DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition
ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 3
Processes

Thread Usage in Nondistributed Systems

- To execute a program, an operating system creates a number of virtual processors, each one for running a different program.
- Like a process, a thread executes its own piece of code, independently from other threads.

Thread Usage in Nondistributed Systems

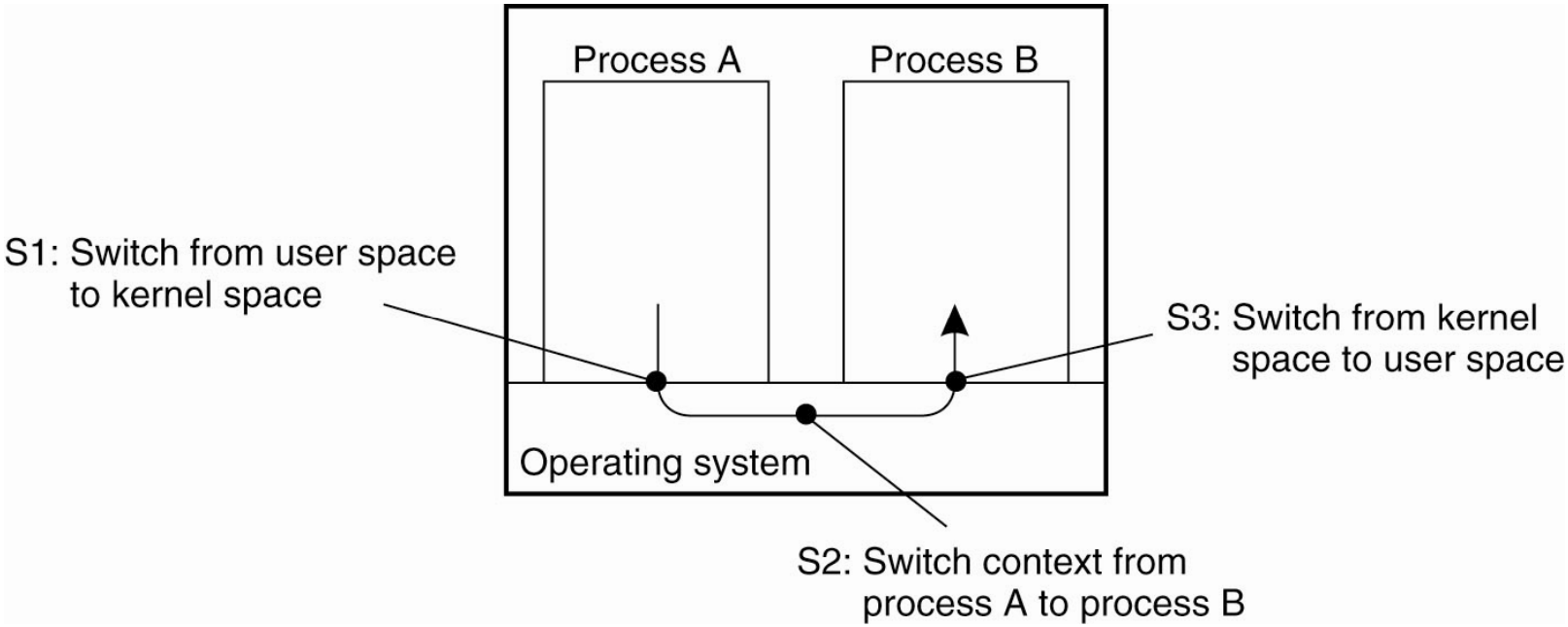

Figure 3-1. Context switching as the result of IPC.

Advantages of Multi-threading

- One advantage of multithreading is that it becomes possible to exploit parallelism when executing the program on a multiprocessor system.
  - In that case, each thread is assigned to a different CPU while shared data are stored in shared main memory.
- 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.
- 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.

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.

Thread Implementation (cont.)

- here 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.

Thread Implementation (cont.)

- A user-level thread library has a number of advantages and disadvantages:
  - 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.
  - 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.

Thread Implementation (cont.)

- The problems of user-level threads 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 kernel, requiring a system call.
- Switching thread contexts may now become as expensive as switching process contexts.

Thread Implementation (cont.)

- 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.

Thread Implementation (cont.)

- 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.

Thread Implementation (cont.)

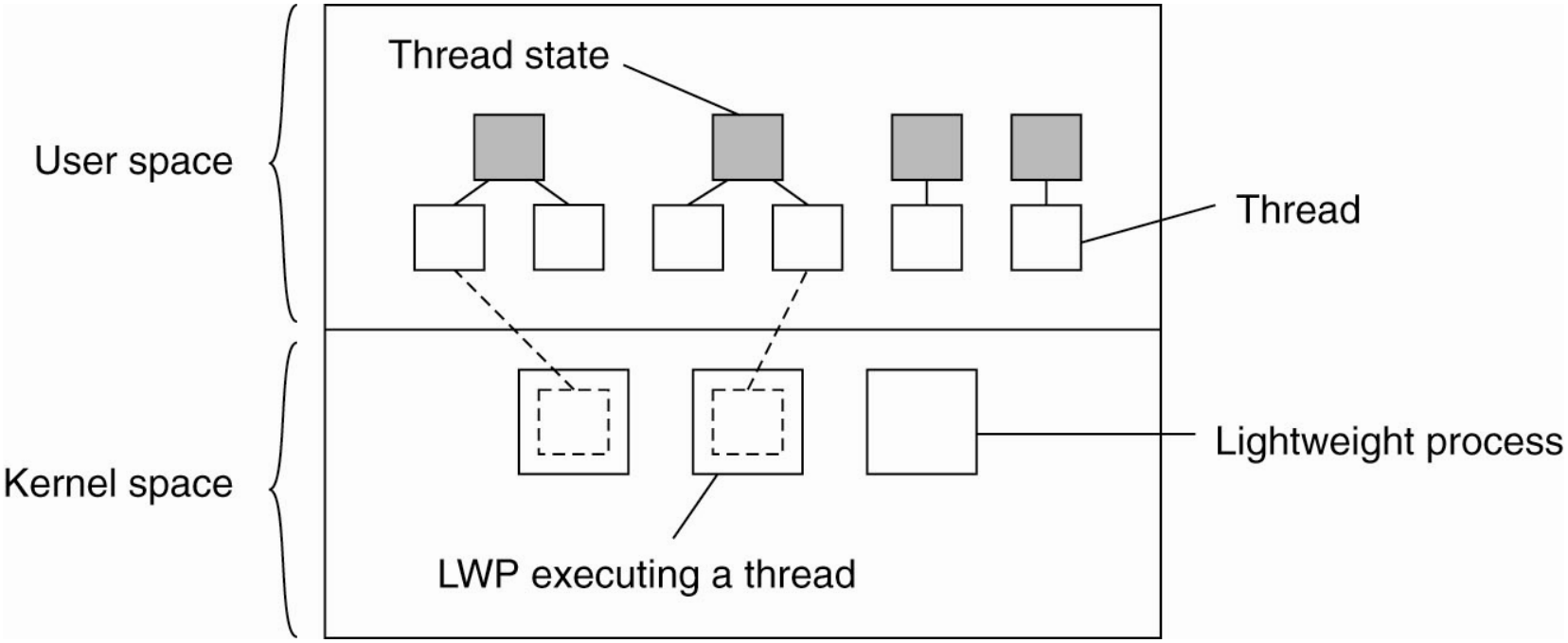

Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

Multithreaded Clients

- A Web browser
  - 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.

Multithreaded Servers

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

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

Multithreaded Servers (cont.)

