DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 3
Processes

*** Modified by Prof. Rivalino Matias, Jr.**

Outline

- Processes
- Threads
- Virtualization
- Organization of Clients & Servers

Processes

- Different types of processes play a crucial role in distributed systems.
- The concepts of processes and threads originate from the field of Operating Systems (*).
  - Understand these concepts is essential to design and program DSs.
- Although processes form a building block in DSs, practice indicates that this level of granularity is not enough.
  - It turns out that having a finer granularity in the form of multiple threads per process makes it much easier to build DSs to get better performance.

(*) See supplemental slides for a detailed discussion on processes (from my OS course).

Threads

- Definition (*)
- Implementation models (*)
  - User-space (M:1), Kernel-space (1:1), and Hybrid (M:N).
- Usage in nondistributed systems
- Usage in distributed systems
  - Multithreaded clients
  - Multithreaded servers

(*) See supplemental slides for a detailed discussion on threads (from my OS course).

Thread Usage in Nondistributed Systems

- The most important benefit is to have independency of the multiple execution contexts inside a process.
  - This becomes evident when using blocking (system) calls.
- Another advantage is to exploit the parallelism when executing a program on a multiprocessor system.
- Multithreading is also useful to save system's resources (e.g., RAM) and computing cycles.
  - E.g., Avoid the IPC overhead caused by multiple cooperating processes, when using multiple cooperating threads.
- Finally, many applications are simply easier to structure as a collection of cooperating threads.
  - IDE, Game, GUI, Text editor, Spreadsheet, etc.

Thread Usage in Nondistributed Systems

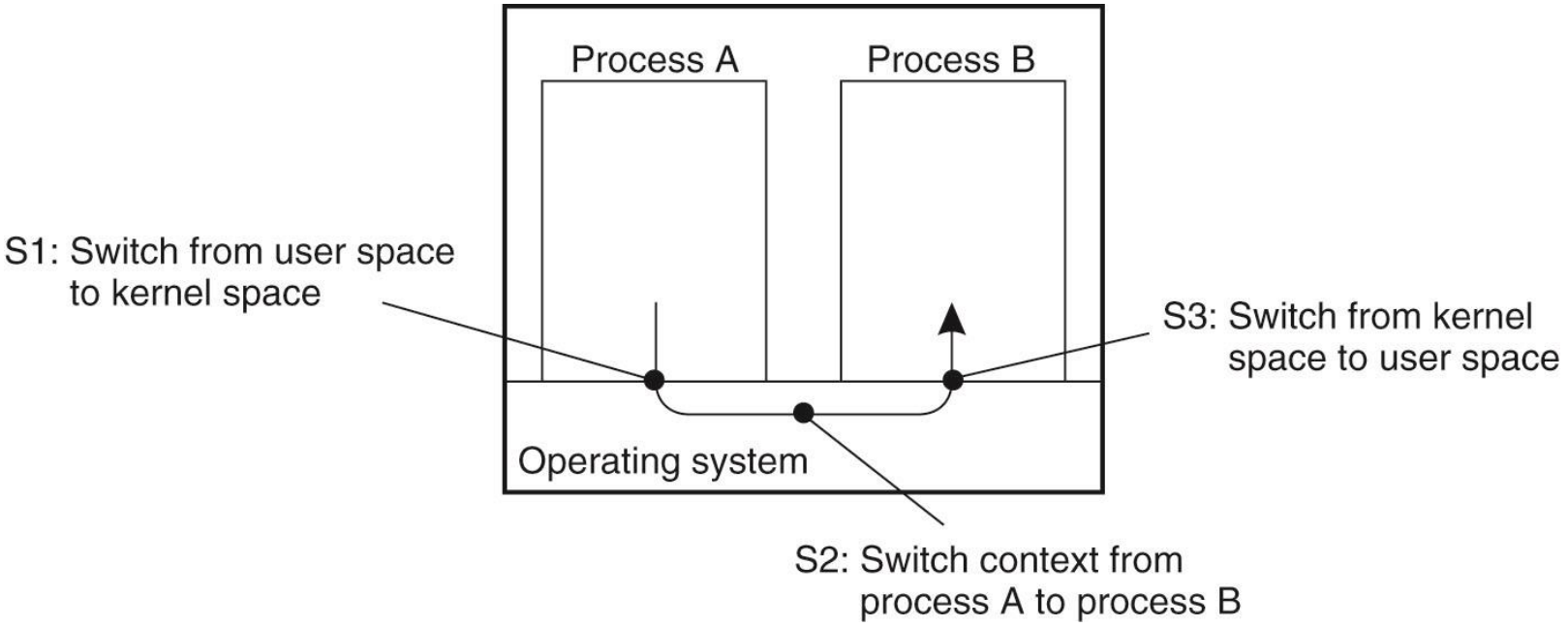

Figure 3-1. Context switching as the result of IPC.

Thread Implementation

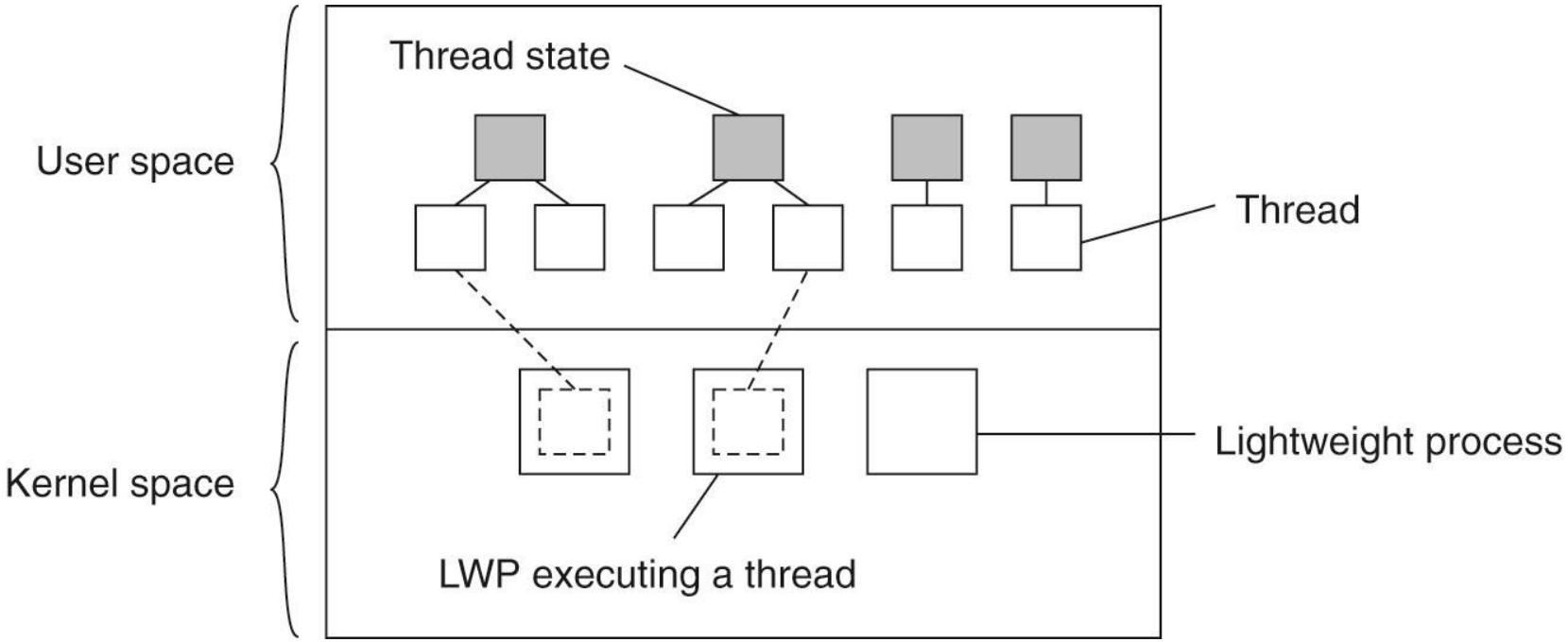

Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

Threads in Distributed Systems (1)

- They can provide a convenient means of allowing blocking system calls without blocking the entire process.
  - This makes them particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time.
- In addition, parallelism can be explored in both client and server programs.

