DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN
By: Dr. Faramarz Safi
Islamic Azad University
Najafabad Branch

Chapter 3
Processes

مدهای پردازنده

هر پردازنده دارای حداقل دارای دو مد می باشد:

۱- مد کاربر

۲- مد کرنل

در مد کرنل کلیه دستورات قابل اجرا می باشند ولی در مد کاربر بعضی از دستورات (دستورات سیستمی) قابل اجرا نمی باشند. بعنوان مثال دستوراتی از قبیل تغییر مد پردازنده، خواندن و نوشتن از/روی سخت افزارهای I/O از این قبیل می باشند.

Kernel vs. Microkernel

میکروکرنل هسته اصلی سیستم عامل می باشد که وظایف اصلی جهت ماژولار کردن سیستم عامل را بعهده دارد. قابلیت انعطاف، گسترش، حمل، اطمینان، پشتیبانی از سیستم های توزیع شده و استاندارد سازی اجزاء سیستم عامل از جمله مزایای این سیستم ها می باشد.

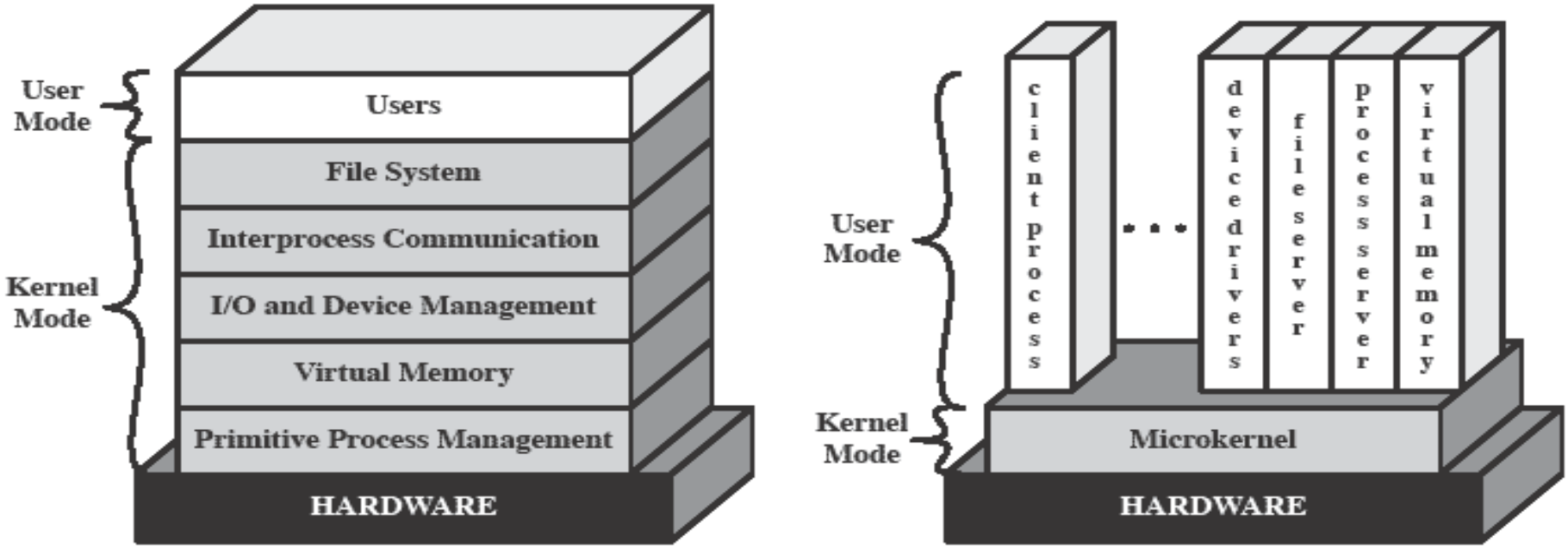


Figure 4.10 Kernel Architecture

Processes and Threads

- برنامه یا Program به خروجی کامپایلر گفته می شود که حاوی دستورات سطح ماشین (زبان ماشین) می باشد.
- پروسه به برنامه در حال اجرا گفته می شود و یا بعبارتی روح اجرایی یک برنامه می باشد.
- وظایف در یک سیستم عامل به پروسه ها شکسته می شود.
- برای نگاهداری اطلاعات زمان اجرایی یک پروسه از ساختاری بنام Process Control Block یا PCB استفاده می شود.
- در یک پروسه وظایف بین Thread ها (نخ ها) تقسیم می شود.
- اطلاعات اجرایی یک نخ در ساختاری بنام Thread Control Block یا TCB نگاهداری می شود.
- هنگام اجرای نخ ها به هر یک برش زمانی اختصاص می یابد و پس از اتمام برش زمانی متن نخ اجرایی عوض می شود(Context Switch) . بعبارتی آخرین وضعیت ثباتها و اطلاعات سیستمی در ساختار TCB نخ فعلی ذخیره می شود و آخرین وضعیت نخ آماده برای اجرای بعدی از TCB نخ بعدی بارگذاری شده و اجرا می گردد.

Multithreading

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

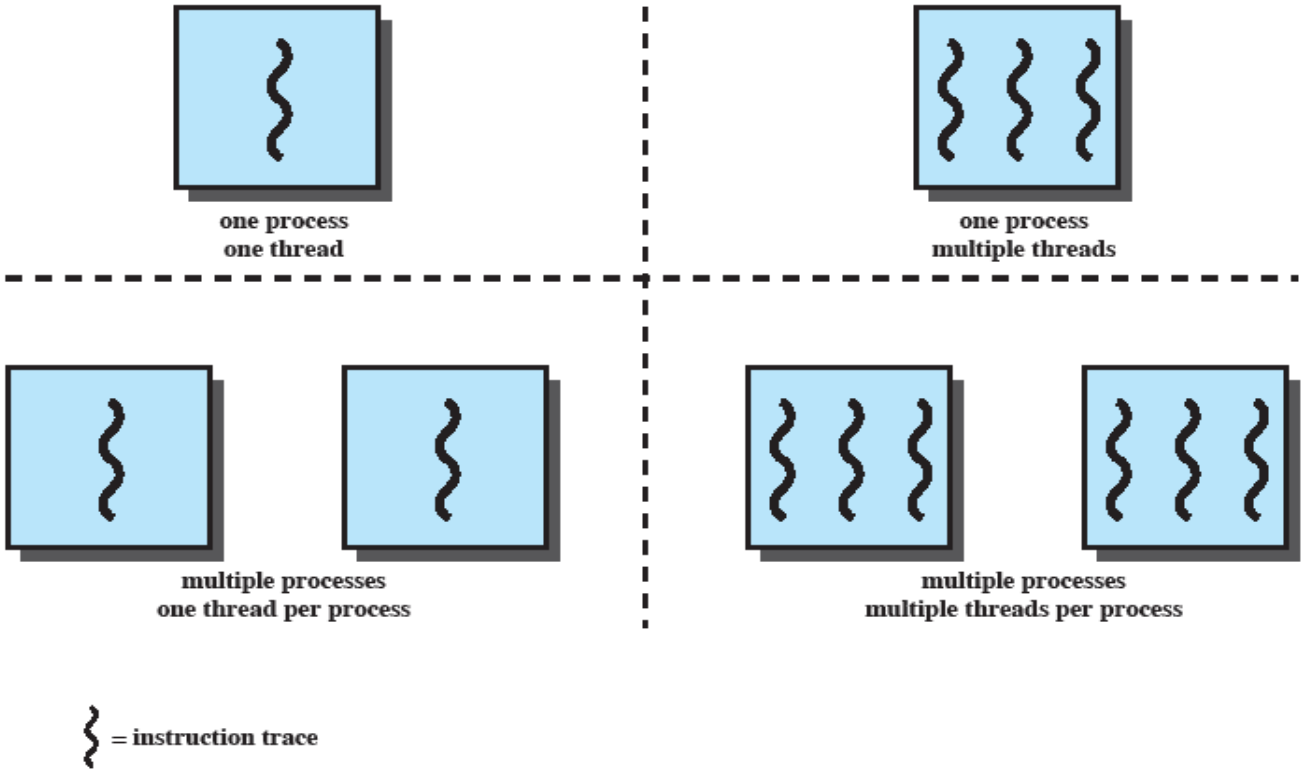


Figure 4.1 Threads and Processes [ANDE97]

Threads vs. processes

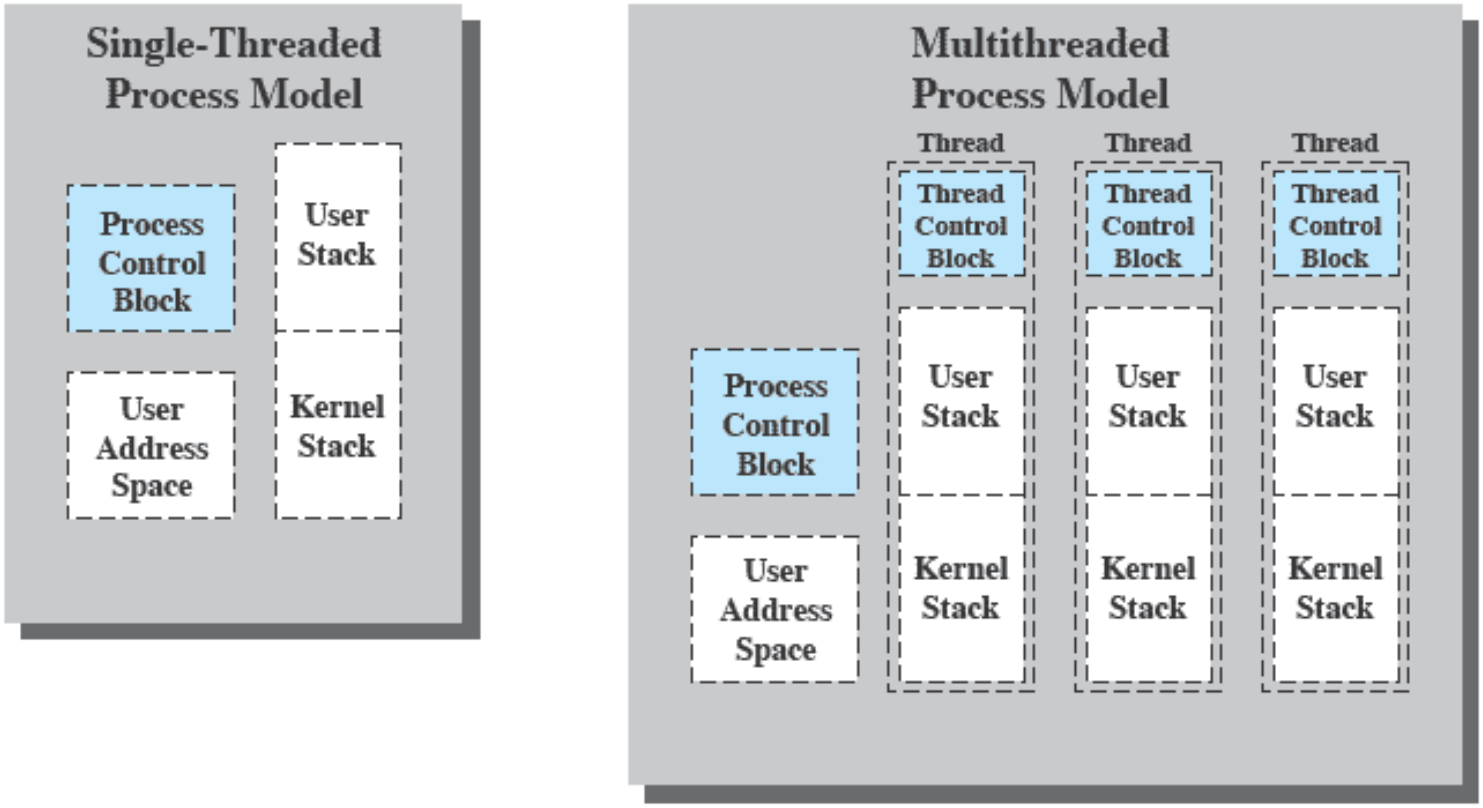


Figure 4.2 Single Threaded and Multithreaded Process Models

Introduction to Processes and Threads

To understand the role of threads in distributed systems, it is important to understand what a process is, and how processes and threads relate. To execute a program, an operating system creates a number of **virtual processors**, each one for running a different program.

