Processes, Threads and Virtualization

The role of processes in distributed systems

Introduction

- To be efficient, a client/server system can use asynchronous communication to overlap communication latencies with local processing.
  - Structure processes with multiple threads
- Virtual machines make it possible for multiple servers to execute securely on a single platform, and to migrate servers from one platform to another.
- Process migration

Concurrency Transparency

- Traditionally, operating systems used the process concept to provide concurrency transparency to executing processes.
  - Virtual processors; hardware support

- Today, multithreading provides concurrency with less overhead (so better performance)
  - Also less transparency – application must provide memory protection for threads.

Large Applications

- Early operating systems (e.g., UNIX)
  - Supported large apps by supporting the development of several cooperating programs via *fork( )* system call (Parent process *fork*s multiple child processes)
  - Rely on IPC mechanisms to exchange info
  - Pipes, message queues, shared memory
- Overhead: numerous context switches
- Possible benefits of multiple threads vs multiple programs (processes)
  - Less communication overhead
  - Easier to handle asynchronous events
  - Easier to handle priority scheduling

Thread

- Conceptually, one of concurrent execution paths contained in a process.

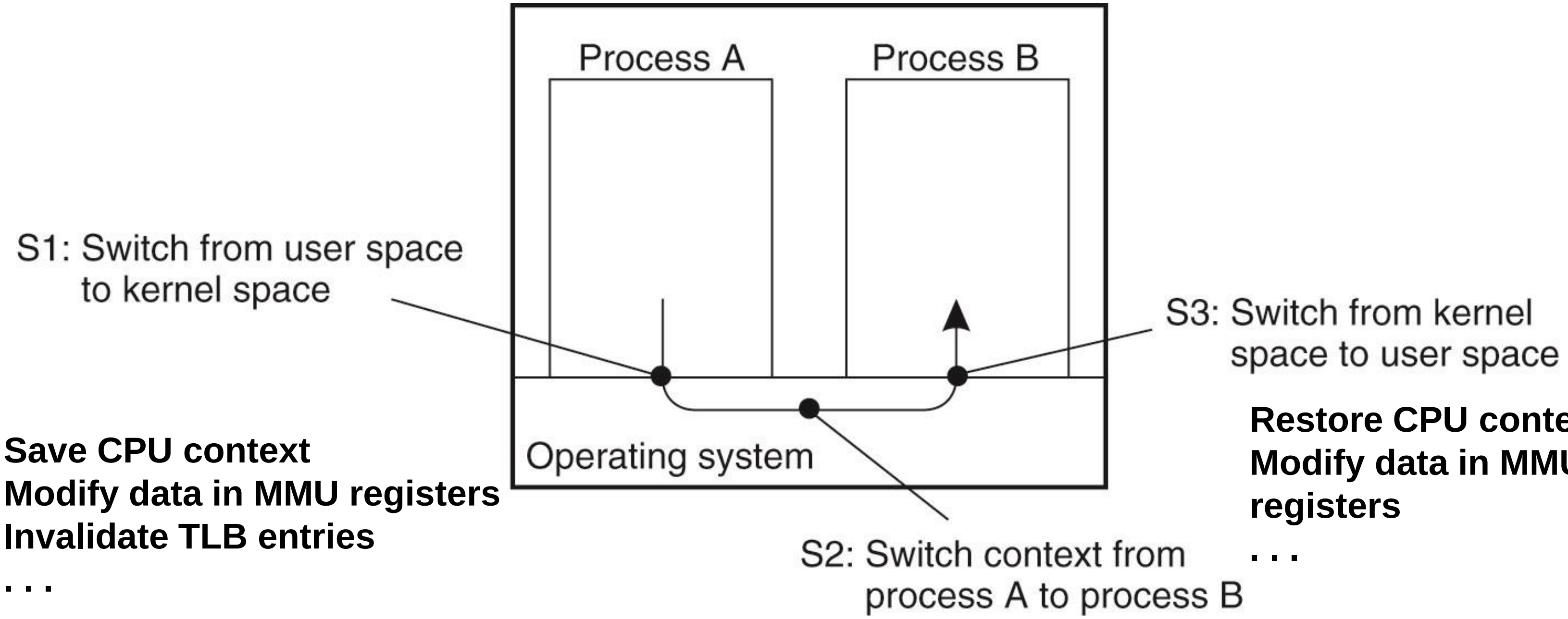
- If two *processes* want to share data or other resources, the OS must be involved.
  - Overhead: system calls, mode switches, context switches, extra execution time.

- Two threads in a single process can share global data automatically – as easily as two functions in a single process

Threads

- Multithreading is useful in the following kinds of situations:
  - To allow a program to do I/O and computations at the “same” time: one thread blocks to wait for input, others can continue to execute
  - To allow separate threads in a program to be distributed across several processors in a shared memory multiprocessor
  - To allow a large application to be structured as cooperating threads, rather than cooperating processes (avoiding excess context switches)
- Multithreading also can simplify program development (divide-and-conquer)

Overhead Due to Process Switching



Save CPU context
Modify data in MMU registers
Invalidate TLB entries
. . .

Restore CPU context
Modify data in MMU registers
. . .

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

Thread Implementation

- User-level
  - Less overhead; faster execution
- Kernel-level
  - Support multiprocessing
  - Independently schedulable by OS
  - Can continue to run if one thread blocks on a system call.
- Light weight processes (LWP)
  - Example: in Sun’s Solaris OS

User-level Threads

- User-level threads are created by calling functions in a user-level library.
    - Less overhead; faster execution
- The advantage here is that they are even more efficient
    - no mode switches are involved in thread creation or switching.
- The process that uses user-level threads appears (to the OS) to be a single threaded process
    - there is no way to distribute the threads in a multiprocessor or block only part of the process.
    - Blocking system call will immediately block the entire process

Kernel-level Threads

- The kernel is aware of the threads and schedules them independently as if they were processes.
- One thread may block for I/O, or some other event, while other threads in the process continue to run.
- Unfortunately, there is a high price to pay: every thread operation will have to be carried out by the kernel.
  - requiring a system call. Switching thread contexts
  - most of the performance benefits of using threads instead of processes then disappears

Hybrid Threads –Lightweight Processes (LWP)

- LWP is similar to a kernel-level thread:
  - It runs in the context of a regular process
  - The process can have several LWPs created by the kernel in response to a system call.
- User level threads are created by calls to the user-level thread package.
- The thread package also has a scheduling algorithm for threads, runnable by LWPs.