Multithreaded Clients

- To achieve a high degree of distribution transparency, DSs that operate in WANs may need to hide long interprocess message propagation times.
  - The round-trip delay in a WAN can easily be in the order of hundred of milliseconds, or sometimes even seconds.
- The usual way to hide communication latencies is to initiate communication and immediately perform something else.
  - E.g., a typical example is in Web browsers.

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, but also makes it to exploit parallelism to gain high performance.
- To understand the benefits of threads for writing server code, let's to look at the organization of a **file server**.
  - Waits for an incoming request for a file operation.
  - Carries out the request thru interaction with the OS, which most of the time must block waiting for the disk.
  - Once it gets the OS reply then sends it back to the client.

Multithreaded Servers (1)

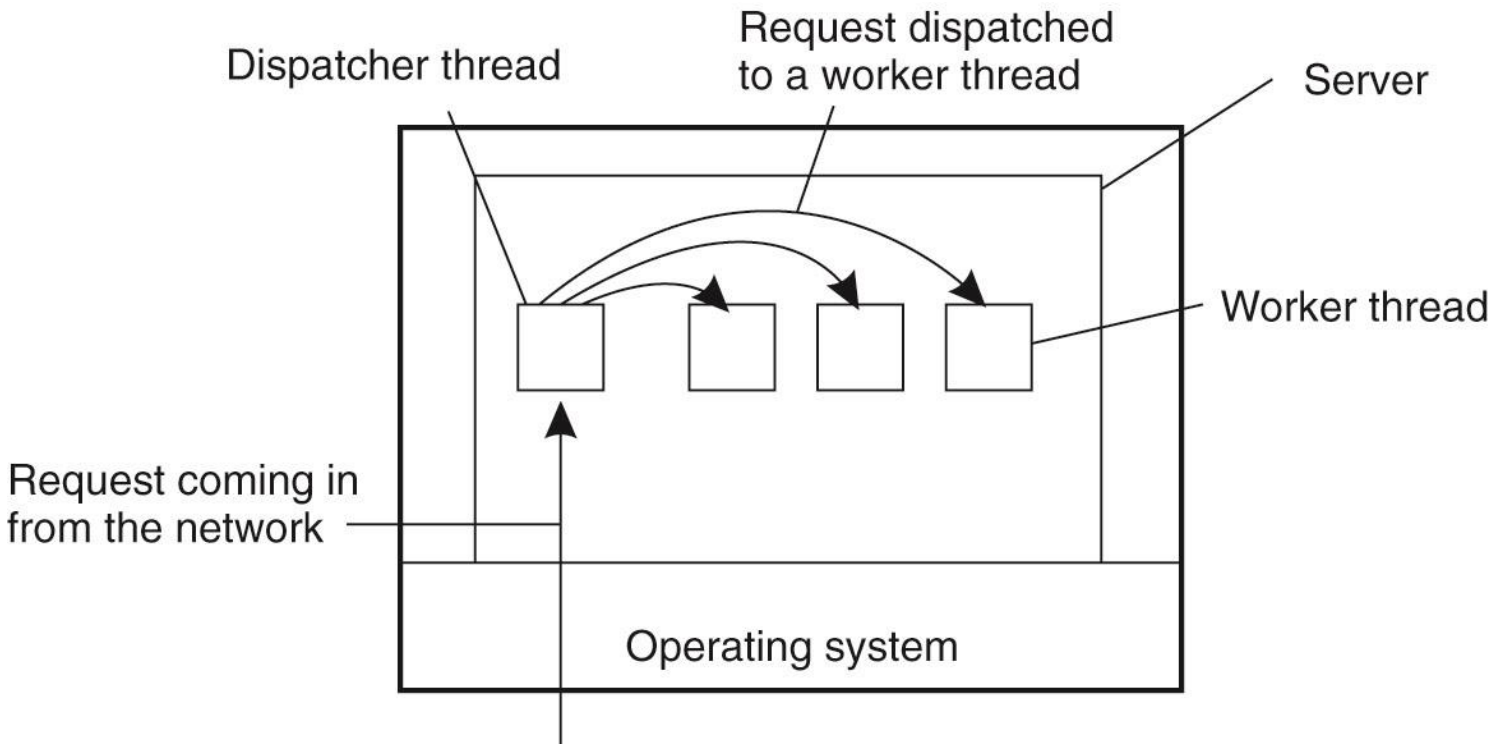

Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

Multithreaded Servers (2)

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

- **Virtualization** exists since 1970; this is not new!
  - The most important reason to be introduced in 70's was to allow legacy software to be executed in new computers (mainframes).
  - Different OS platforms also were an issue, since each computer manufacturer adopted a different proprietary OS platform.
  - As the hardware became cheaper and more standardized, it reduced the need for virtualization (80's and 90's).
- However, virtualization gained again increasingly importance.
  - Hardware changes faster than software.
  - Easy of **portability** and code migration.
  - **Consolidation** of multiple servers (OS and HW).
  - **Isolation** of failing or attacked components.

The Role of Virtualization in Distributed Systems

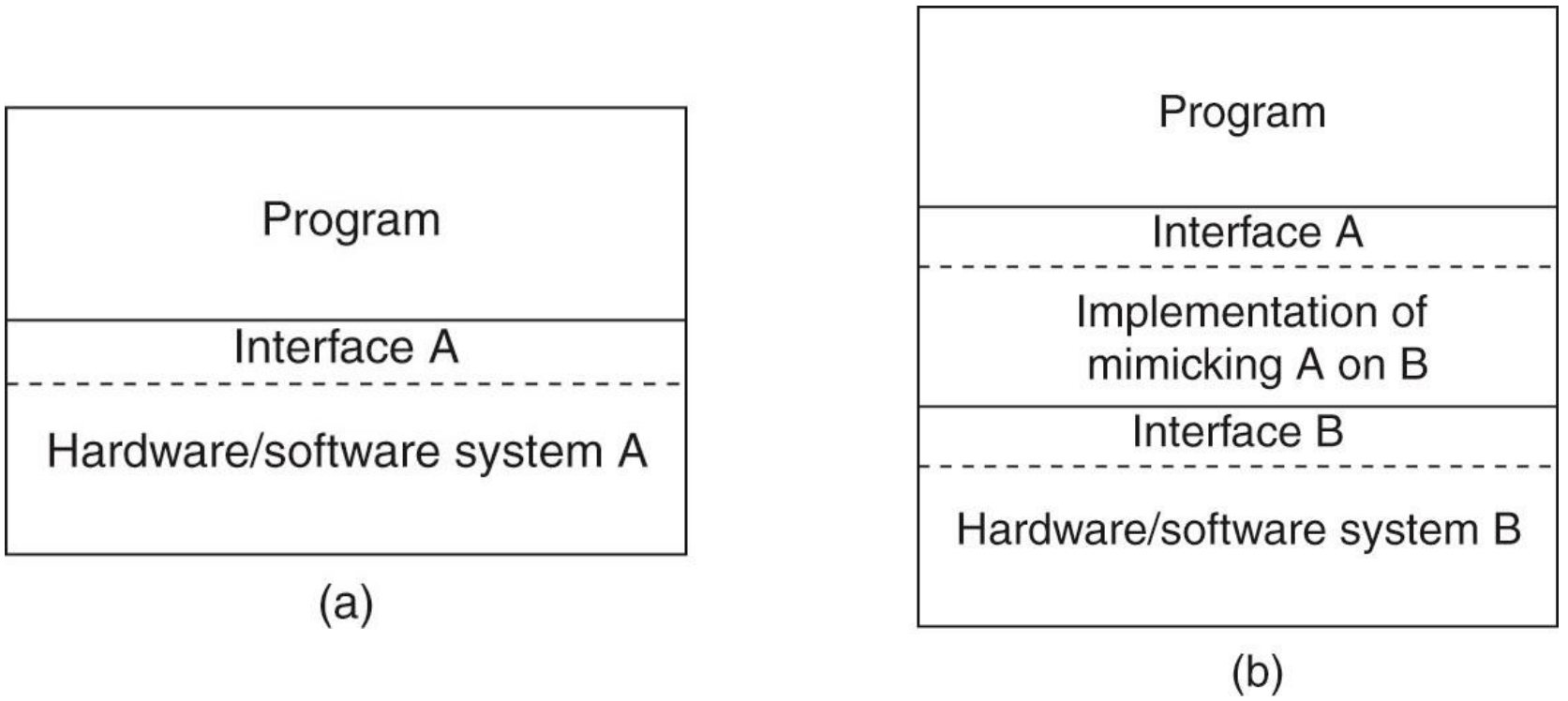

Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

Architectures of Virtual Machines (1)

