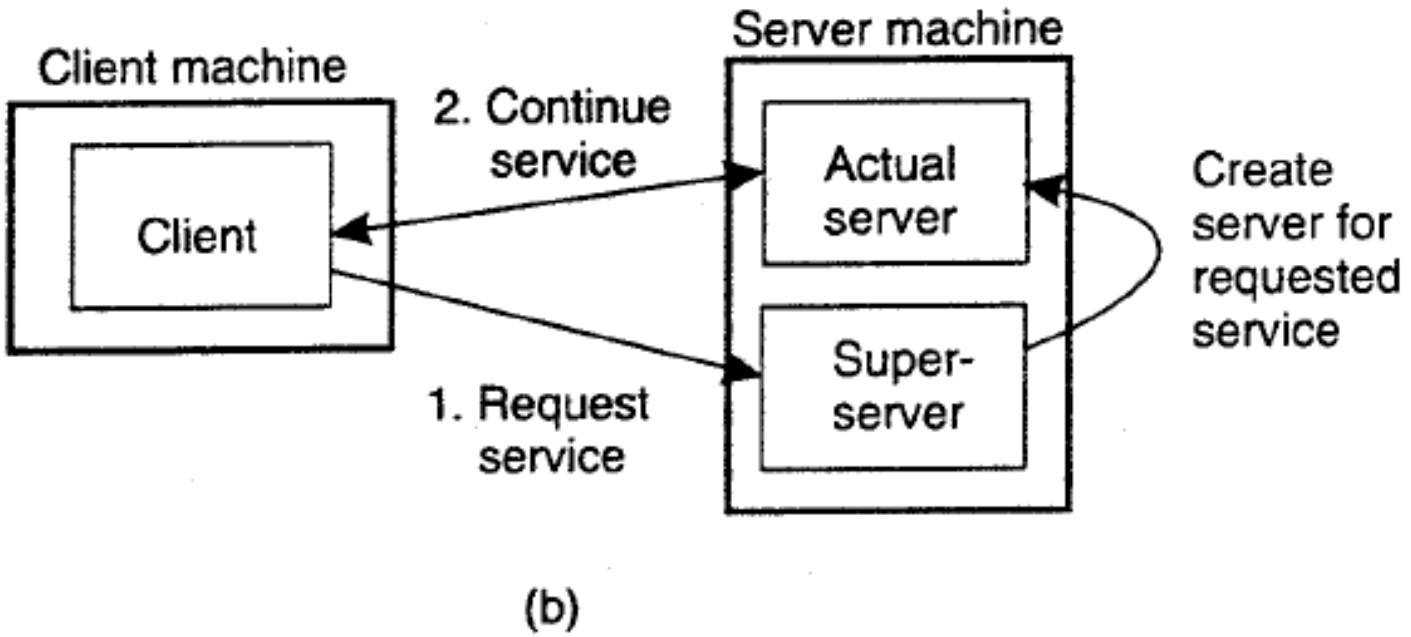


(a)

Superserver

- It is common to associate an end point with a specific service. However, actually implementing each service by means of a separate server may be a waste of resources.
- For example, in a typical UNIX system, it is common to have lots of servers running simultaneously, with most of them passively waiting until a client request comes in.
- 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.
- This is the approach taken, for example, with the *inetd* daemon in UNIX. *Inetd* listens to a number of well-known ports for Internet services. When a request comes in, the daemon forks a process to take further care of the request. That process will exit after it is finished.

Superserver



(b)

Figure 3-11. (a) Client-to-server binding using a daemon. (b) Client-to-server binding using a superserver..

Server interruption

- Another issue that needs to be taken into account when designing a server is whether and how a server can be interrupted.
- For example, consider a user who has just decided to upload a huge file to an FTP server. Then, suddenly realizing that it is the wrong file, he wants to interrupt the server to cancel further data transmission.
- There are several ways to do this. One approach that works only too well in the current Internet (and is sometimes the only alternative) is for the user to abruptly exit the client application (which will automatically break the connection to the server), immediately restart it, and pretend nothing happened.
- The server will eventually tear down the old connection, thinking the client has probably crashed.

Out-of-band communication

Issue

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

Solution 1

- Use a separate port for urgent data:
  - Server has a separate thread/process for urgent messages
  - Urgent message comes in => associated request must wait
    - Note: we require OS supports priority-based scheduling

Solution 2

- Use out-of-band communication facilities of the transport layer:
  - Example: TCP allows for urgent messages in same connection
  - Urgent messages can be caught using OS signaling techniques

Stateless Servers

- **Stateless servers**

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

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

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

Stateful servers

- **Stateful servers**

Keeps track of the status of its clients:

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

- **Observation**

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

Temporary and permanent state

- We can argue that one should actually make a distinction between (temporary) session state and permanent state.
- A series of operations by a single user could be maintained for some time, but not indefinitely.
- Session state is often maintained in three-tiered client-server architectures, where the application server actually needs to access a database server through a series of queries before being able to respond to the requesting client.
- What remains for permanent state is typically information maintained in databases, such as customer information, keys associated with purchased software, etc.

Cookies

- In other cases, a server may want to keep a record on a client's behavior so that it can more effectively respond to its requests.
  - For example, Web servers sometimes offer the possibility to immediately direct a client to his favorite pages.
- This approach is possible only if the server has history information on that client.
- When the server cannot maintain state, a common solution is then to let the client send along additional information on its previous accesses.
- In the case of the Web, this information is often transparently stored by the client's browser in what is called a cookie, which is a small piece of data containing client-specific information that is of interest to the server.
- Cookies are never executed by a browser; they are merely stored.

Cookies

- The first time a client accesses a server, the latter sends a cookie along with the requested Web pages back to the browser.
- Each subsequent time the client accesses the server, its cookie for that server is sent along with the request.
- Although in principle, this approach works fine, the fact that cookies are sent back for safekeeping by the browser is often hidden entirely from users.