Thread Implementation

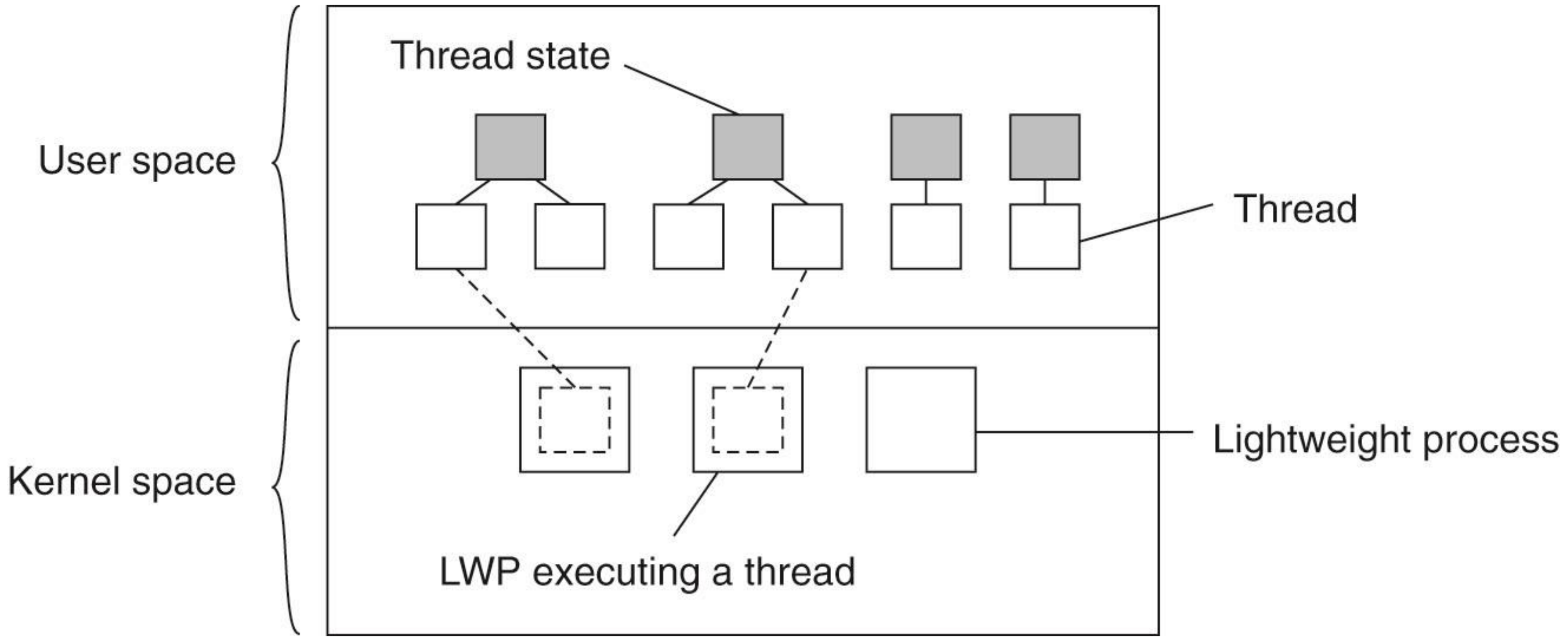


Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

Hybrid threads – LWP

- The OS schedules an LWP which uses the thread scheduler to decide which thread to run.

- Thread synchronization and context switching are done at the user level; LWP is not involved and continues to run.

- If a thread makes a blocking system call control passes to the OS (mode switch)
  - The OS can schedule another LWP or let the existing LWP continue to execute, in which case it will look for another thread to run.

Advantages of the hybrid approach

- Most thread operations (create, destroy, synchronize) are done at the user level
- Blocking system calls need not block the whole process
- Applications only deal with user-level threads
- LWPs can be scheduled in parallel on the separate processing elements of a multiprocessor.

Threads in Distributed Systems

- Threads gain much of their power by sharing an address space
  - But … no sharing in distributed systems

- However, multithreading can be used to improve the performance of individual nodes in a distributed system.
  - A process, running on a single machine; *e.g.*, a client or a server, can be multithreaded to improve performance

Multithreaded Clients

- Main advantage: hide network latency
  - Addresses problems such as delays in downloading documents from web
- Hide latency by starting several threads
  - One to download text (display as it arrives)
  - Others to download photographs, figures, etc.
- All threads execute simple blocking system calls; easy to program this model
- Browser displays results as they arrive.

Multithreaded Clients

- Even better: if servers are **replicated**, the multiple threads may be sent to separate sites.
  - Data can be downloaded in several parallel streams, improving performance even more.
  - Designate a thread in the client to handle and display each incoming data stream.

Multithreaded Servers

- Improve performance, provide better structuring
- Consider what a server does:
  - Wait for a request
  - Execute request (may require blocking I/O)
  - Send reply to client
- Several models for programming the server
  - Single threaded
  - Multi-threaded
  - Finite-state machine

Threads in Distributed Systems - Servers

- A single-threaded (iterative) server processes one request at a time – other requests must wait.
  - Possible solution: create (fork) a new server process for a new request.
  - This approach creates performance problems

- Creating a new server thread is much more efficient.
  - Processing is overlapped and shared data structures can be accessed without extra context switches.

Multithreaded Servers

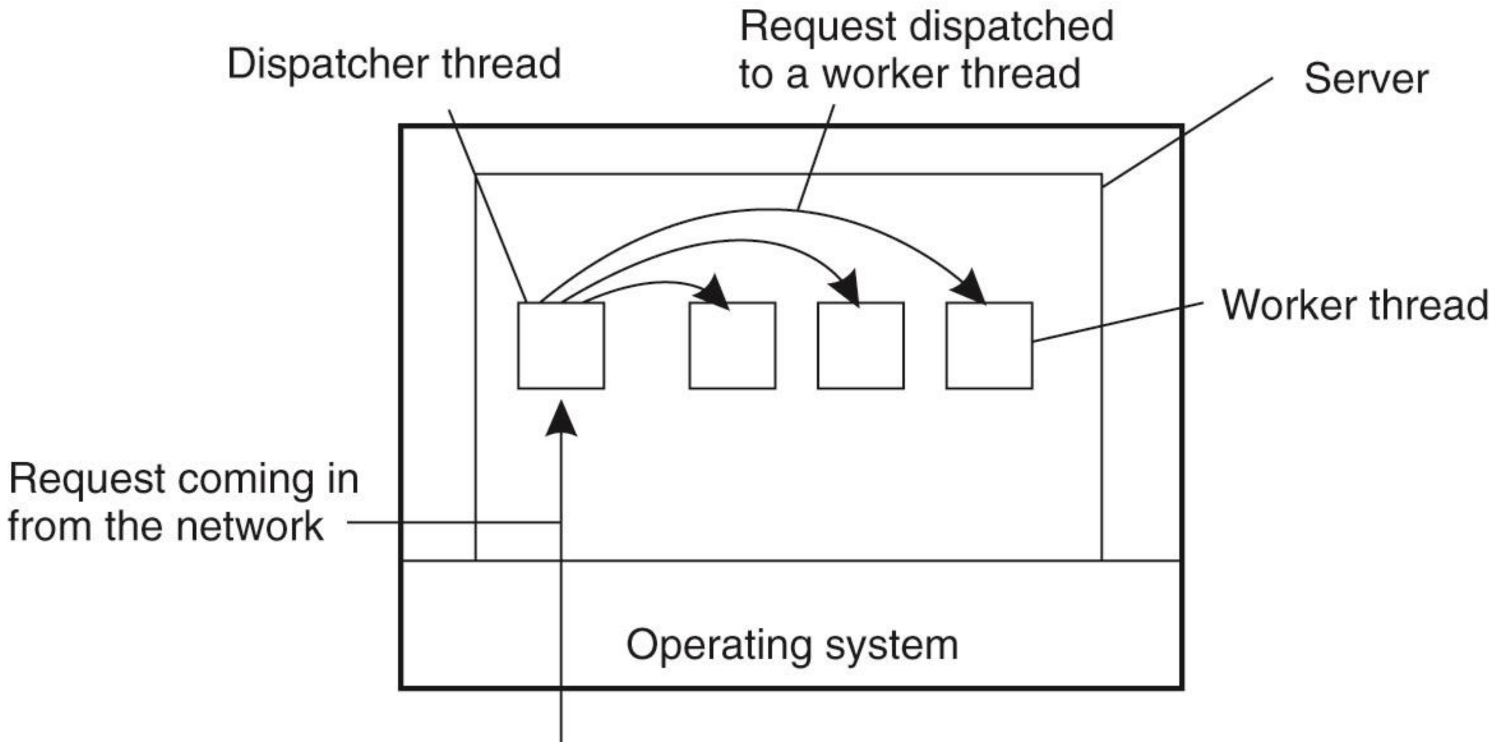


Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

Finite-state machine

- The file server is single threaded but doesn't block for I/O operations
- Instead, save state of current request, switch to a new task – client request or disk reply.
- Outline of operation:
  - Get request, process until blocking I/O is needed
  - Save state of current request, start I/O, get next task
  - If task = completed I/O, resume process waiting on that I/O using saved state, else service a new request if there is one.

Virtualization

- Multiprogrammed operating systems provide the illusion of simultaneous execution through *resource virtualization*
  - Use software to make it look like concurrent processes are executing simultaneously
- Virtual machine technology creates separate virtual machines, capable of supporting multiple instances of different operating systems.