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

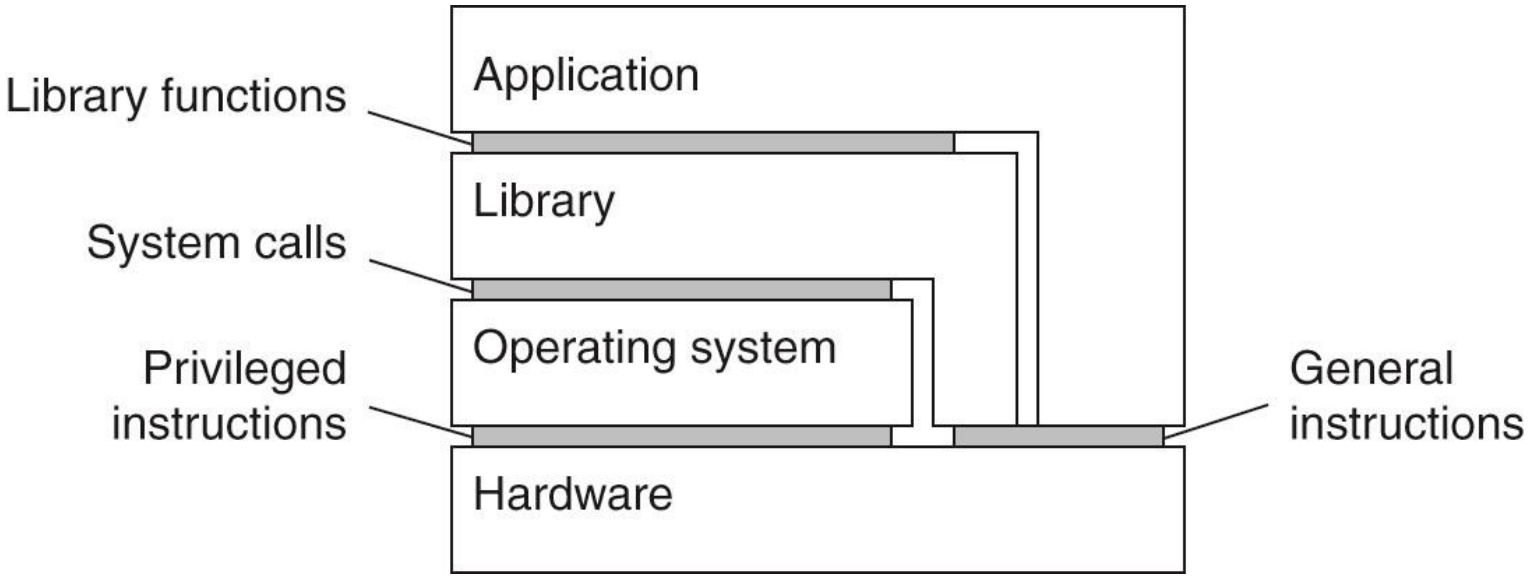

Figure 3-6. Various interfaces offered by computer systems.

Architectures of Virtual Machines (2)

- Basically, there are two types of virtualization:
  - **Process VM**: A program is compiled to intermediate portable code, which is executed by a runtime system (e.g., Java VM).
  - **VM Monitor**: A separate software layer mimics the instruction set of hardware (e.g., VMWare, VirtualBox).

Architectures of Virtual Machines (3)

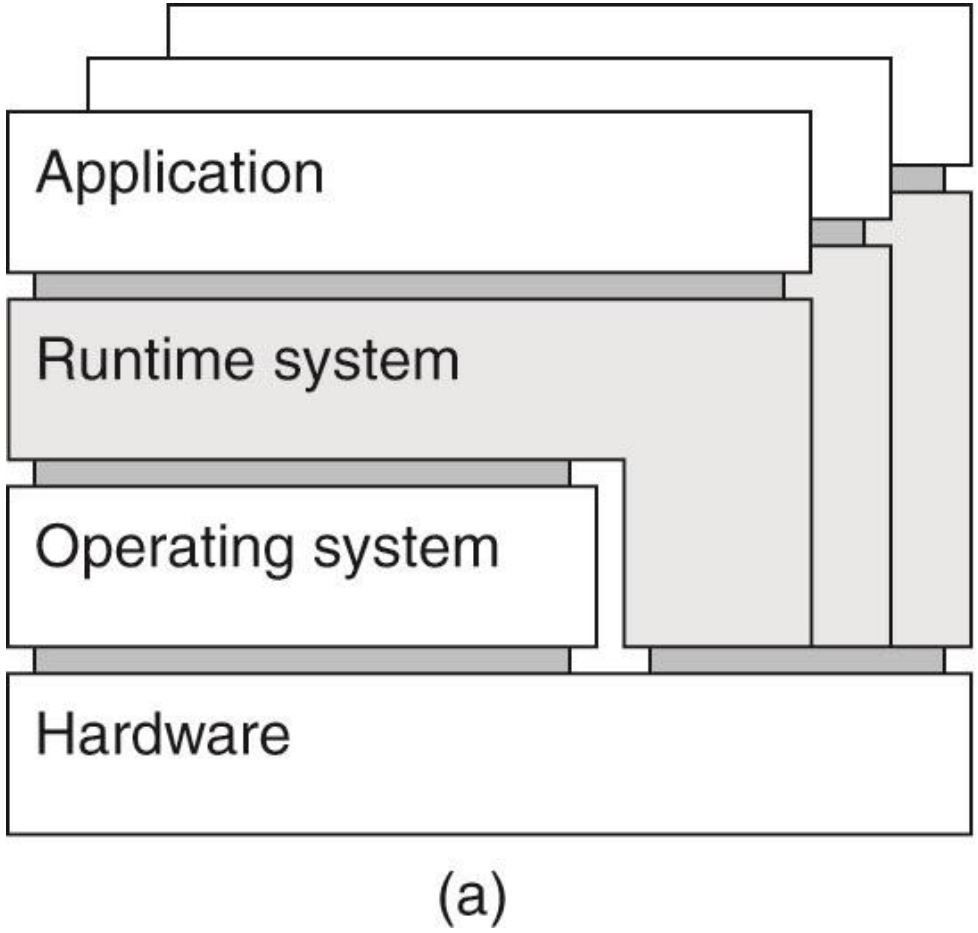

Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations.