Server Clusters

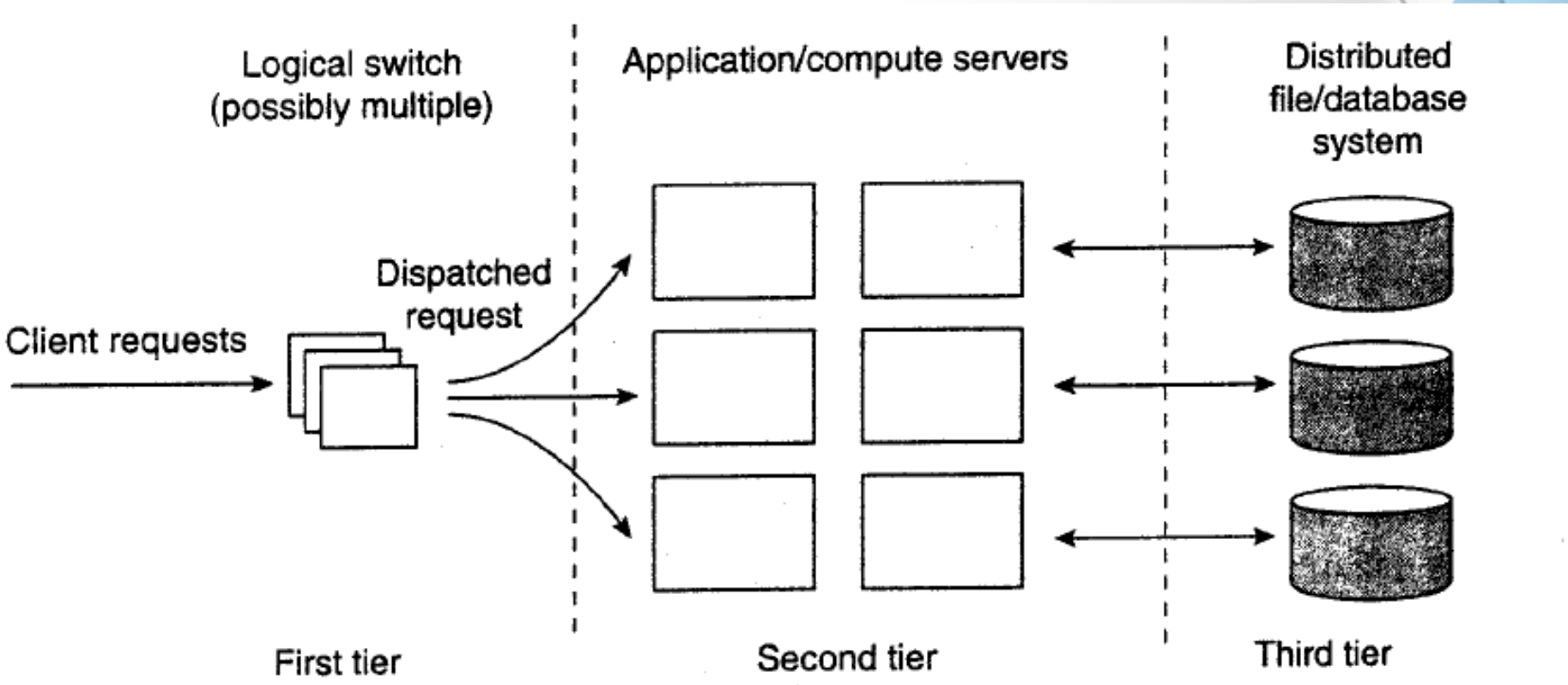


Figure 3-12. The general organization of a three-tiered server cluster.

Critical element

The first tier is generally responsible for passing requests to an appropriate server.

Enterprise Server Clusters

- As in any multitiered client-server architecture, many server clusters also contain servers dedicated to application processing.
  - In cluster computing, these are typically servers running on high-performance hardware dedicated to delivering compute power.
- However, in the case of enterprise server clusters, it may be the case that applications need only run on relatively low-end machines, as the required compute power is not the bottleneck, but access to storage is.
- This brings us the third tier, which consists of data-processing servers, notably file and database servers.
- Again, depending on the usage of the server cluster, these servers may be running an specialized machines, configured for high-speed disk access and having large server-side data caches.

Server clusters for streaming media

- Of course, not all server clusters will follow this strict separation in 3 parts.

- It is frequently the case that each machine is equipped with its own local storage, often integrating application and data processing in a single server leading to a two tiered architecture.

- For example, when dealing with streaming media by means of a server cluster, it is common to deploy a two-tiered system architecture, where each machine acts as a dedicated media server.

Code Migration

- When a server cluster offers multiple services, it may happen that different machines run different application servers. As a consequence, the switch will have to be able to distinguish services or otherwise it cannot forward requests to the proper machines.

- As it turns out, 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.

- As a consequence, we may find that certain machines are temporarily idle, while others are receiving an overload of requests.

- What would be useful is to temporarily migrate services to idle machines. A solution is to use virtual machines allowing a relative easy migration of code to real machines.

Request handling: potential bottleneck

- Having the first tier handle all communication from/to the cluster may lead to a bottleneck.
- Client applications running on remote machines should have no need to know anything about the internal organization of the cluster. This access transparency is invariably offered by means of a single access point, in turn implemented through some kind of hardware switch such as a dedicated machine.
- The switch forms the entry point for the server cluster, offering a single network address.
- For scalability and availability, a server cluster may have multiple access points, where each access point is then realized by a separate dedicated machine.