Benefits

- Hardware changes faster than software
  - Suppose you want to run an existing application and the OS that supports it on a new computer: the VMM layer makes it possible to do so.
- Compromised systems (internal failure or external attack) are isolated.
- Run multiple different operating systems at the same time

Role of Virtualization in Distributed Systems

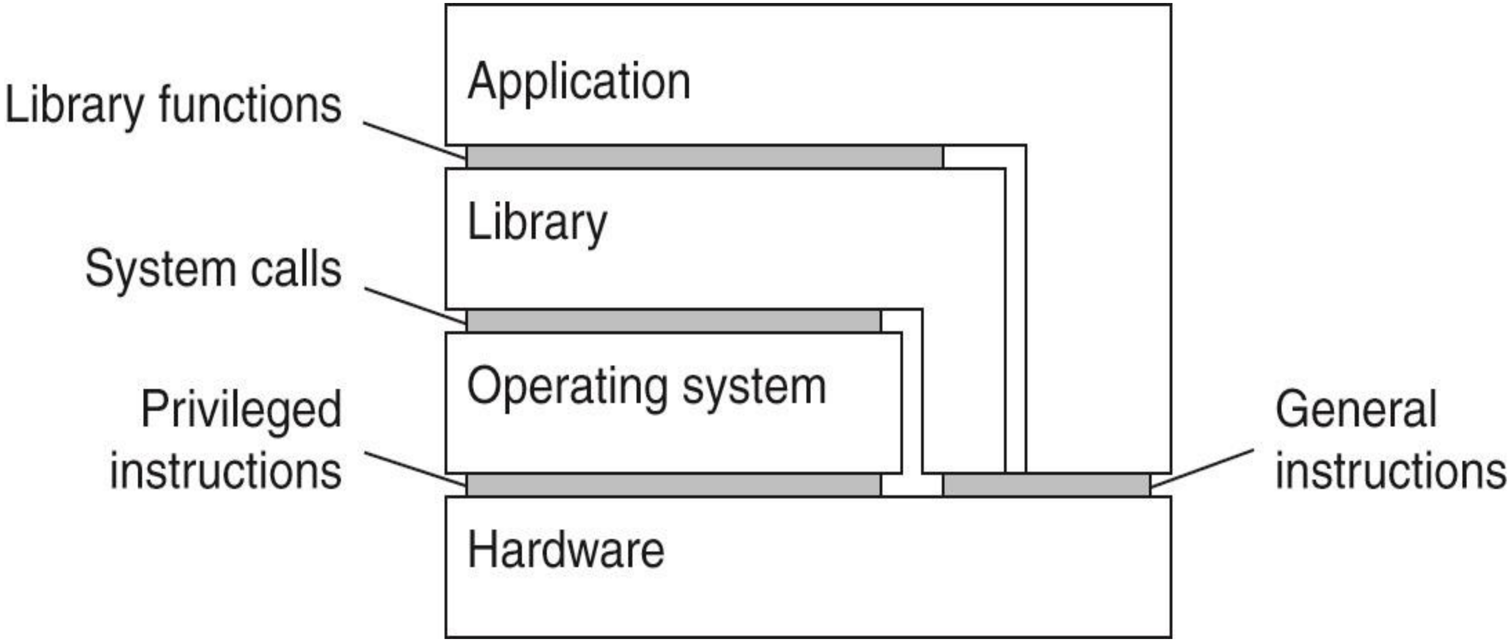
- Portability of virtual machines supports moving (or copying) servers to new computers

- Multiple servers can safely share a single computer

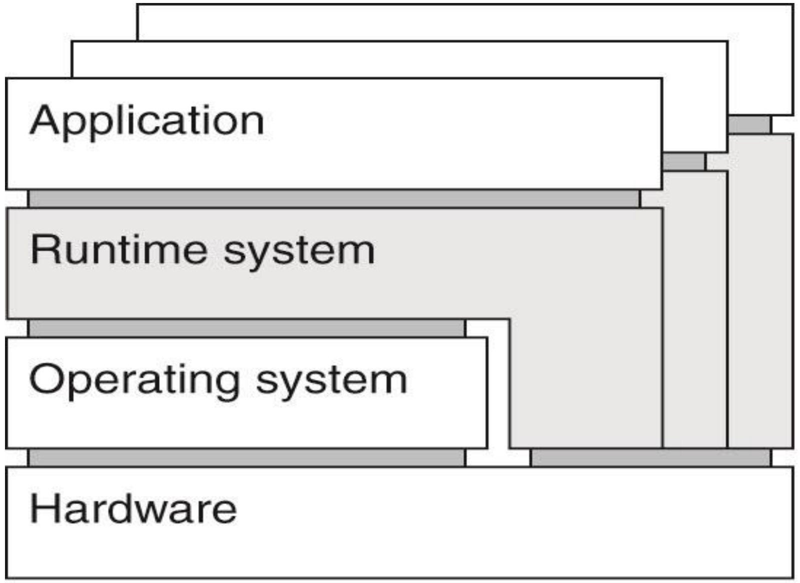
- Portability and security (isolation) are the critical characteristics.

Interfaces Offered by Computer Systems

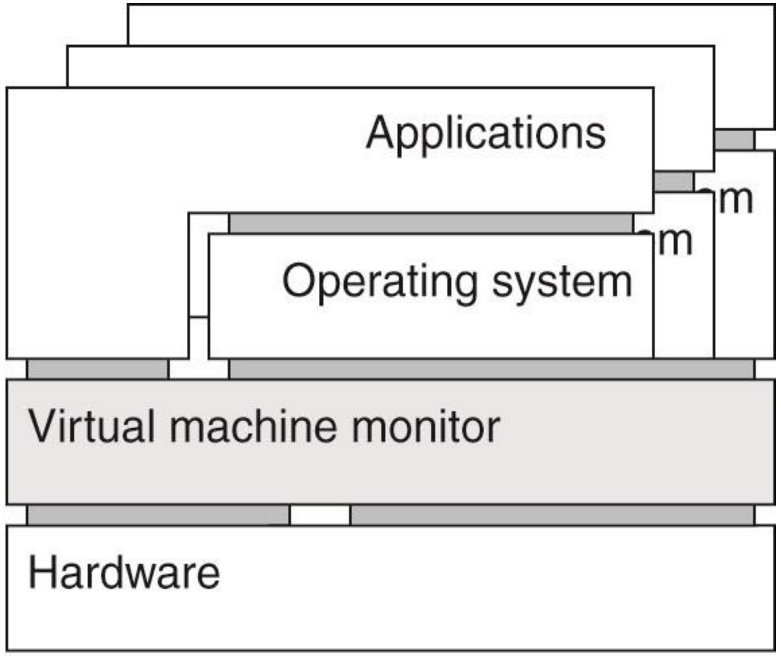
- Unprivileged machine instructions: available to any program
- Privileged instructions: hardware interface for the OS/other privileged software
- System calls: interface to the operating system for applications & library functions
- API: An OS interface through library function calls from applications.



Two Ways to Virtualize



(a)



(b)

<u>Process Virtual Machine</u>:
program is compiled to intermediate code, executed by a runtime system

<u>Virtual Machine Monitor:</u>
software layer mimics the instruction set; supports an OS and its applications

Processes in a Distributed System

Clients, Servers, and Code Migration

Another Distributed System Definition

"*Distributed systems* are networked computers in which the different components of a software application program run on different computers on a network, but all of the distributed components work cooperatively as if all were running on the same machine."