To keep track of these virtual processors, the operating system has a process table, containing entries to store CPU register values, memory maps, open files, accounting information, privileges, etc.

A process is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors.

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

Introduction to Processes and Threads

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, a data segment, copying the associated program into a code segment, and setting up a stack segment for temporary data.

Likewise, **switching the CPU between two processes** may be relatively **expensive** as well. Apart from saving the CPU context (which consists of register values, program counter, stack pointer, etc.), the operating system will also have to modify registers of the memory management unit (MMU) and invalidate address translation caches such as in the translation look-aside buffer (TLB).

In addition, if the operating system supports more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.

Introduction to Processes and Threads

Like a process, a thread executes its own piece of code, independently from other threads. 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. For example, a thread system may keep track of the fact that a thread is currently blocked on a mutex variable, so as not to select it for execution.

Information that is not strictly necessary to manage multiple threads is generally ignored. For instance, protecting data against inappropriate access by threads within a single process is left entirely to application developers.

Two implications of multi-thread systems

There are two important implications of this approach.:

First of all, the performance of a multithreaded application is not worse than that of its single-threaded counterpart. In fact, in many cases, multithreading leads to a performance gain.

Second, because threads are not automatically protected against each other, development of multithreaded applications requires additional intellectual effort.

Threads in non-distributed systems

- Dividing a job among several threads and different control points. i.e. Microsoft Excel uses different threads to control different aspects.
- Another advantage of multithreading is that it becomes possible to exploit parallelism when executing the program on a multiprocessor system.
- 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. Such as Unix process that use Inter Process Communication (IPC) mechanism which impose several problems. Alternatively, there is also a pure software engineering reason to use threads: many applications are simply easier to structure as a collection of cooperating threads. In word processor, separate threads can be used for handling user input, spelling and grammar checking, document layout, index generation, etc.

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

Because IPC requires kernel intervention, a process will generally:

- First have to switch from user mode to kernel mode. This requires changing the memory map in the MMU, as well as flushing the TLB.
- Within the kernel, a process context switch takes place, after which the other party can be activated by switching from kernel mode to user mode again.
- The latter switch again requires changing the MMU map and flushing the TLB.

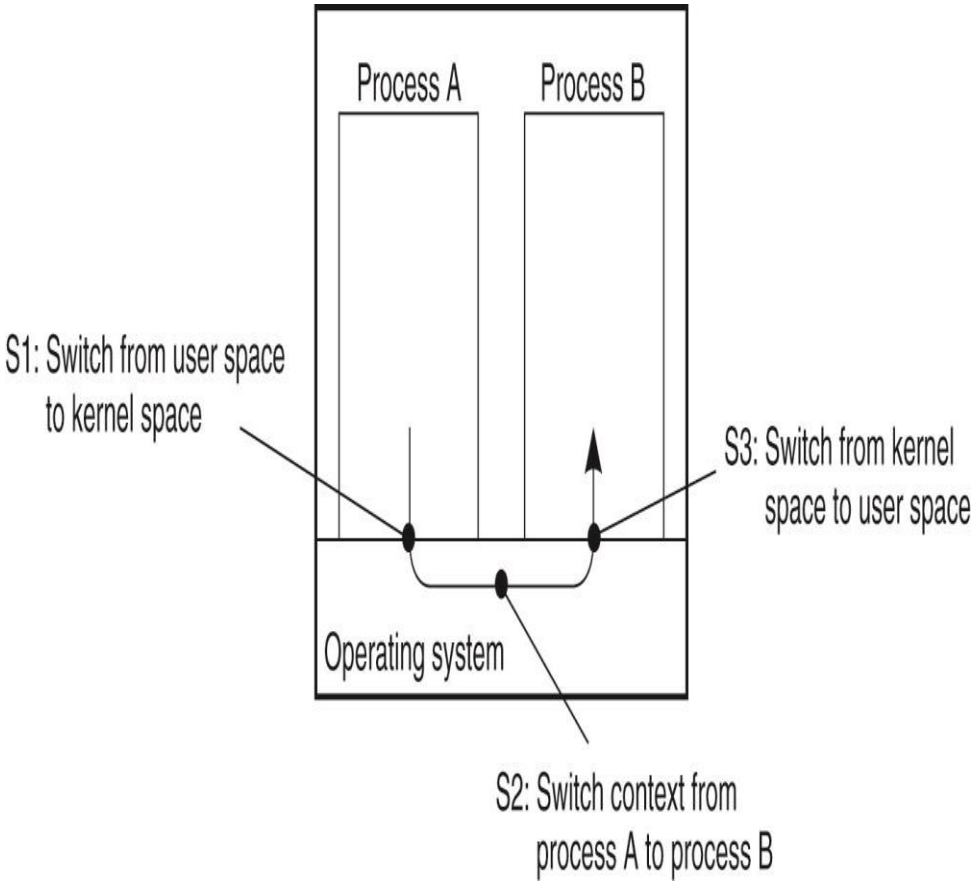


Figure 3-1. Context switching as the result of IPC.

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

Instead of using processes, an application can also be constructed such that different parts are executed by separate threads.
Communication between those parts is entirely dealt with by using shared data.

Thread switching can sometimes be done entirely in user space, although in other implementations, the kernel is aware of threads and schedules them. The effect can be a dramatic improvement in performance.

Thread Implementation

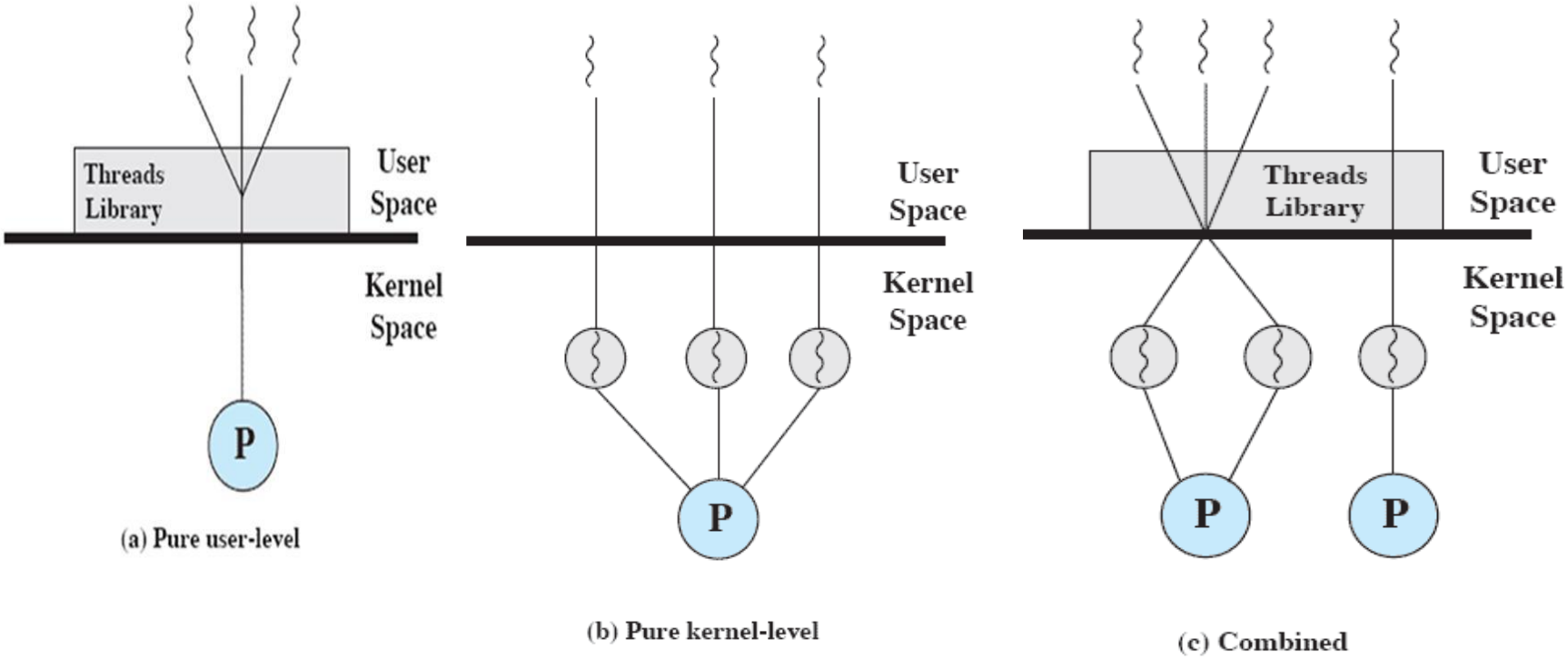
Threads are often provided in the form of a thread package. Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes and condition variables. There are basically **three** approaches to implement a thread package.

The first approach is to construct a thread library that is executed entirely in user mode, called **User Level Threads (ULT)**.

The second approach is to have the kernel be aware of threads and schedule them, called **Kernel Level Threads (KLT)**.

The third approach is a **Hybrid** Form.

ULT Vs. KLT


Threads
Library
User
Space
Kernel
Space
P
(a) Pure user-level
User
Space
Kernel
Space
P
(b) Pure kernel-level
Threads
Library
User
Space
Kernel
Space
P
P
(c) Combined

User-Level Threads
Advantages

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.
- Analogously, destroying a thread mainly involves freeing memory for the stack.

- A second advantage of user-level threads is that switching thread context can often be done in just a few instructions.

- Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched. There is no need to change memory maps, flush the TLB, do CPU accounting, and so on.

User-level Threads
Disadvantages

- A major drawback of user-level threads is that invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process.

Threads are particularly useful to structure large applications into parts that could be logically executed at the same time. In that case, blocking on I/O should not prevent other parts to be executed in the meantime.

Kernel-Level Threads

These problems can be mostly circumvented by implementing threads in the operating system's kernel. Unfortunately, there is a high price to pay:

Every thread operation (creation, deletion, synchronization, etc.) will have to be carried out by the kernel which require a system call.

Switching thread contexts may now become as expensive as switching process contexts. As a result, most of the performance benefits of using threads instead of processes then disappears.

Kernel-Level Threads Advantages

1) The kernel can simultaneously schedule multiple threads from the same process on multiple processors.