TCP-Handoff

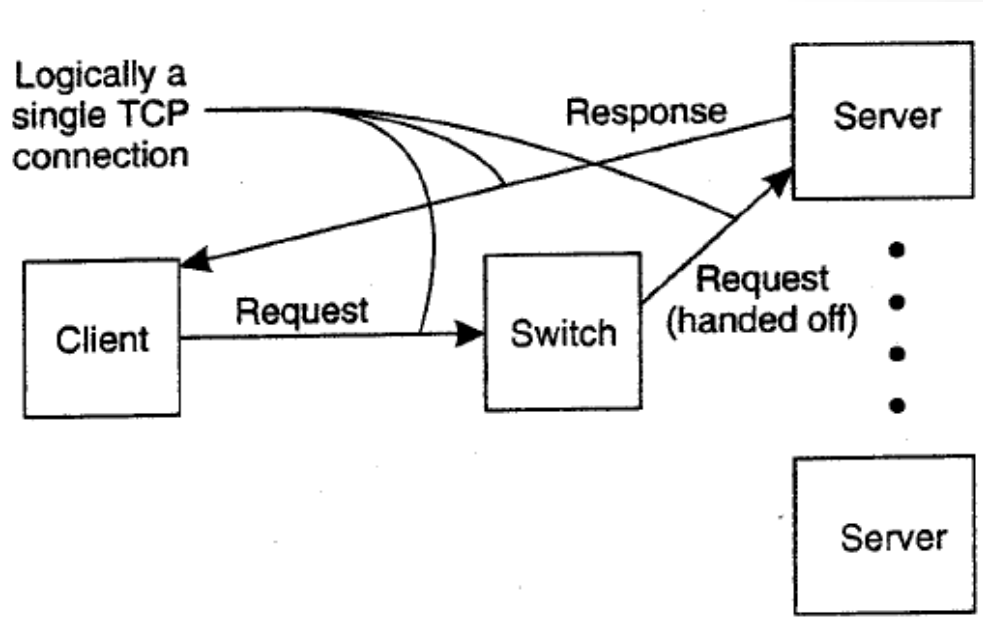


Figure 3-13. The principle ofTCP handoff.

- The server, in turn, will send an acknowledgment back to the requesting client by inserting the switch's IP address as the source field of the header of the IP packet carrying the TCP segment.
- Note that this spoofing is necessary for the client to continue executing the TCP protocol: it is expecting an answer back from the switch, not from some arbitrary server it is has never heard of before.
- Clearly, a TCP-handoff implementation requires operating-system level modifications.

Distributed Servers

- The server clusters discussed so far are generally rather statically configured. In these clusters, 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.

- 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.

Distributed Servers: route optimization

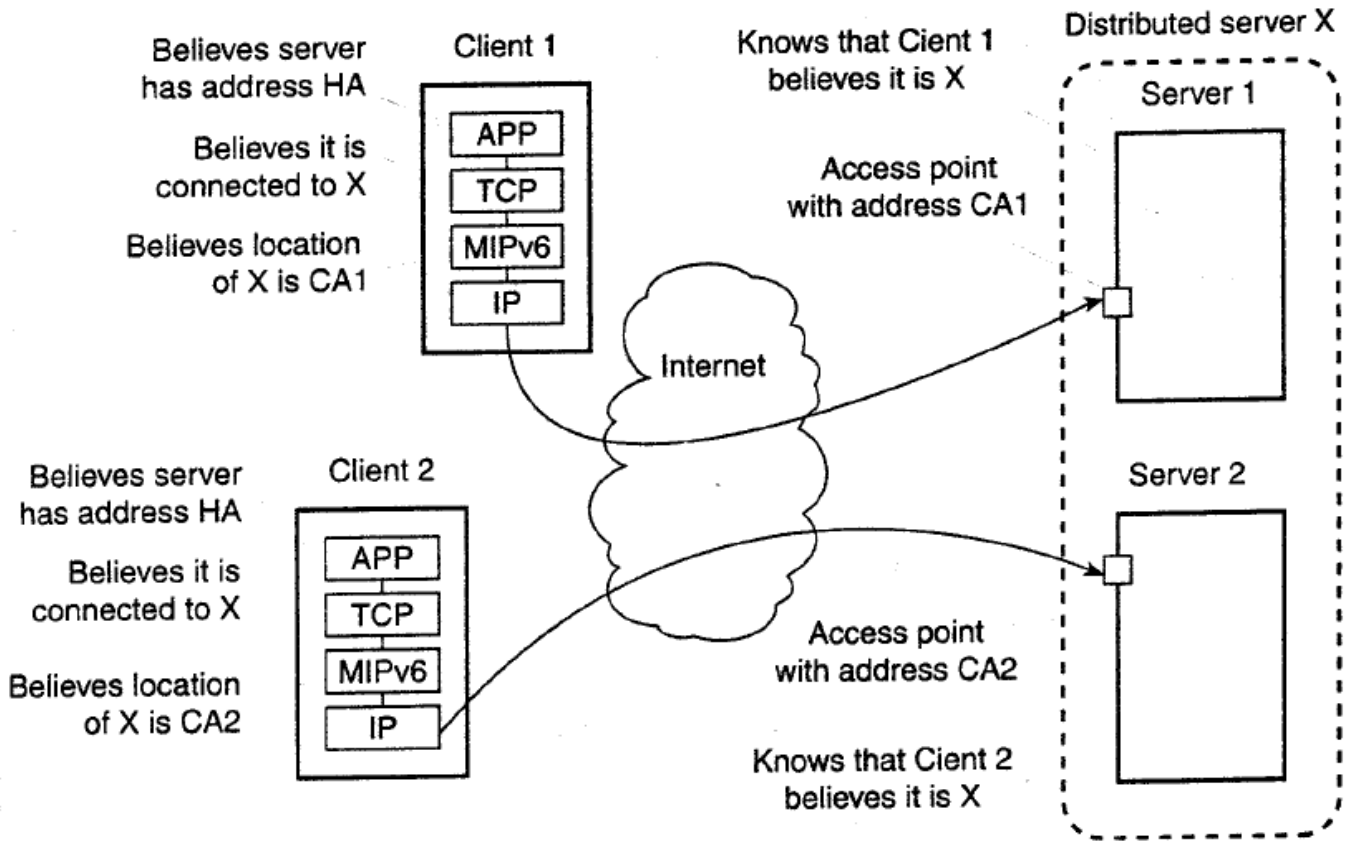


Figure 3-14. Route optimization in a distributed server.