Networked User Interfaces

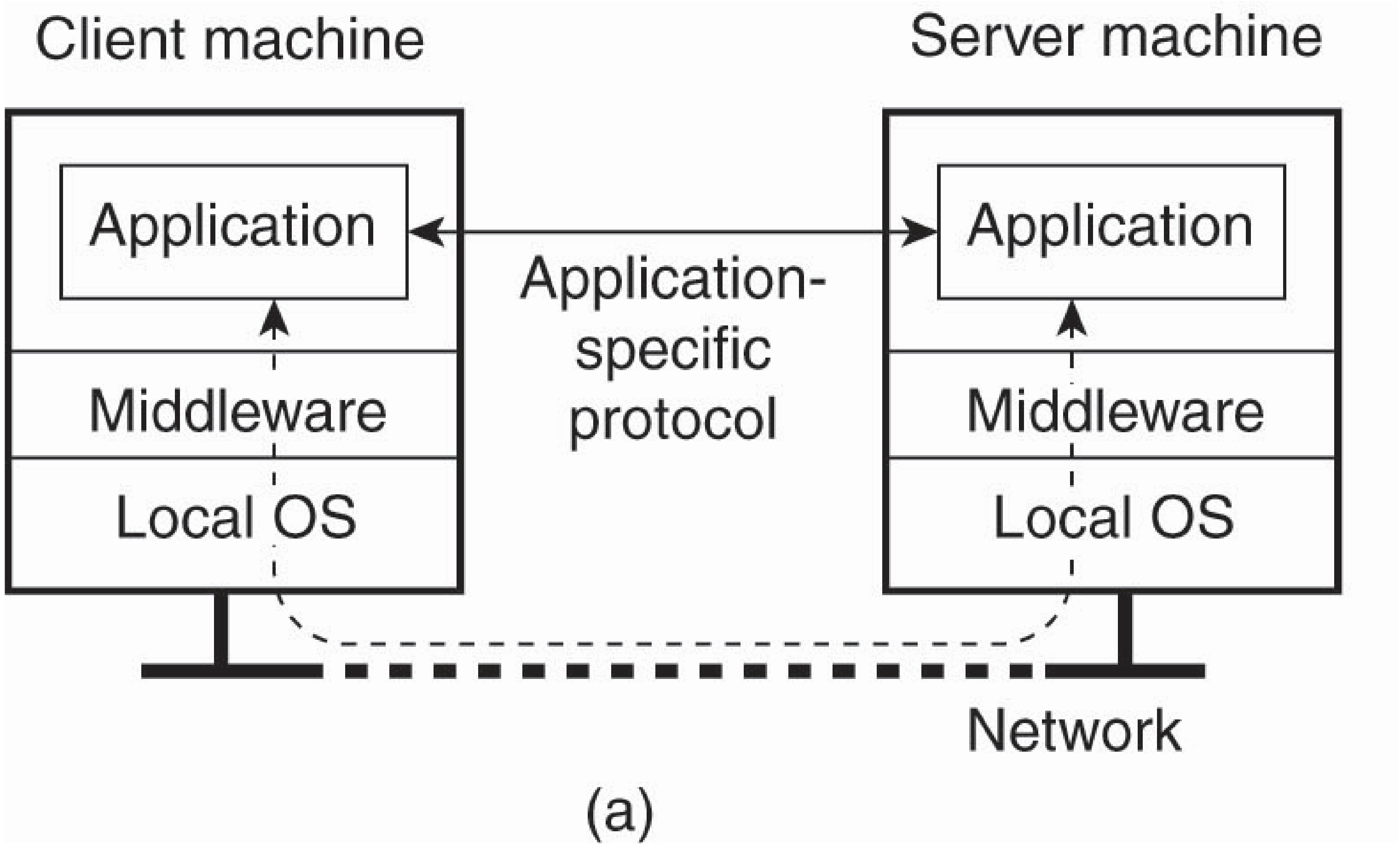


Figure 3-8. (a) A networked application with its own protocol

Networked User Interfaces

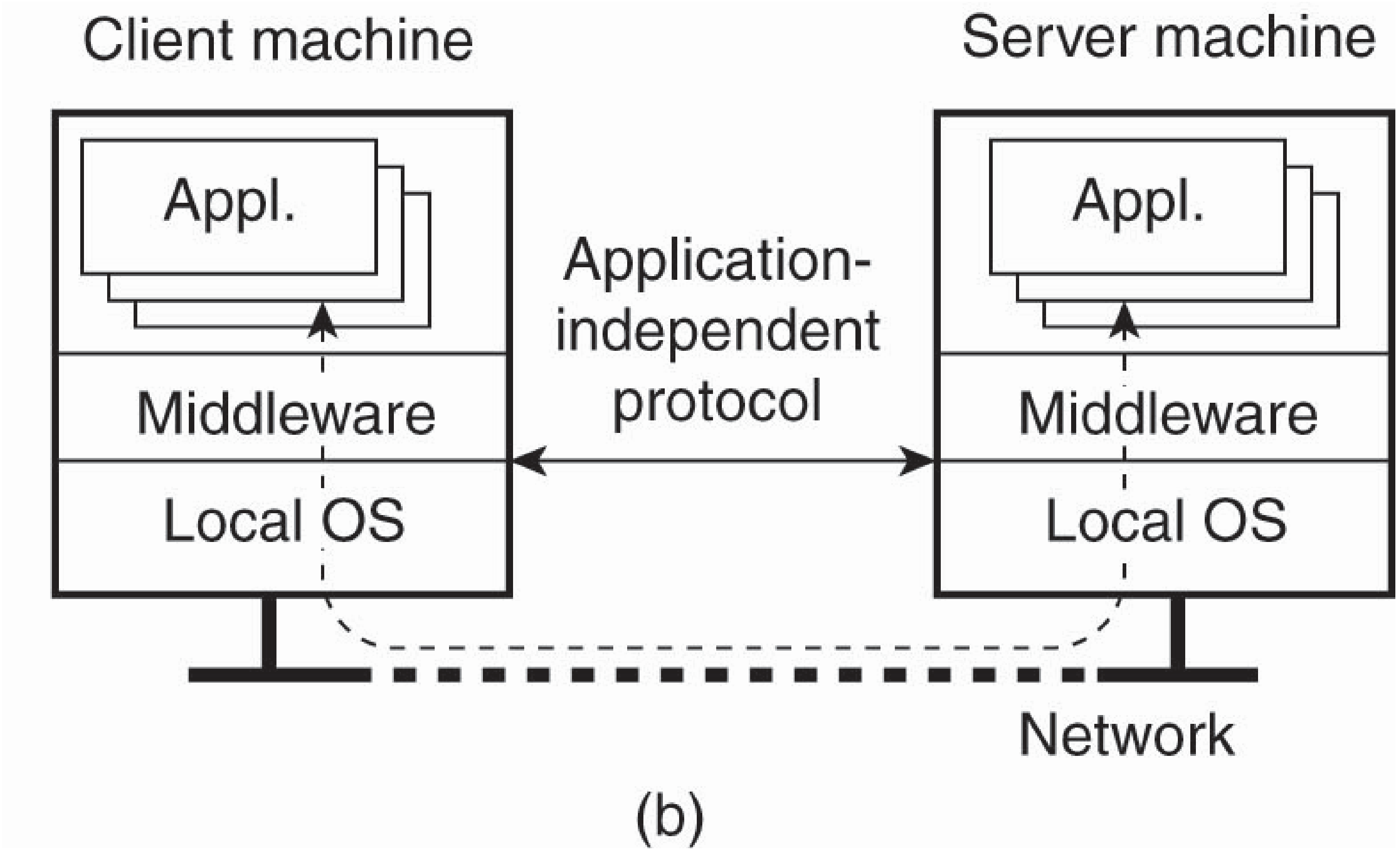
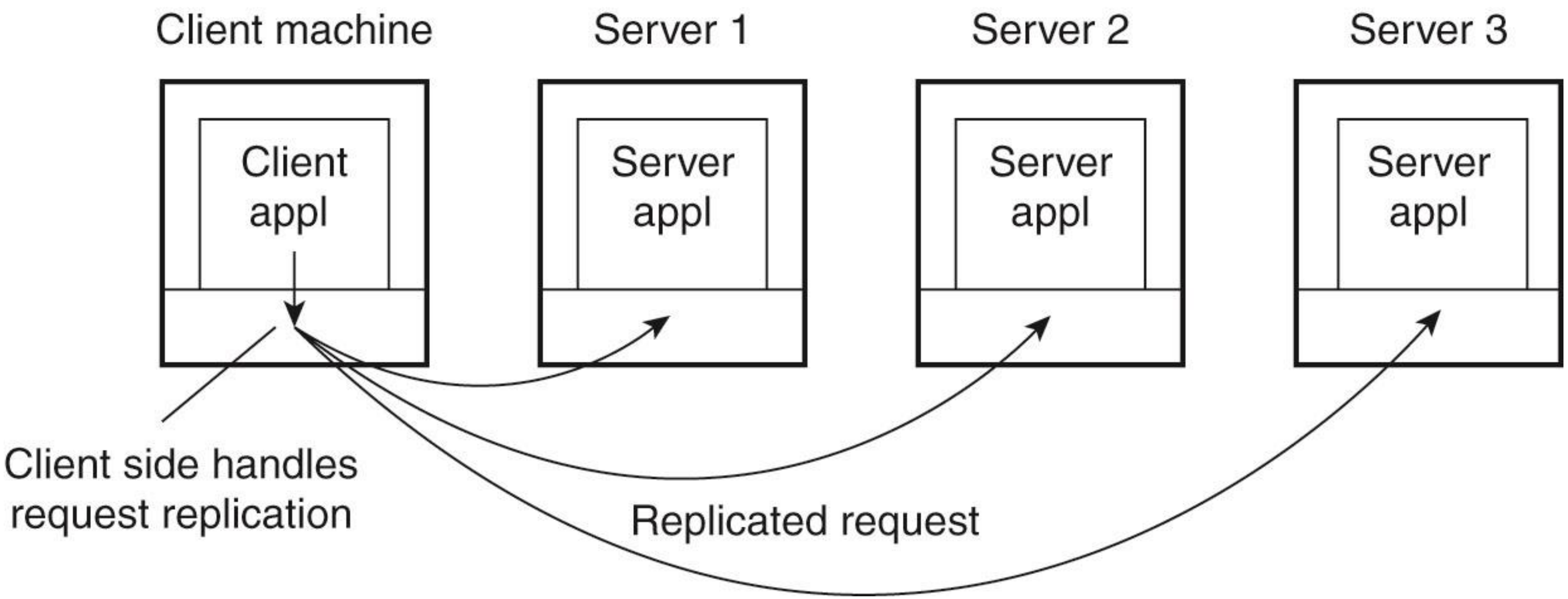


Figure 3-8. (b) A general solution to allow access to remote applications.

