Distributed Systems Lecture 3

Professor Louise E. Moser

Spring 2013

Basic Idea of Threads

- Build **virtual processors** in **software**, on top of physical hardware processors
- **Processor:** Provides a set of instructions, along with the ability to execute a sequence of such instructions
- **Process:** A software processor in whose context one or more threads are executed
- **Thread:** A minimal software processor in whose context a sequence of instructions are executed
  - **Executing a thread** means executing a sequence of instructions in the context of that thread
  - **Saving a thread context** means stopping the current execution and saving all of the data needed to continue the execution at a later time

Threads and Operating Systems

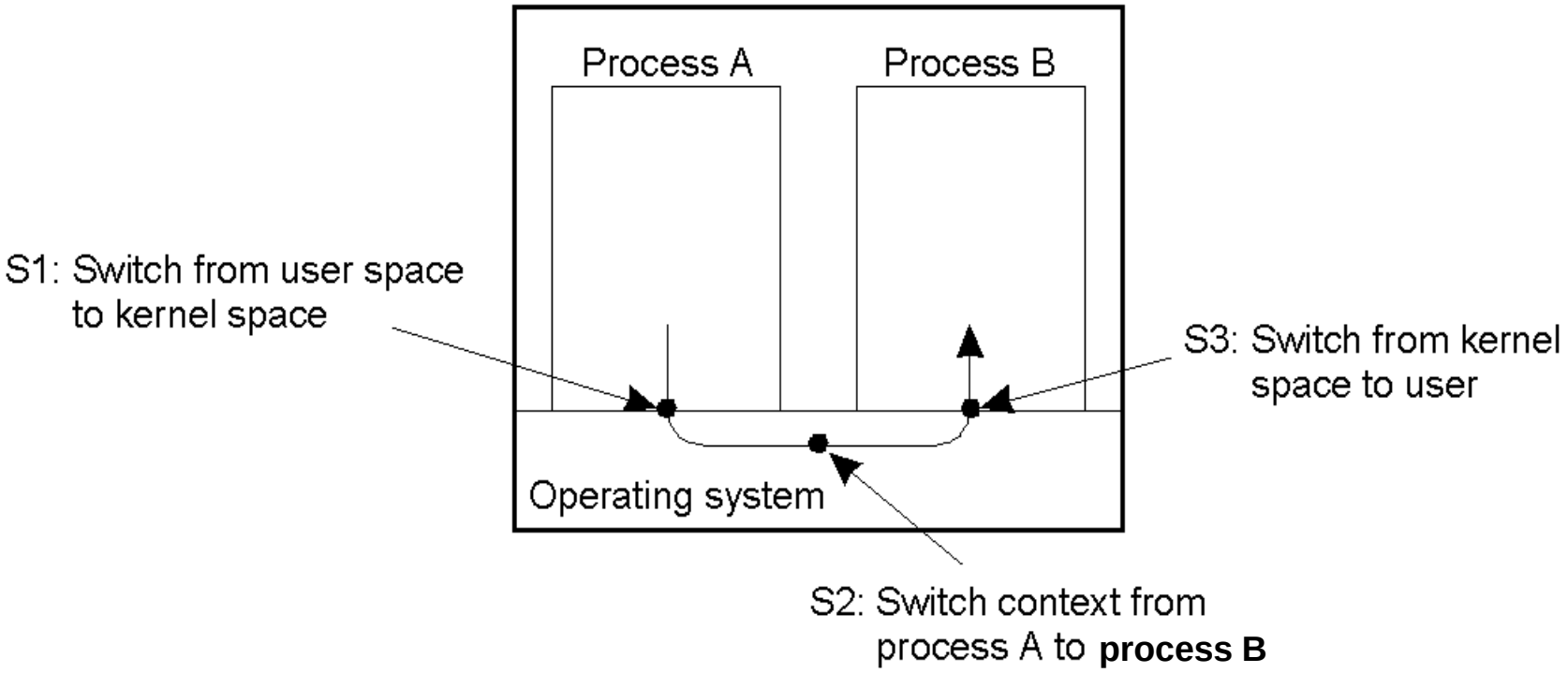
- **Main issue:** Should the OS kernel provide threads, or should they be implemented as a user-level package?
- **User-level solution:** The implementation of the thread package is contained in user space
  - **Operations can be completely handled within a single process**
    - PRO: Implementation can be extremely efficient
  - ***All* services that the kernel provides are performed for the *entire* process in which the thread resides**
    - CON: If the kernel decides to block a thread, the entire process is blocked
  - **In practice, we want to use threads when there are many external events with blocking of a thread on a per-event basis**
    - CON: If the kernel can't distinguish between threads, how can it signal events and block them individually

Threads and Operating Systems

- **Kernel-level solution:** The kernel contains the implementation of the thread package, which means that *all* operations return as *system calls*
  - PRO: Operations that block a thread are no longer a problem: The kernel simply schedules another thread in the same process
  - PRO: Handling external events is simple: The kernel (which catches all events) schedules the thread associated with the event
  - CON: Loss of efficiency due to the fact that each thread operation requires a trap to the kernel
- **Conclusion:** Try to mix user-level and kernel-level threads into a single concept