Route optimization can be used to make different clients believe they are communicating with a single server, where, in fact, each client is communicating with a different member node of the distributed server

Managing Server Clusters

- By far the most common approach to managing a server cluster is to extend the traditional managing functions of a single computer to that of a cluster.
  - In its most primitive form, this means that an administrator can log into a node from a remote client and execute local managing commands to monitor, install, and change components.

- Somewhat more advanced is to hide the fact that you need to login into a node and instead provide an interface at an administration machine that allows to collect information from one or more servers, upgrade components, add and remove nodes, etc.
  - The main advantage of the latter approach is that collective operations, which operate on a group of servers, can be more easily provided. This type of managing server clusters is widely applied in practice, exemplified by management software such as Cluster Systems Management from IBM

Managing Server clusters

- However, as soon as clusters grow beyond several tens of nodes, this type of management is not the way to go.

- Many data centers need to manage thousands of servers, organized into many clusters but all operating collaboratively. Doing this by means of centralized administration servers is simply out of the question. ☺

- Moreover, it can be easily seen that very large clusters need continuous repair management (including upgrades). To simplify matters, if *p* is the probability that a server is currently faulty, and we assume that faults are independent, then for a cluster of *N* servers to operate without a single server being faulty is $(1-p)^N$. With $p=0.001$ and $N=1000$, there is only a 36 percent chance that all the servers are correctly functioning. ☹/☺

- It turns out that support for very large server clusters is almost always ad hoc.

Example: PlanetLab

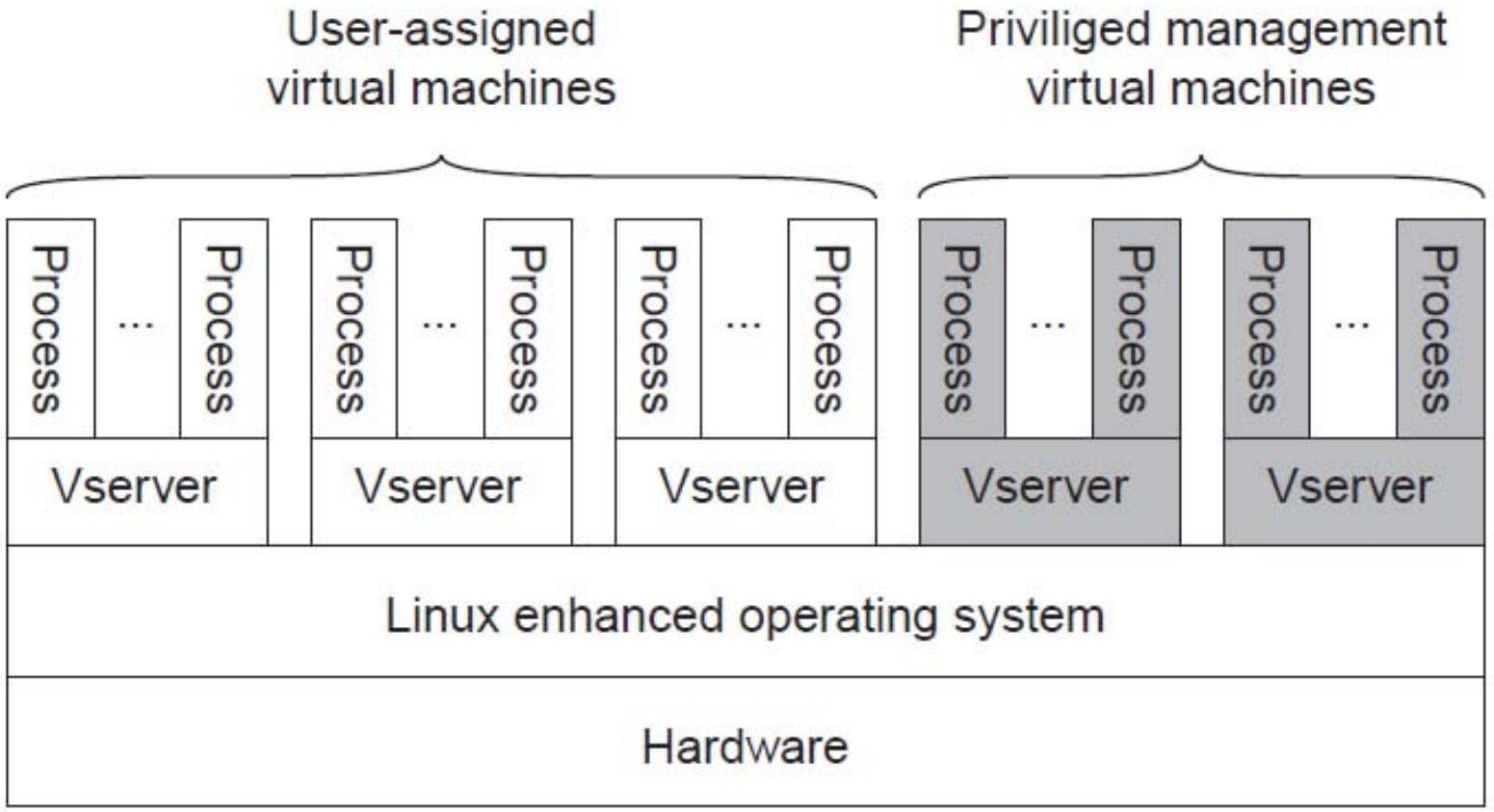
- Let us now take a closer look at a somewhat unusual cluster server. PlanetLab is a collaborative distributed system in which different organizations each donate one or more computers, adding up to a total of hundreds of nodes.

- Together, these computers form a 1-tier server cluster, where access, processing, and storage can all take place on each node individually.

- Management of PlanetLab is by necessity almost entirely distributed.

- In PlanetLab, an organization donates one or more nodes, where each node is easiest thought of as just a single computer, although it could also be itself a cluster of machines.