Architectures of Virtual Machines (4)

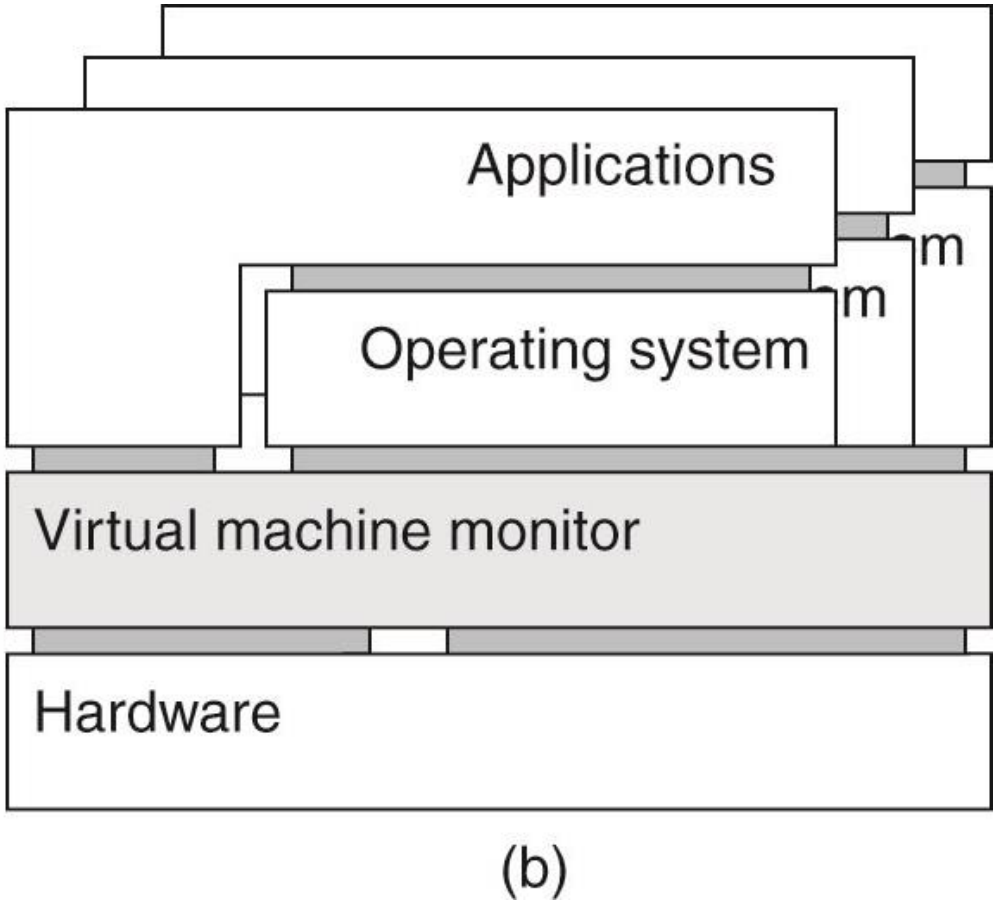

Figure 3-7. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

Client & Server Organization

- A major part of client-side software is focused on (graphical) user interfaces.
  - Mainly two modes of operation are used:
    - For each service requested there is a piece of software on the client side that interacts to the remote service thru a protocol.
    - There is no client-side software related to the remote service. The client simply provide direct access to a remote service behaving as a dumb terminal.
- Thick vs. Thin clients.
  - **Thick (or rich client)**: Applications use local resources (CPU, Memory, Storage) for to perform their jobs.
  - **Thin client**: Applications are executed completely at server side, thus using no resource from the client machine.

Networked User Interfaces (1)

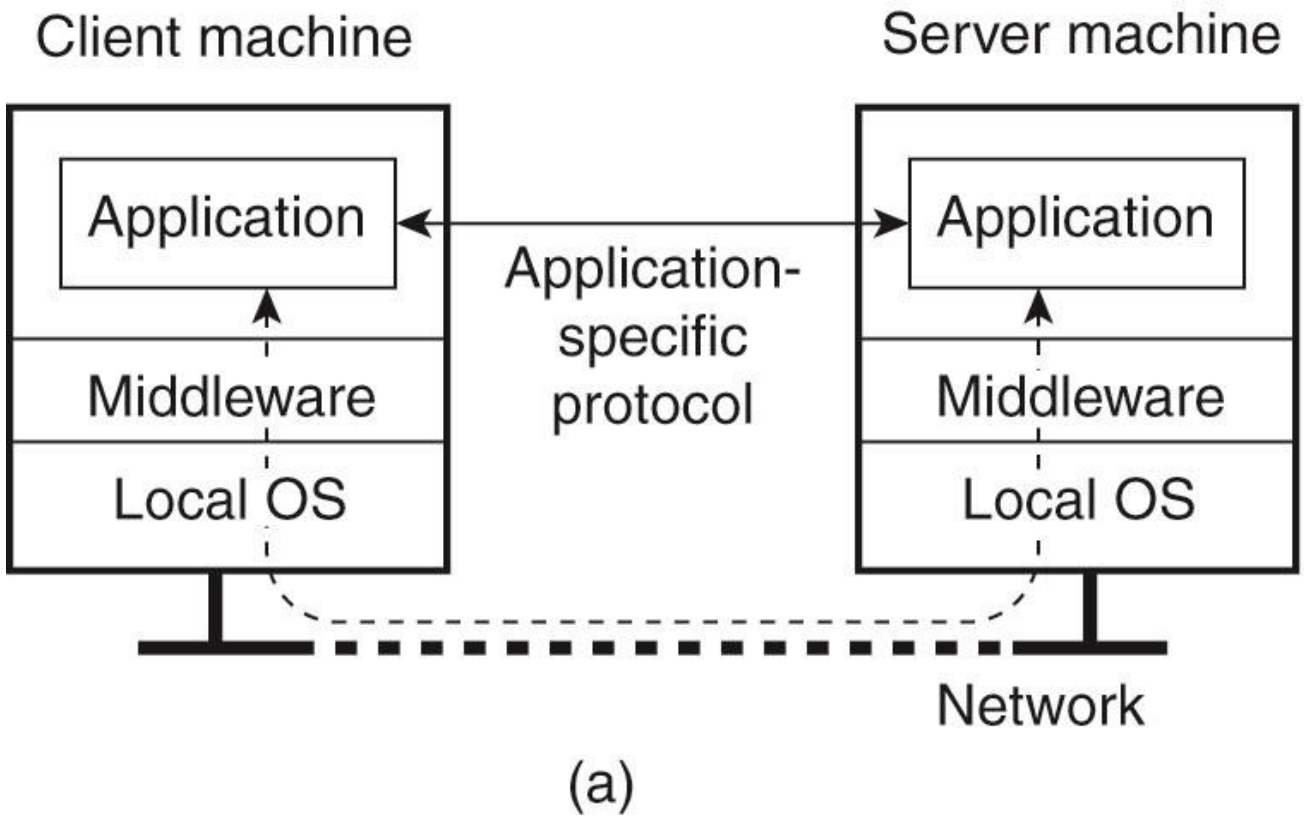

Figure 3-8. (a) A networked application with its own protocol.

Networked User Interfaces (2)