Client Side Software

- Manages user interface
- Parts of the processing and data (maybe)
- Support for <u>distribution transparency</u>
  - **Access** transparency: Client side stubs hide communication and hardware details.
  - **Location**, migration, and relocation transparency rely on naming systems, among other techniques
  - **Failure** transparency (e.g., client middleware can make multiple attempts to connect to a server)

Client-Side Software for Replication Transparency



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

Here, the client application is shielded from replication issues by client-side software that takes a single request and turns it into multiple requests; takes multiple responses and turn them into a single response.

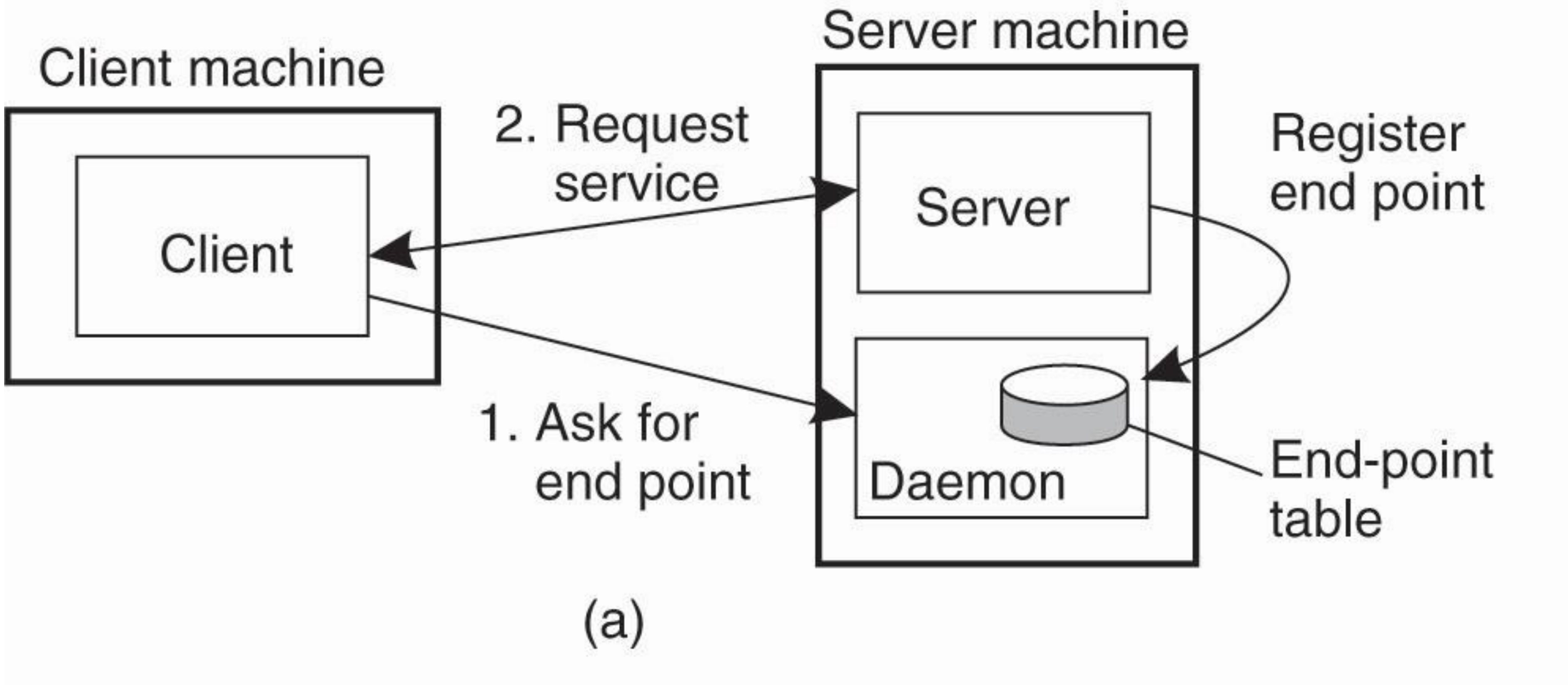
Servers

- Processes that implement a service for a collection of clients
  - Passive: servers wait until a request arrives
- Server Design:
  - **Iterative** servers: handles one request at a time, returns response to client
  - **Concurrent** servers: act as a central receiving point
    - Multithreaded servers versus forking a new process

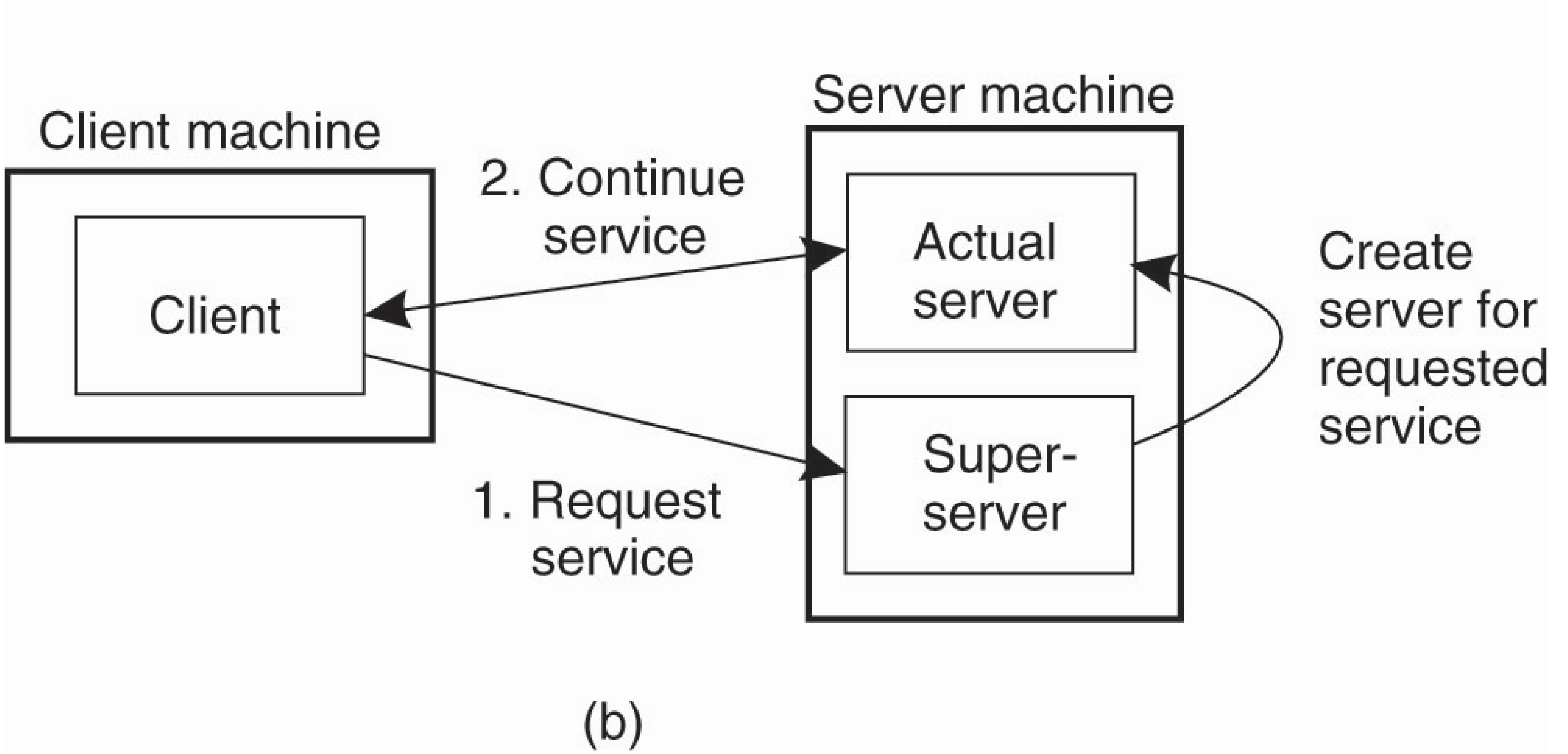
Contacting the Server

- Client requests are sent to an **end point**, or **port**, at the server machine.
- How are port numbers located?
  - **Global**: e.g; 21 for FTP requests and 80 for HTTP
  - Or, contact a **daemon** on a server machine that runs multiple services.
- For services that don't need to run continuously, superservers can listen to several ports, create servers as needed.