A Node of PlanetLab



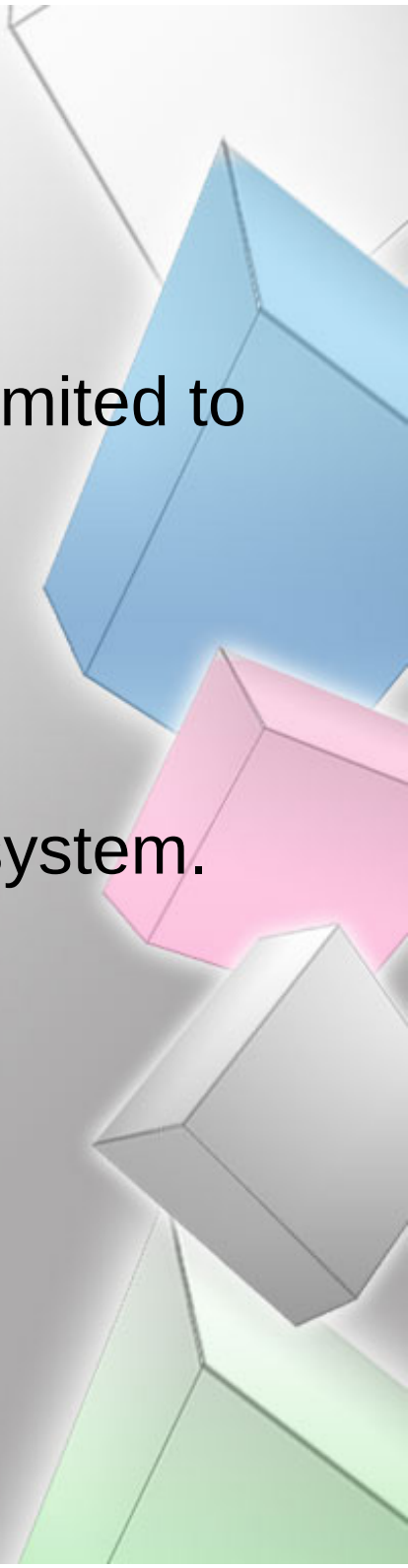
Vserver: Independent and protected environment with its own libraries, server versions, and so on.
Distributed applications are assigned a collection of vservers distributed across multiple machines (slice).

Code Migration

- Until now we have been mainly concerned with distributed systems in which communication is limited to passing data.

- However, there are situations in which passing programs, sometimes even while they are being executed, simplifies the design of a distributed system.

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



Migration: why bother with it?

- Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so.
- That reason has always been performance.
  - The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines.
- Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

- Load distribution algorithms by which decisions are made concerning the allocation and redistribution of tasks with respect to a set of processors, play an important role in compute-intensive systems.
  - However, in many modern distributed systems, optimizing computing capacity is less an issue than, for example, trying to minimize communication.

Migrating from client to server

- Consider, as an example, a client-server system in which the server manages a huge database.

- 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.
  - Otherwise, the network may be swamped with the transfer of data from the server to the client.

- In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

Migrating from server to client

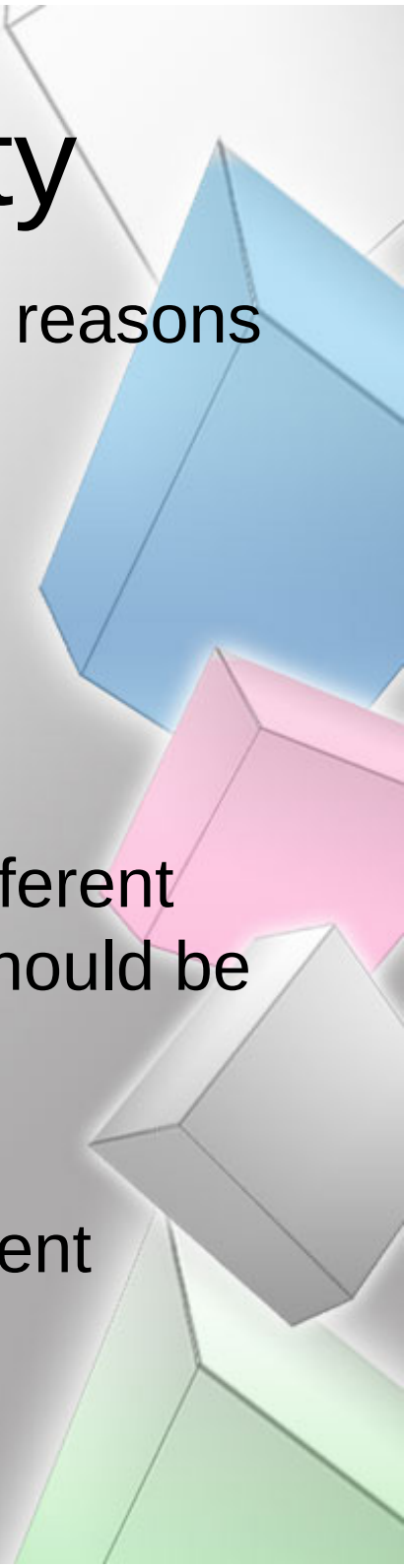
- The performance reason can be used for migrating parts of the server to the client.
  - For example, in many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations.

- Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network.

- The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.

Migrating code: Flexibility

- Besides improving performance, there are other reasons for supporting code migration as well.
- The most important one is that of flexibility.
- 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.
- This approach, for example, has led to the different multitiered client-server applications.