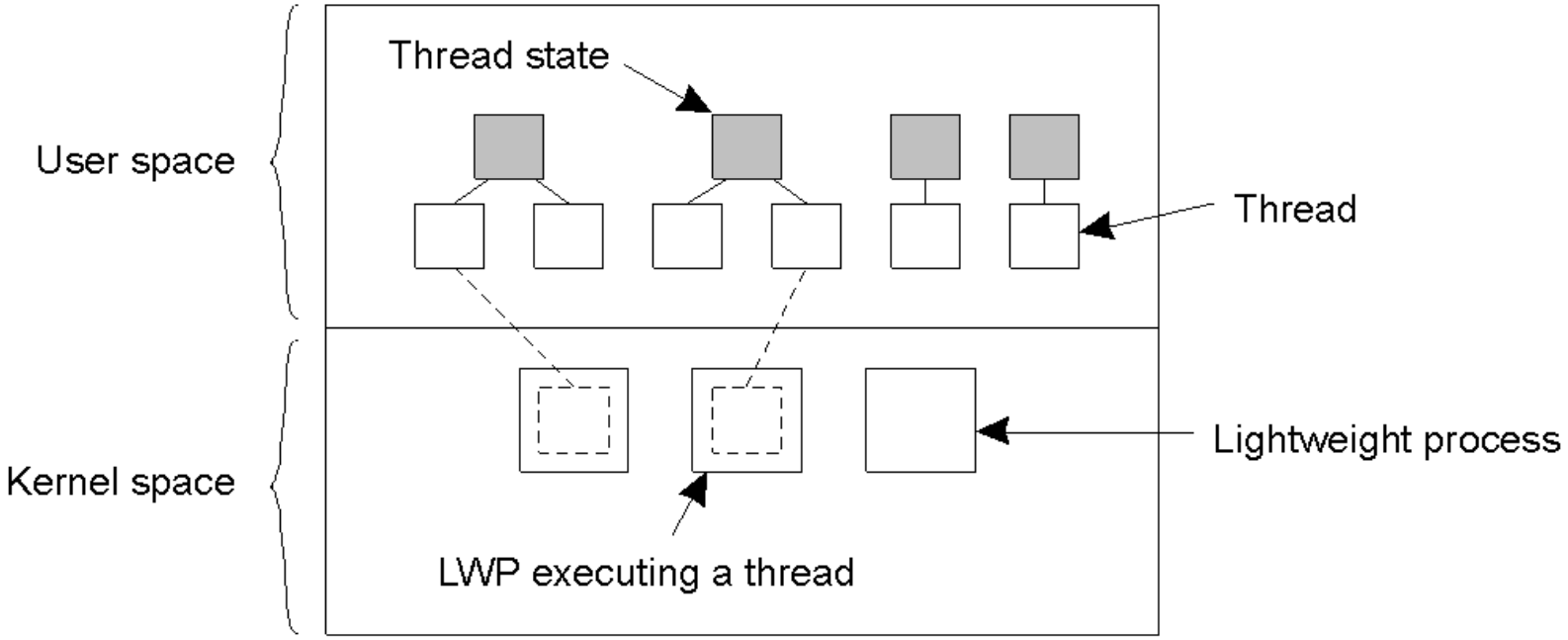
Threads in Non-Distributed Systems

- **Inter-Process Communication (IPC)** which results in **context switching** and thus performance overhead



Thread Implementation

- Associate threads with **Lightweight Processes (LWP)**, so they can be scheduled by the kernel
- Think of a Lightweight Process as an empty slot in the kernel's scheduling table



Threads in Distributed Systems

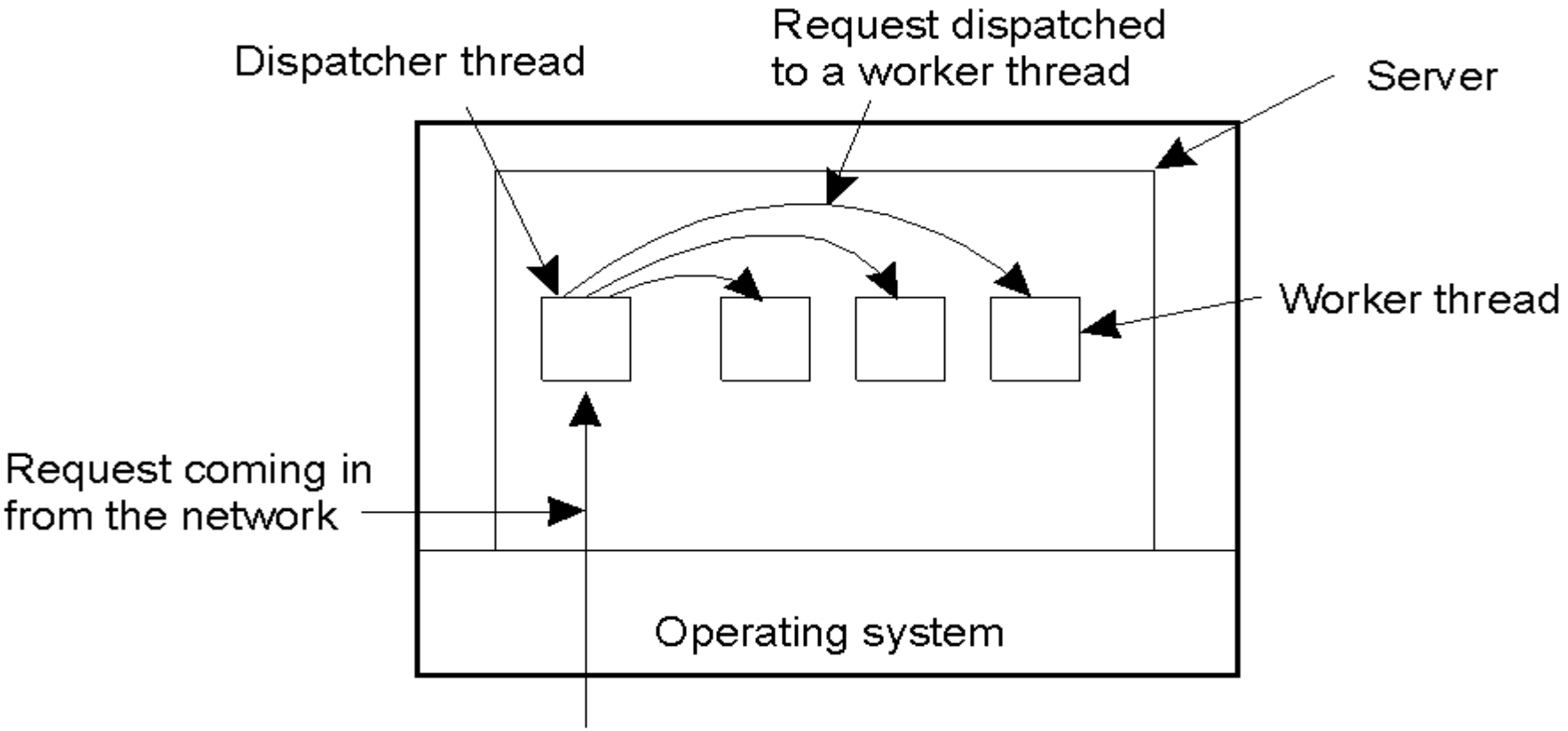
- **Multithreaded clients:** Main issue is hiding network latency
- **Multithreaded Web client**
  - Client's Web browser scans an incoming HTML page, and finds that more files (such as image files) need to be fetched
  - Each file is fetched by a separate thread over a separate TCP connection; each thread does a (blocking) HTTP request
  - As the files arrive, the client's Web browser displays them
- **Multiple RPCs**
  - A client does several RPCs at the same time (in parallel), each using a different thread and TCP connection
  - The client then waits until all results have been returned
  - If the RPCs are to different servers, we might see a linear speed-up compared to doing RPCs sequentially one after the other

Threads in Distributed Systems

- **Multithreaded servers:** Main issues are improved performance and better structure
- **Improved performance**
  - **Starting a new thread to handle an incoming request is *much* cheaper than starting a new process**
  - **Having a multithreaded server supports scaling the server to serve multiple clients**
  - **As with clients, network latency can be hidden by reacting to the next request while replying to the previous request**
- **Better structure**
  - **Most servers have high I/O demands: Use well-understood blocking calls to simplify the overall structure**
  - **Multithreaded programs tend to be smaller and easier to understand, due to simplified control flow**