2) If one thread in a process is blocked, the kernel can schedule another thread of the same process.

3) Kernel routines themselves can be multithreaded.

Kernel-Level Threads Disadvantages

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

Hybrid Approach

A solution lies in a hybrid form of user-level and kernel-level threads, generally referred to as **lightweight processes** (LWP). An LWP runs in the context of a single (heavy-weight) process, and there can be several LWPs per process.

In addition to having LWPs, a system also offers a user-level thread package. Offering applications the usual operations for creating and destroying threads.

In addition, the package provides facilities for thread synchronization such as mutexes and condition variables.

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

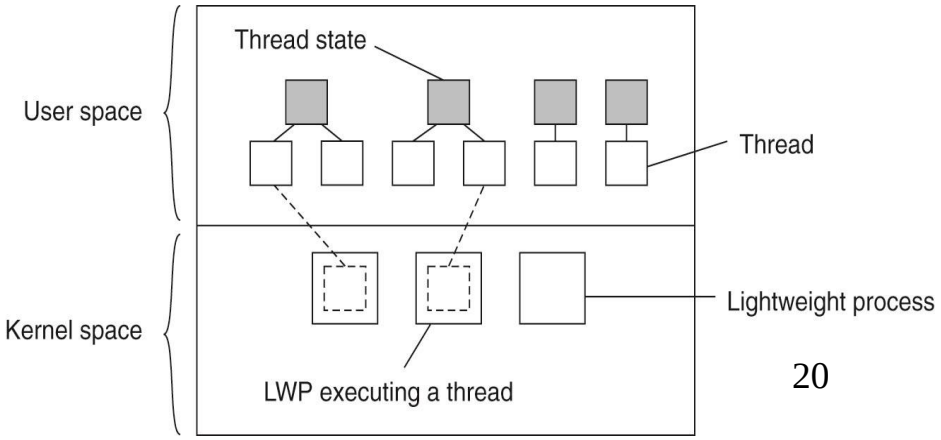


Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

Hybrid Approach

The combination of (user-level) threads and LWPs works as follows:

The thread package has a single routine to schedule the next thread. When creating an LWP (which is done by means of a system call), the LWP is given its own stack, and is instructed to execute the scheduling routine in search of a thread to execute.

If there are several LWPs, then each of them executes the scheduler. When an LWP finds a runnable thread, it switches context to that thread.

The beauty of all this is that the LWP executing the thread need not be informed: the context switch is implemented completely in user space and appears to the LWP as normal program code.

Hybrid Approach
Blocking Call

Now let us see what happens when a thread does a blocking system call.
The execution changes from user mode to kernel mode, but still continues in the context of the current LWP.

At the point where the current LWP can no longer continue, the operating system may decide to switch context to another LWP, which also implies that a context switch is made back to user mode.

The selected LWP will simply continue where it had previously left off.

Hybrid Approach
Advantages and Disadvantages

There are several advantages to using LWPs in combination with a user-level thread package.

First, creating, destroying, and synchronizing threads is relatively cheap and involves no kernel intervention at all.

Second, provided that a process has enough LWPs, a blocking system call will not suspend the entire process.

Third, there is no need for an application to know about the LWPs. All it sees are user-level threads.

Fourth, LWPs can be easily used in multiprocessing environments, by executing different LWPs on different CPUs. This multiprocessing can be hidden entirely from the application.

The only **drawback** of lightweight processes in combination with user-level threads is that we still need to create and destroy LWPs, which is just as expensive as with kernel-level threads. However, creating and destroying LWPs needs to be done only occasionally, and is often fully controlled by the operating system.

Hybrid Approach scheduler activations

The most essential difference between scheduler activations and LWPs is that when a thread blocks on a system call, the kernel does an ***upcall*** *to the thread package, effectively calling the* scheduler routine to select the next runnable thread.

The same procedure is repeated when a thread is unblocked. The advantage of this approach is that it saves management of LWPs by the kernel.

However, the use of upcalls is considered less elegant, as it violates the structure of layered systems, in which calls only to the next lower-level layer are permitted.

Threads in Distributed Systems
Multithreaded Clients

The usual way to hide communication latencies is to initiate communication and immediately proceed with something else. A typical example where this happens is in Web browsers.
A Web browser often starts with fetching the HTML page and subsequently displays it.

To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in.

As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts. Each thread sets up a separate connection to the server and pulls in the data. Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process.

Threads in Distributed Systems
Multithreaded Servers

Although there are important benefits to multithreaded clients, the main use of multithreading in distributed systems is found at the server side.

Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uni-processor systems.

However, now that multiprocessor computers are widely available as general-purpose workstations, multithreading for parallelism is even more useful.

To understand the benefits of threads for writing server code, consider the organization of a file server that occasionally has to block waiting for the disk.

Multithreaded 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.

Multithreaded Servers

The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply. In Fig. 3-3, a **dispatcher** thread, reads incoming requests for a file operation. After examining the request, the server chooses an idle (i.e., blocked) **worker thread** and hands it the request**.**

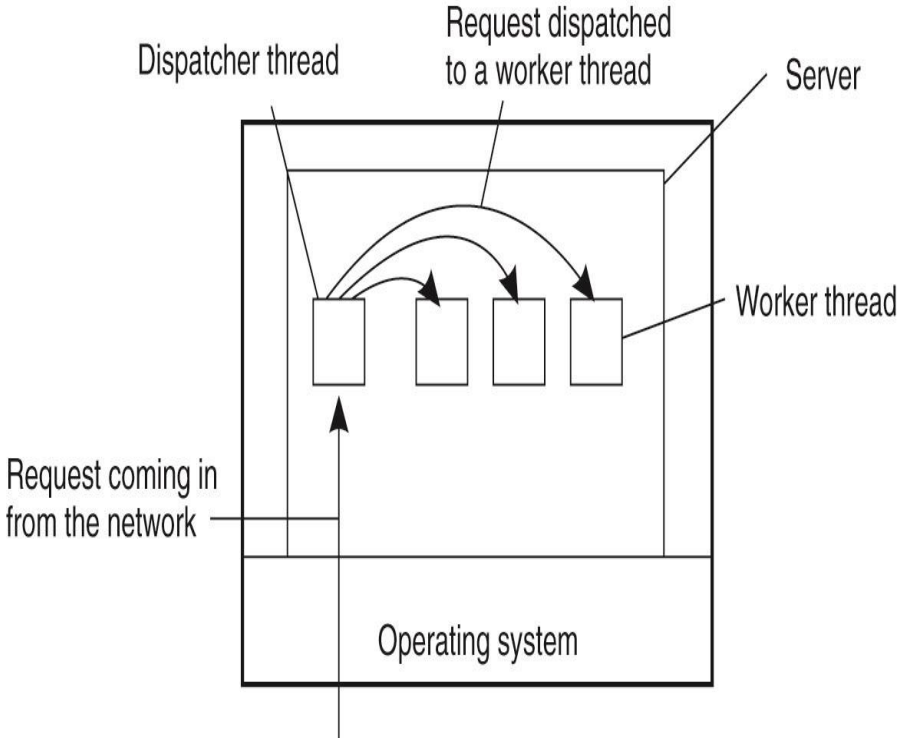


Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

Single-thread Server

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

The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one.

While waiting for the disk, the server is idle and does not process any other requests. Consequently, requests from other clients cannot be handled.

In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk.

The net result is that many fewer requests/sec can be processed.

Finite-sate machine Server

A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.

However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message.

The next message may either be a request for new work or a reply from the disk about a previous operation.
If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed and subsequently sent to the client.

In this scheme, the server will have to make use of non-blocking calls to send and receive.

The Role of Virtualization in Distributed Systems

In practice, every (distributed) computer system offers a programming interface (API) to higher level software, as shown in Fig. 3-5(a).

There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware systems.

In its essence, virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system, as shown in Fig.3-5(b).

The Role of Virtualization in Distributed Systems

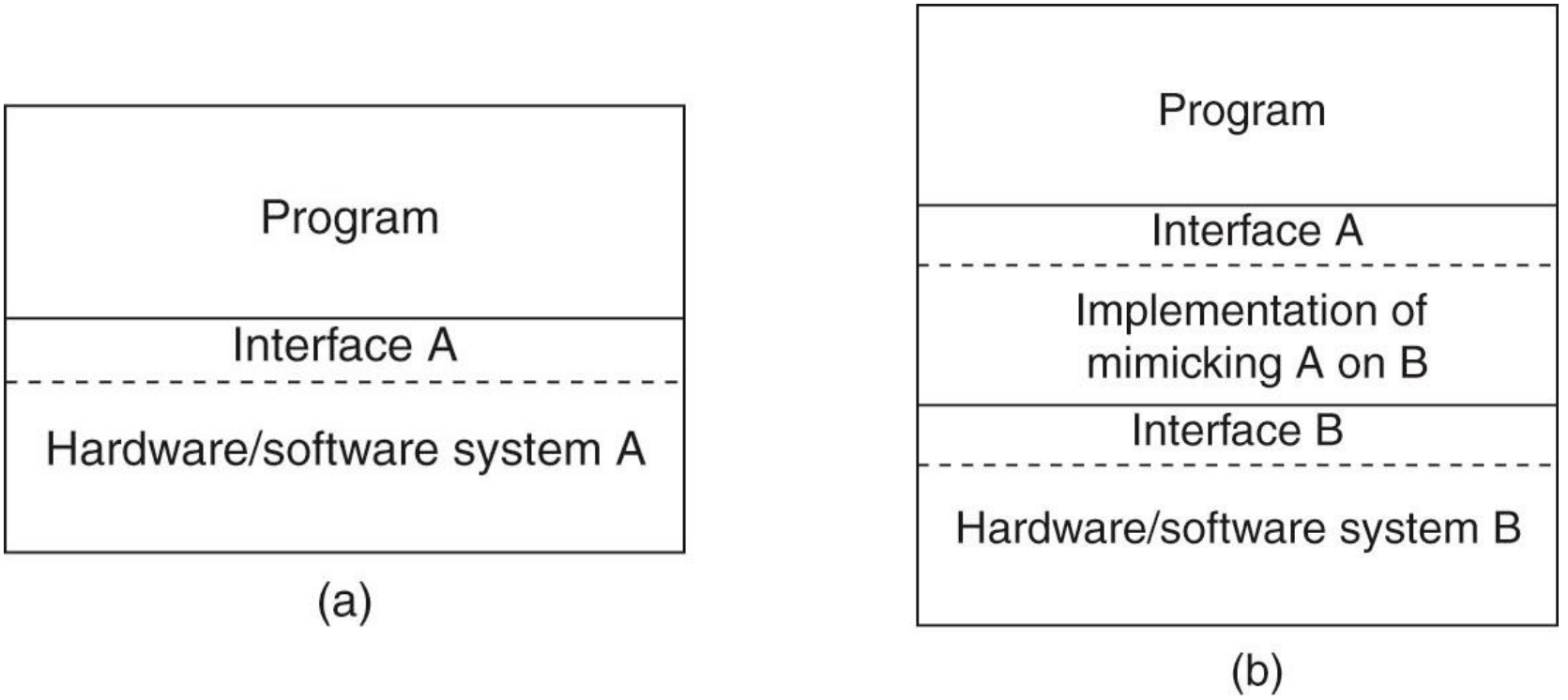


Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

Architectures of Virtual Machines (1)