Migrating code: dynamic configuration of DSs

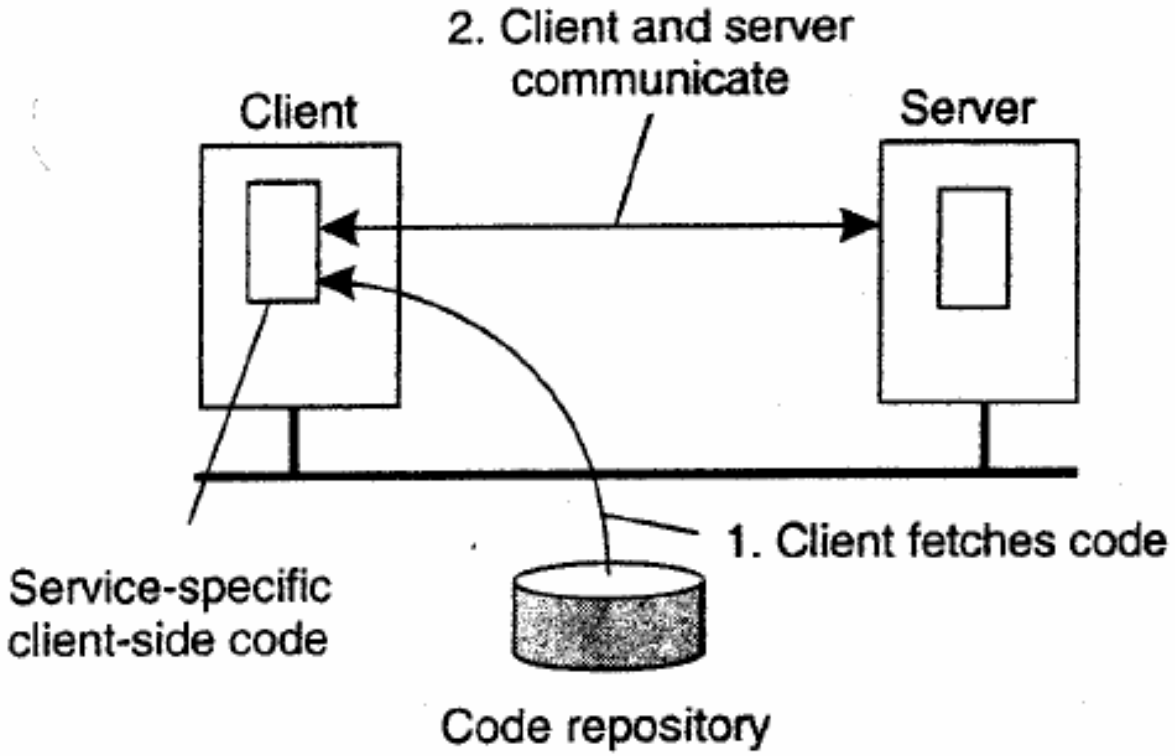


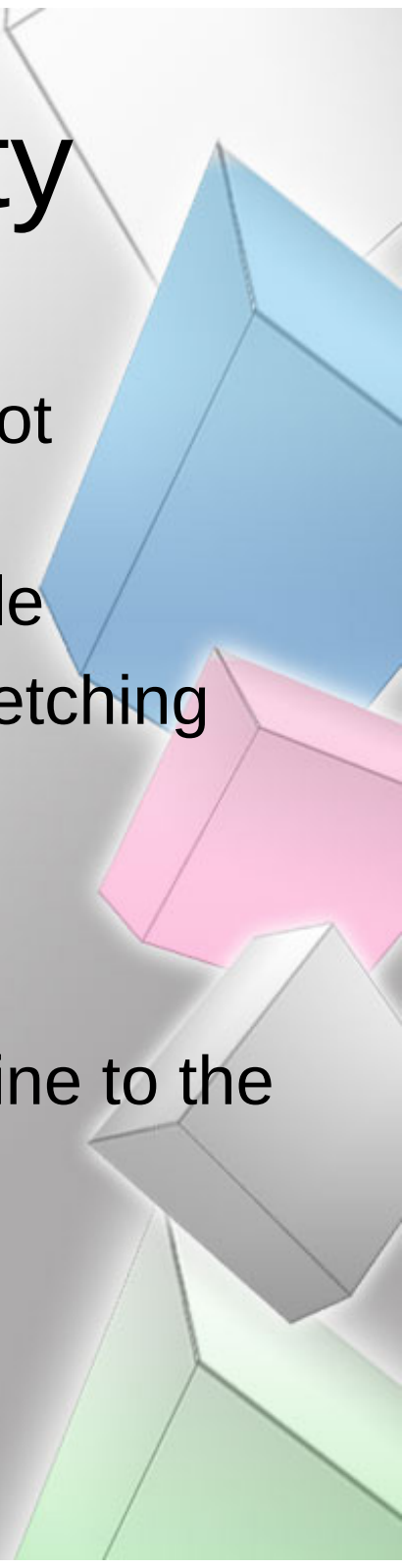
Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

Approaches to Code Migration: background

- 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:
  - The *code segment* is the part that contains the set of instructions that make up the program that is being executed.
  - The *resource segment* contains references to external resources needed. by the process, such as files, printers, devices, other processes, and so on.
  - The *execution segment* is used to store the current execution state of a process, consisting of private data, the stack, and, of course, the program counter.

Strong and weak mobility

- **Weak mobility**
  - Move only code and data segment (and reboot execution):
  - Relatively simple, especially if code is portable
  - Distinguish code shipping (push) from code fetching (pull)
- **Strong mobility**
  - Move component, including execution state
  - Migration: move entire object from one machine to the other
  - Cloning: start a clone, and set it in the same execution state.