Client-to-server binding using a daemon



Client-to-server binding using a superserver



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.
- Solutions
  - The user exit the client application, immediately restart it
  - Send **out-of-band** data

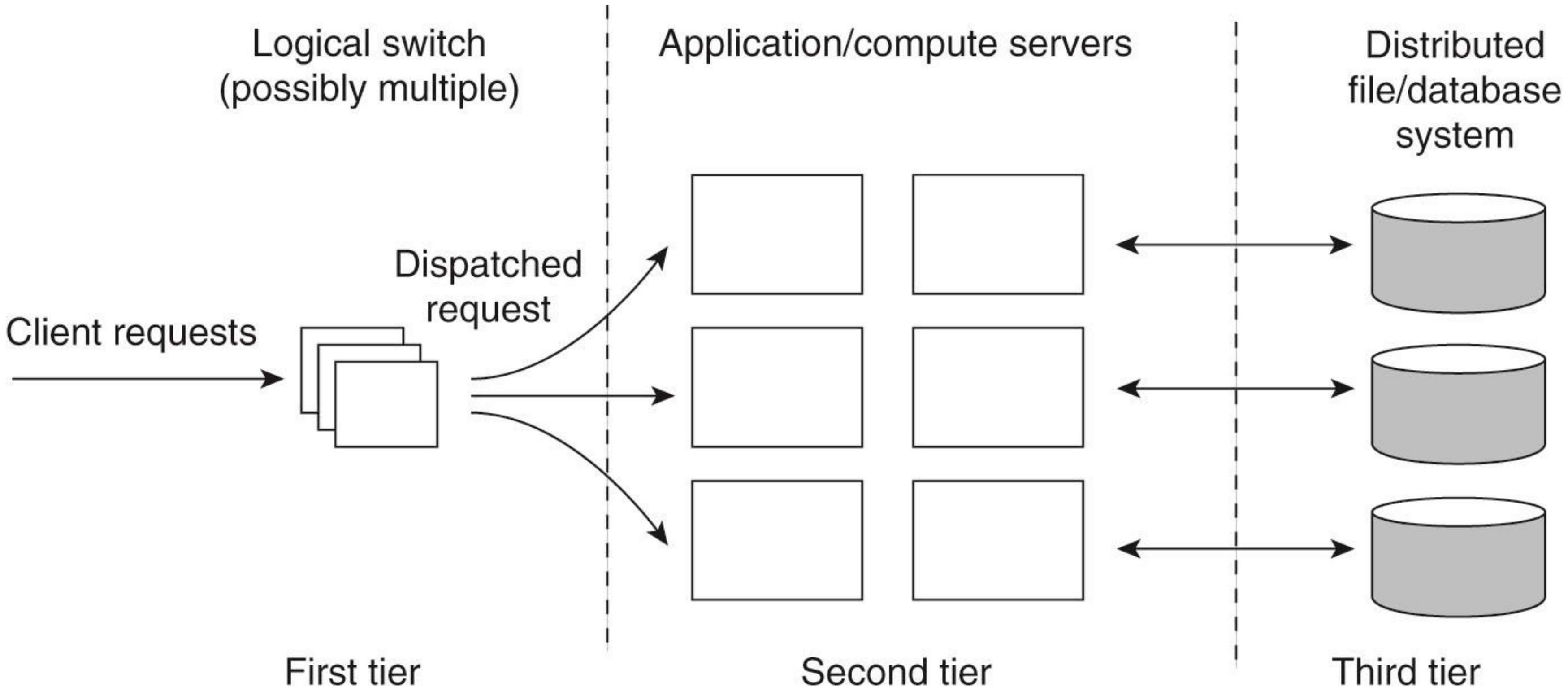
Stateful vs. Stateless

- Some servers keep no information about clients (Stateless)
  - Example: a web server which honors HTTP requests doesn't need to remember which clients have contacted it.

- Stateful servers retain information about clients and their current state, e.g., updating file.
  - Loss of state may lead to permanent loss of information.

Server Clusters

- A server cluster is a collection of machines, connected through a network, where each machine runs one or more services.
- Often clustered on a LAN
- Three tiered structure is common
  - Client requests are routed to one of the servers through a front-end switch



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

Three tiered server cluster

- Tier 1: the switch (access/replication transparency)

- Tier 2: the servers
  - Some server clusters may need special compute-intensive machines in this tier to process data

- Tier 3: data-processing servers, e.g. file servers and database servers
  - For other applications, the major part of the workload may be here

Server Clusters

- In some clusters, all server machines run the same services
- In others, different machines provide different services
  - May benefit from load balancing
  - One proposed use for virtual machines