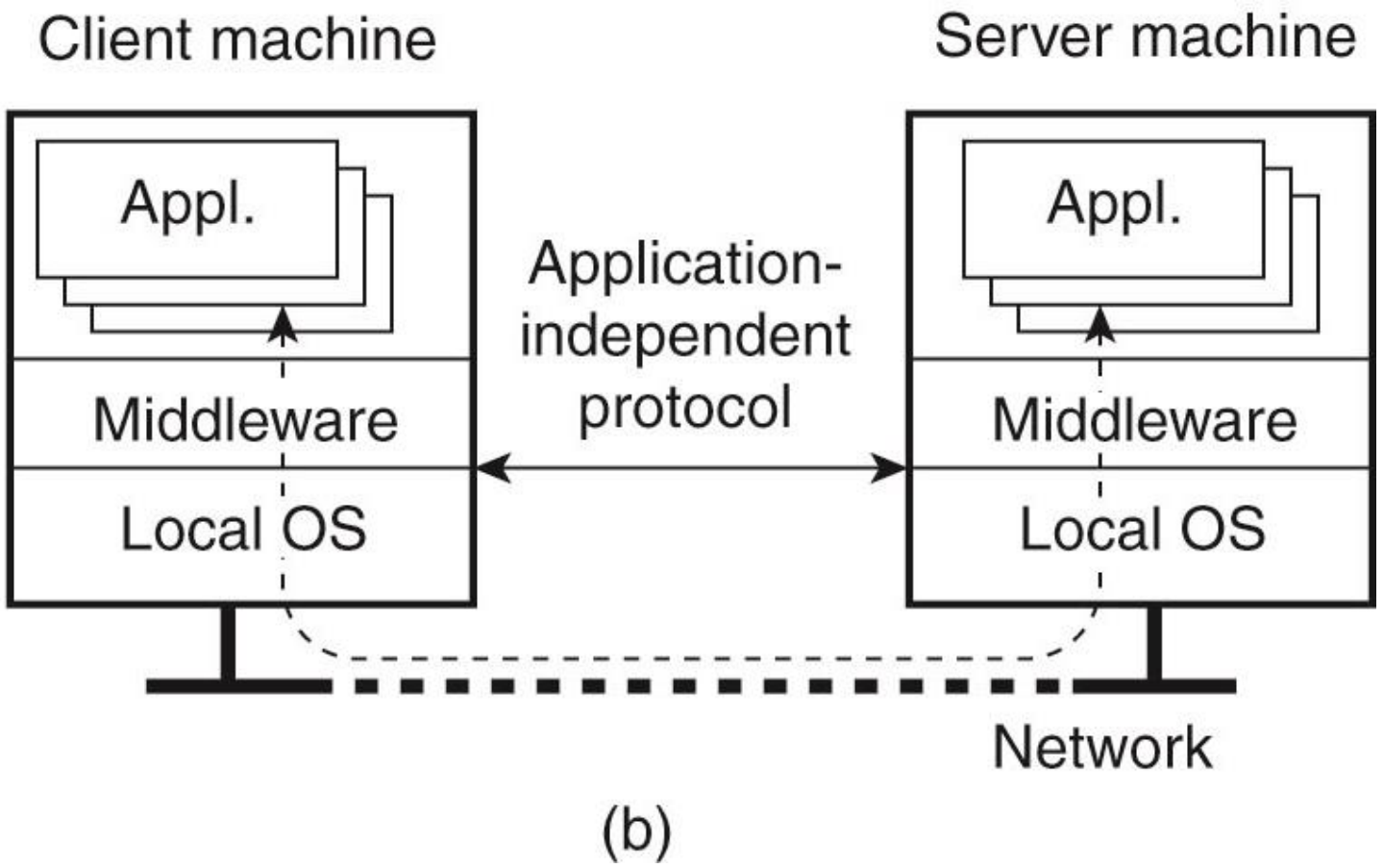

Figure 3-8. (b) A general solution to allow access to remote applications.

Example: The XWindow System

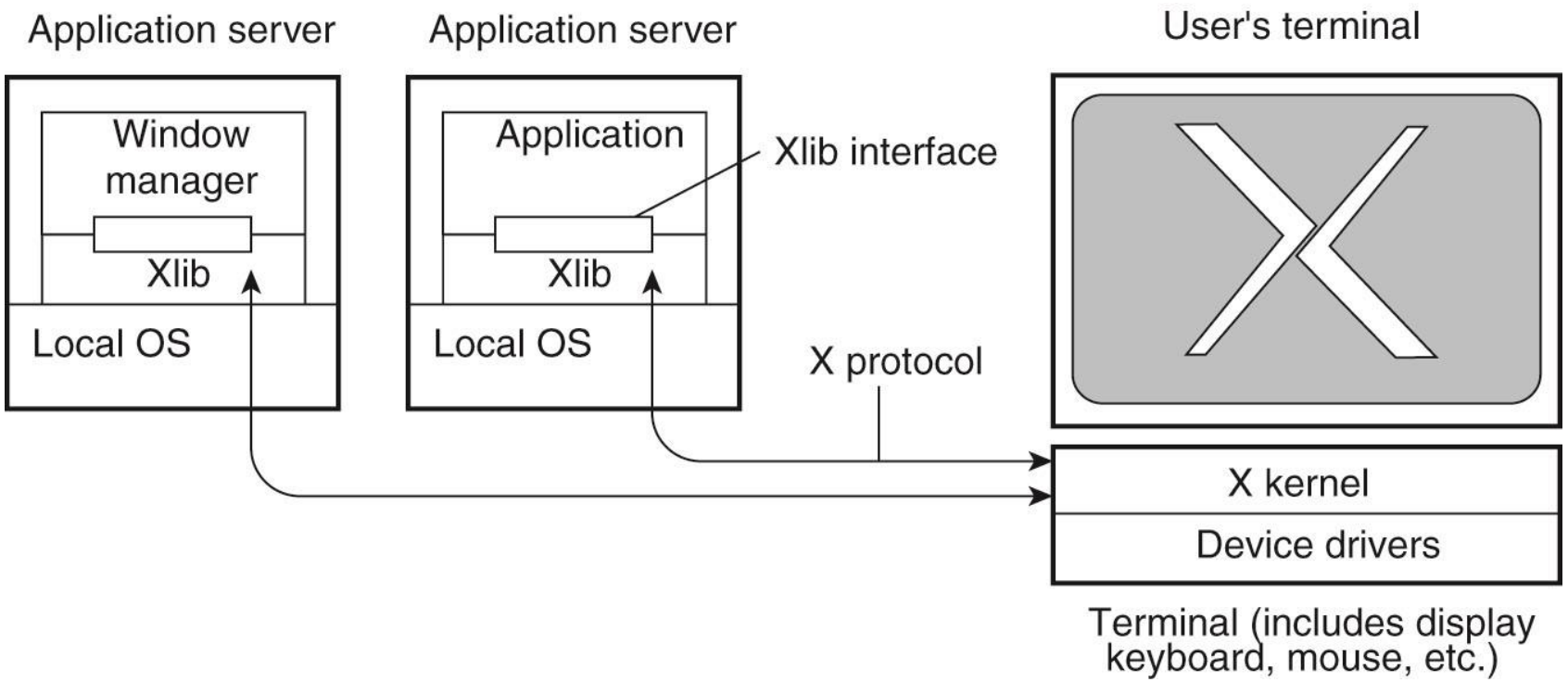

Figure 3-9. The basic organization of theXWindow System.

Client & Server Organization

- Generally tailored for distribution transparency
  - **Access transparency**: e.g., Client-side stubs for RPC.
  - **Location/migration transparency**: let client-side software keep track of actual location.
  - **Replication transparency**: multiple invocations handled by client stub.
  - **Failure transparency**: can often be placed only at client (we're trying to mask server and communication failures).