Senders and receivers

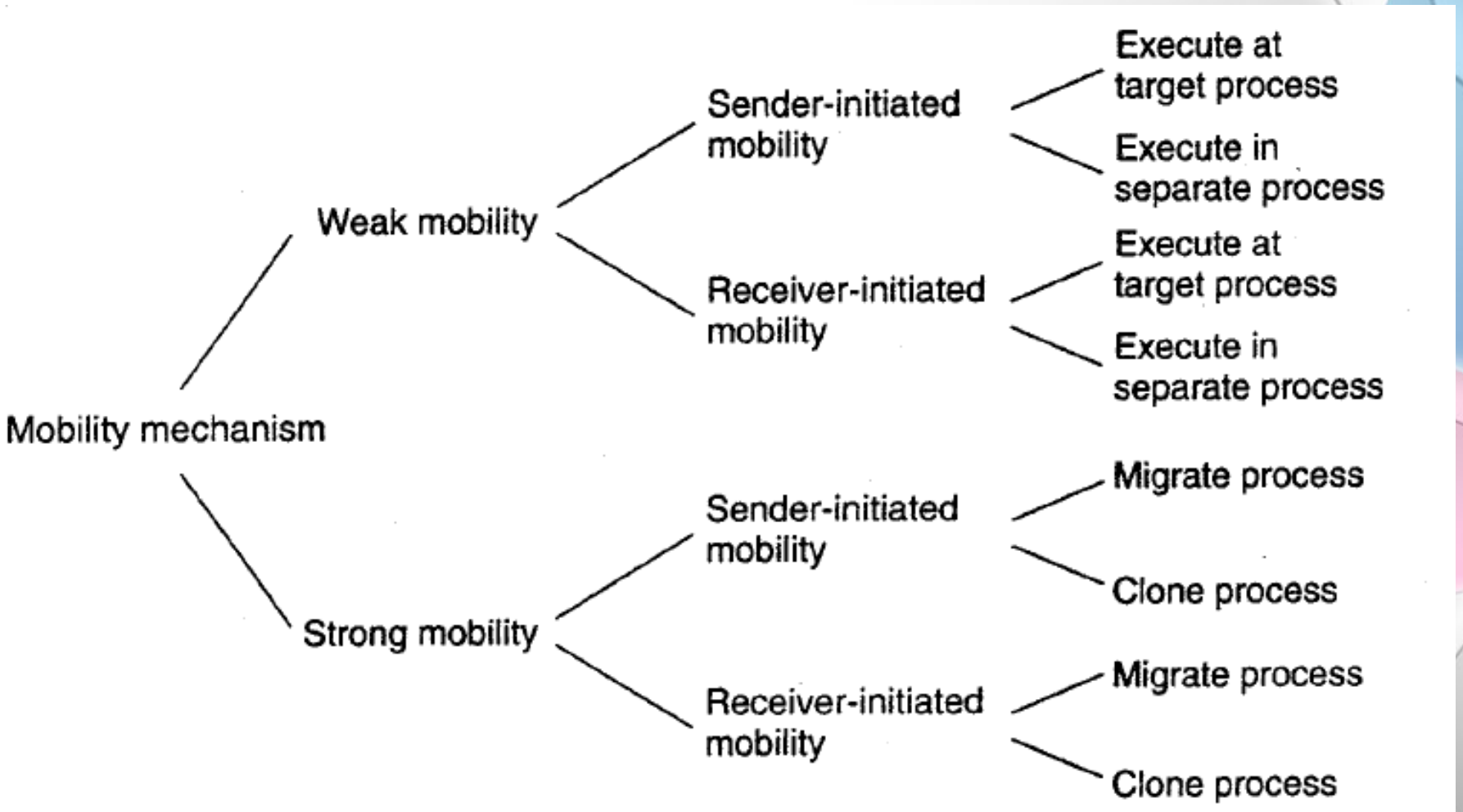


Figure 3-18. Alternatives for code migration.

Target Process

- 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. The benefit of this approach is that there is no need to start a separate process, thereby avoiding communication at the target machine.
- The main drawback is that the target process needs to be protected against malicious or inadvertent code executions.
  - A simple solution is to let the operating system take care of that by creating a separate process to execute the migrated code.

Remote Cloning

- 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.

- In UNIX systems, remote cloning takes place by forking off a child process and letting that child continue on a remote machine.

- The benefit of cloning is that 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.

- For example, suppose a process holds a reference to a specific TCP port through which it was communicating with other (remote) processes.

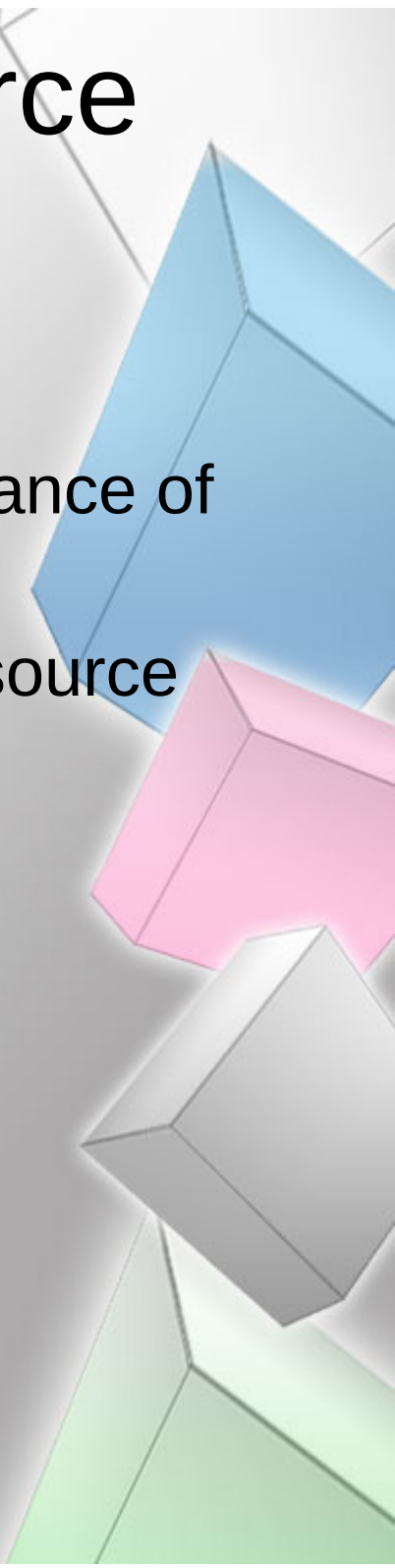
- Such a reference is held in its resource segment. When the process moves to another location, it will have to give up the port and request a new one at the destination.

- In other cases, transferring a reference need not be a problem. For example, a reference to a file by means of an absolute URL will remain valid irrespective of the machine where the process that holds the URL resides.