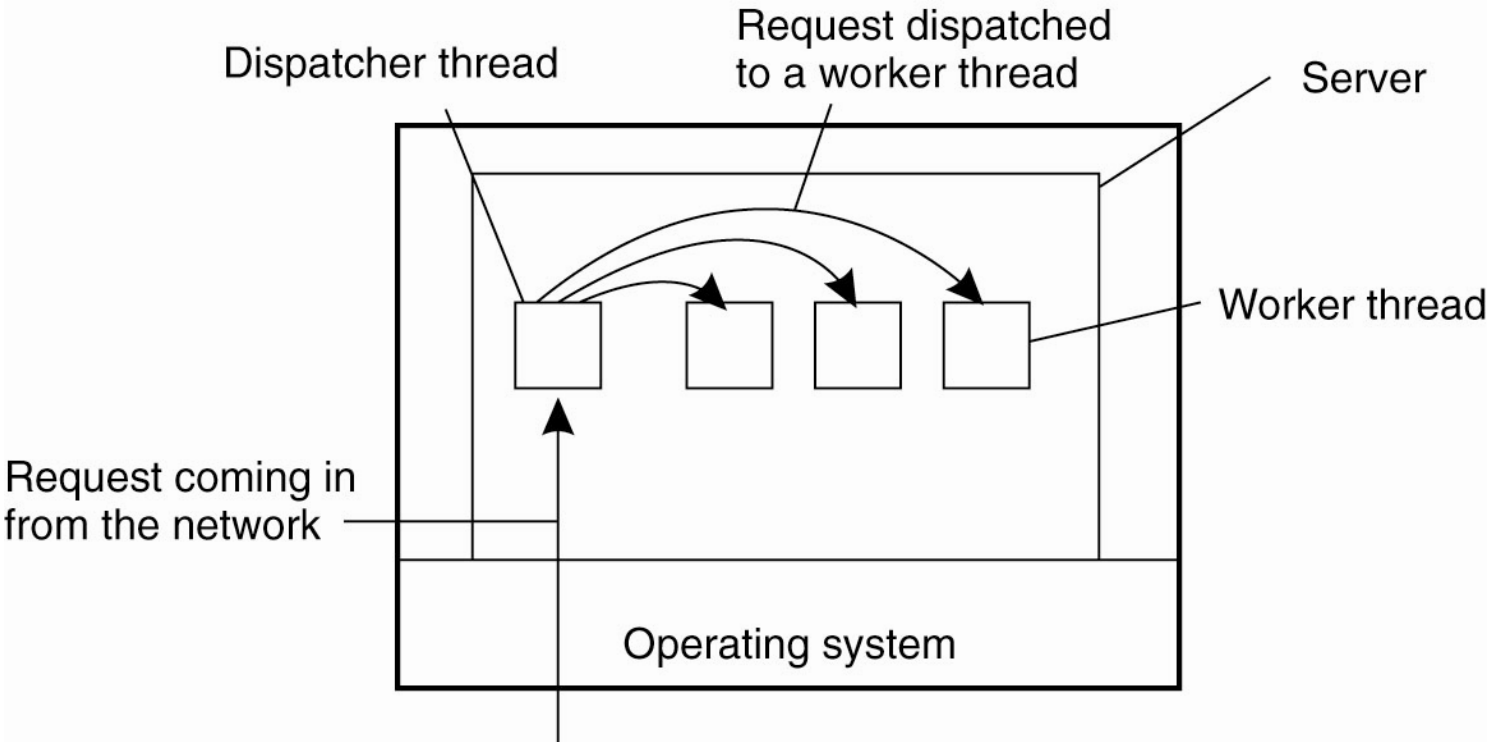

Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

Multithreaded Servers (cont.)

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.

Virtualization

- In practice, every (distributed) computer system offers a programming interface to higher level software.
- 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.
- Virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system.

Virtualization (cont.)

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

- A **legacy system** is an old computer system or application program which continues to be used because the user (typically an organization) does not want to replace or redesign it.

The Role of Virtualization in Distributed Systems

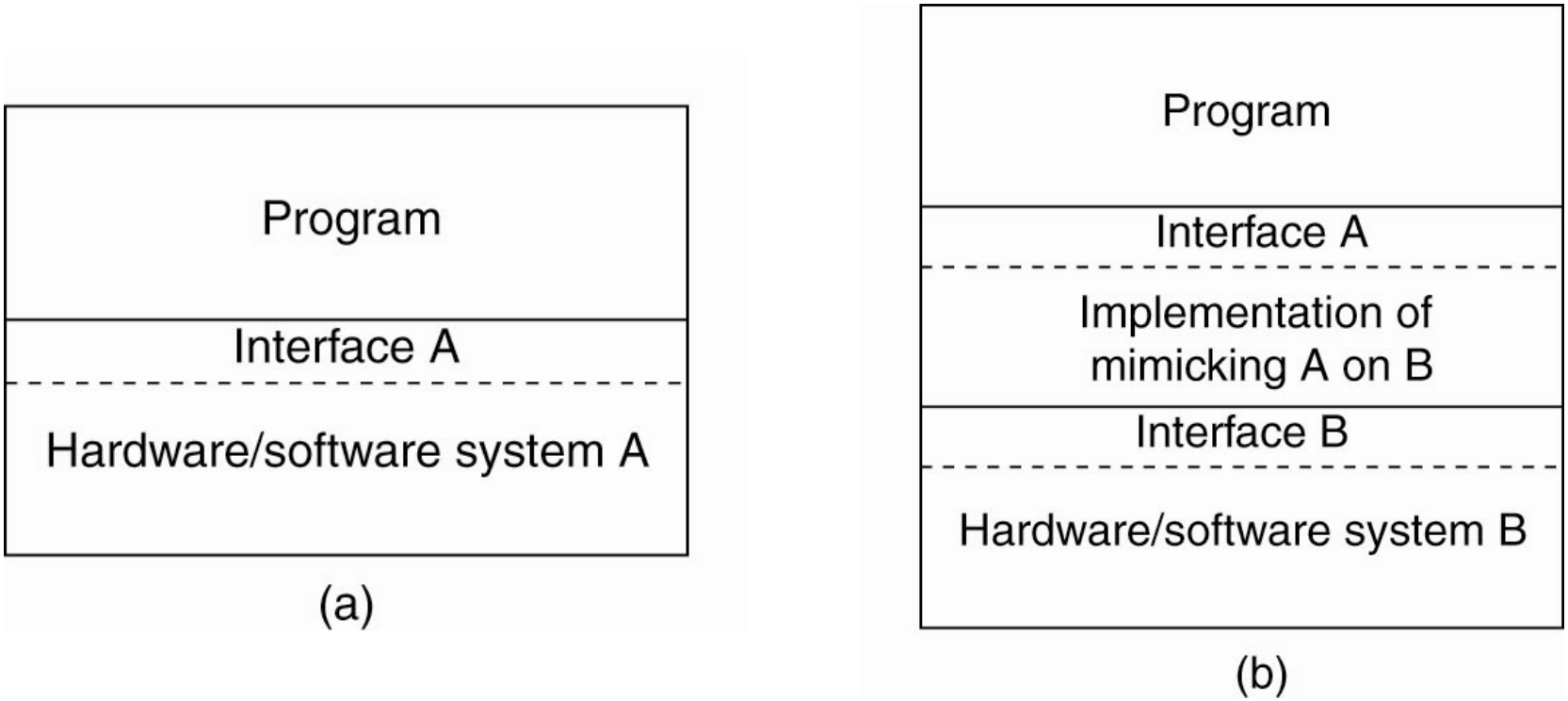

Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

Architectures of Virtual Machines

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 (cont.)

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 (cont.)