Multithreaded Servers

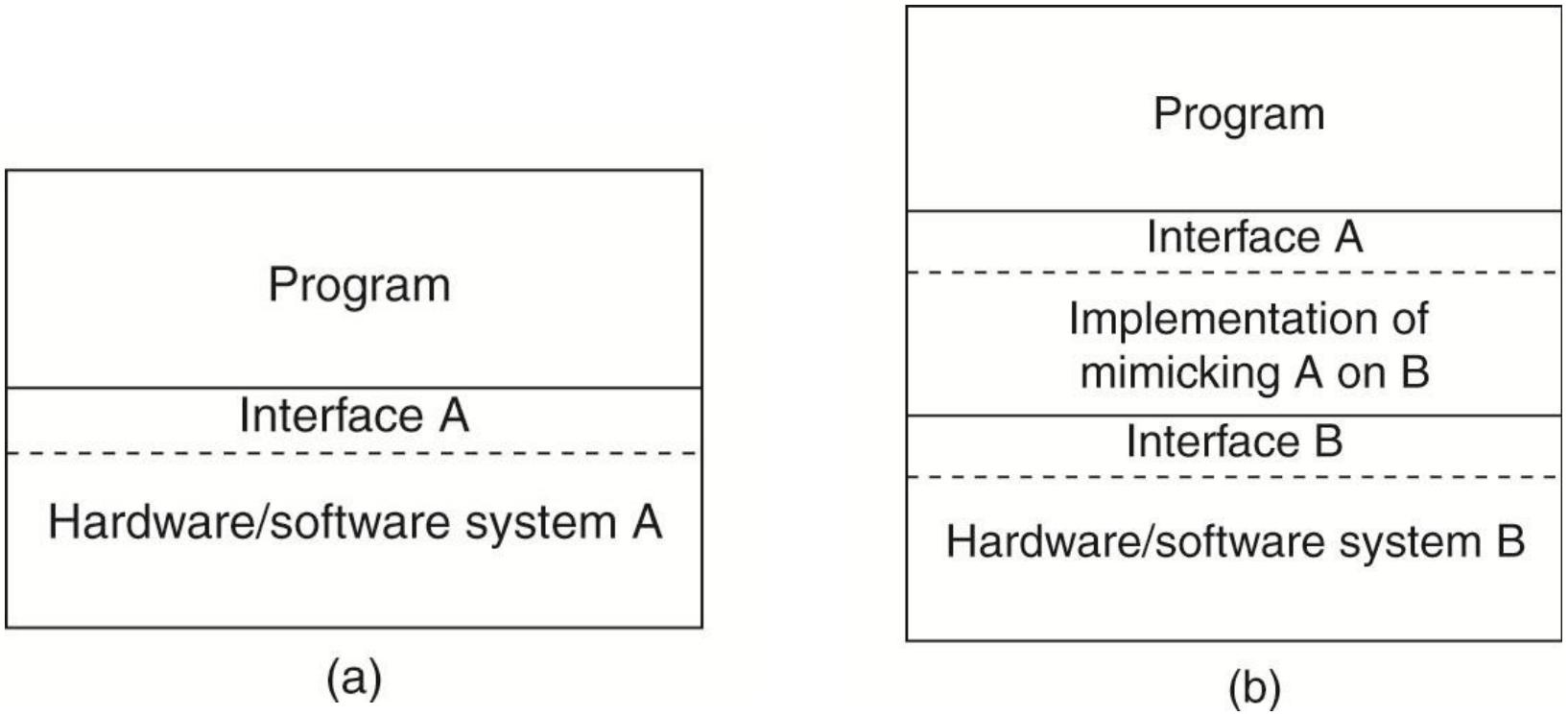
- A multithreaded server organized as a dispatcher thread and multiple worker threads



Virtualization in Distributed Systems

(a) Organization of a program, interface, and system

(b) Organization of a virtual system A on top of system B

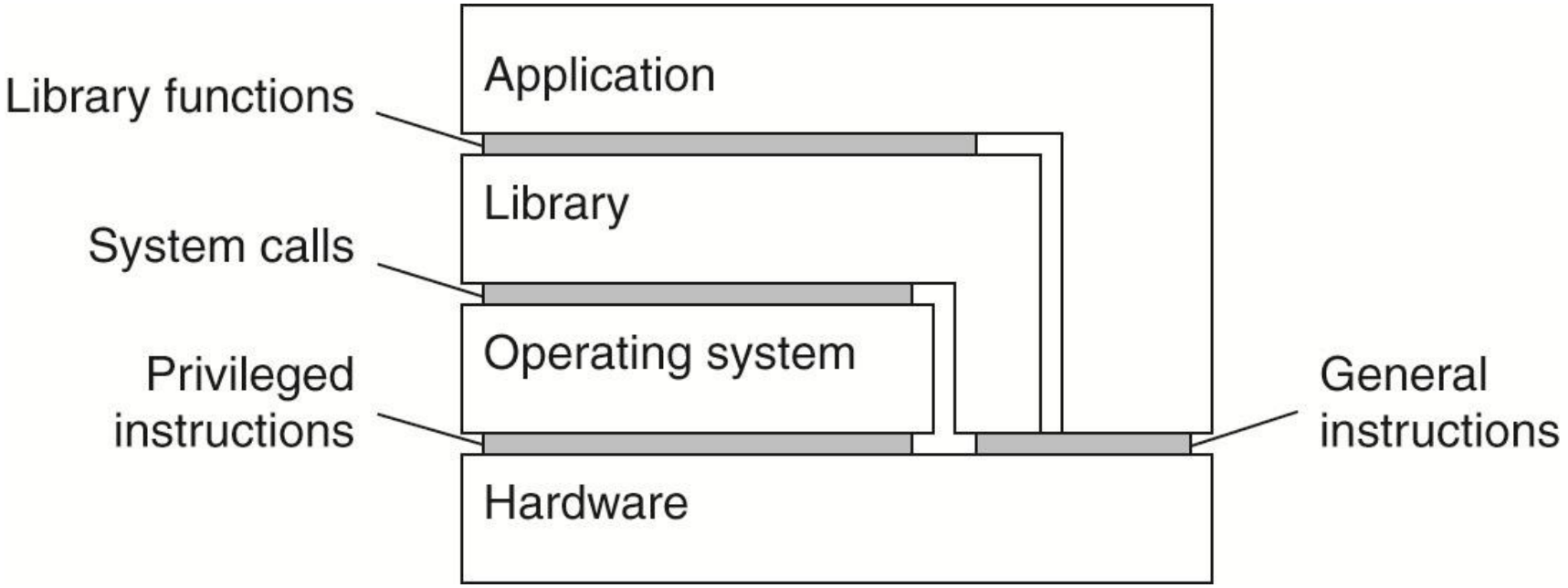


Virtual Machine Architectures

- Interfaces at different levels
  - **An interface consisting of an Application Programming Interface (API) with library calls that hide the system calls**
  - **An interface consisting of system calls that the operating system offers**
  - **An interface between the hardware and the software consisting of machine instructions that can be**
    - Invoked by any program
    - Invoked only by privileged programs, such as the operating system