Interfaces at different levels

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

Architectures of Virtual Machines (2)

Interfaces at different levels

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

Architectures of Virtual Machines (3)

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

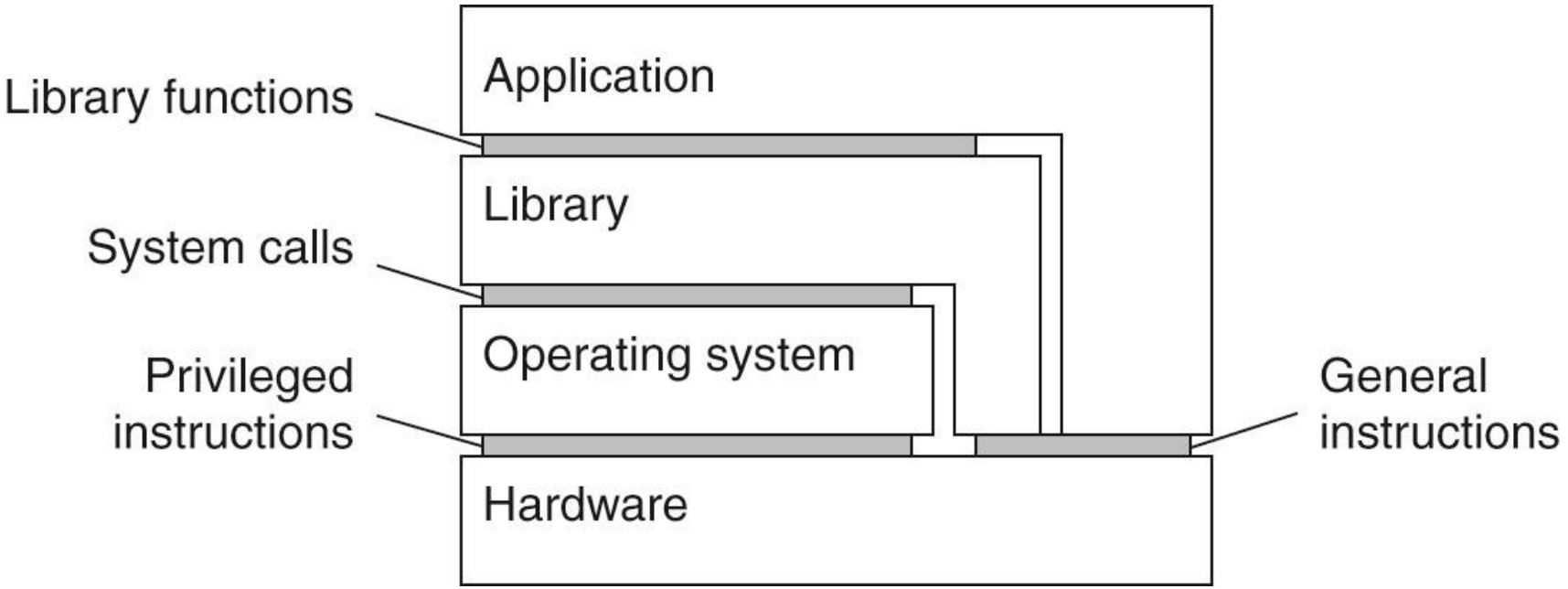


Figure 3-6. Various interfaces offered by computer systems.

Two Different Ways of Making Virtual Machines

Virtualization can take place in two different ways:

1- First, we can build a runtime system that essentially provides an abstract instruction set that is to be used for executing applications. Instructions can be **interpreted** (i.e. the Java runtime environment), but could also be **emulated** (i.e. running Windows applications on UNIX platforms). This type of virtualization leads to **process virtual machine**, stressing that virtualization is done essentially **only for a single process**.

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

Architectures of Virtual Machines (4)

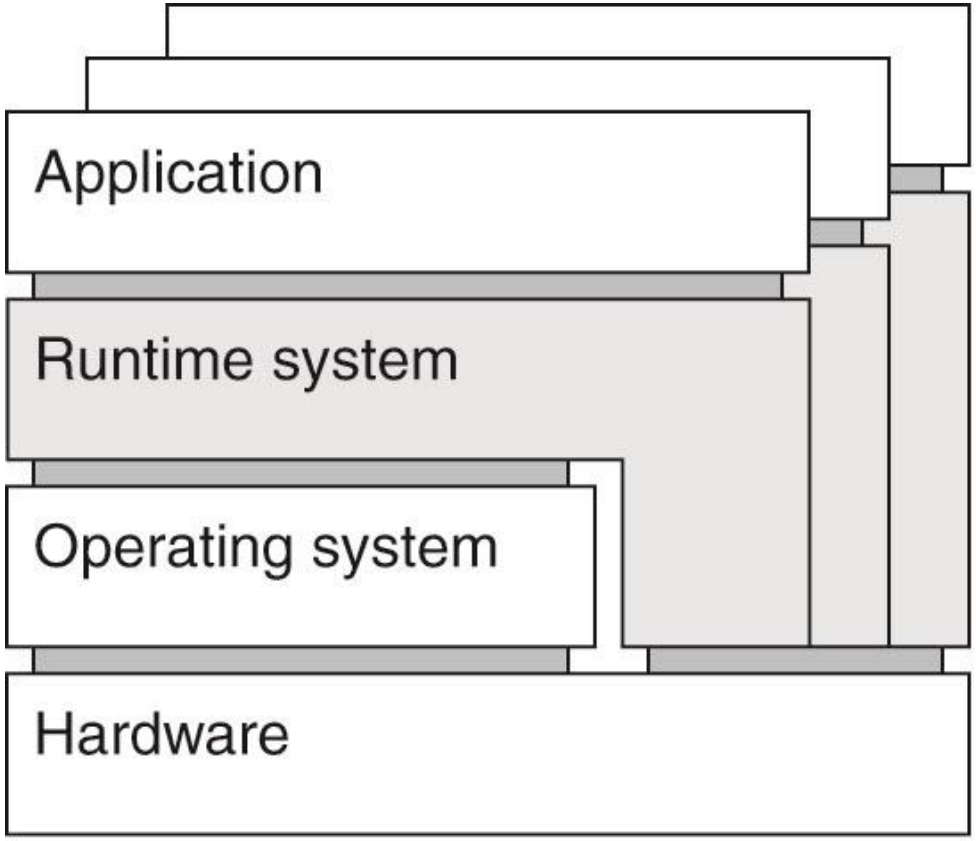


Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations.

Architectures of Virtual Machines (5)

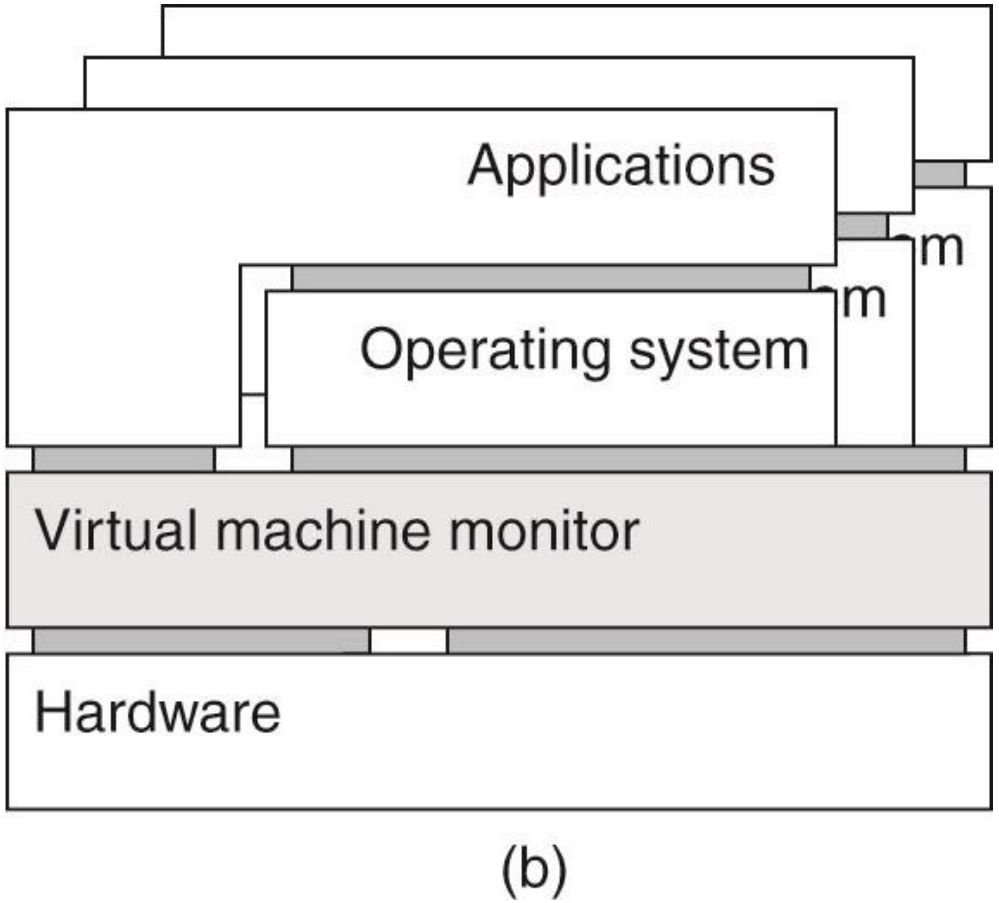


Figure 3-7. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

Clients

A major task of client machines is to provide the means for users to interact with remote servers. There are roughly two ways in which this interaction can be supported.

First, for each remote service the client machine will have a separate counterpart that can contact the service over the network (Fig. 3-8(a), Fig. 3-8(b)).

A second solution is to provide direct access to remote services by only offering a convenient user interface. Effectively, this means that the **client machine is used only as a terminal** with no need for local storage, leading to an application neutral solution as shown in (Fig.3-9).