Client-Side Software for Distribution Transparency

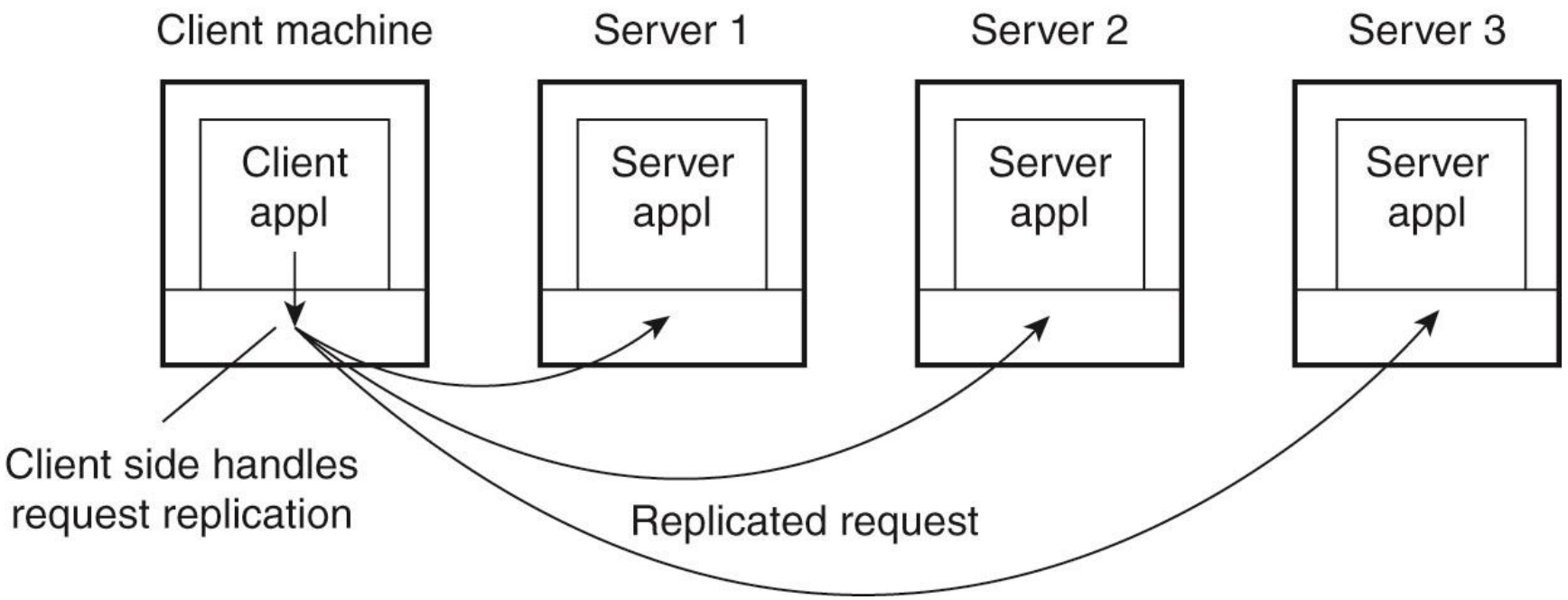

Figure 3-10. Transparent replication of a server using a client-side solution.

Client & Server Organization

- A server is a process implementing a specific service on behalf of a collection of clients.
  - File, Backup, Web, Database (DBMS), Applications, Communication (eMail, IM, Chat, …), Security (Antivirus/IDS/ Firewall/VPN), Naming/Directory, Time, Installation/Updates, …
- In general, each server is organized similarly:
  - It waits for an incoming request from a client and ensures that the request is taken care of, after which it waits for the next request.
- Generally, there are two types of servers:
  - **Iterative** vs. **Concurrent** servers:
    - Iterative servers can handle only one client at a time, in contrast to concurrent servers.

Client & Server Organization

- Contacting a server: **end points**
  - An important server design aspect is where clients contact the server.
    - In all cases, clients send requests to an **end point**, also called a **port**, at the machine where the server is running.
  - How do clients know the **end point** of a service?
    - One approach is to globally assign end points for well-known services, such as FTP (TCP port 21) and HTTP (TCP port 80).
    - These end points have been assigned by the Internet Assigned Numbers Authority (IANA) and are documented.
    - With assigned end points, the client needs to find only the network address of the machine where the server is running.
      - Name services (e.g., DNS) can be used for that purpose.

Client & Server Organization

- There are services that do not require a preassigned end point.
  - For example, a time-of-day server may use an **end point** that is dynamically assigned to it by its local operating system.
  - In that case, a client will first have to look up the end point.
- One solution is to have a **special process** (*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 the next slide.

General Design Issues (1)

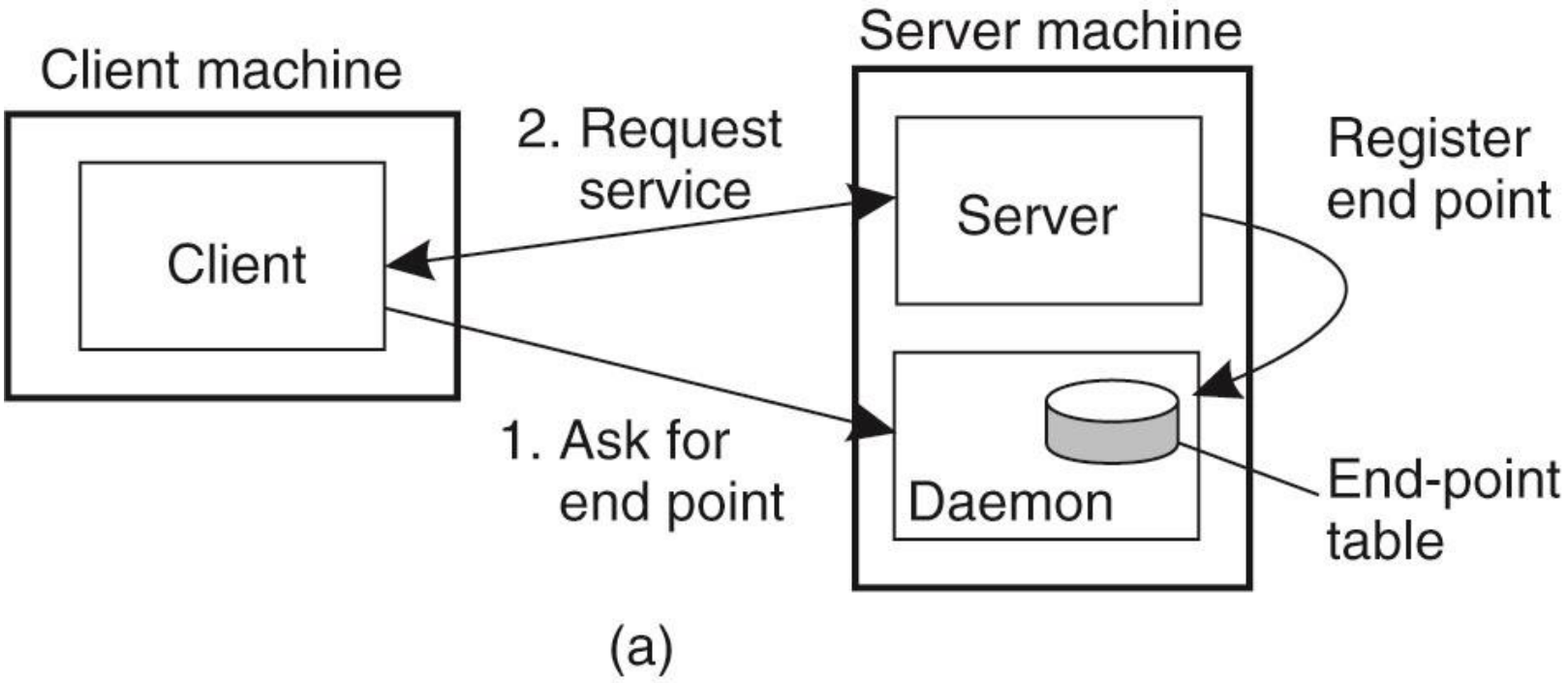

Figure 3-11. (a) Client-to-server binding using a daemon.

Client & Server Organization

- Another solution is instead of having to keep track of so many passive processes, it is often more efficient to have a single superserver.
  - This **superserver** listens to each end point associated with a specific service.
  - For example, the **inetd** daemon in Unix systems.
    - It listens to a number of well-known ports for Internet services.
    - When a request comes in, the daemon forks a process to handle it. That process will exit when finished.