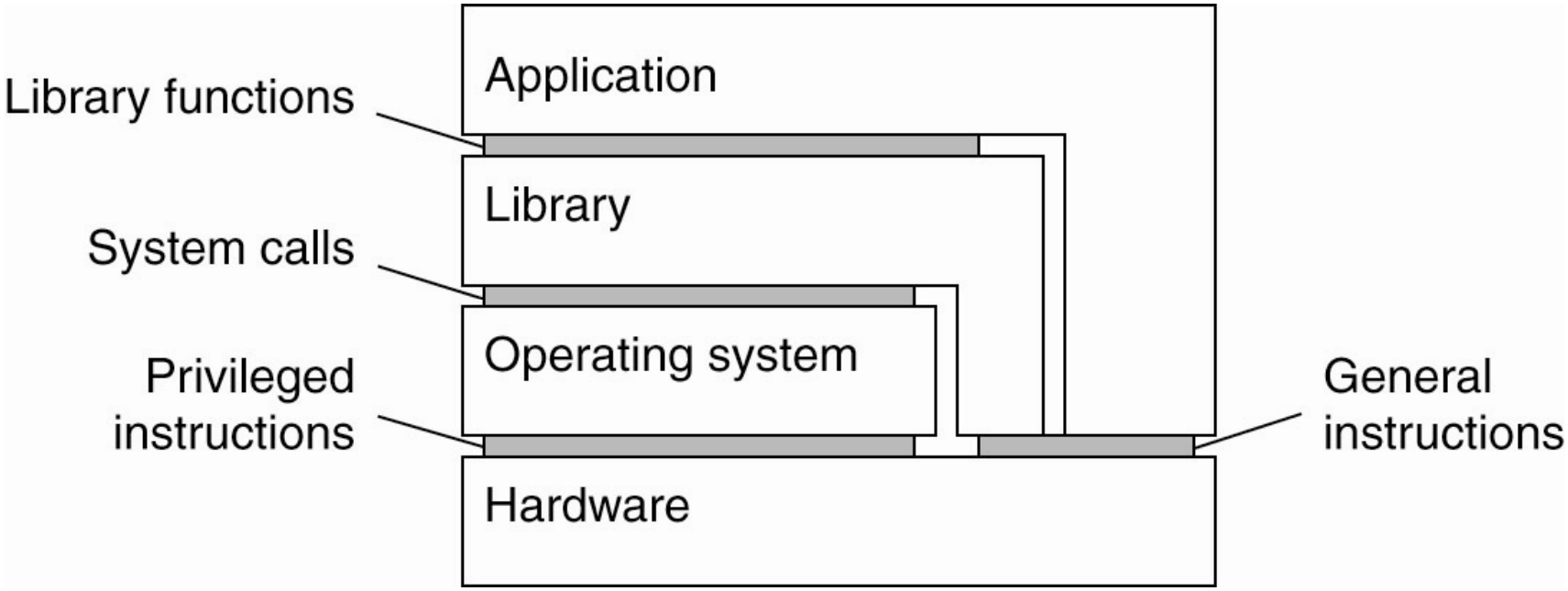

Figure 3-6. Various interfaces offered by computer systems.

Types of Virtualization

Virtualization can take place in two different ways:

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

- This type of virtualization leads to what Smith and Nair (2005) call a **process virtual machine**, stressing that virtualization is done essentially only for a single process.

Types of Virtualization (cont.)

- An alternative approach toward virtualization is to provide a system that is essentially implemented as a layer completely shielding the original hardware, but offering the complete instruction set of that same (or other hardware) as an interface.

- 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)**.

Architectures of Virtual Machines

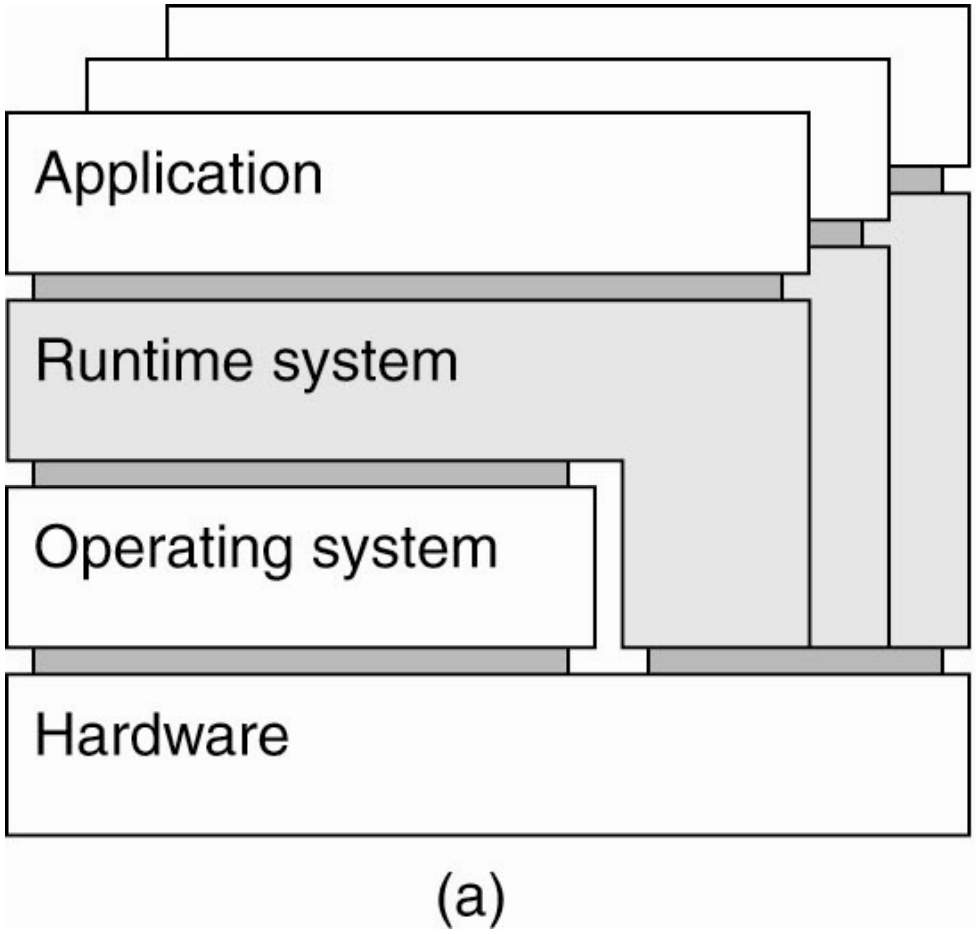

Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations.

Architectures of Virtual Machines (cont.)

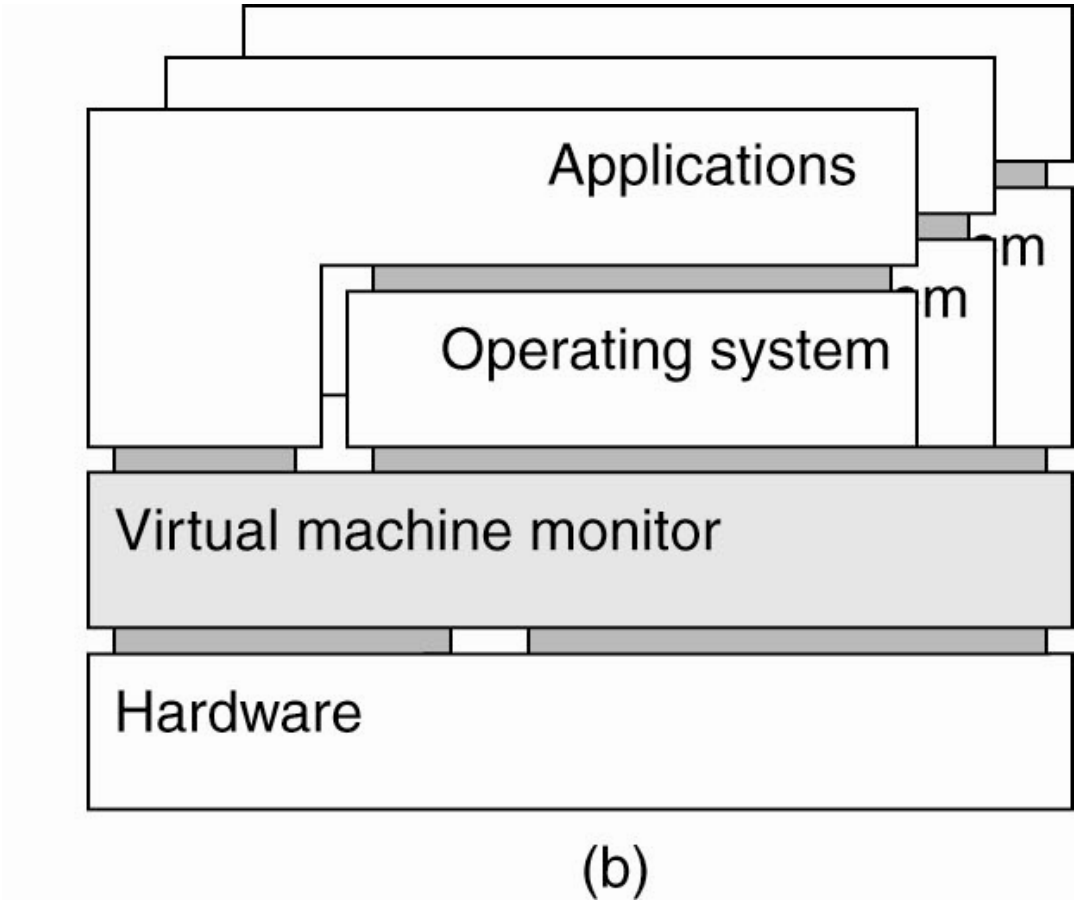

Figure 3-7. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

Networked User Interfaces

Mainly two ways for a client to connect to a server:

- For each remote service the client m/c will have a separate counterpart that can contact the service over the network application specific protocol required

- Direct access to remote services by only offering a convenient user interface (client m/c = terminal with no local storage)

Networked User Interfaces (cont.)