Clients
Networked User Interfaces (1)

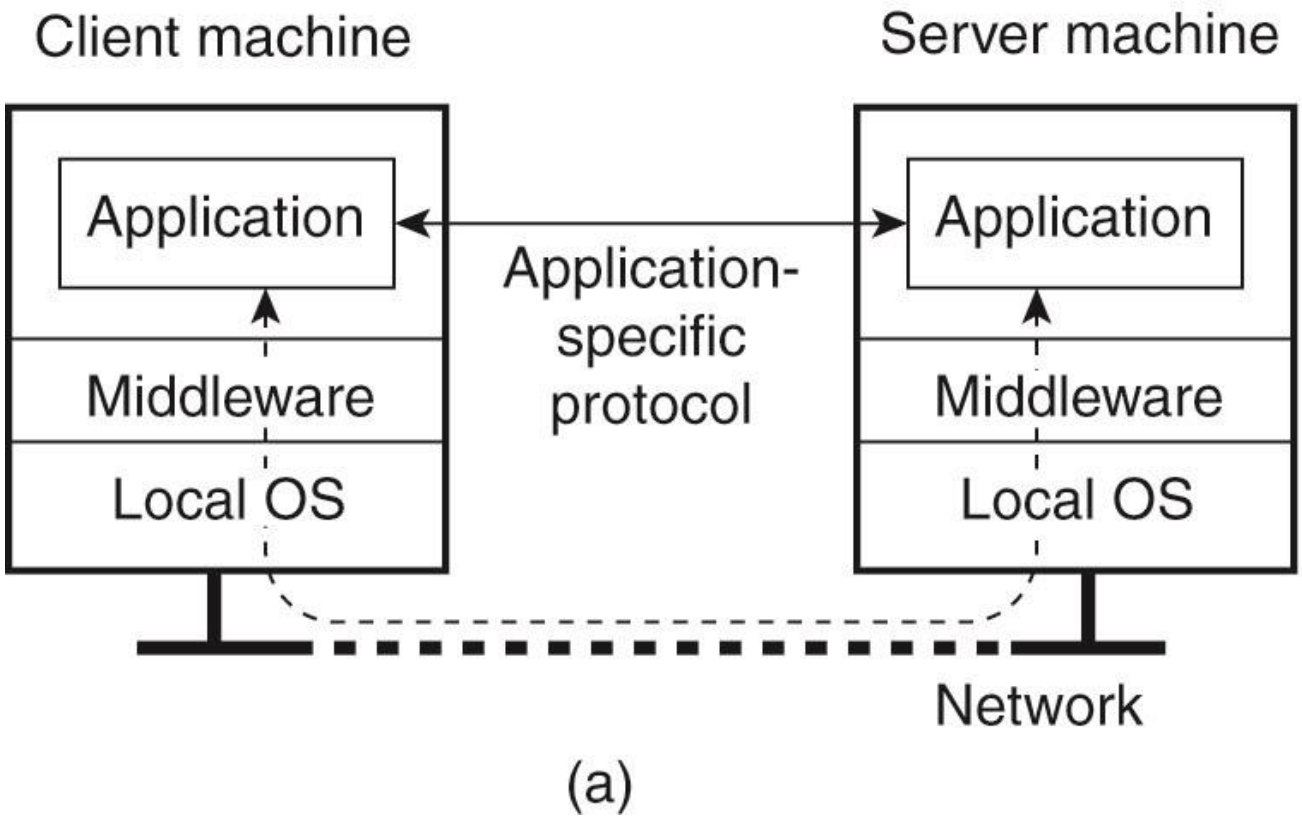


Figure 3-8. (a) A networked application with its own protocol.

Clients
Networked User Interfaces (2)

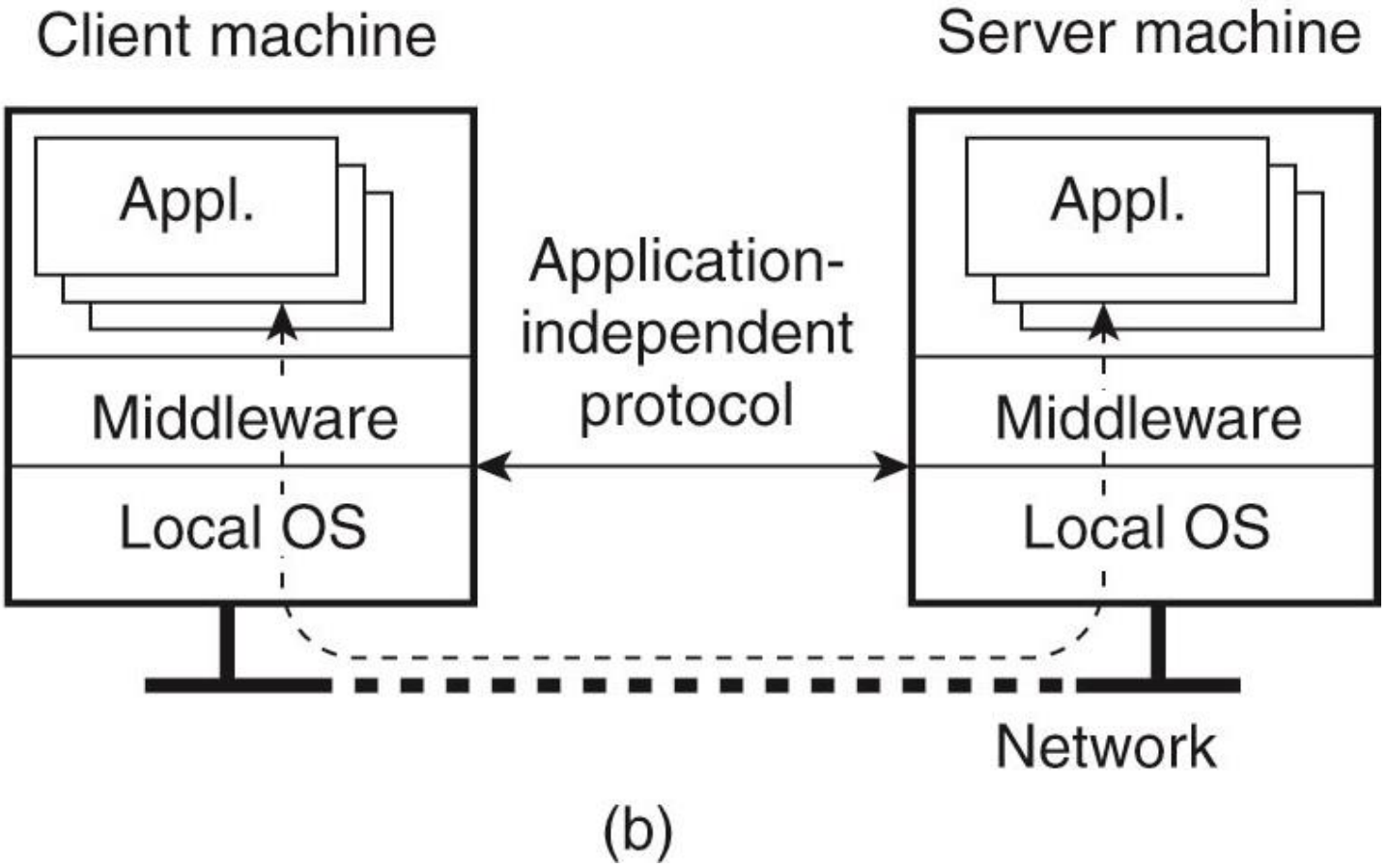


Figure 3-8. (b) A general solution to allow access to remote applications.

Clients
Example: The XWindow System

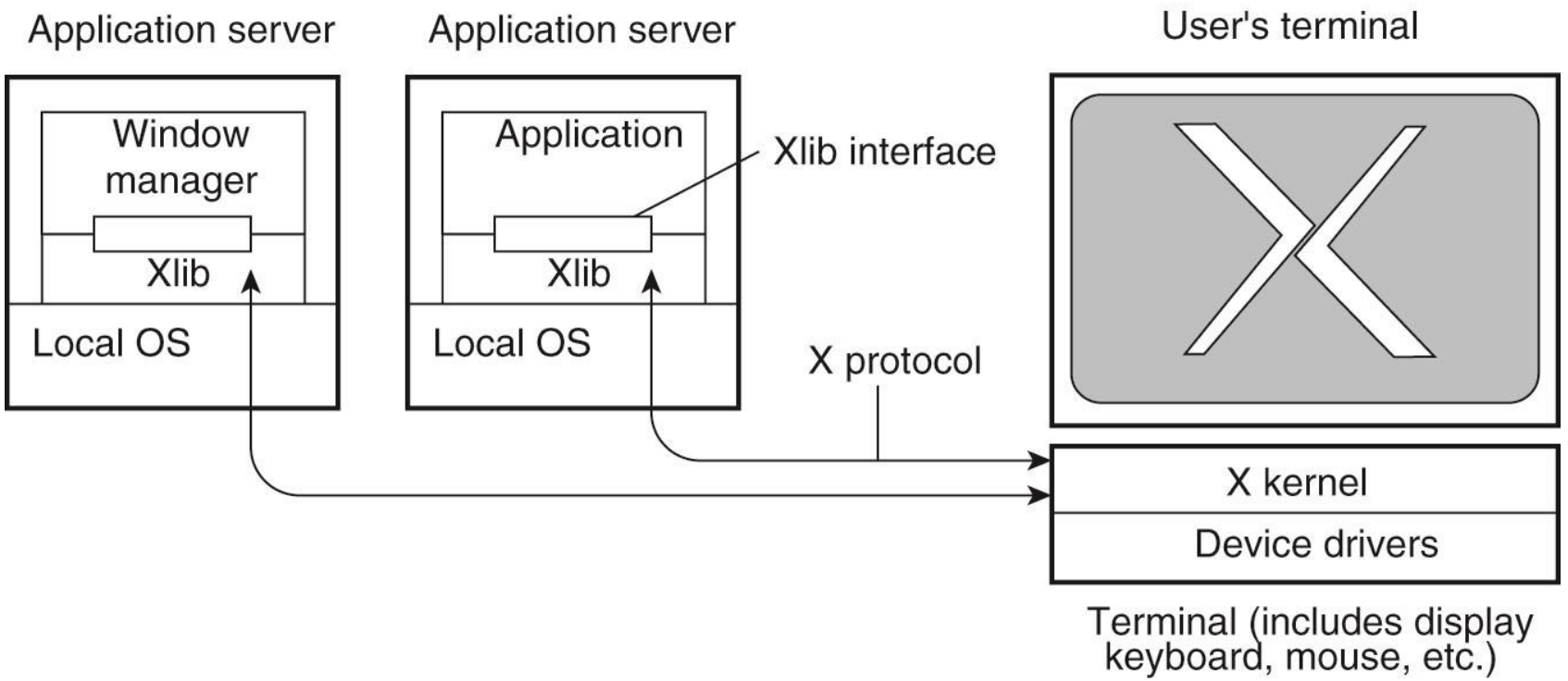


Figure 3-9. The basic organization of theXWindow System.

Clients
Client-Side Software for Distribution Transparency

Client software comprises more than just user interfaces. In many cases, parts of the processing and data level in a client-server application are executed on the client side as well. A special class is formed by embedded client software, such as for automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, etc.

Besides the user interface and other application-related software, client software comprises components for achieving **distribution transparency**. **Access transparency** is generally handled through the generation of a **client stub** from an **interface definition (IDL)** of what the server has to offer. There are different ways to handle **location, migration, and relocation transparency**. Using a convenient **naming system** is crucial. Many distributed systems implement **replication transparency** by means of client-side solutions.