Virtual Machine Architectures

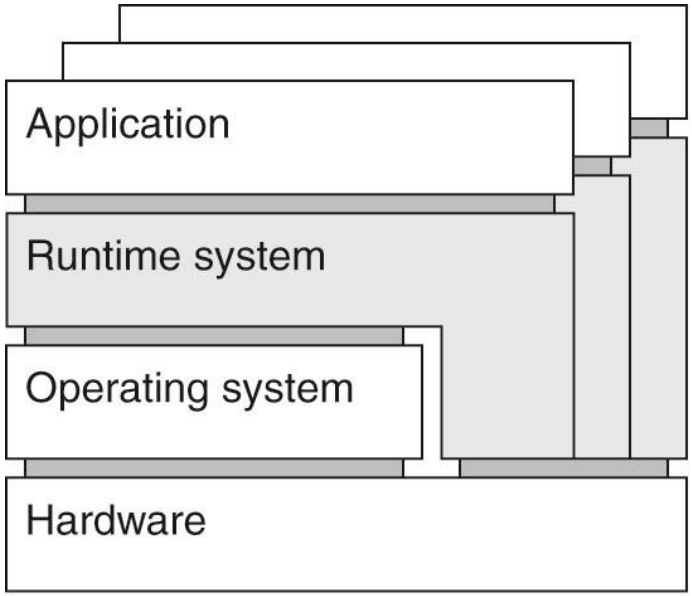
- Interfaces at different levels



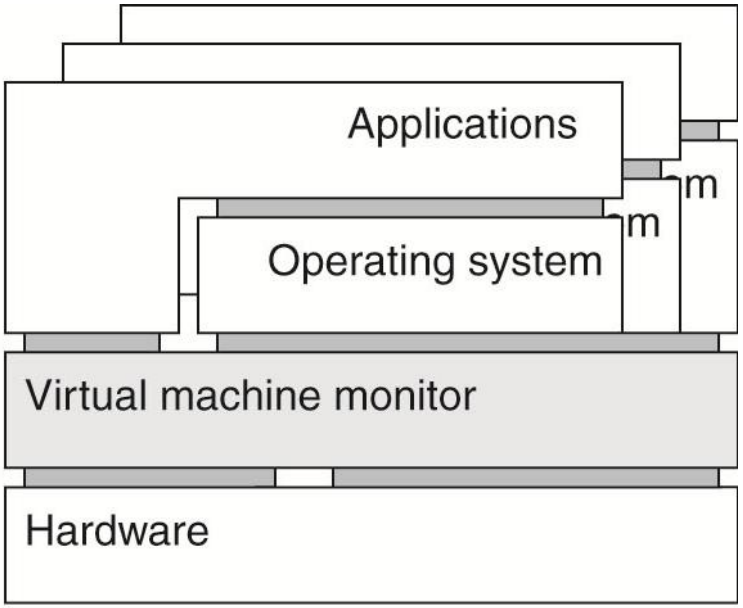
Virtual Machine Architectures

(a) A **process virtual machine**, with multiple instances of (application, runtime system) combinations

(b) A **virtual machine monitor**, with multiple instances of (application, operating system) combinations



(a)



(b)

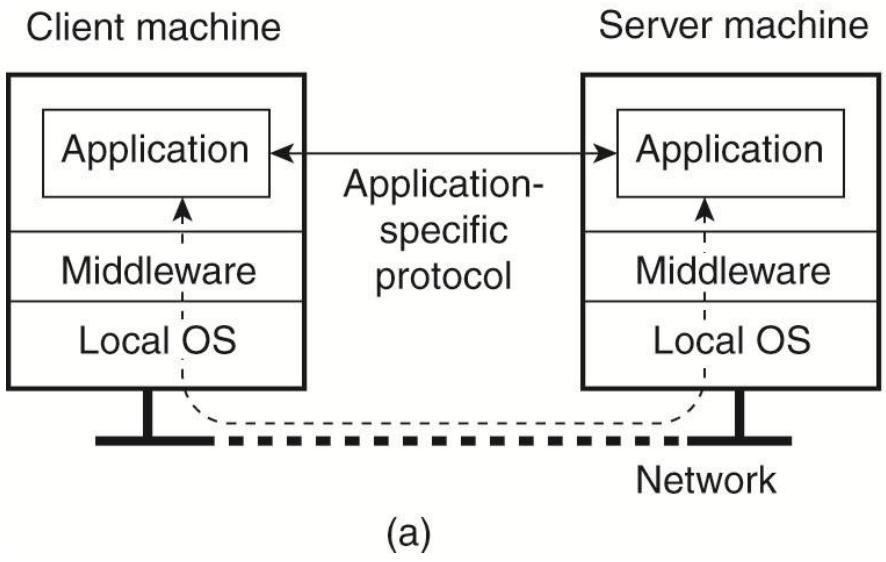
Client-Side Software

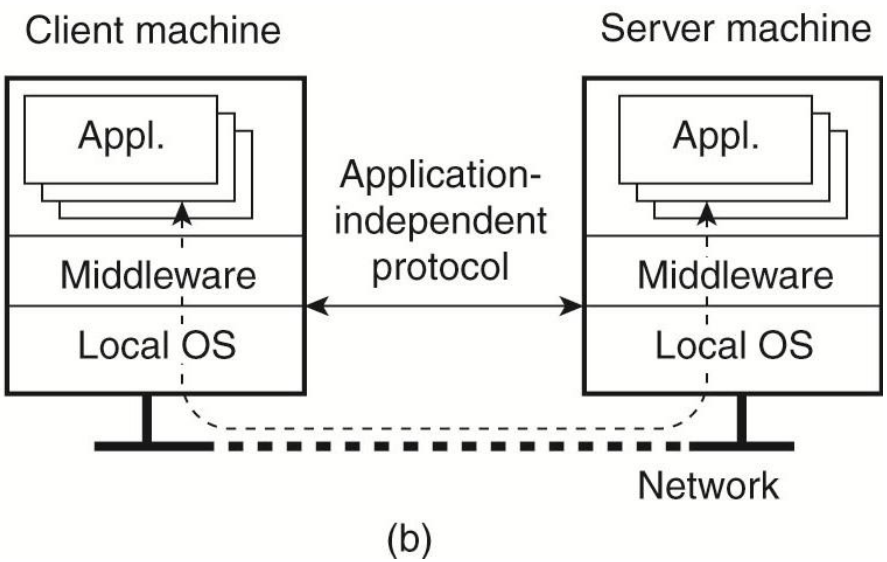
- **Essence:** A major part of client-side software is focused on **Graphical User Interfaces (GUIs)**
- **Compound documents:** To enable communication between applications, make the user interface **application-aware**
  - **Drag-and-drop: Move objects to other positions on the screen, possibly causing interactions with other applications**
  - **In-place editing: Integrate several applications at the user-interface level, e.g., word processing and drawing facilities**

Networked User Interfaces

(a) An **application-specific protocol**, such as the iTrust over HTTP protocol (see Lecture 2, slides 30-32)