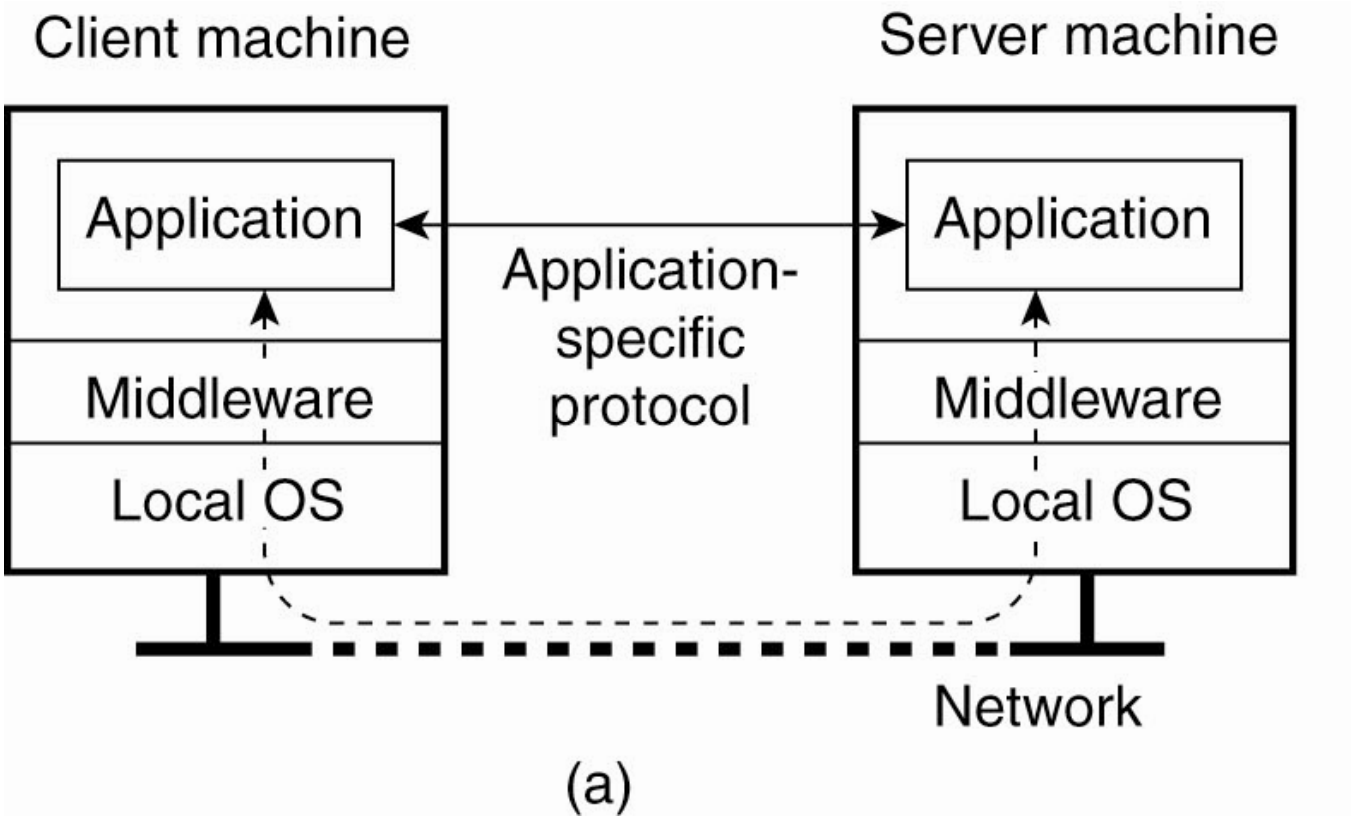

Figure 3-8. (a) A networked application with its own protocol. e.g., PDA synchronizes shared agenda with a remote (handled by application level protocol

Networked User Interfaces (cont.)

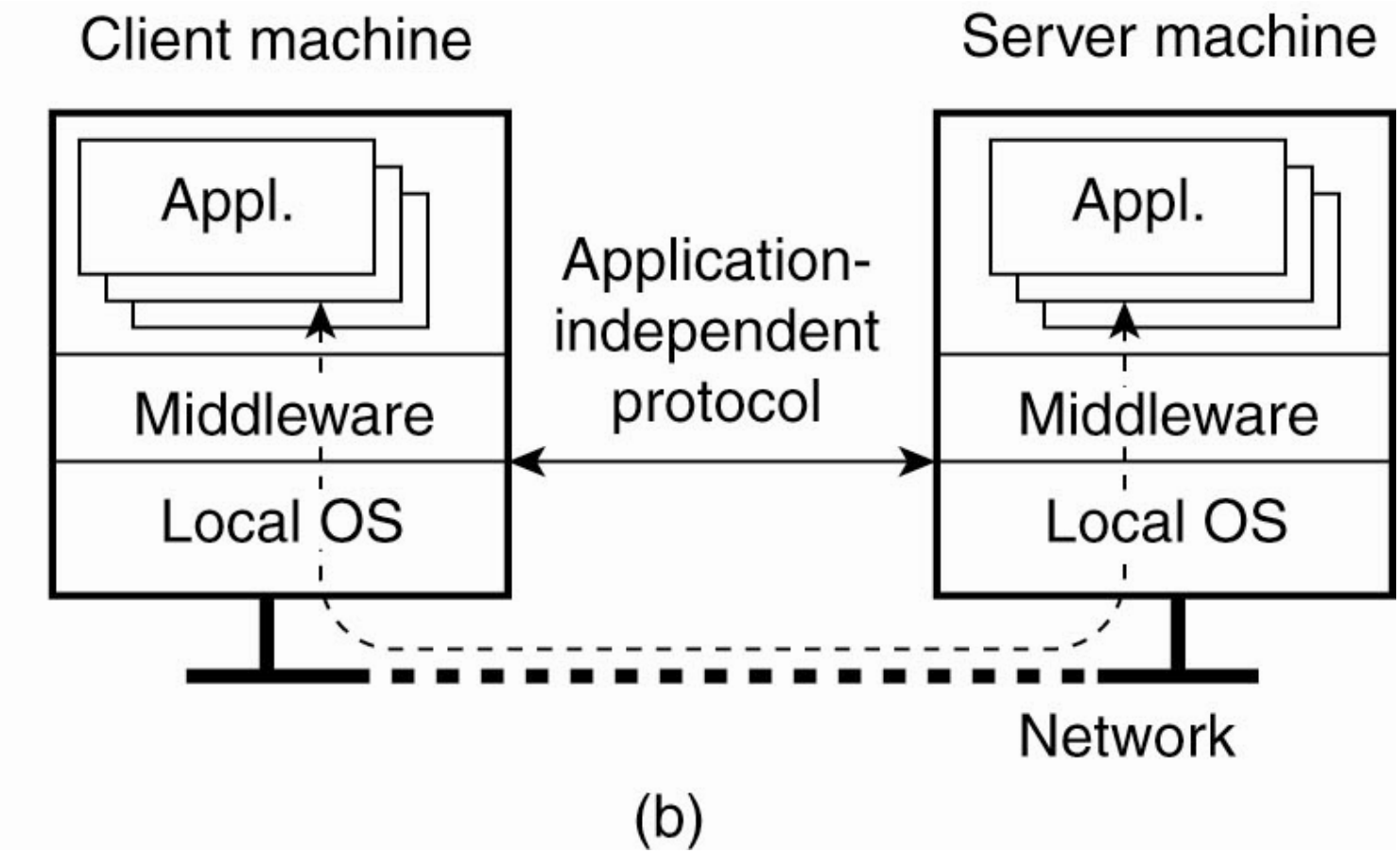

Figure 3-8. (b) A general solution to allow access to remote applications.