Clients
Client-Side Software for Distribution Transparency

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

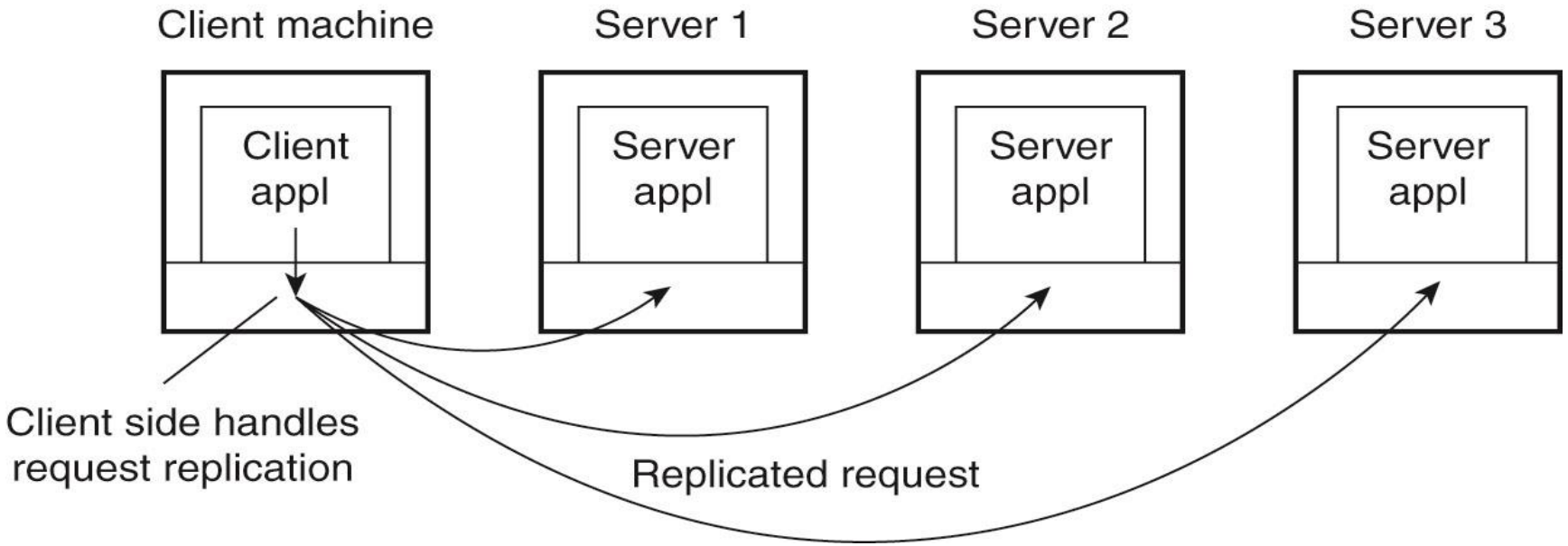


Figure 3-10. Transparent replication of a server using a client-side solution.

Servers
General Issues

A server is a process implementing a specific service on behalf of a collection of clients. In essence, each server is organized in the same way: it waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.

In the case of an **iterative server**, the server itself handles the request and, if necessary, returns a response to the requesting client.

A **concurrent server** does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request. A **multithreaded server** is an example of a concurrent server. An alternative implementation of a concurrent server is **to fork a new process for each new incoming request**.

Servers
General Issues

An issue is where clients contact a server. In all cases, clients send requests to an **end point**, also called a **port**, at the machine where the server is running. Each server listens to a specific end point.
How do clients know the end point of a service?

The approach is to globally **assign end points for well-known services**. For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80.

Servers
General Issues

There are many services that do not require a pre-assigned end point (i.e. a time-of-day server). So, a client will first have to look up the end point.

One solution is to have a special **daemon** running on each machine that runs servers. The daemon keeps track of the current end point of each service implemented by a co-located server. The **daemon itself listens to a well-known end point**. A client will first contact the daemon, request the end point, and then contact the specific server, as shown in Fig. 3-11(a).

It is often more efficient to have a single **super-server** listening to each end point associated with a specific service, as shown in Fig. 3-1l(b). For example, the *inetd daemon in UNIX istens to a number of* well-known ports for Internet services. When a request comes in, the daemon forks a process to process the request. That process will exit after it is finished.

Servers
General Design Issues

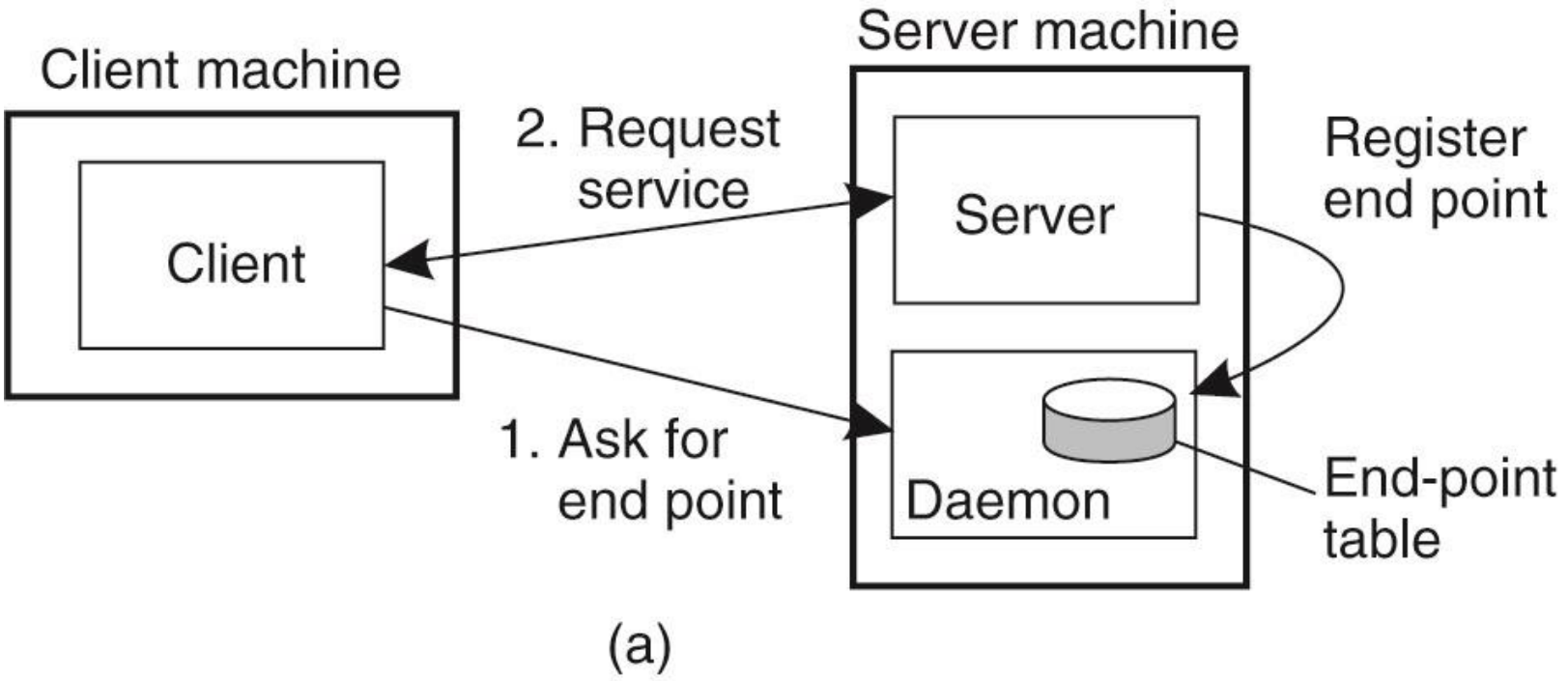


Figure 3-11. (a) Client-to-server binding using a daemon.

Servers
General Design Issues

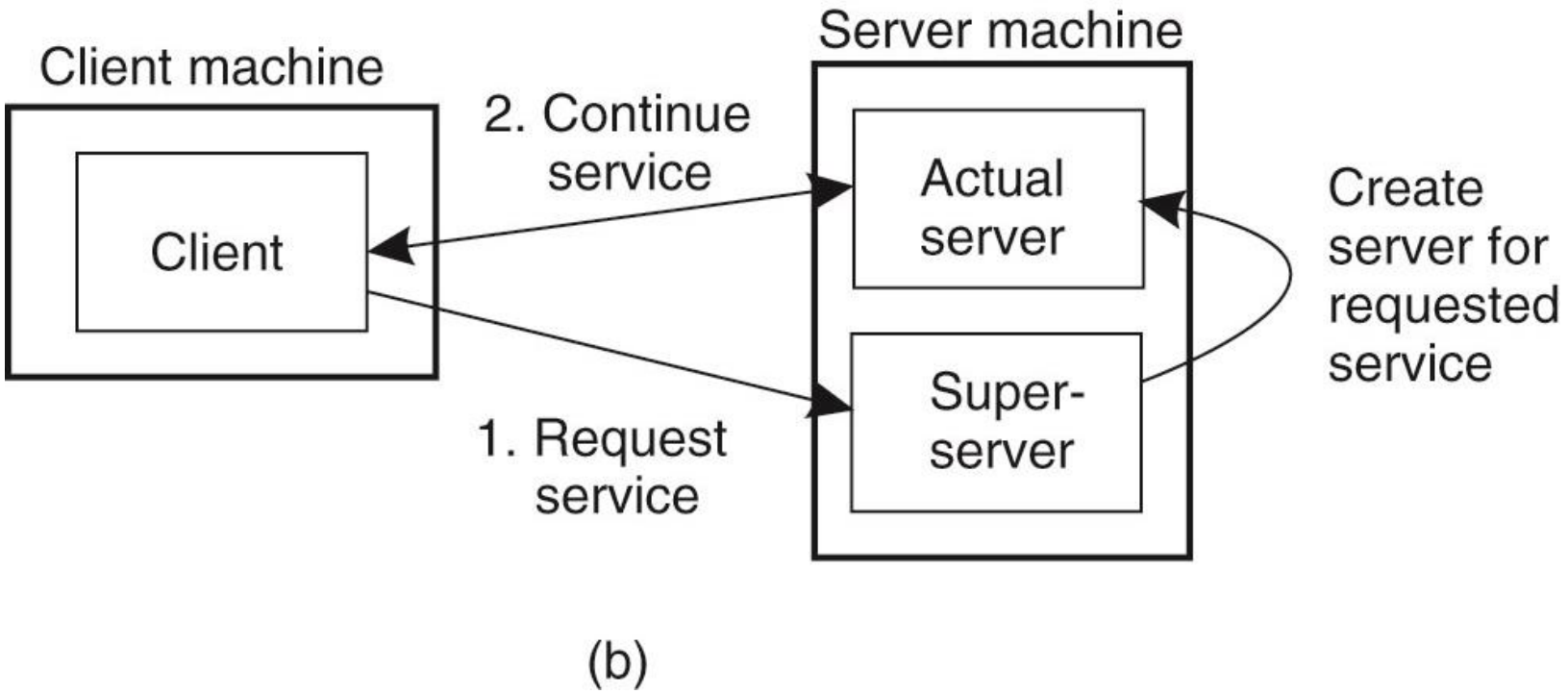


Figure 3-11. (b) Client-to-server binding using a superserver.

Servers
General Issues

A final, important design issue, is whether or not the server is **stateless or stateful**.