Server Clusters

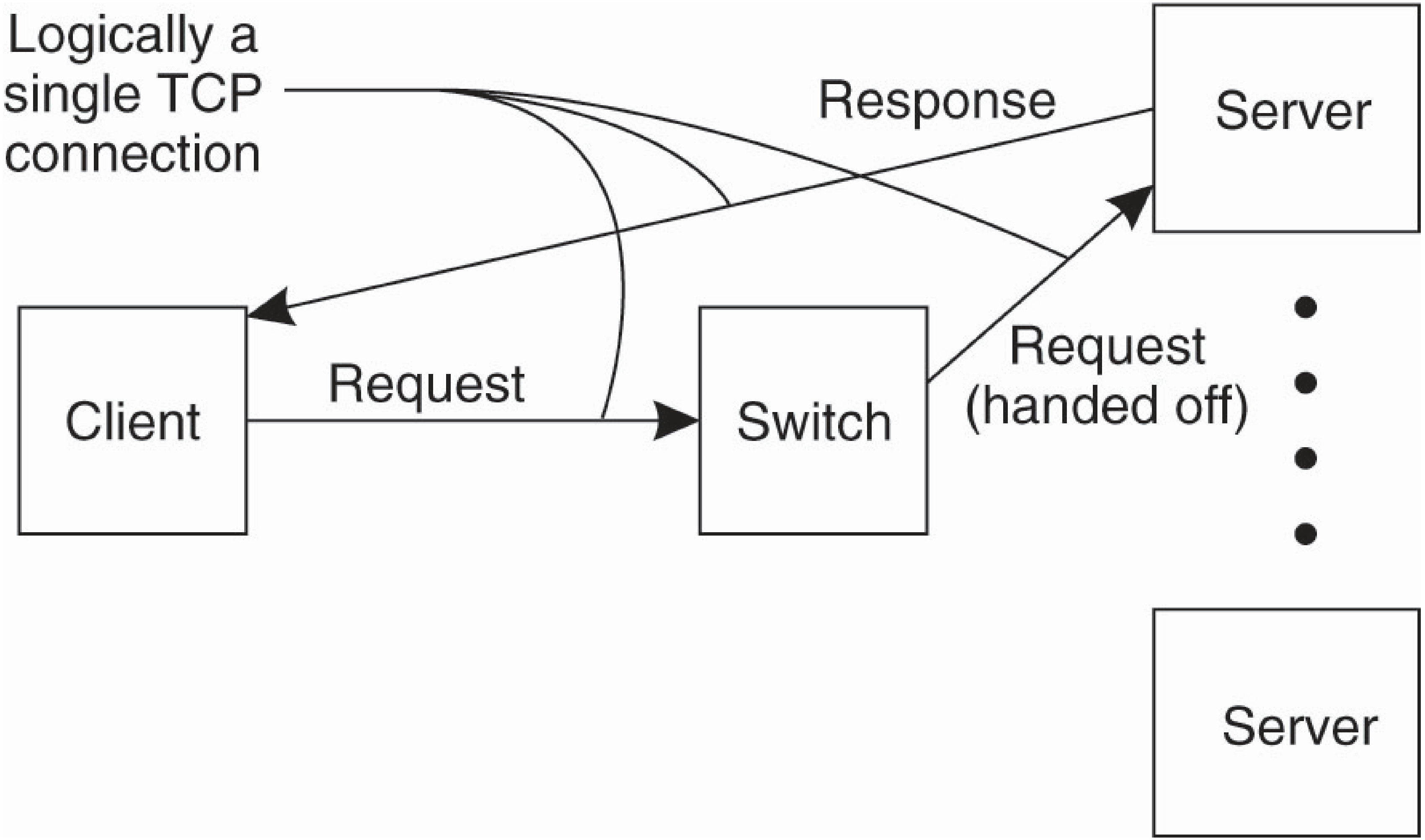


Figure 3-13. The principle of TCP handoff

Distributed Servers

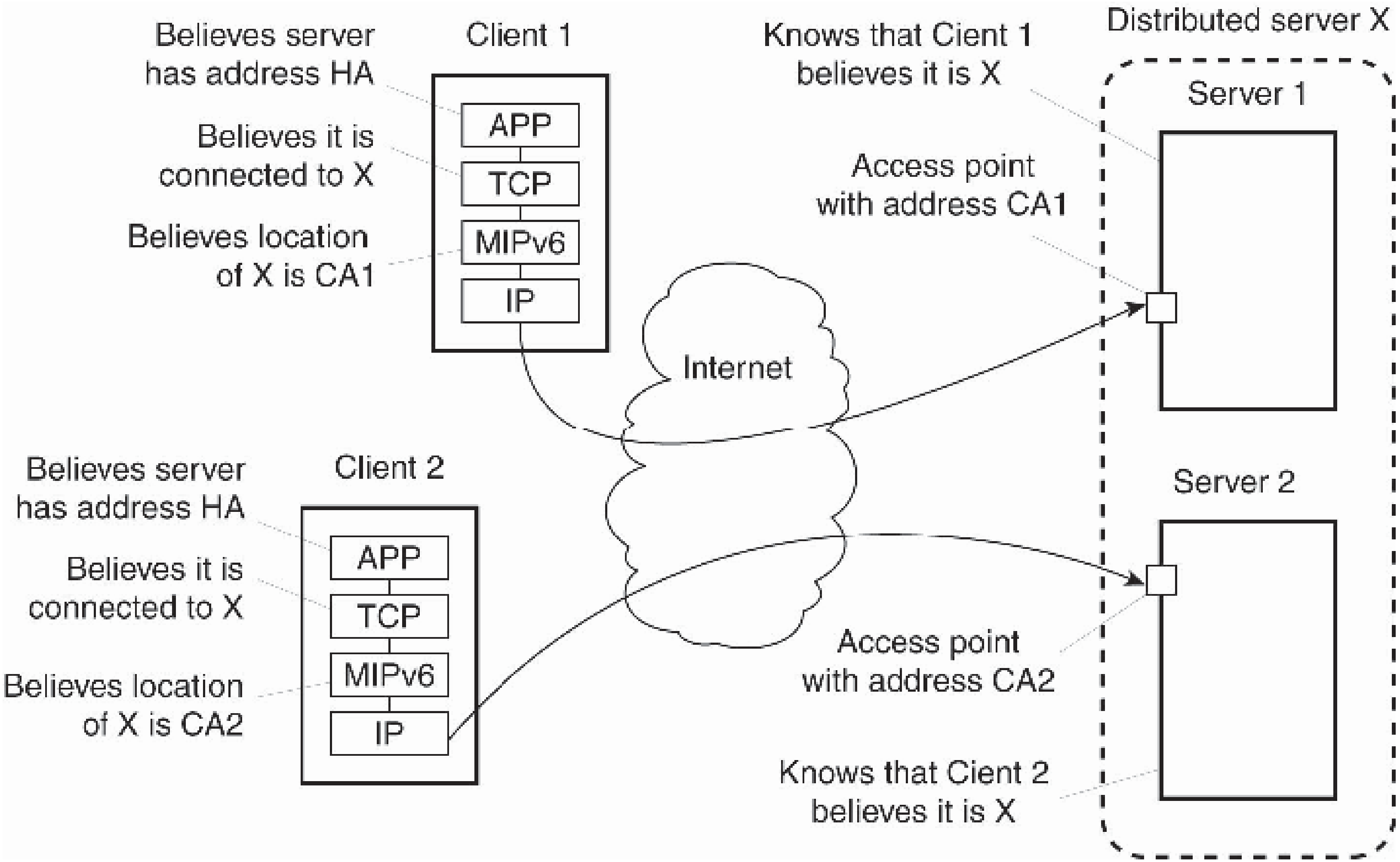


Figure 3-14. Route optimization in a distributed server.

Code Migration: Overview

- Instead of distributed system communication based on passing **data**, why not pass **code** instead?
  - Load balancing
  - Reduce communication overhead
  - Parallelism; e.g., mobile agents for web searches
  - Flexibility – configure system architectures dynamically

Code Migration: Overview

- Process migration may require moving the entire process state;
- Early DS's focused on process migration & tried to provide it transparently

Client-Server Examples

- Example 1: (Send Client code to Server)
  - Server manages a huge database. If a client application needs to perform many database operations, it may be better to ship part of the client application to the server and send only the results across the network.
- Example 2: (Send Server code to Client)
  - In many interactive DB applications, clients need to fill in forms that are subsequently translated into a series of DB operations. Reduce network traffic, improve service. Security issues?

Examples

- Mobile agents: independent code modules that can migrate from node to node in a network and interact with local hosts; e.g. to conduct a search at several sites in parallel
- Dynamic configuration of DS: Instead of pre-installing client-side software to support remote server access, download it dynamically from the server when it is needed.

Code Migration

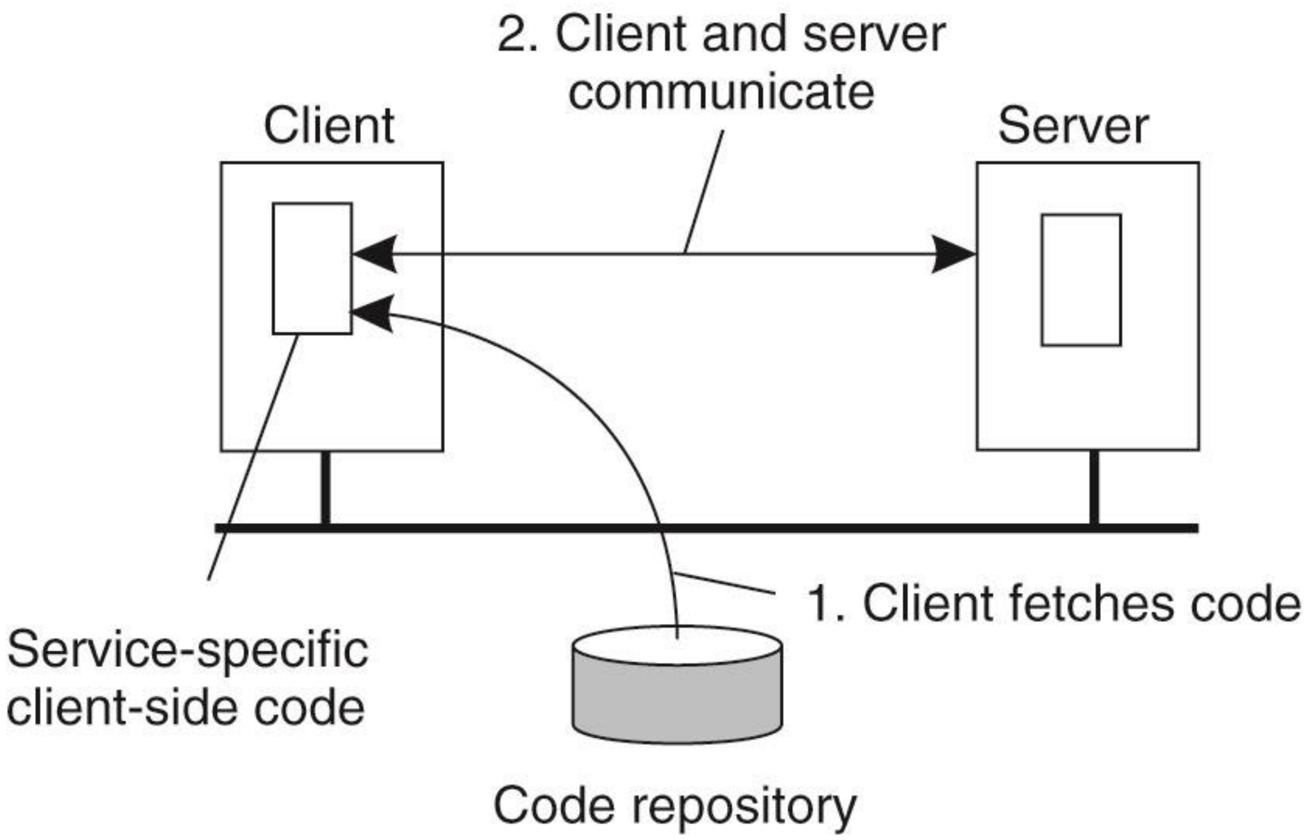


Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

A Model for Code Migration (1)

as described in Fuggetta et. al. 1998

- Three components of a process:
  - **Code segment**: the executable instructions
  - **Resource segment**: references to external resources (files, printers, other processes, etc.)
  - **Execution segment**: contains the current state
    - Private data, stack, program counter, other registers, etc. – data that will be saved during a context switch.