Client-Side Software for Distribution Transparency

- A client is more than a user interface
- Access transparency
  - Client generates a stub based on an interface provided by the server (hides the m/c architecture and communication)
- Location, migration, relocation transparency
- Concurrency transparency
  - Usually handled by (intermediate) server
- Failure transparency
  - Client middleware
- Replication transparency
  - Client proxy

Client-Side Software for Distribution Transparency

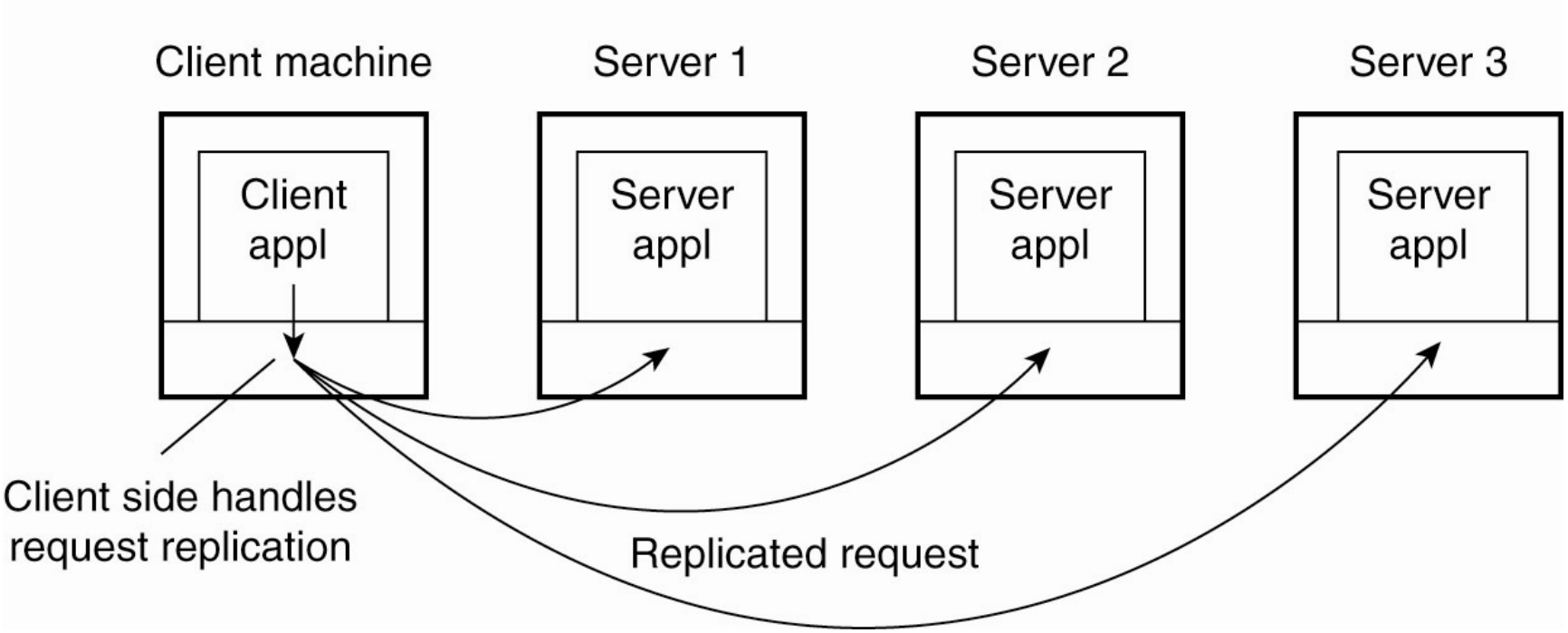

Figure 3-10. Transparent replication of a server using a client-side solution.

Servers Design Issues – Types of Servers

- Iterative serverAccess transparency
  - The server itself handles the request, and if necessary, returns a response to the requesting client

- Concurrent server
  - Does not handle the request itself. Instead it generates a thread or process to handle the request and to return the required response

Servers Design Issues – Types of Servers

- Server interruption
  - A client (user) exits the application at any time and restarts it again as if nothing has happened
  - Sending out of band data (mandatory urgent data) using a different connection between a client and a server
  - Using the same connection to send out of band data and normal data

Types of Servers

- Stateless server
  - Does not keep information of the client states, changes its own state without informing the clients , web server
  - Some web server keeps some information about the client using cookies
- Stateful server
  - Maintains information about the states of the clients, file server

General Design Issues

- Server endpoints: clients contact point at the server
  - a) Client-to-server binding using a daemon as in DCE
  - Daemon: a client first contacts a daemon for end point information
  - b) Client-to-server binding using a superserver as in UNIX
  - Superserver: client requested server creates another server for the service

General Design Issues (cont.)

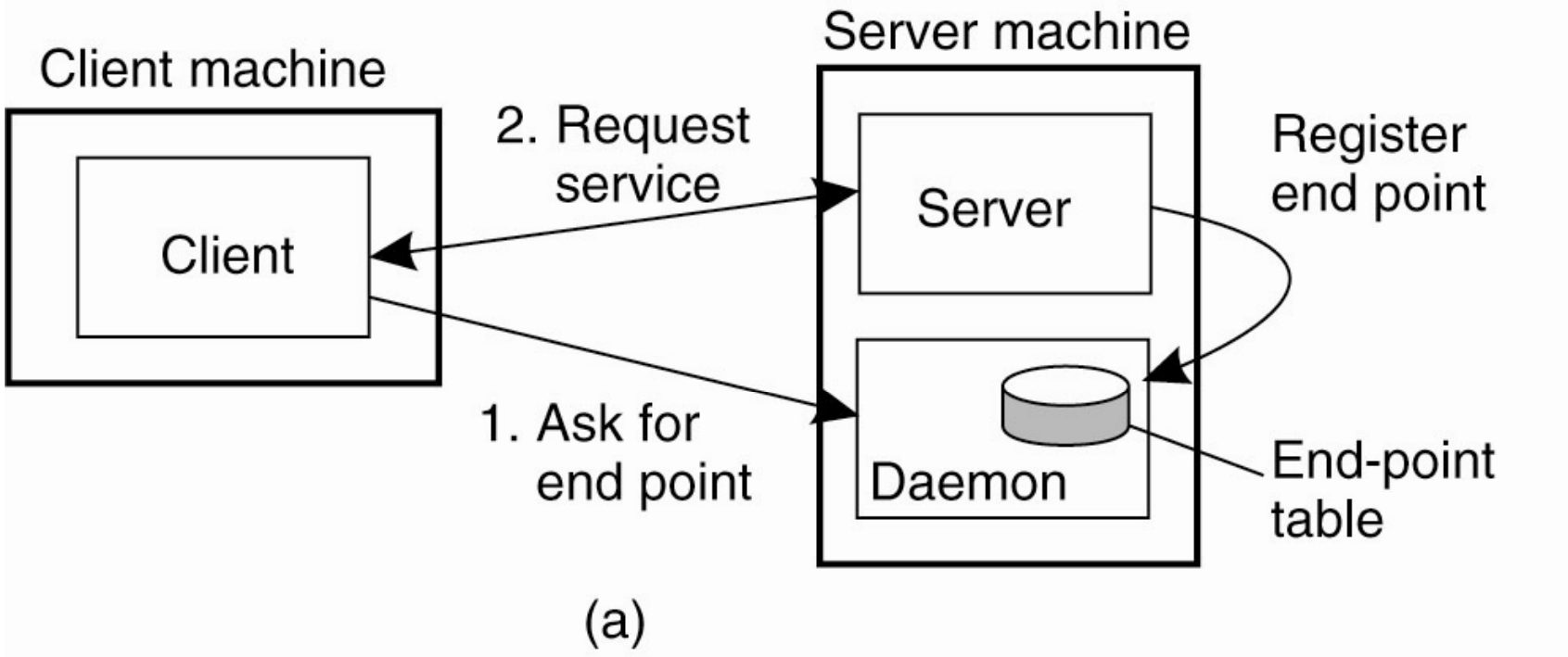

Figure 3-11. (a) Client-to-server binding using a daemon.

General Design Issues (cont.)