General Design Issues (2)

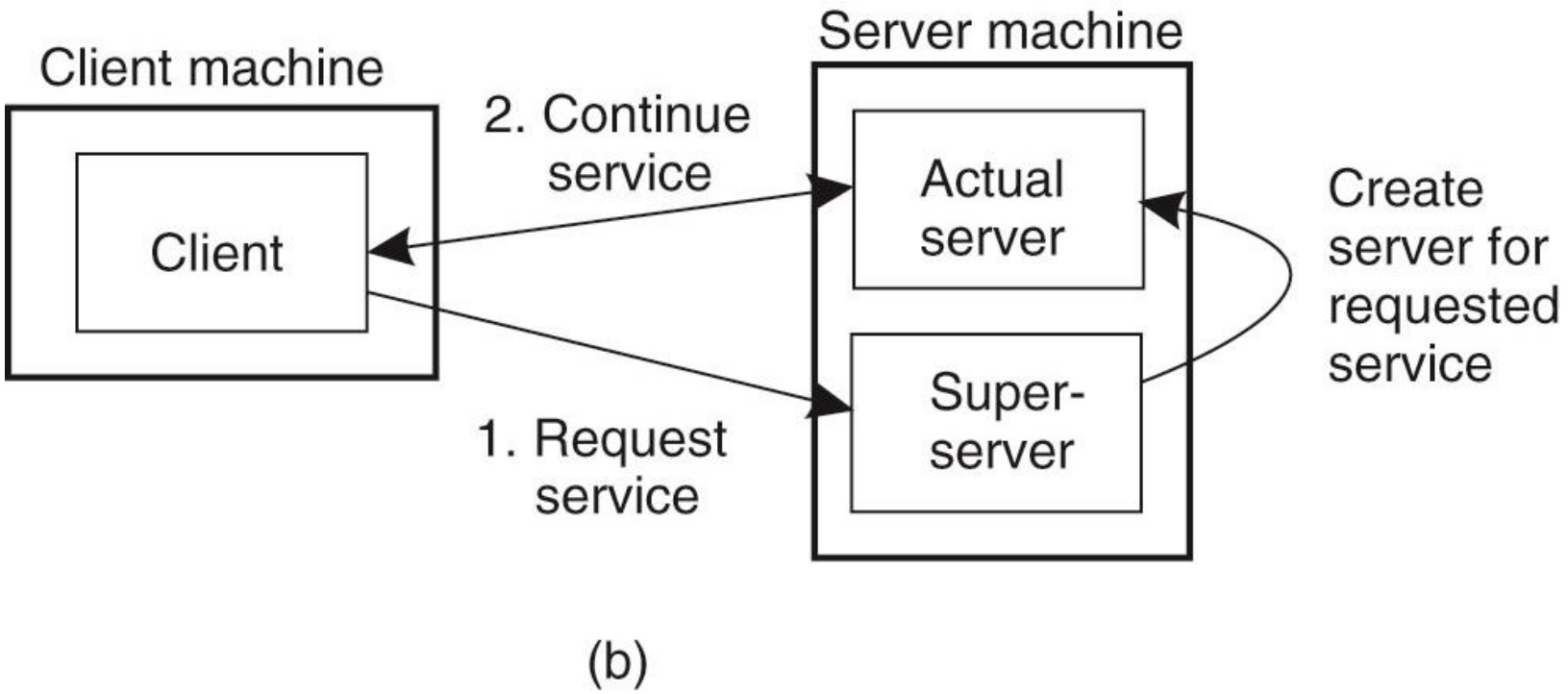

Figure 3-11. (b) Client-to-server binding using a superserver.

Client & Server Organization

- Interrupting a server.
  - Another issue that needs to be taken into account when designing a server is whether and how a server can be interrupted.
    - E.g., consider a user who decided to upload a huge file to an FTP server. Then, realizing that it is the wrong file, s/he wants to interrupt the server to cancel further data transmission.
  - **There are several ways to do this**:
    - One approach that works well is to abruptly exit the client (which will automatically break the connection to the server).
      - The server will eventually tear down the old connection, thinking the client has probably crashed.

Client & Server Organization

- Interrupting a server (cont'd).
  - Another approach is to develop the client and server such that it is possible to send **out-of-band** data.
    - **OOB** data is a type of data that is to be processed by the server before any other data from that client.
  - One solution is to let the server listen to a separate control end point to which the client sends OOB data, while listening (lower priority) to the end point through which the normal data passes.
  - Another solution is to send OOB data across the same connection through which the client is sending the original request.
    - In TCP, it is possible to transmit **urgent data**.
    - When **urgent data** are received at the server, the latter is interrupted, after which it can inspect the data and handle them accordingly.

Client & Server Organization

- Stateless vs. Stateful servers
  - Another important server design is whether or not the server is **stateless**.
    - 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.
    - A server processes requests based solely on information provided with each request and does not rely on information from earlier requests. The server does not need to maintain state information between requests.
    - For example, browsing static Web pages in a Web server.
  - In many stateless designs, the server does maintain information on its clients, but crucial is the fact that if this information is lost, it will not lead to a disruption of the service offered by the server.
  - Some systems keep the so-called **soft state**
    - This is a temporary state that is valid only for a limited time.

Client & Server Organization

- Stateless vs. Stateful servers
  - A **stateful server** is one that processes requests based on both the information provided with each request and information stored from earlier requests.
  - For stateless interactions, it does not matter whether different requests are processed by different servers.
  - However, for **stateful interactions**, the server that processes a request needs access to the state information necessary to service that request.
    - Either the same server can process all requests that are associated with the same state information, or the state information can be shared by all servers that require it.
      - In the latter case, accessing the shared state information from the same server minimizes the processing overhead associated with accessing the shared state information from multiple servers.

Server Clusters (1)

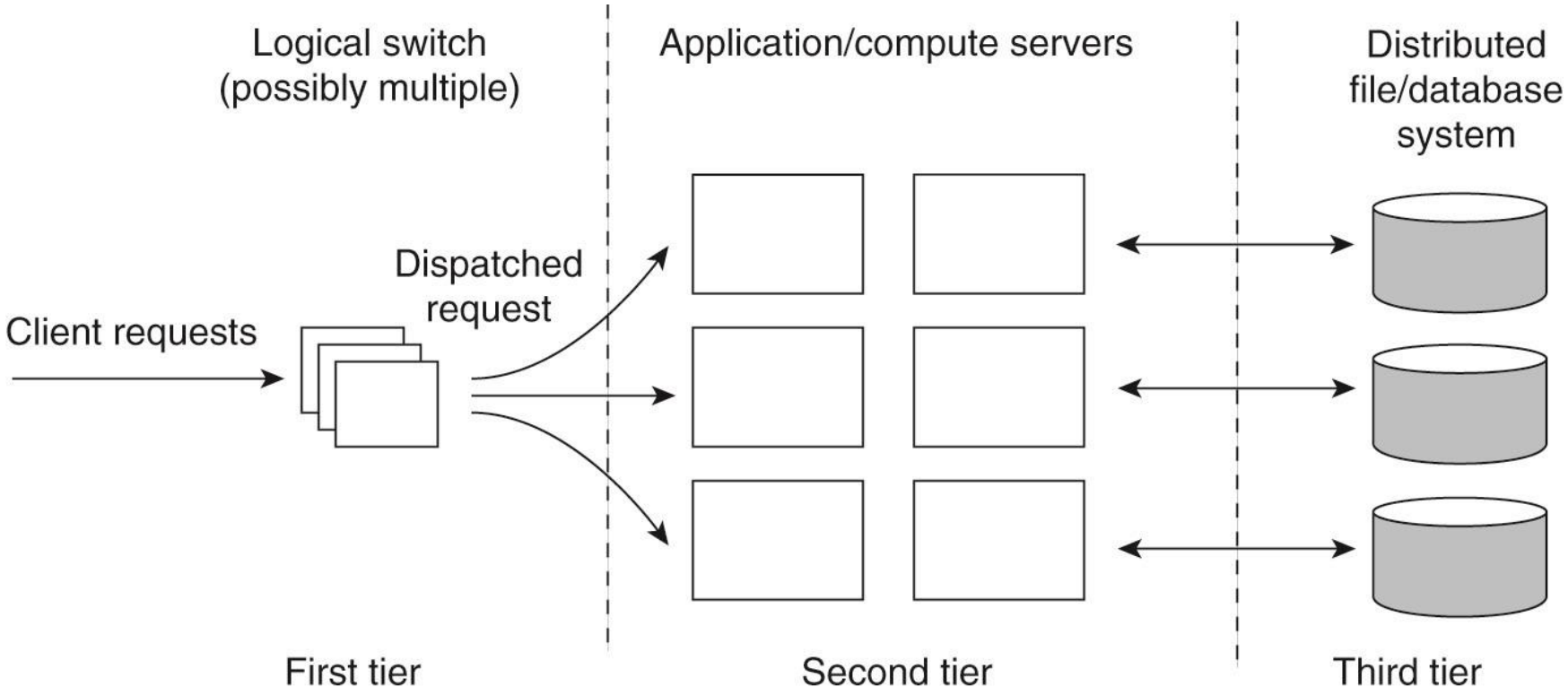

Figure 3-12. The general organization of a three-tiered server cluster.