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

Servers
Server Clusters (1)

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

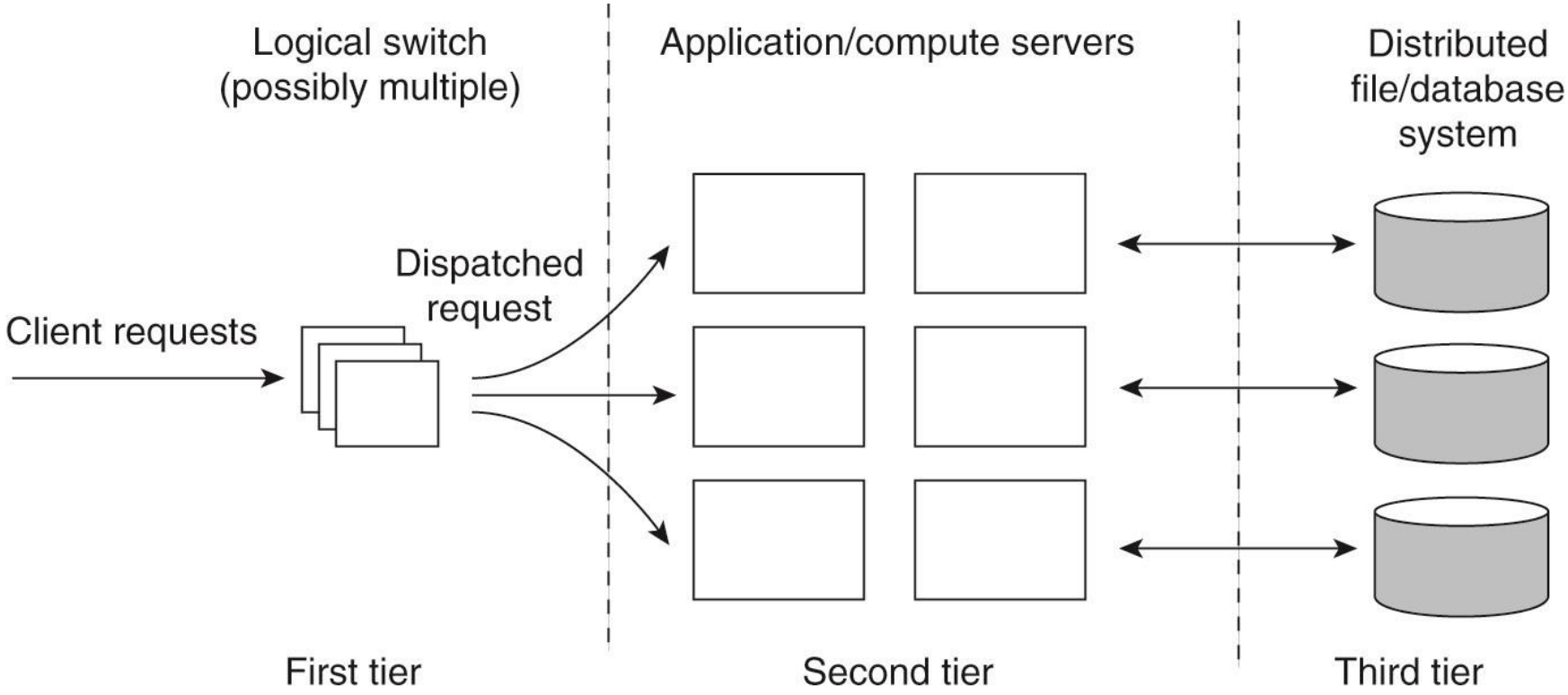


Figure 3-12. The general organization of a three-tiered server cluster.

Servers
Server Clusters (2)

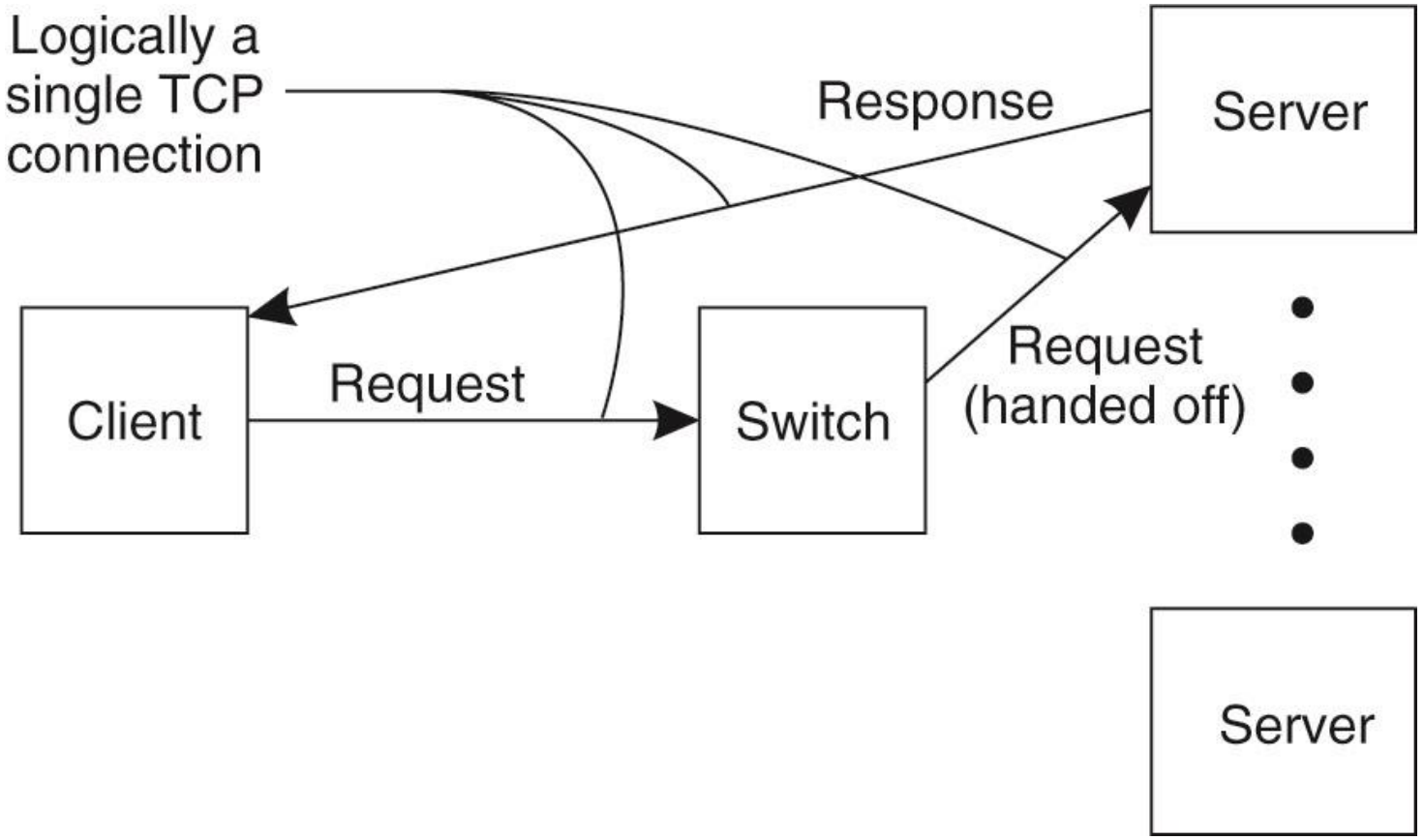


Figure 3-13. The principle of TCP handoff.

Servers
Distributed Servers

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 and powerful machine. The design of such a distributed server is given in “Szymaniak et al. (2005)”.

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.

Servers
Distributed Servers

So far, server clusters are generally:

1- rather statically configured. There is often a separate administration machine that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.

2- offer a single access point. When that point fails, the cluster becomes unavailable.

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

To eliminate this potential problem, several access points can be provided, of which the addresses are made publicly available.

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

Servers
Distributed Servers

Let us concentrate on how a **stable access point** can be achieved in such a system. The main idea is to make use of available networking services, notably **mobility support** for IP version 6 (MIPv6).

In MIPv6, a mobile node is assumed to have a **home network** where it normally resides and for which it has an associated stable address, known as its **Home Address (HoA)**. This home network has a special router attached, known as the **home agent**, which will take care of traffic to the mobile node when it is away.

To this end, when a mobile node attaches to a foreign network, it will receive **a temporary Care-of Address (CoA)** where it can be reached. This care-of address is reported to the node's home agent who will then see to it that all traffic is forwarded to the mobile node. Note that applications communicating with the mobile node **will only see the address associated with the node's home network**. They will **never** see the care-of address.

Servers
Distributed Servers

This principle can be used **to offer a stable address of a distributed server**. In this case, **a single unique contact address is initially assigned to the server cluster**. The contact address will be the server's life-time address to be used in all communication with the outside world.

At any time, **one node in the distributed server will operate as an access point using that contact address**, but this role can easily be taken over by another node. What happens is that **the access point records its own address as the care-of address at the home agent associated with the distributed server**.
At that point, all traffic will be directed to the access point, who will then take care in distributing requests among the currently participating nodes. If the access point fails, a simple fail-over mechanism comes into place by which another access point reports a new care-of address.

Servers
Distributed Servers

This simple configuration would **make the home agent as well as the access point a potential bottleneck** as all traffic would flow through these two machines. This situation can be avoided by using an MIPv6 feature known as ***route optimization***. Route optimization works as follows.

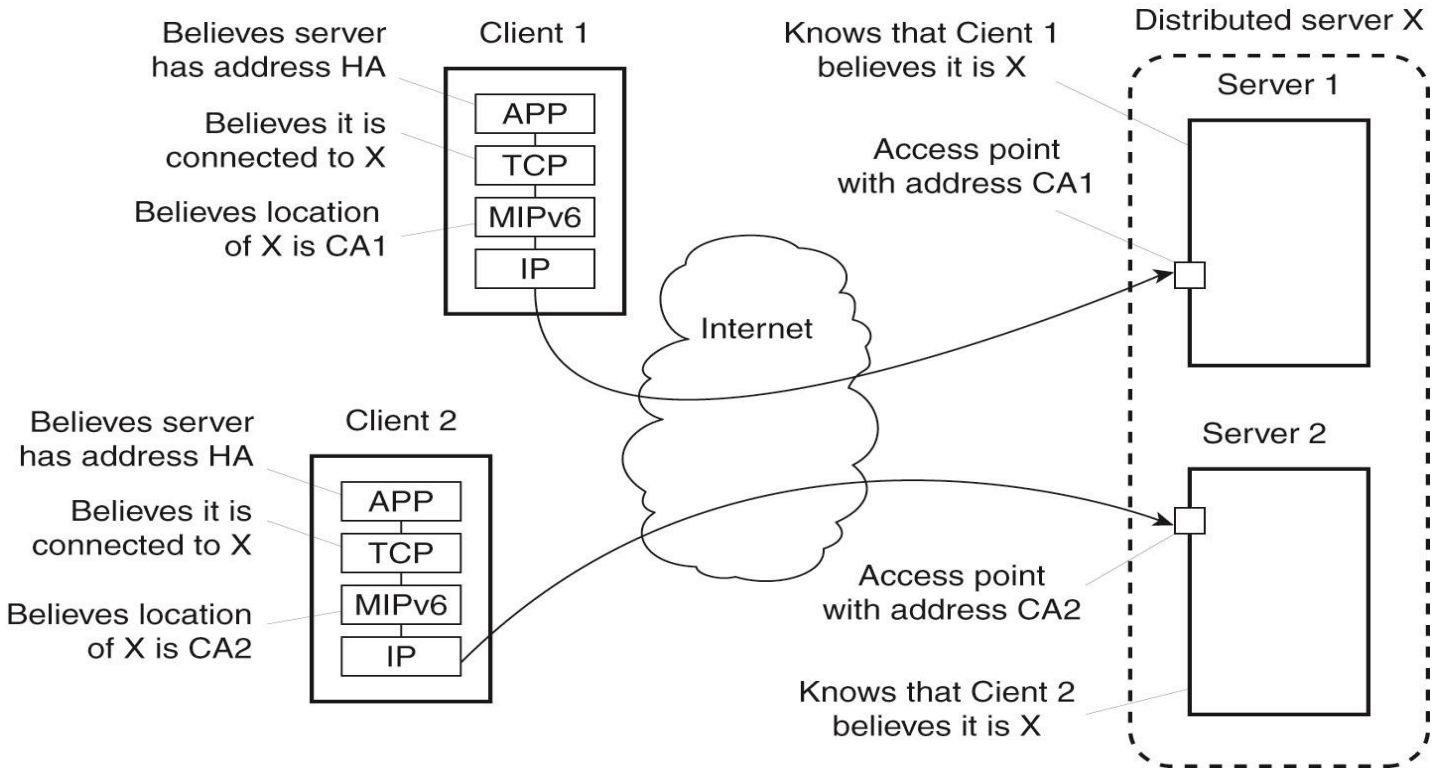


Figure 3-14. Route optimization in a distributed server.