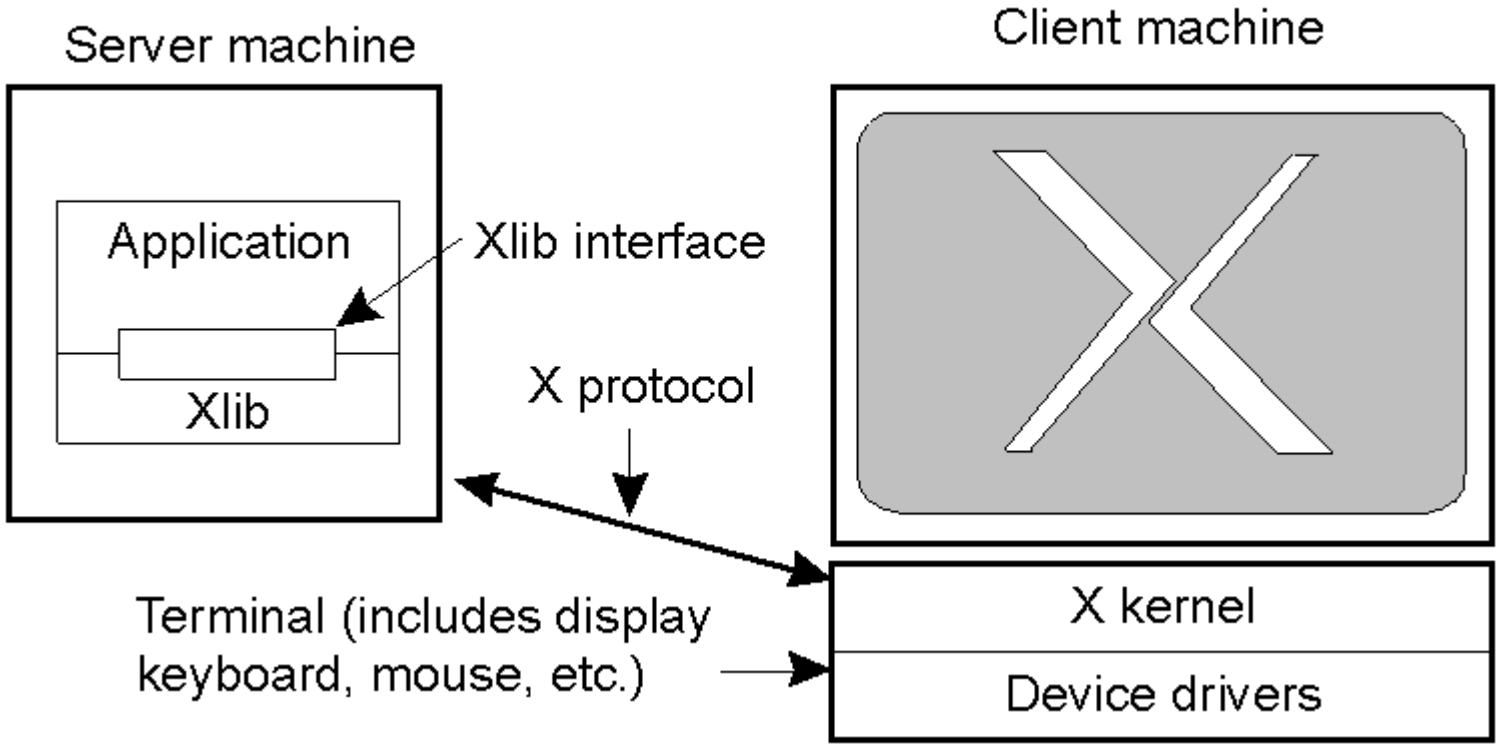
(b) An **application-independent protocol** that allows access to remote applications, such as HTTP or the X protocol (next slide)





Example: The X-Window System

- The X protocol is an application-independent protocol, for the X-Window System, originally designed for Linux

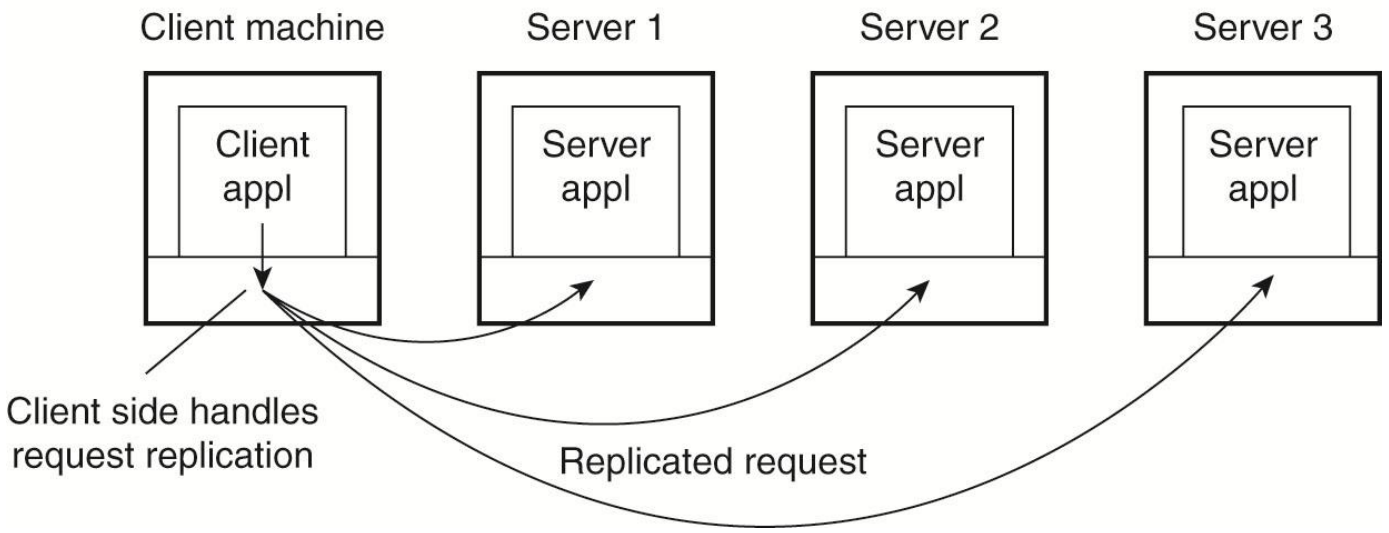


Client-Side Software

- **Essence:** Often focuses on providing **distribution transparency**
- **Access transparency:** Client-side stubs for RPCs and RMIs
- **Location / migration transparency:** Client-side software to keep track of the location of the servers
- **Replication transparency:** Client-side software to handle multiple invocations to different server replicas
- **Failure transparency:** Client-side software to mask server and communication failures

Client-Side Software

- Transparent replication of a server using client-side software



Servers

- **Basic model:** A **server** is a process that waits for incoming service requests at a specific Transport address, i.e., **port number**
- In practice, there is a one-to-one mapping between a service and a port

ftp-data	20	File Transfer Protocol [Default Data]
ftp	21	File Transfer Protocol [Control]
telnet	23	Telnet
	24	Any private mail system
smtp	25	Simple Mail Transfer Protocol
login	49	Login Host Protocol
http	80	Hyper-Text Transfer Protocol