Server Clusters (2)

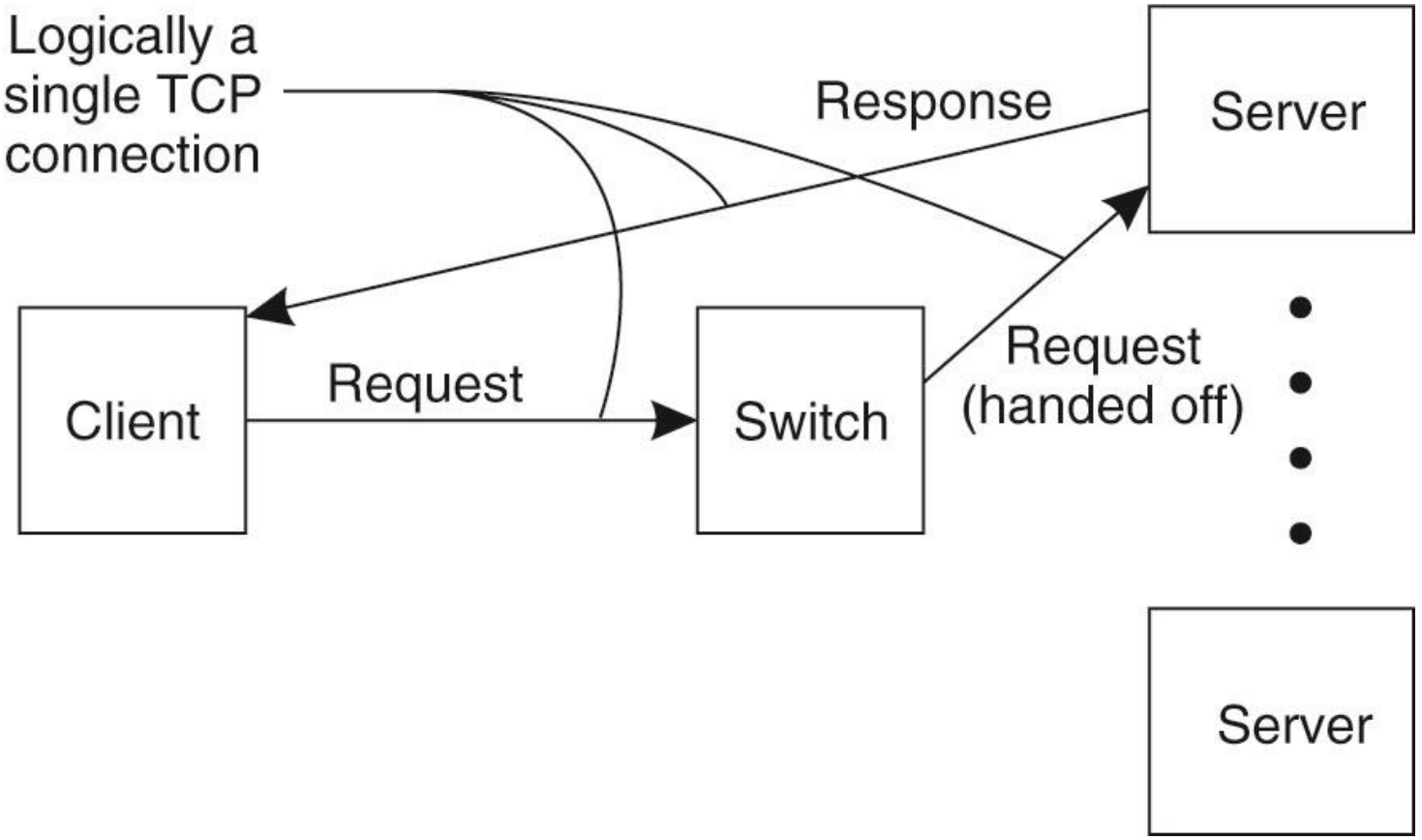

Figure 3-13. The principle of TCP handoff.

Distributed Servers

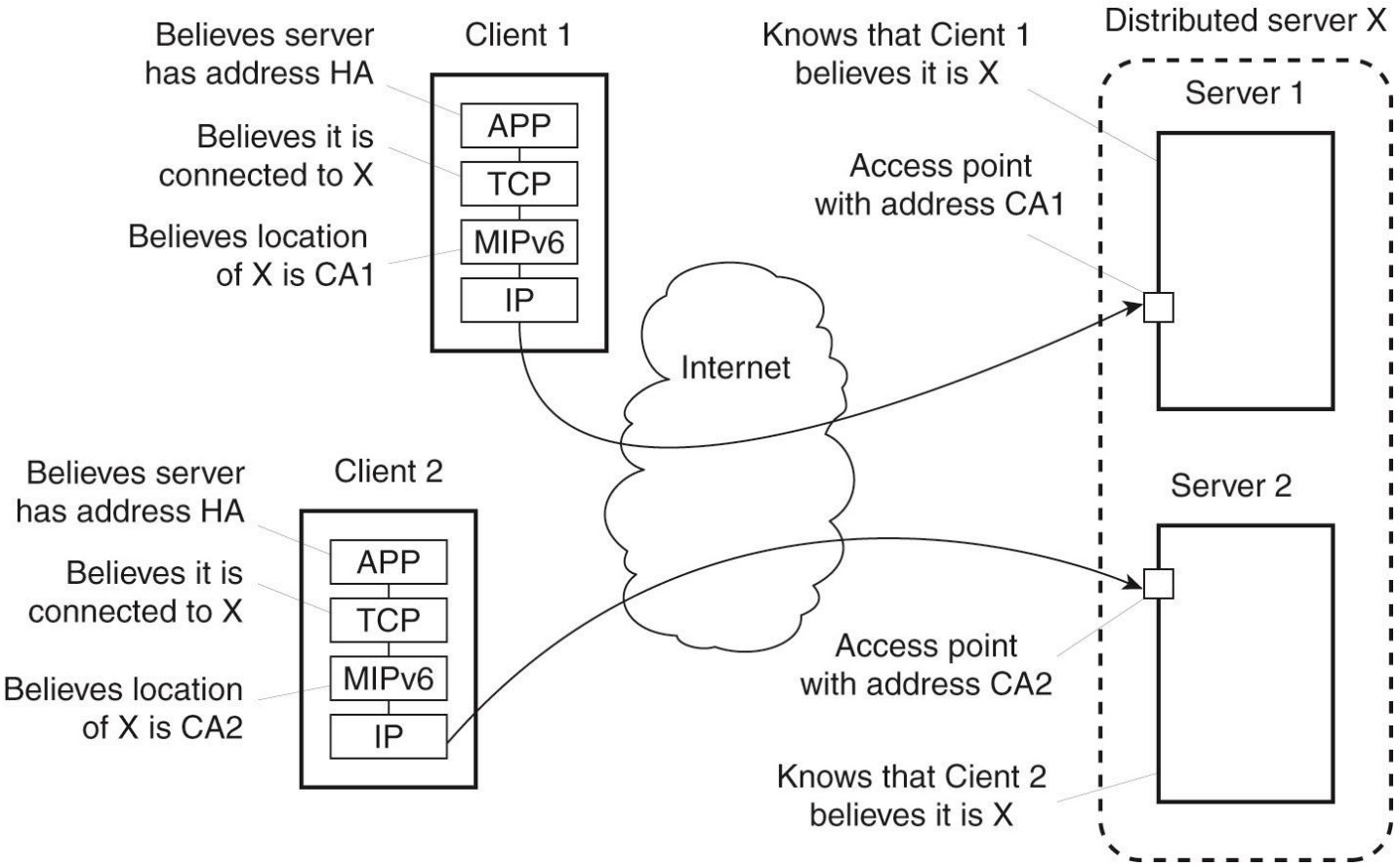

Figure 3-14. Route optimization in a distributed server.