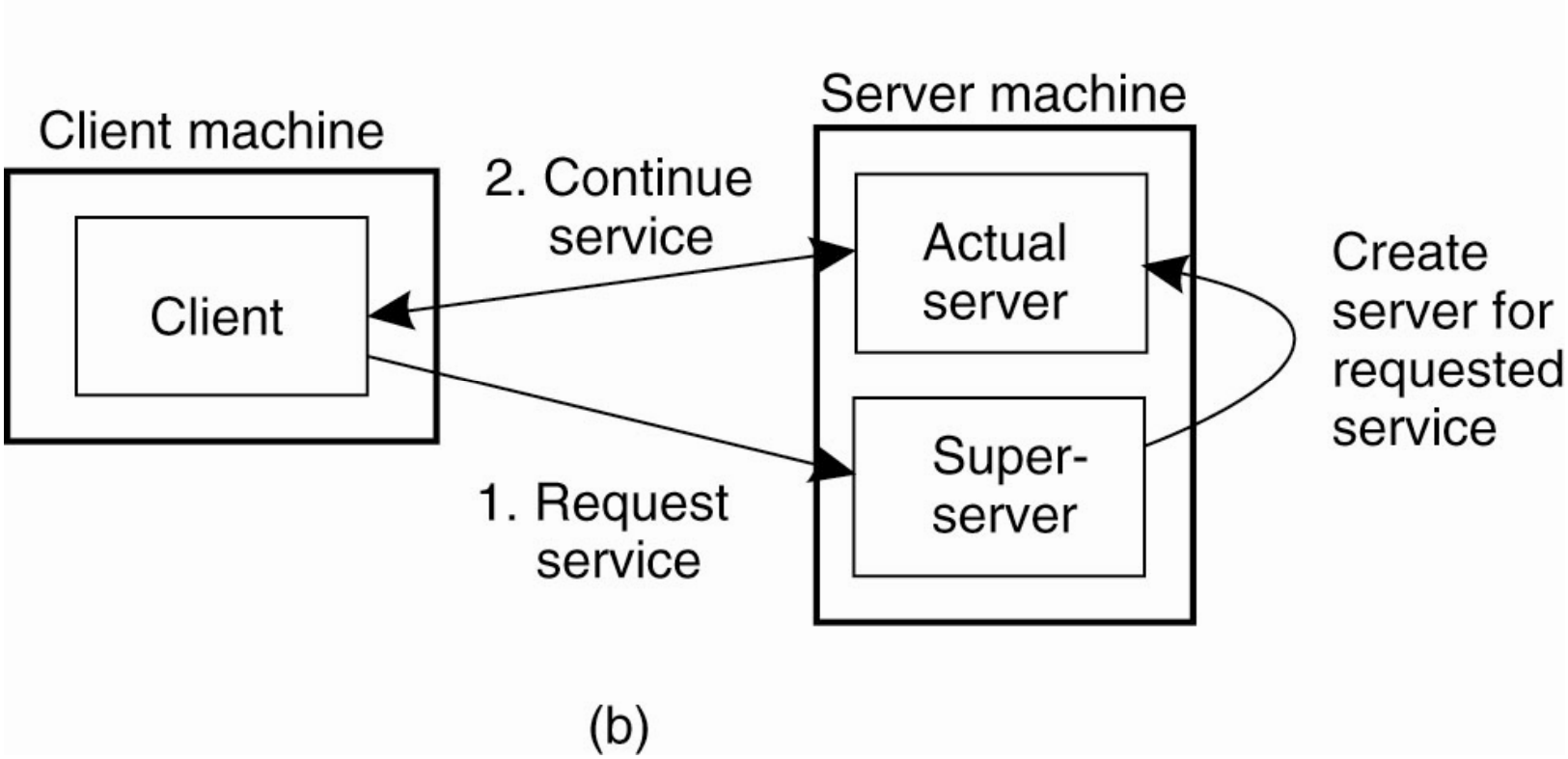

Figure 3-11. (b) Client-to-server binding using a superserver.

Server Clusters

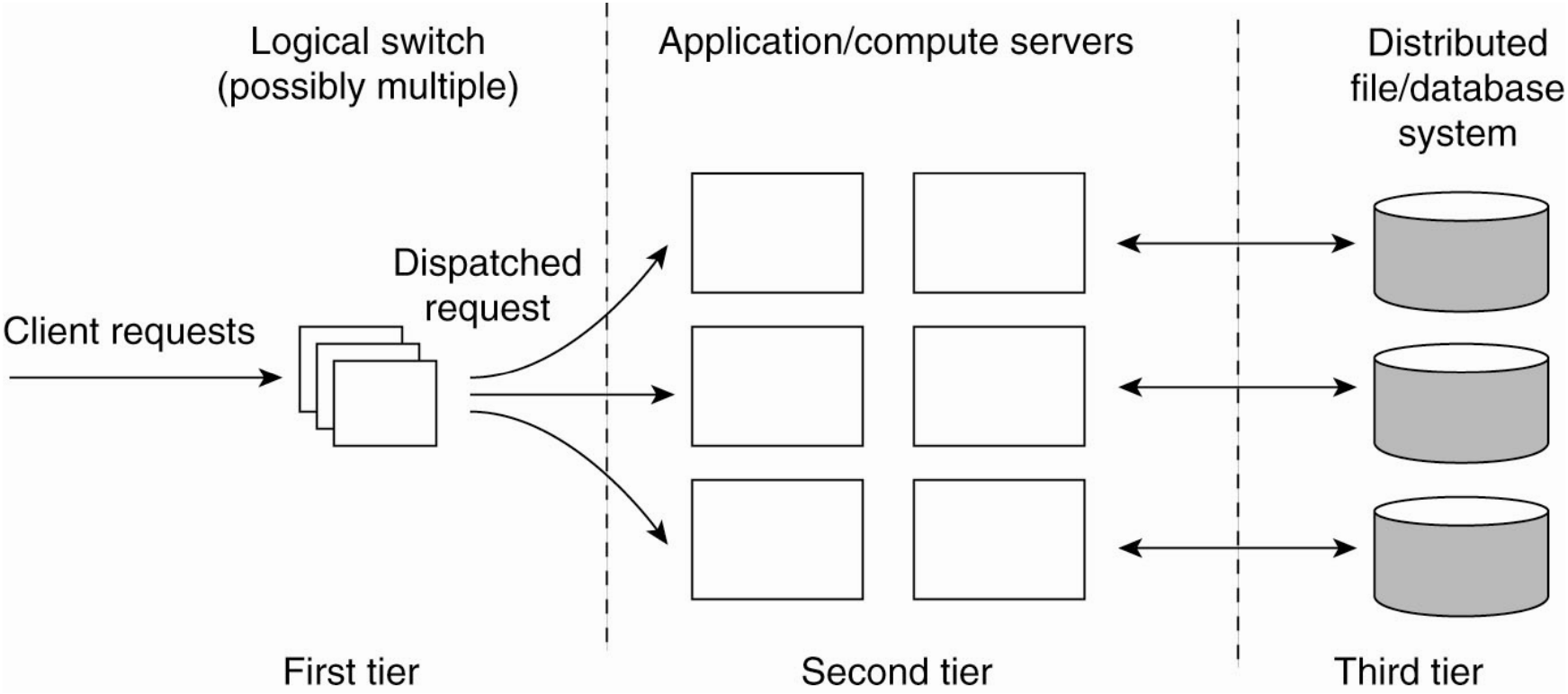

Figure 3-12. The general organization of a three-tiered server cluster.

Server Clusters (cont.)