A Model for Code Migration (2)

- Weak mobility: transfer the code segment and possibly some initialization data.
  - Process can only migrate before it begins to run, or perhaps at a few intermediate points.
  - Requirements: portable code
  - Example: Java applets
- Strong mobility: transfer code segment and execution segment.
  - Processes can migrate after they have already started to execute
  - Much more difficult

A Model for Code Migration (3)

- Sender-initiated: initiated at the “home” of the migrating code
  - e.g., upload code to a compute server; launch a mobile agent, send code to a DB
- Receiver-initiated: host machine downloads code to be executed locally
  - e.g., applets, download client code, etc.
- If used for load balancing, sender-initiated migration lets busy sites send work elsewhere; receiver initiated lets idle machines volunteer to assume excess work.

Security in Code Migration

- Code executing remotely may have access to remote host's resources, so it should be trusted.
  - For example, code uploaded to a server might be able to corrupt its disk
- Question: should migrated code execute in the context of an existing process or as a separate process created at the target machine?
  - Java applets execute in the context of the target machine's browser
  - Efficiency (no need to create new address space) versus potential for mistakes or security violations in the executing process.

Cloning v Process Migration

- Cloned processes can be created by a *fork* instruction (as in UNIX) and executed at a remote site
    - Migration by cloning improves distribution transparency because it is based on a familiar programming model
    - UNIX has a clone() function that connects to a remote host, copies the process over, executes a fork() & exec() to start it.

Models for Code Migration

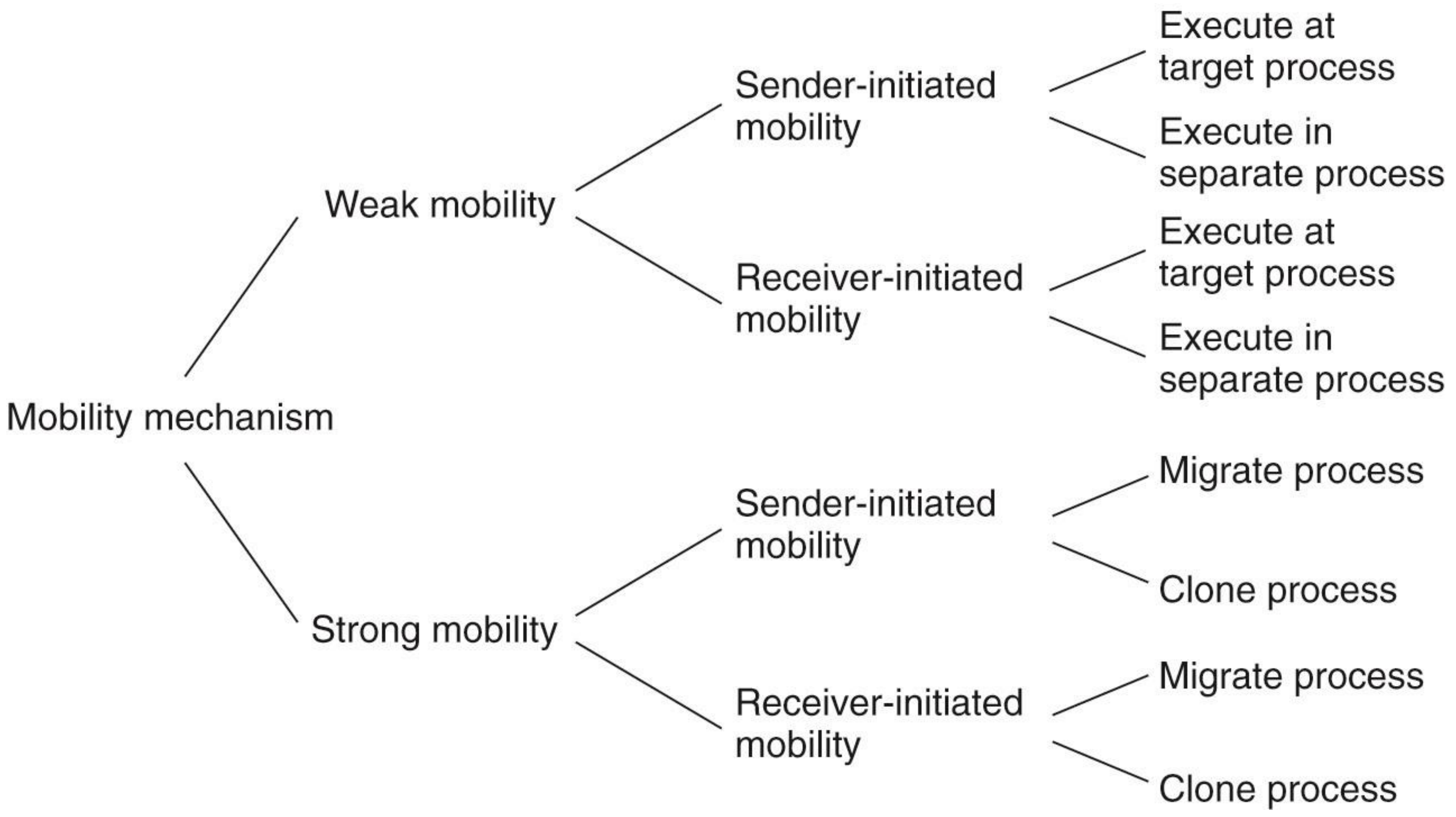


Figure 3-18. Alternatives for code migration.

Resource Migration

- Resources are bound to processes
  - By identifier: resource reference that identifies a particular object; e.g. a URL, an IP address, local port numbers.
  - By value: reference to a resource that can be replaced by another resource with the same “value”, for example, a standard library.
  - By type: reference to a resource by a type; e.g., a printer or a monitor

- Code migration cannot change (weaken) the way processes are bound to resources.

Resource Migration

- How resources are bound to machines:
  - Unattached: easy to move; my own files
  - Fastened: harder/more expensive to move; a large DB or a Web site
  - Fixed: can’t be moved; local devices

- Global references: meaningful across the system
  - Rather than move fastened or fixed resources, try to establish a global reference

Migration and Local Resources

		Resource-to-machine binding		
		Unattached	Fastened	Fixed
Process-to-resource binding	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

- Different computers, different operating systems – migrated code is not compatible

- Can be addressed by providing process virtual machines:
  - Directly interpret the migrated code at the host site (as with scripting languages)
  - Interpret intermediate code generated by a compiler (as with Java)

Migrating Virtual Machines

- A virtual machine encapsulates an entire computing environment.
- If properly implemented, the VM provides strong mobility since local resources may be part of the migrated environment
- “Freeze” an environment (temporarily stop executing processes) & move entire state to another machine
  - e.g. In a server cluster, migrated environments support maintenance activities such as replacing a machine.

Migration of Virtual Machines

- Example: real-time (“live”) migration of a virtualized operating system with all its running services among machines in a server cluster on a local area network.
- Problems:
  - Migrating the memory image (page tables, in-memory pages, etc.)
  - Migrating bindings to local resources

Memory Migration in Virtual Machines

- Three possible approaches
  - **Pre-copy**: push memory pages to the new machine and resend the ones that are later modified during the migration process.
  - **Stop-and-copy**: pause the current virtual machine; migrate memory, and start the new virtual machine.
  - Let the new virtual machine pull in new pages as needed, using demand paging

Resource Migration in a Cluster

- Migrating local resource bindings is simplified in this example because we assume all machines are located on the same LAN.
  - "Announce" new address to clients
  - If data storage is located in a third tier, migration of file bindings is trivial.