Migration: process-to-resource bindings

Three types of binding:

1. By identifier: the object requires a specific instance of a resource (e.g. a specific database)
2. By value: the object requires the value of a resource (e.g. the set of cache entries)
3. By type: the object requires that only a type of resource is available (e.g. a color monitor)



Binding by identifier

- The strongest binding is when a process refers to a resource by its identifier.
  - In this case, the process requires precisely the referenced resource, and nothing else.
- An example of such a binding by identifier is when a process uses a URL to refer to a specific Web site or when it refers to an FTP server by means of that server's Internet address.
- In the same line of reasoning, references to local communication end points also lead to a binding by identifier.

Binding by value

- A weaker form of process-to-resource binding is when only the value of a resource is needed.
- In this case, the execution of the process would not be affected if another resource would provide that same value.
- A typical example of binding by value is when a program relies on standard libraries, such as those for programming in C or Java.
  - Such libraries should always be locally available, but their exact location in the local file system may differ between sites. Not the specific files, but their content is important for the proper execution of the process.

Migration and local resources types

Problem

- An object uses local resources that may or may not be available at the target site.

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

- If, and exactly how reference should be changed, depends on whether that resource can be move along with the code to the target machine. More specifically, we need to consider the resource-to-machine bindings,

- **Resource types**
  - Fixed: the resource cannot be migrated, such as local hardware
  - Fastened: the resource can, in principle, be migrated but only at high cost
  - Unattached: the resource can easily be moved along with the object (e.g. a cache)

Migration and local resources

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

GR = Establish global systemwide reference

MV = Move the resource

CP = Copy the value of the resource

RB = Re-bind to a locally available resource

When the resource is unattached, it is generally best to move it along with the migrating code.

However, when the resource is shared by other processes, an alternative is to establish a global reference, that is, a reference that can cross machine boundaries.

Migration in Heterogeneous Systems

- So far, we have tacitly assumed that the migrated code can be easily executed at the target machine.
- This assumption is in order when dealing with homogeneous systems.
- 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.
- Also, we need to ensure that the execution segment can be properly represented at each platform.

Heterogeneity and portability

- The problems coming from heterogeneity are in many respects the same as those of portability.
  - Not surprisingly, solutions are also very similar.
- For example, at the end of the 1970s, a simple solution to alleviate many of the problems of porting Pascal to different machines was to generate machine-independent intermediate code for an abstract virtual machine.
  - That machine, of course, would need to be implemented on many platforms, but it would then allow Pascal programs to be run anywhere.
- Although this simple idea was widely used for some years, it never really caught on as the general solution to portability problems for other languages, notably C.

Heterogeneity: moving processes on Process VMs

- About 25 years later, code migration in heterogeneous systems is being attacked by scripting languages and highly portable languages such as Java.
  - In essence, these solutions adopt the same approach as was done for porting Pascal.

- All such solutions have in common that they rely on a (process) virtual machine that either directly interprets source code (as in the case of scripting languages), or otherwise interprets intermediate code generated by a compiler (as in Java).
- Being in the right place at the right time is also important for language developers.

Heterogeneity: moving entire computing environment

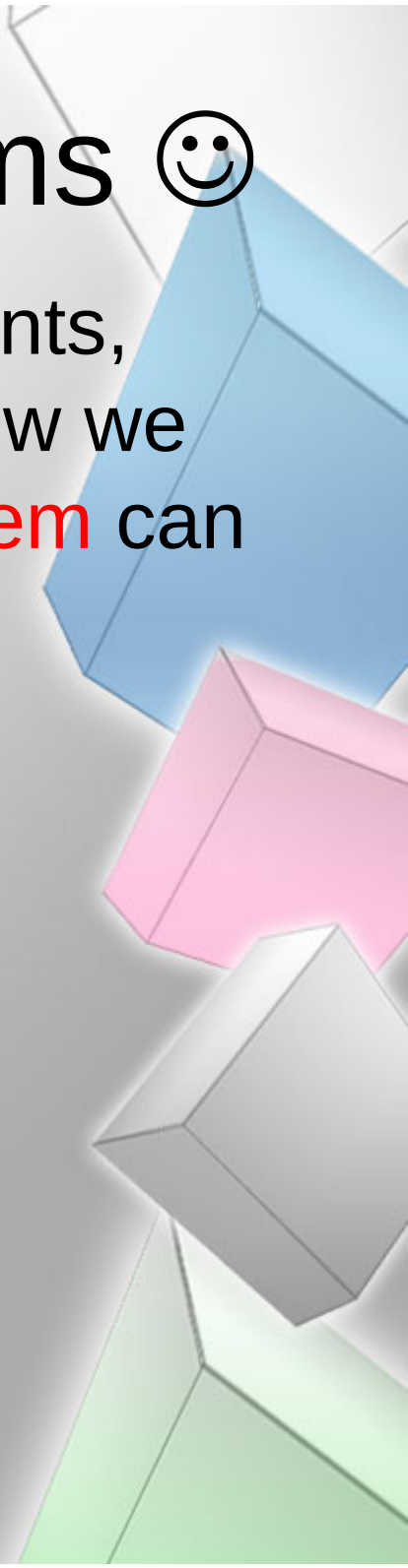
- Recent developments have started to weaken the dependency on programming languages.
- In particular, solutions have been proposed not only to migrate processes, but to migrate entire computing environments.
  - The basic idea is to compartmentalize the overall environment and to provide processes in the same part their own view on their computing environment.
- If the compartmentalization is done properly, it becomes possible to decouple a part from the underlying system and actually migrate it to another machine.

Migrating entire environments

- There are several reasons for wanting to migrate entire environments, but perhaps the most important one is that it allows continuation of operation while a machine needs to be shutdown.
- For example, in a server cluster, the systems administrator may decide to shut down or replace a machine, but will not have to stop all its running processes.
  - Instead, it can temporarily freeze an environment, move it to another machine (where it sits next to other, existing environments), and simply unfreeze it again.
- Clearly, this is an extremely powerful way to manage long-running compute environments and their processes.

Migrating Operating Systems ☺

- The overall effect of migrating environments, instead of migrating processes, is that now we actually see that an entire operating system can be moved between machines.



End of Lesson 3

- Readings
  - Distributed Systems, Chapter 3.