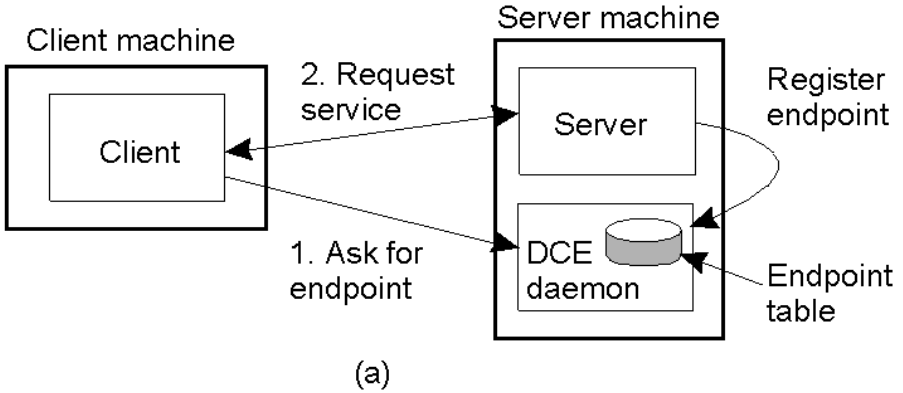
Servers

- **Super-servers:** Servers that listen on several ports, i.e., that provide several independent services
  - **In practice, when a service request comes in, they start a sub-process to handle the request (in Unix)**
- **Iterative servers vs. concurrent servers**
  - **Iterative servers can handle only one client at a time**
  - **Concurrent servers can handle multiple clients at the same time**

Servers: General Design Issues

(a) Client-to-server binding using a **daemon** (**name server**), as in the Distributed Computing Environment (DCE)

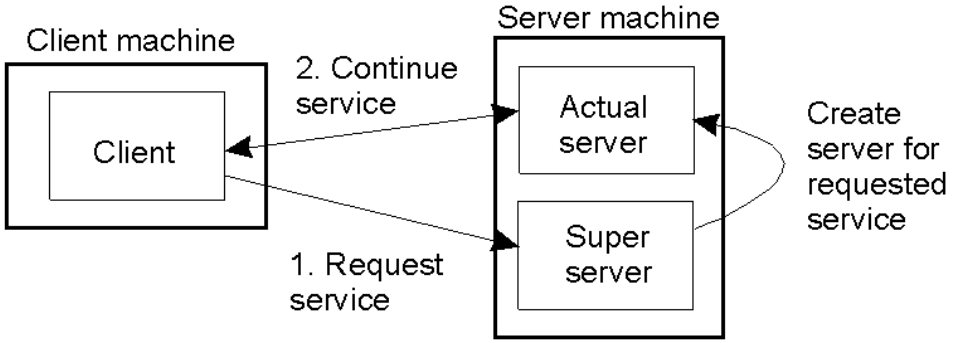
- **The server process is created by the DCE daemon and registers with the DCE daemon before the client requests service**



(a)

(b) Client-to-server binding using a **super-server**, as in Unix

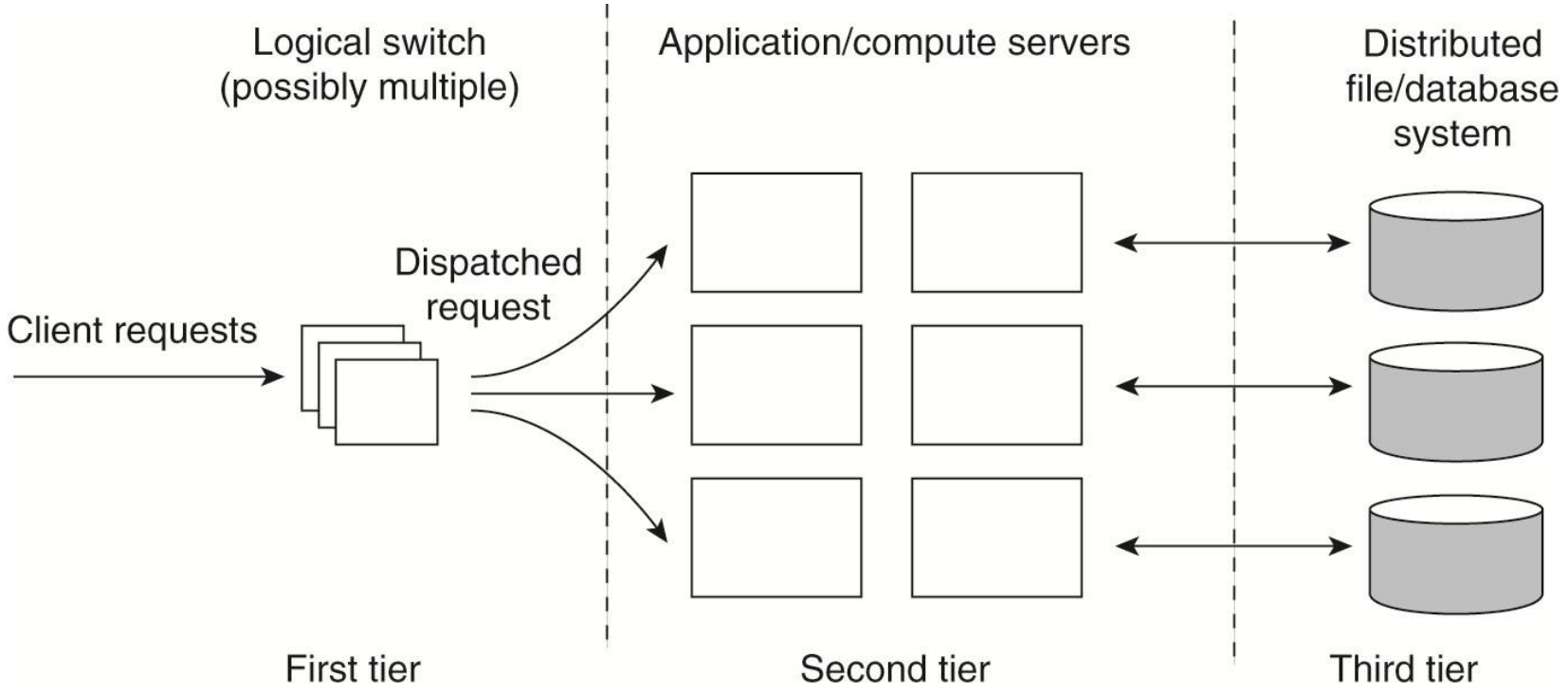
- **The server process or thread for the client is created on demand when the client requests service (connects)**



(b)