Code Migration

So far, communication is limited to passing data in distributed systems. However, there are situations in which passing programs, simplifies the design of a distributed system.

- What code migration actually is.
- Different approaches to code migration,
- How to deal with the local resources that a migrating program uses.

Reasons for Migrating Code

1- Overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines.

2- To minimize communication. If a client application needs to perform many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. This same reason can be used for migrating parts of the server to the client.

3- To improve performance by exploiting parallelism. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent, that moves from site to site.

4- Flexibility is another reason. An approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed.

5- Dynamically configure distributed systems.

Reasons for Migrating Code

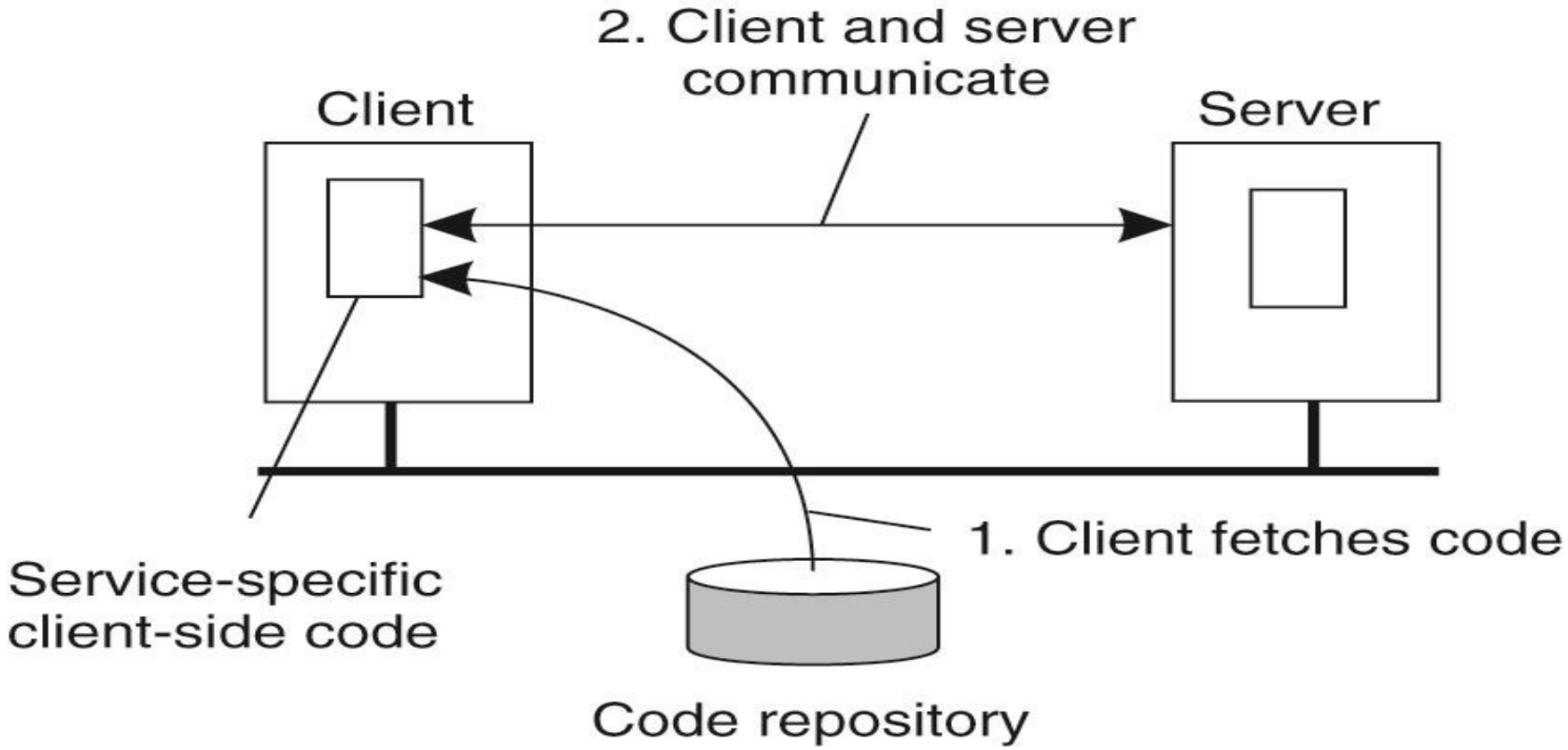


Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

Models for Migrating Code

In **weak mobility**, it is possible to transfer **only the code segment**, **along with perhaps some initialization data**. for example, with Java applets, which always start execution from the beginning. The benefit of this approach is its simplicity.

In systems that support **strong mobility** the **execution segment can be transferred as well**. The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution. Clearly, strong mobility is much more general than weak mobility, but also much harder to implement.

Models for Code Migration

In **sender-initiated** migration, migration is initiated at the machine where the code currently resides or is being executed. (i.e. uploading programs to a compute server or sending a search program across the Internet to a Web database server to perform the queries at that server.
In **receiver-initiated** migration, the initiative for code migration is taken by the target machine. Java applets are an example of this approach. (i.e. Java applets)
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 main drawback is that the target process needs to be protected against malicious or inadvertent code executions.
A simple solution is creating a **separate process** to execute the migrated code.

Models for Code Migration

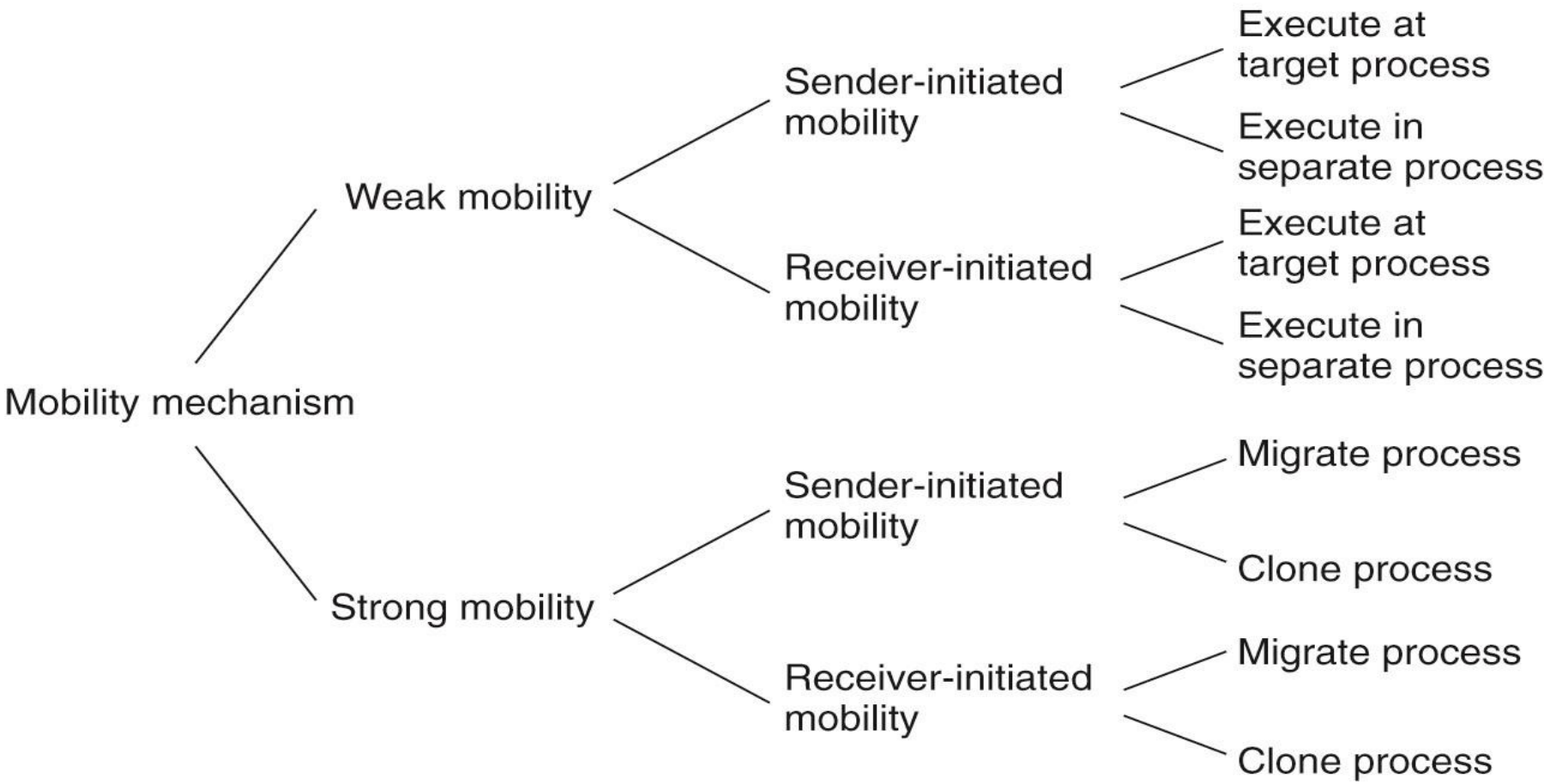


Figure 3-18. Alternatives for code migration.

Migration and Local Resources

So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration so difficult is that:

the resource segment cannot always be simply transferred along with the other segments without being changed. 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.

process-to-resource binding

Fuggetta et al. (1998) distinguish **three types of process-to-resource bindings**:

1. The **strongest binding** is when a process refers to a resource by its **identifier**. The process requires precisely the referenced resource, and nothing else. An example 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**.
2. A **weaker form** of process-to-resource binding is when 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 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

process-to-resource binding

3. the **weakest form of binding** is when a process indicates it needs only a **resource of a specific type**. This binding by type is exemplified by references to local devices, such as monitors, printers, and so on.

Migration and Local Resources
resource-to-machine binding
Unattached, Fastened, and Fixed Resources

Unattached resources can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated.

In contrast, moving or copying a **fastened resource** may be possible, but only at relatively high costs. Typical examples of fastened resources are 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.

Finally, **fixed resources** are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices. Another example of a fixed resource is a local communication end point.

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. These nine combinations are shown in Fig. 3-19.

Migration and Local Resources

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

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

Figure 3-19: Actions to be taken with respect to the references to local resources when migrating code to another machine.

Migrating Code in Heterogeneous Systems

So far, we have 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.