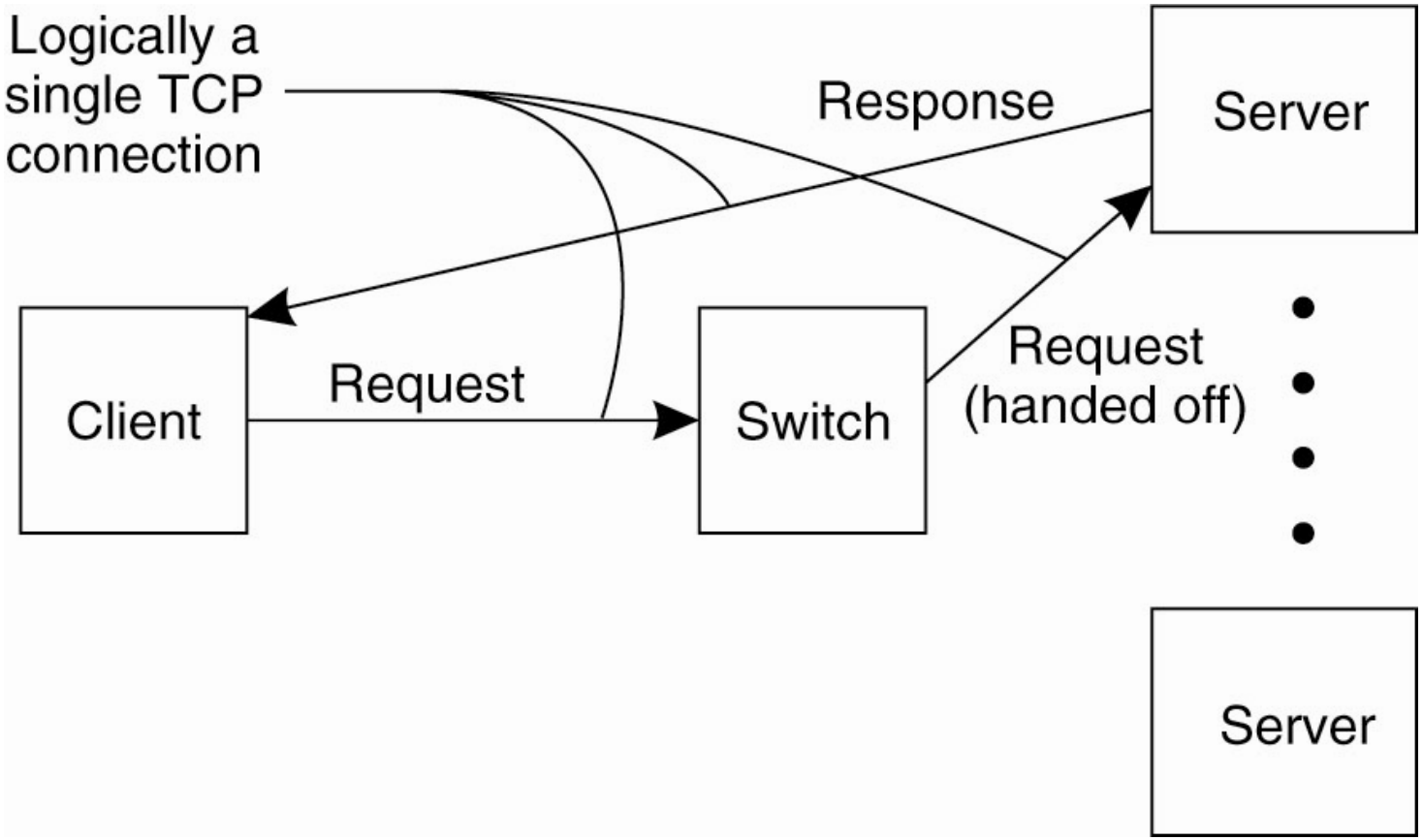

Figure 3-13. The principle of TCP handoff.

Distributed Servers

- Dynamically changing set of machines, varying access points, appearing to the outside world as a single, powerful Machine

- Robust, high-performance, stable server
  - Several access points are provided so that the cluster can still be used when an access point fails
  - High performance mainframes with very high mean time to failure
  - By grouping simpler machines transparently into a cluster instead of relying on the availability of a singe machine
  - By offering a stable address to a distributed server

Distributed Servers

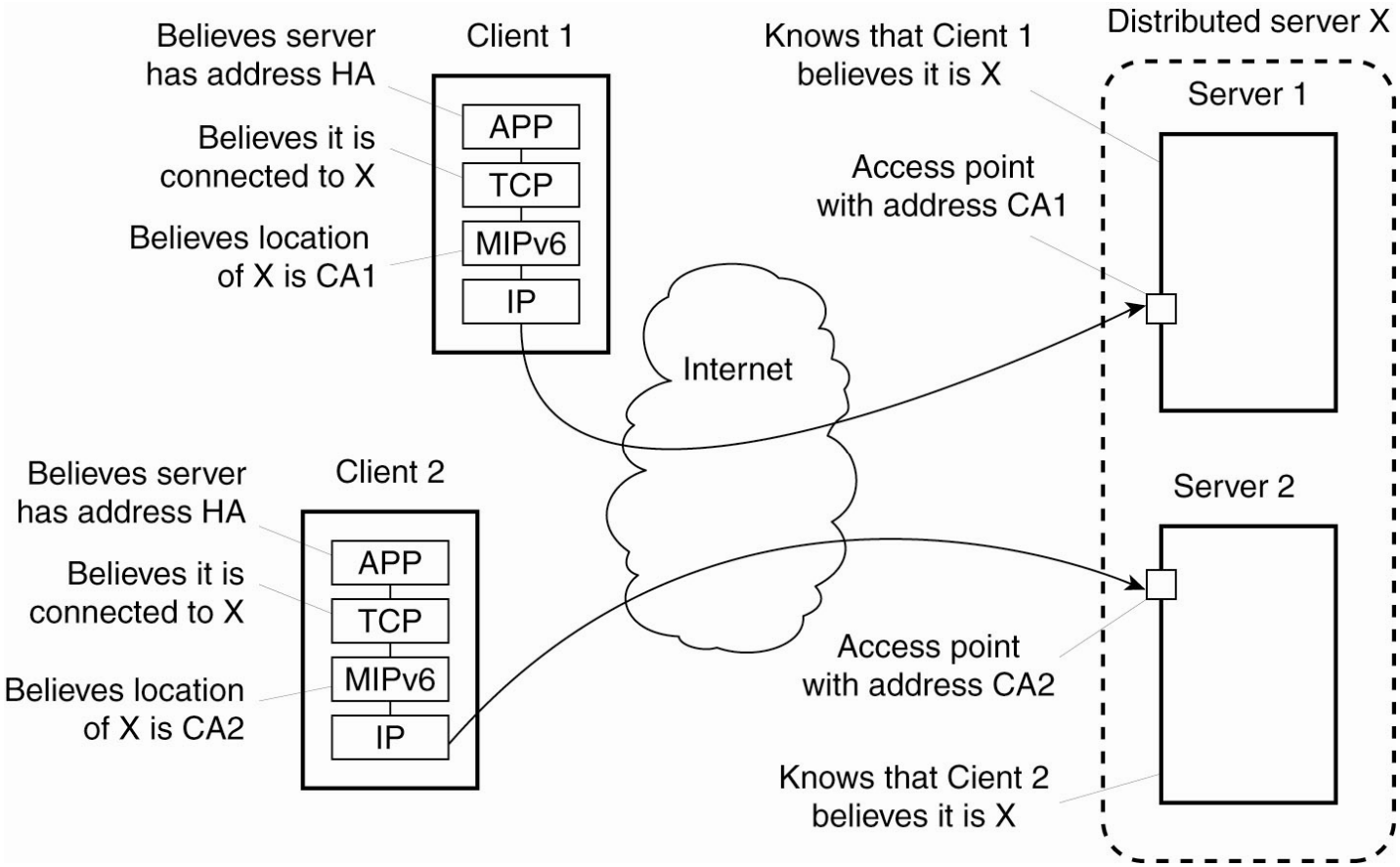

Figure 3-14. Route optimization in a distributed server.

Code Migration

- Passing programs, sometimes even while they are being executed
- Traditionally code migrations in DS was process migration an entire process was moved from one m/c to another for load balancing
- However, in many modern DSs, optimization of computing power is a less pressing issue than minimizing communication costs

Why code migration?

- To process data close to where those data reside – migrate part of the server program to the client

- Improve performance by parallelism – make several copies of a program and send each off to different sites – linear speed up compared to using just a single program instance
  - Mobile agents
- Flexibility – dynamic configuration
  - Traditional approach – partition the application into different parts, and decide in advance where each part should be executed - multitiered client server applications
  - Code migration – dynamically configure distributed systems

Dynamic Configuration Through Code Migration

- The software is readily available to the remote client
- The server provides client implementation only when the client binds to the server
  - The client first fetches the necessary software, and then invokes the server
  - Needs standard for downloading and initializing protocol
  - The client should be able to execute the downloaded code

Dynamic Configuration Through Code Migration

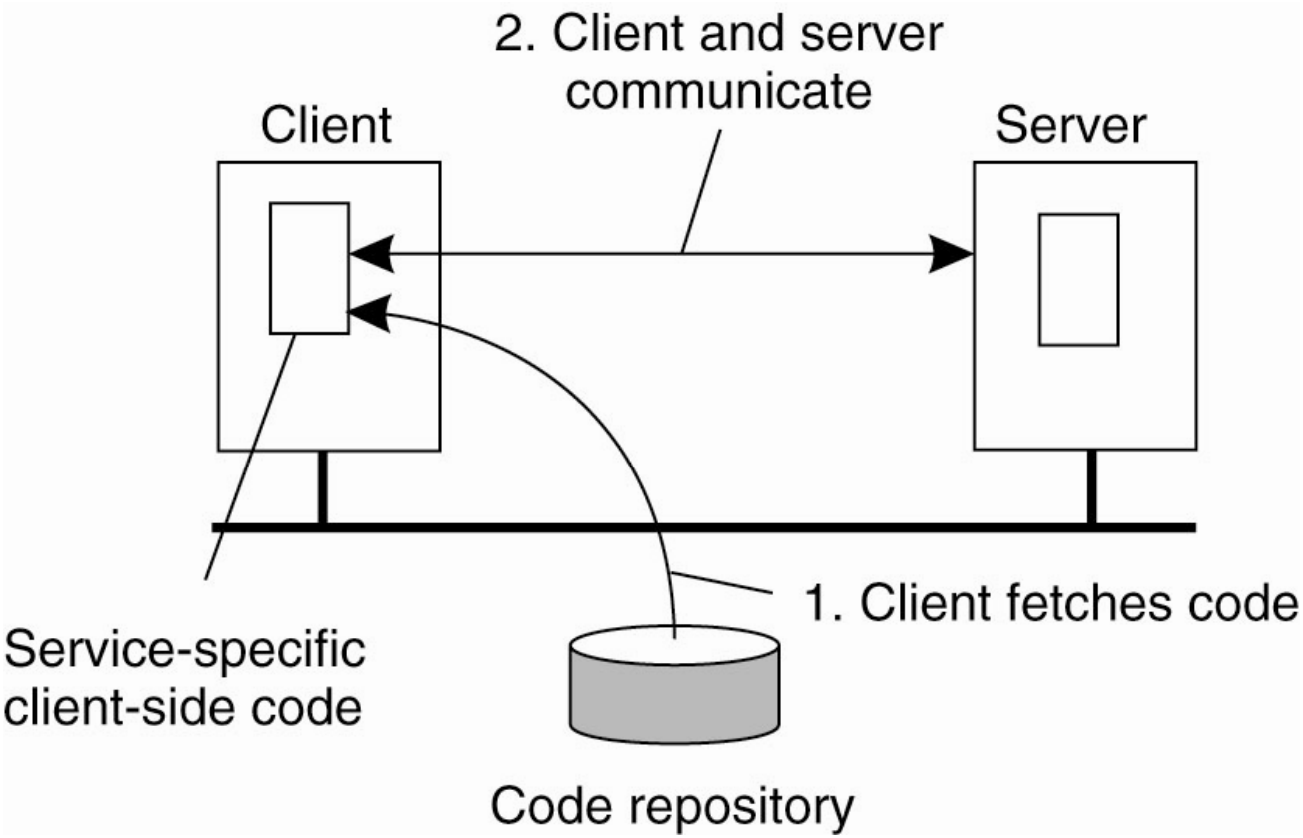

Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

Models for Code Migration

- Code – the set of instructions that make up the executing program
  - Resource – references to external resources (files, printers, devices, etc.) needed by the process
  - Execution – stores the current execution state (private data, the stack, and the program counter) of a process

Models for Code Migration (cont.)

- Alternatives based on transfer of segments of a process

  – Code segment only along with perhaps some initialization data –**weak mobility**
    - Transferred program always starts from its initial state – simplicity, **Java applets**
    - Only target m/c can execute the code – requires portability

  – Both code and execution segment – strong mobility
    - A running process can be stopped and subsequently transferred to a different m/c, and resume execution where it left off
    - Powerful than weak mobility, but harder to implement

Models for Code Migration

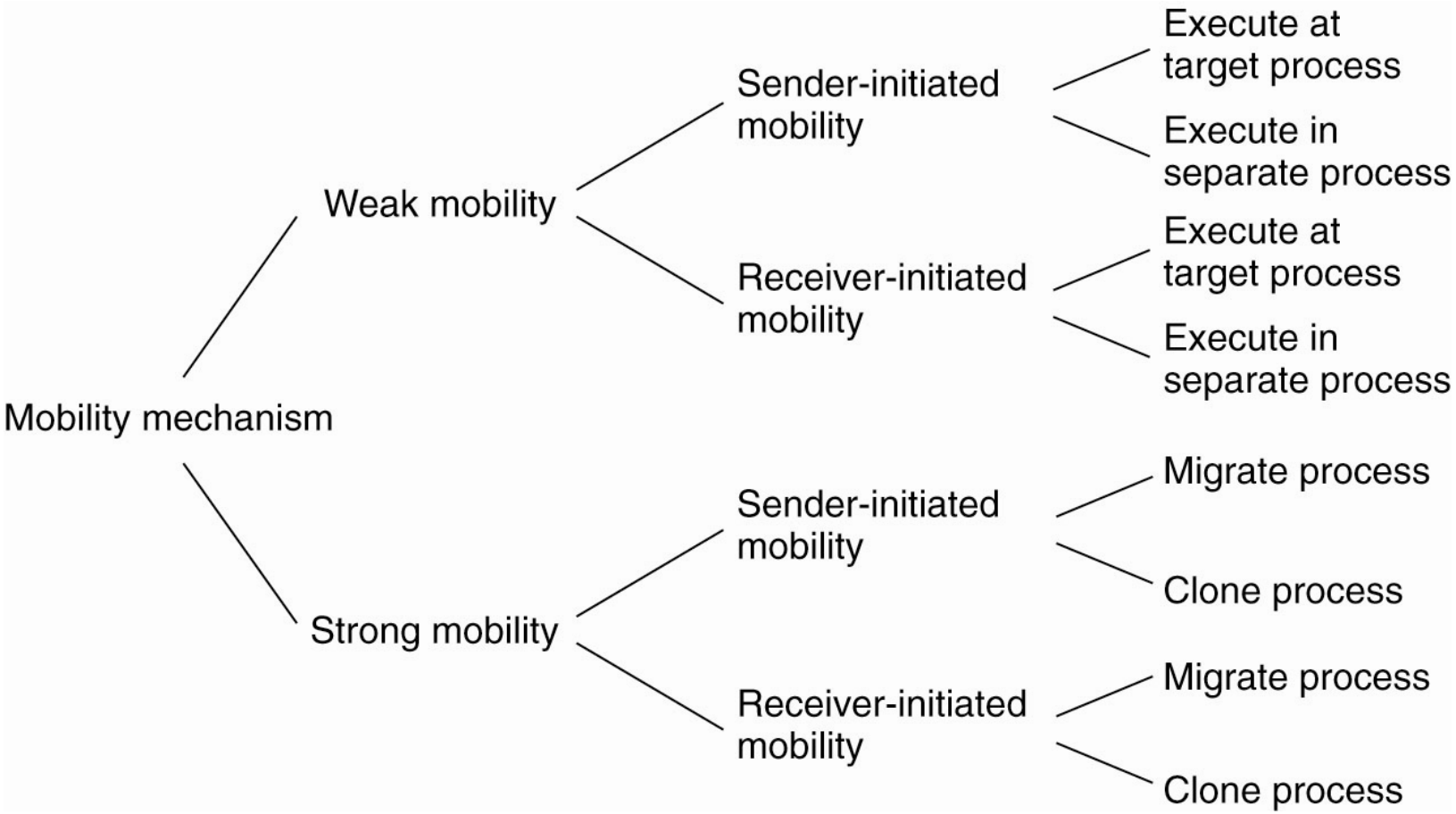

Figure 3-18. Alternatives for code migration.

Resource Migration

- Segments of a Process
  - Code – the set of instructions that make up the executing program
  - Resource – references to external resources (files, printers, devices, etc.) needed by the process
  - Execution – stores the current execution state (private data, the stack, and the program counter) of a process

- Thus far, we have discussed only the migration of code and data segment
  - Code segment only along with perhaps some initialization data –weak mobility
  - Both code and execution segment

Migration of Local Resources

- Types of process-to-resource binding – in order of strength
  - By identifier – requires a resource uniquely identified by the identifier – referring to URL or FTP server address.
  - By value – Not a specific resource, but the value provided by the resource is important – referring to standard libraries in C or Java.
  - By type – requires a resource of specific type – references to local devices such as monitors, printers.

- In code migration, we need to change the process-to-resource binding – we have to bind the resource-to-target m/c

Migration of Local Resources (cont.)

- Types of resource-to-m/c binding – in order of increased
- cost
  - Unattached – the data files related only with the migrated program
  - Fastened – local databases, complete website
  - Fixed – cannot be moved, local devices or communication
- endpoints

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.

Migration in Heterogeneous Systems

- The migrated system should be executed in different target platform - execution segment contains data that is private to the process, its current stack, and the program counter
- One possible solution – avoid having execution that depend on platform specific data such as register values
- Best solution to handle heterogeneity is to use virtual Machines
  - Process virtual machines – JVM
  - Virtual machines monitor – allow the migration of processes along with their underlying operating system

Migration in Heterogeneous Systems

Three ways to handle migration (which can be combined)

- Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.
- Stopping the current virtual machine; migrate memory, and start the new virtual machine.
- Letting the new virtual machine pull in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.