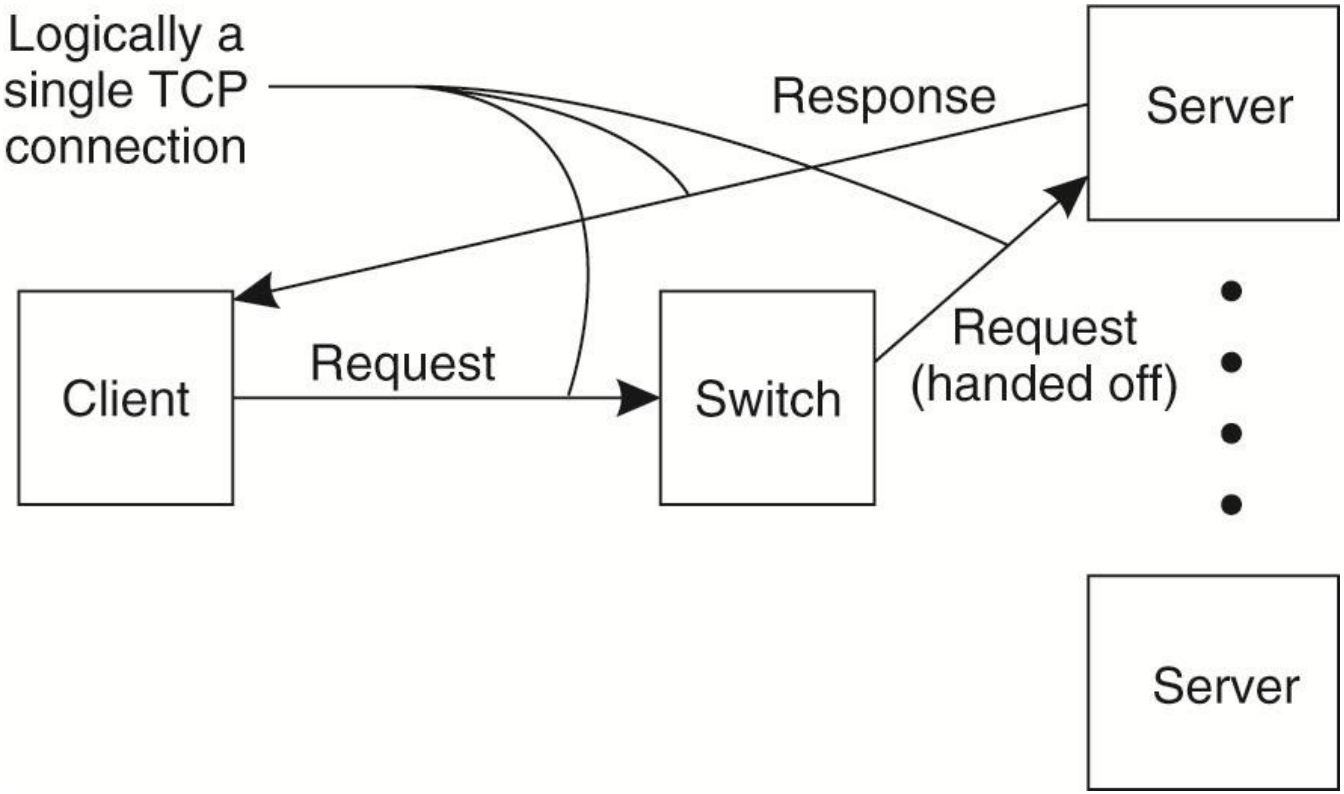
Server Clusters

- Organization of a three-tier server cluster



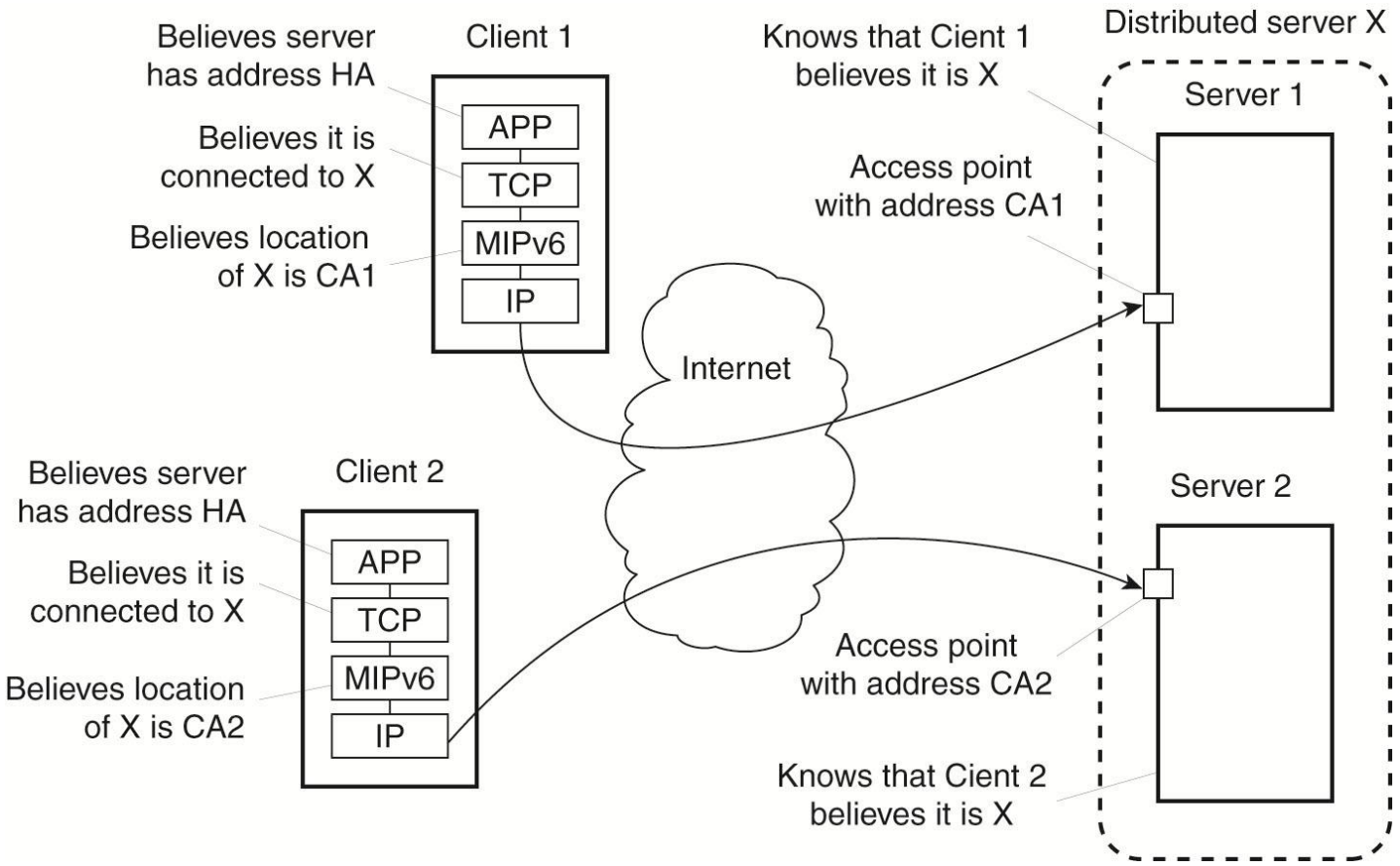
Server Clusters

- The principle of **TCP handoff** by a switch or load balancer to a server



Distributed Servers

- Multiple copies of the same server used by different clients, to optimize distance to the server, load on the server, etc.



Servers and State

- **Stateless server:** Doesn't keep *accurate* information about the status of a client after handling the client's request
  - **Doesn't keep track of its clients**
  - **Doesn't promise to invalidate the data that a client has cached, after the client's request is completed**
  - **Doesn't record whether a file has been opened (simply closes it after access)**
- **Consequences**
  - **Clients and servers are completely independent**
  - **PRO: State inconsistencies, due to client or server crashes, are reduced (ignored)**
  - **CON: Possible loss of performance because, e.g., a server cannot anticipate a client's behavior, e.g., the client might access a file a second time**

Servers and State

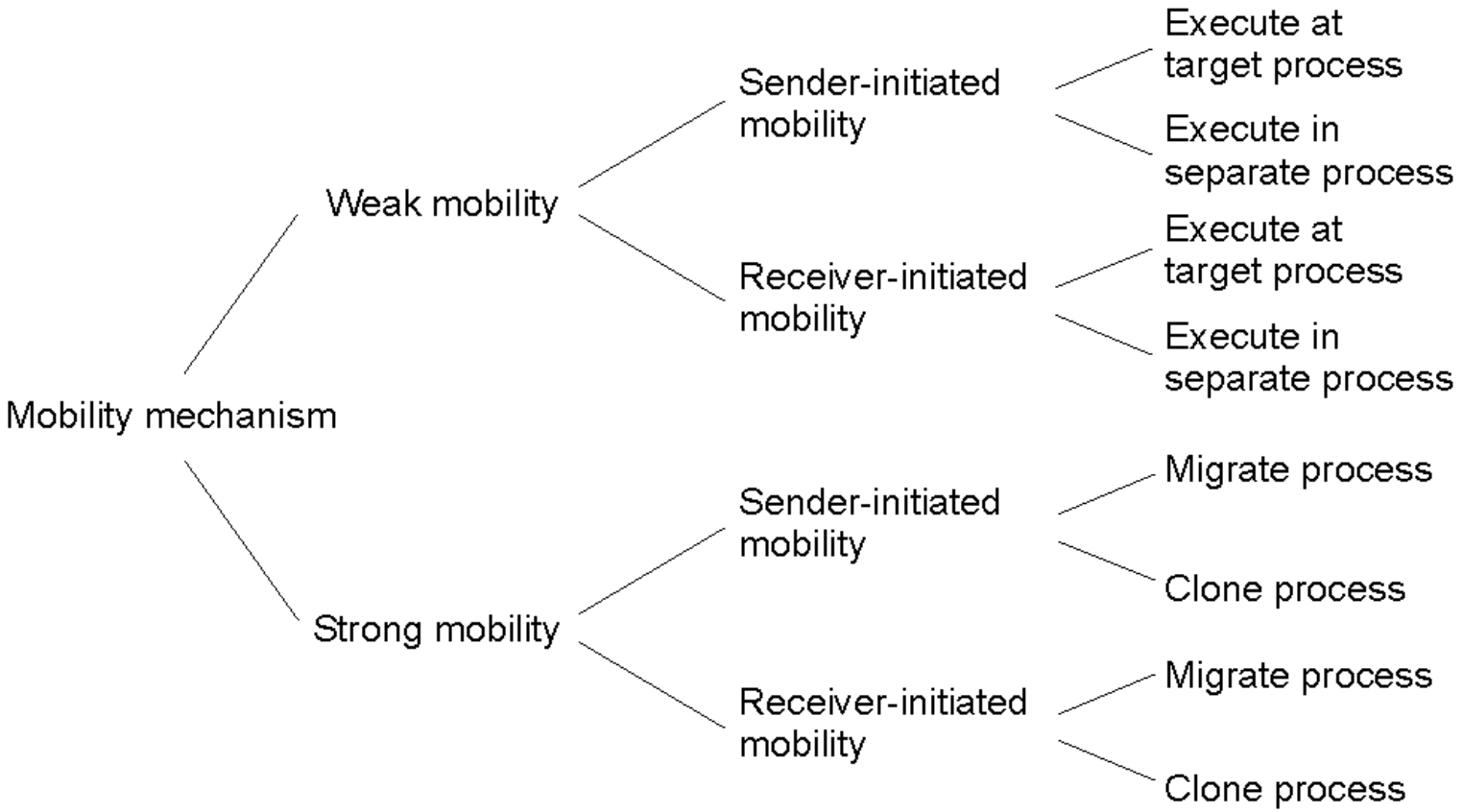
- **Stateful server:** Keeps track of the status of its clients
  - **Knows which data a client has cached, and allows clients to cache local copies of shared data**
  - **Records that a file has been opened, so that prefetching can be done**
- **Consequences**
  - **Clients and servers are coupled together (somewhat)**
  - **PRO: The performance of a stateful server can be extremely high, provided that clients are allowed to cache local copies**
  - **CON: Maintaining consistency between the data at the server and the data that the client has cached can be challenging**

Migration of Objects (Processes)

- **Object components**
  - **Code segment:** **Contains the code of the object**
  - **Data segment:** **Contains the data (state) of the object**
  - **Execution state:** **Contains the context of the thread executing the object's code**
- **Weak mobility:** Move the code segment and the data segment, and start the execution from the beginning after migration
  - **Relatively simple, especially if the code is portable**
  - **Distinguish between code shipping (push) and code fetching (pull)**
- **Strong mobility:** Move all object components, including the execution state
  - **Migration: Move the entire object from one machine to another**
  - **Cloning: Simply start a clone, and set it to the same execution state**

Models for Object Migration

- Alternatives for object migration



Migration and Local Resources

- **Problem:** An object uses local resources that might (or might not) be available at the target site of the object migration
- **Resource types**
  - **Unattached: Resource can easily be migrated along with the object (e.g., contents of a cache)**
  - **Fastened: Resource can, in principle, be migrated, but only at a high cost (e.g., a disk file)**
  - **Fixed: Resource cannot be migrated (e.g., local hardware)**
- **Object-to-resource binding**
  - **By identifier: Object requires a specific instance of a resource (e.g., a specific database)**
  - **By value: Object requires the value of a resource (e.g., the set of cache entries)**
  - **By type: Object requires that only a type of resource is available (e.g., a color monitor)**

Migration and Local Resources

- Actions to be taken (GR, MV, CP, RB), with respect to resources, when migrating objects from one machine to another

Resource types

Object-to-resource binding	Unattached	Fastened	Fixed
By identifier	MV (or GR)	GR (or MV)	GR
By value	CP (or MV, GR)	GR (or CP)	GR
By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

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

Migration in Heterogeneous Systems

- **Main problem**
  - **The target machine might not be suitable for executing the migrated object because**
    - The processor / process / thread context is highly dependent on the local hardware, operating system, and runtime system
- **Only general solution:** Make use of an abstract machine that is implemented on different platforms
- **Current solutions**
  - **Interpreted languages running on a virtual machine, such as Java or scripting languages**
  - **Existing languages that allow migration at specific "transferable" points, such as just before a procedure / function / method call**

Migration in Heterogeneous Systems

- The principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment