Distributed Systems

Mahmoud Abou El-Magd Soliman
Department of Computer Science
Faculty of Computers and Artificial Intelligence
Sohag University

Chapter Three

Processes

Introduction

In this chapter, we take a closer look at how the different types of processes play a crucial role in distributed systems. The concept of a process originates from the field of operating systems where it is generally defined as a program in execution. From an operating system perspective, the management and scheduling of processes are perhaps the most important issues to deal with. However, when it comes to distributed systems, other issues tum out to be equally or more important.

Introduction

For example, to efficiently organize client-server systems, it is often convenient to make use of multithreading techniques. A main contribution of threads in distributed systems is that they allow clients and servers to be constructed such that communication and local processing can overlap, resulting in a high level of performance.

Introduction to 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.

Introduction to Threads

We build virtual processors in software, on top of physical processors:

Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.

Thread: A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.

Process: A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

Context Switching

- Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
- Thread context: The minimal collection of values stored in registers and memory, used for the execution of a series of instructions (i.e., processor context, state).
- Process context: The minimal collection of values stored in registers and memory, used for the execution of a thread (i.e., thread context, but now also at least MMU register values).

Context Switching

Observations

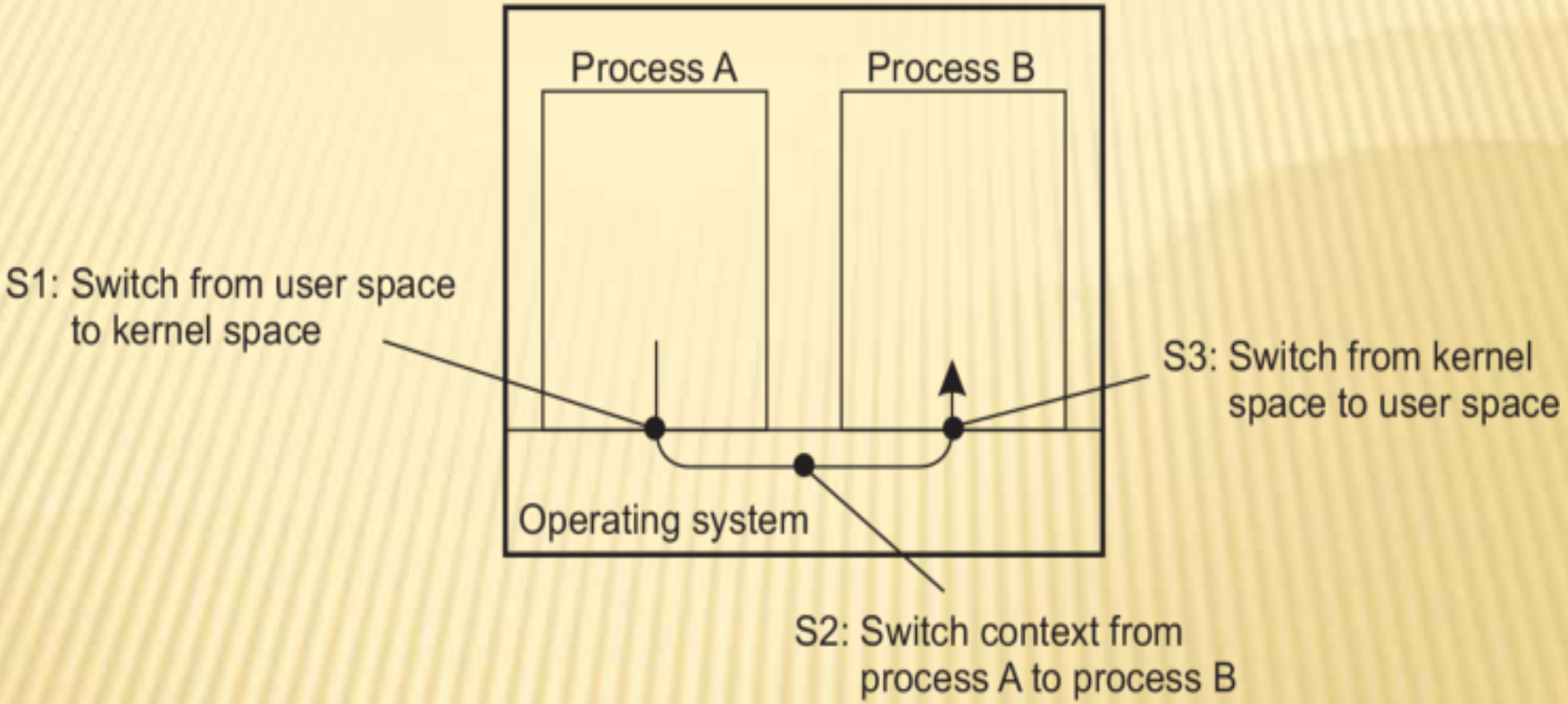
- Threads share the same address space. Thread context switching can be done entirely independent of the operating system.
- Process switching is generally (somewhat) more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.
- Creating and destroying threads is much cheaper than doing so for processes.

Why Use Threads

Some simple reasons

- Avoid needless blocking: a single-threaded process will block when doing I/O; in a multi-threaded process, the operating system can switch the CPU to another thread in that process.
- Exploit parallelism: the threads in a multi-threaded process can be scheduled to run in parallel on a multiprocessor or multicore processor.
- Avoid process switching: structure large applications not as a collection of processes, but through multiple threads.

Avoid Process Switching



Trade-offs

- Threads use the same address space.
- Thread context switching may be faster than process context

Threads in Distributed Systems

An important property of threads is that they can provide a convenient means of allowing blocking system calls without blocking the entire process in which the thread is running. This property makes threads 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. We illustrate this point by taking a closer look at multithreaded clients and servers, respectively.

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. In many cases, a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc. To fetch each element of a Web document, the browser has to set up a TCP/IP connection, read the incoming data, and pass it to a display component.

Using Threads at the Client Side

Multithreaded web client

- Web browser scans an incoming HTML page, and finds that more files need to be fetched.
- Each file is fetched by a separate thread
- As files come in, the browser displays them.

Using Threads at the Client Side

Multiple request-response calls to other machines (RPC):

- A client does several calls at the same time, each one by a different thread.
- It then waits until all results have been returned.
- Note: if calls are to different servers, we may have a linear speed-up.

Multithreaded Clients

There is an important benefit to using multithreaded Web browsers in which several connections can be opened simultaneously. In the previous example, several connections were set up to the same server. If that server is heavily loaded, or just plain slow, no real performance improvements will be noticed.

Multithreaded Clients

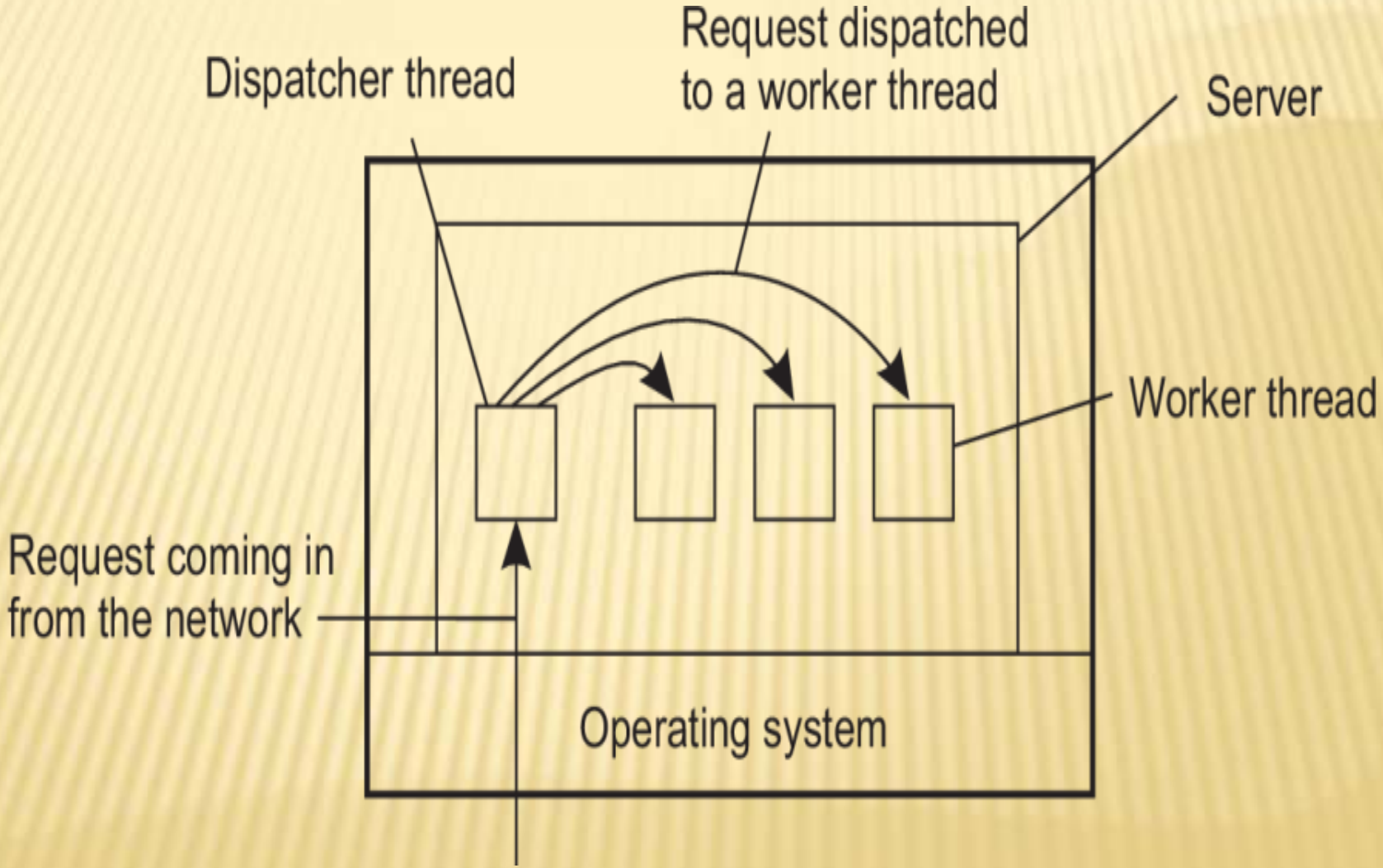
When using a multithreaded client, connections may be set up to different replicas, allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a nonreplicated server. This approach is possible only if the client can handle truly parallel streams of incoming data. Threads are ideal for this purpose.

Multithreaded Servers

Improve performance

- Starting a thread is cheaper than starting a new process.
- Having a single-threaded server prohibits simple scale-up to a multiprocessor system.
- As with clients: hide network latency by reacting to next request while previous one is being replied.
- Multithreaded programs tend to be smaller and easier to understand

Why Multithreading is Popular



Dispatcher/Worker Model

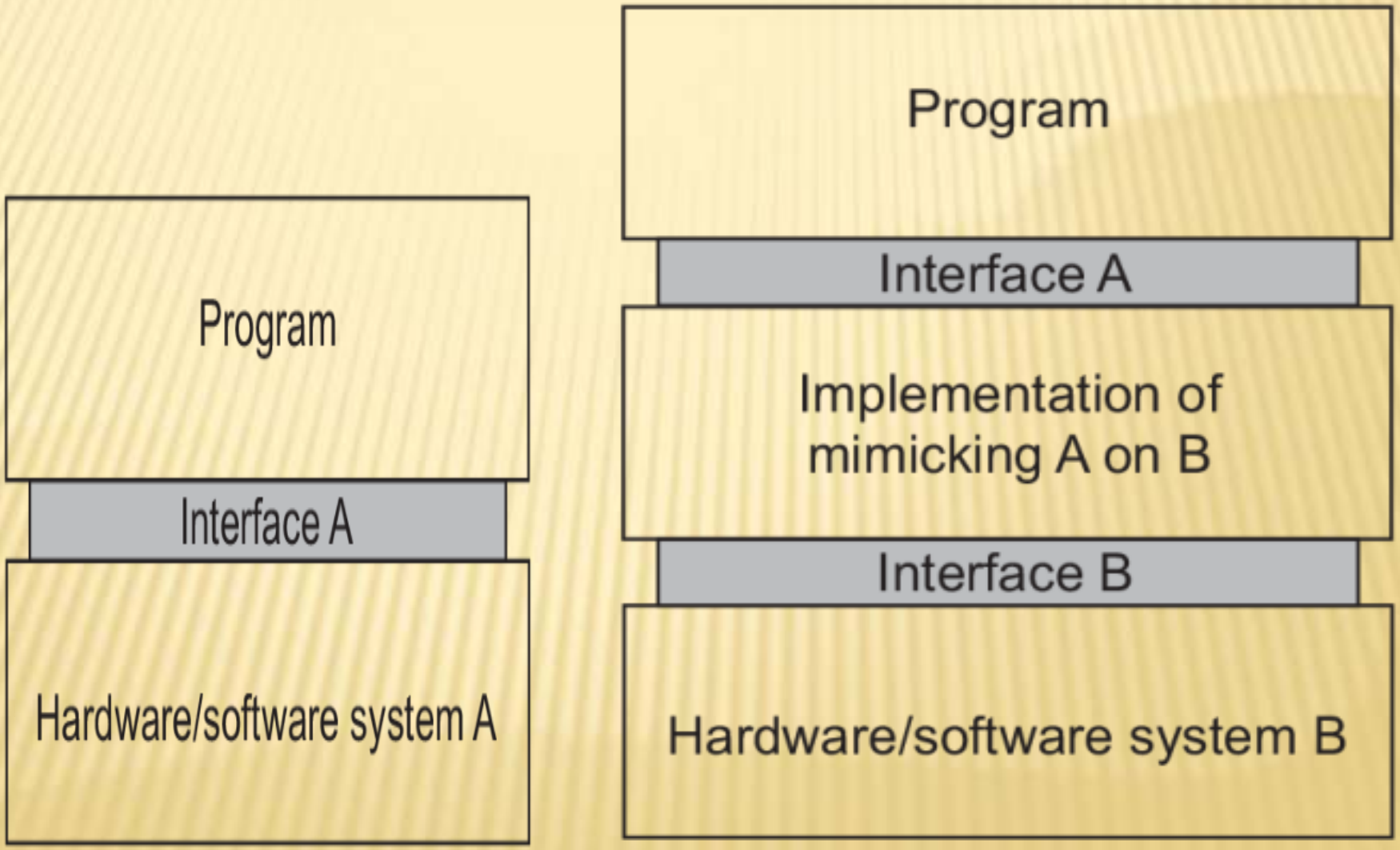
Virtualization

Virtualization is important:

- Hardware changes faster than software
- Ease of portability and code migration
- Isolation of failing or attacked components

Principle: Mimicking Interfaces

Virtualization



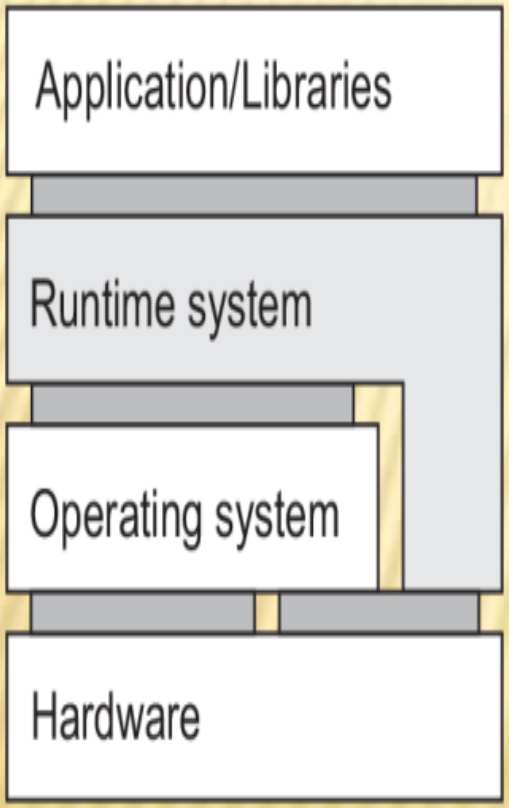
Mimicking Interfaces

Four types of interfaces at three different levels

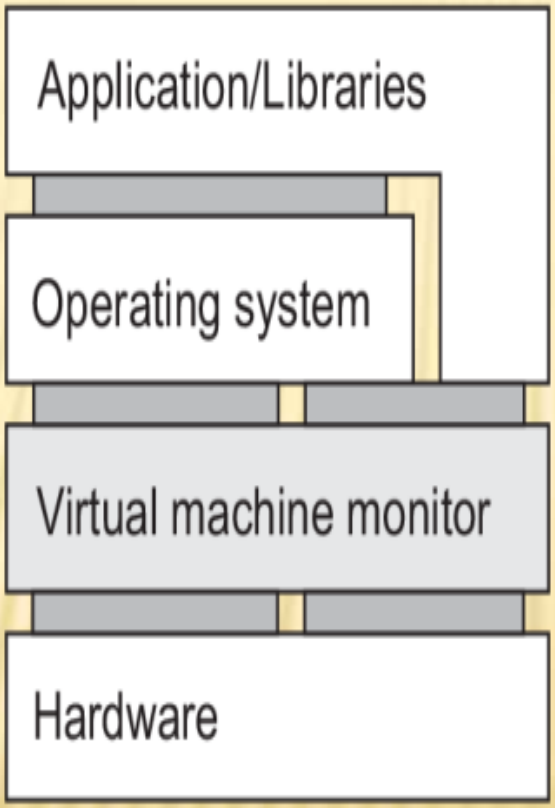
1) **Instruction set architecture:** the set of machine instructions, with two subsets:
   - **Privileged instructions:** allowed to be executed only by the operating system.
   - **General instructions:** can be executed by any program.
2) **System calls** as offered by an operating system.
3) **Library calls,** known as an application programming interface (**API**)

Ways of virtualization

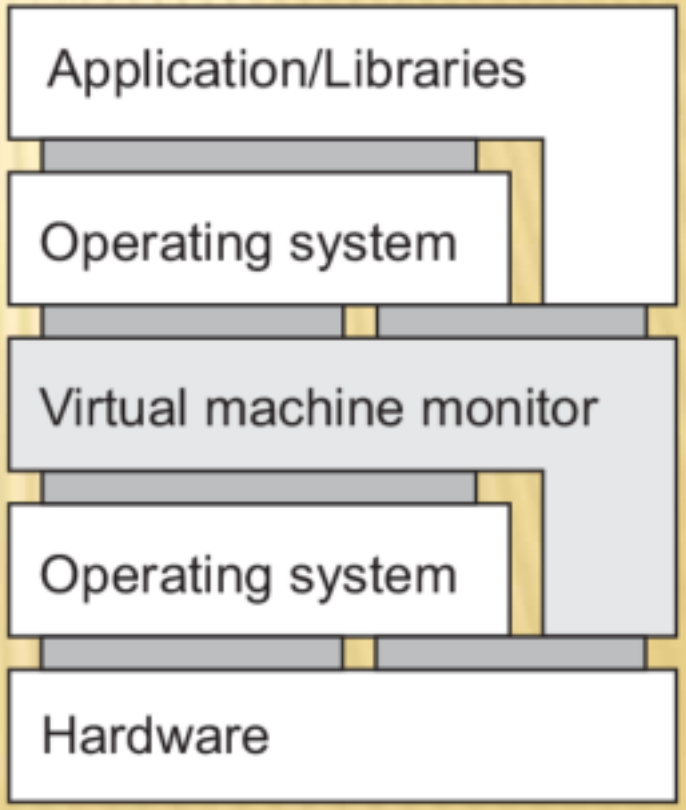
(a) Process VM, (b) Native VMM, (c) Hosted VMM



(a)



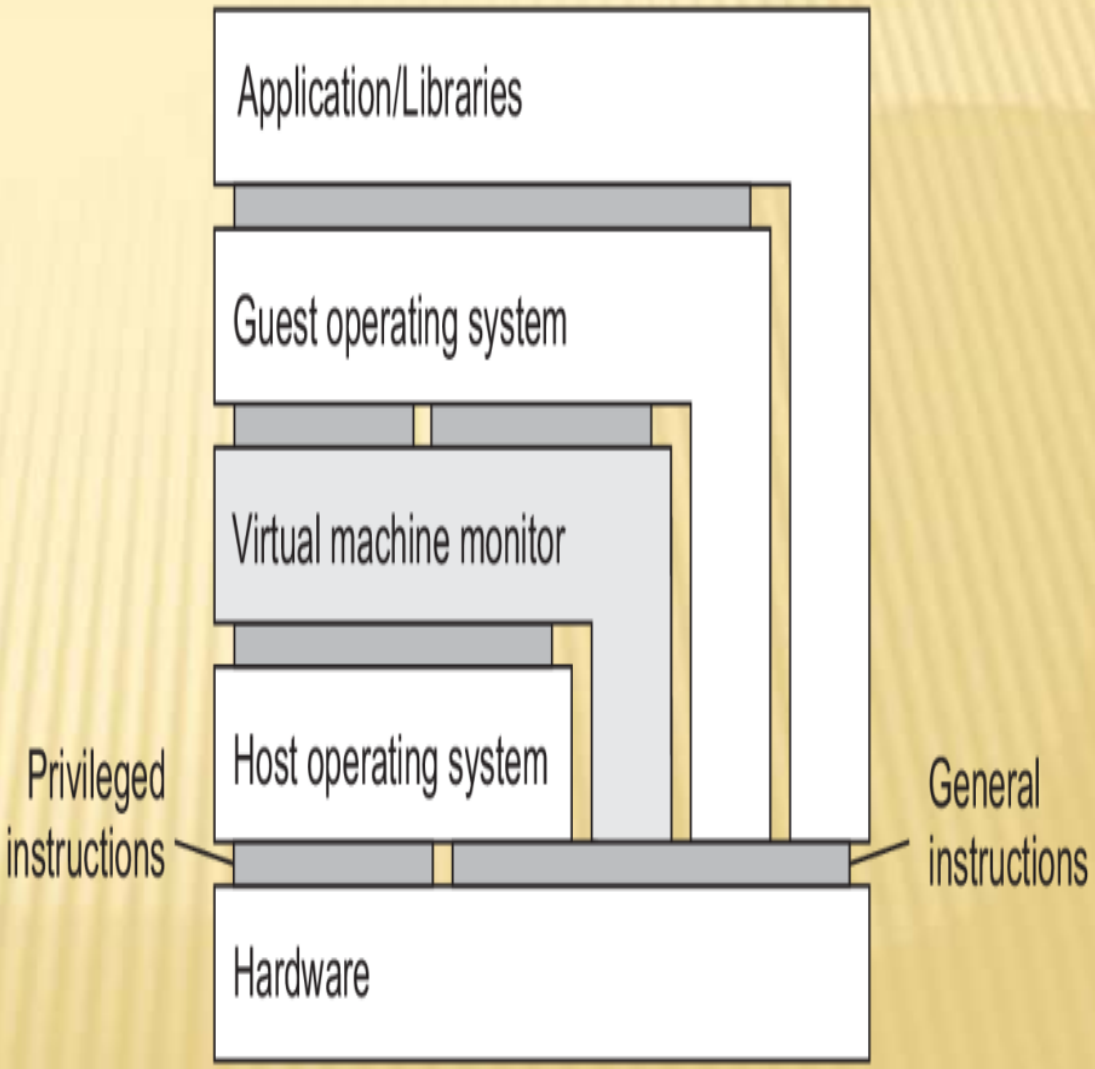
(b)



(c)

Refining the Organization

- **Privileged instruction: if and only if executed in user mode, it causes a trap to the operating system**
- **Nonpriviliged instruction: the rest**



VMs and Cloud Computing

Three types of cloud services:

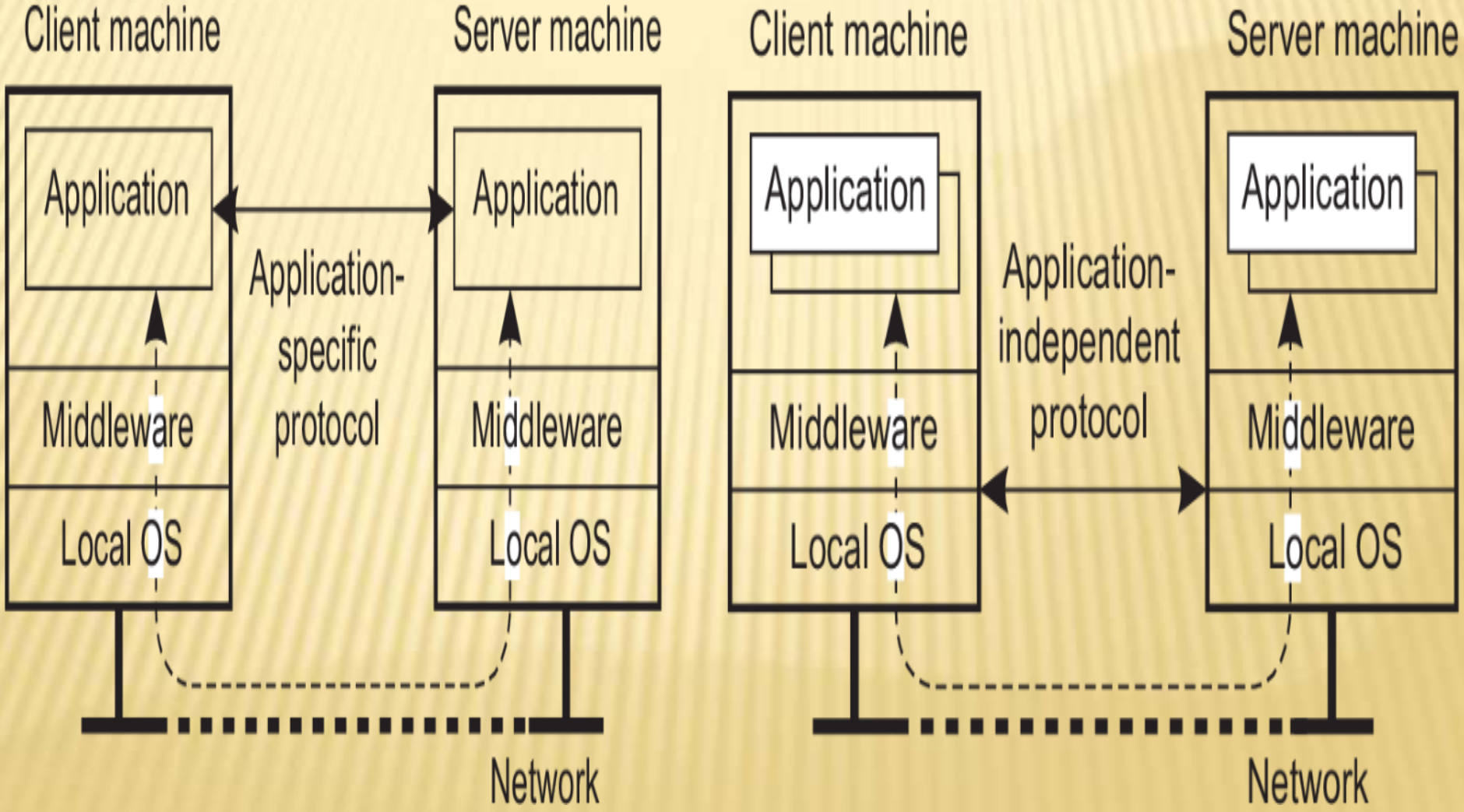
- Infrastructure-as-a-Service covering the basic infrastructure
- Platform-as-a-Service covering system-level services
- Software-as-a-Service containing actual applications

IaaS

Instead of renting out a physical machine, a cloud provider will rent out a VM (or VMM) that may possibly be sharing a physical machine with other customers.

Client-Server Interaction

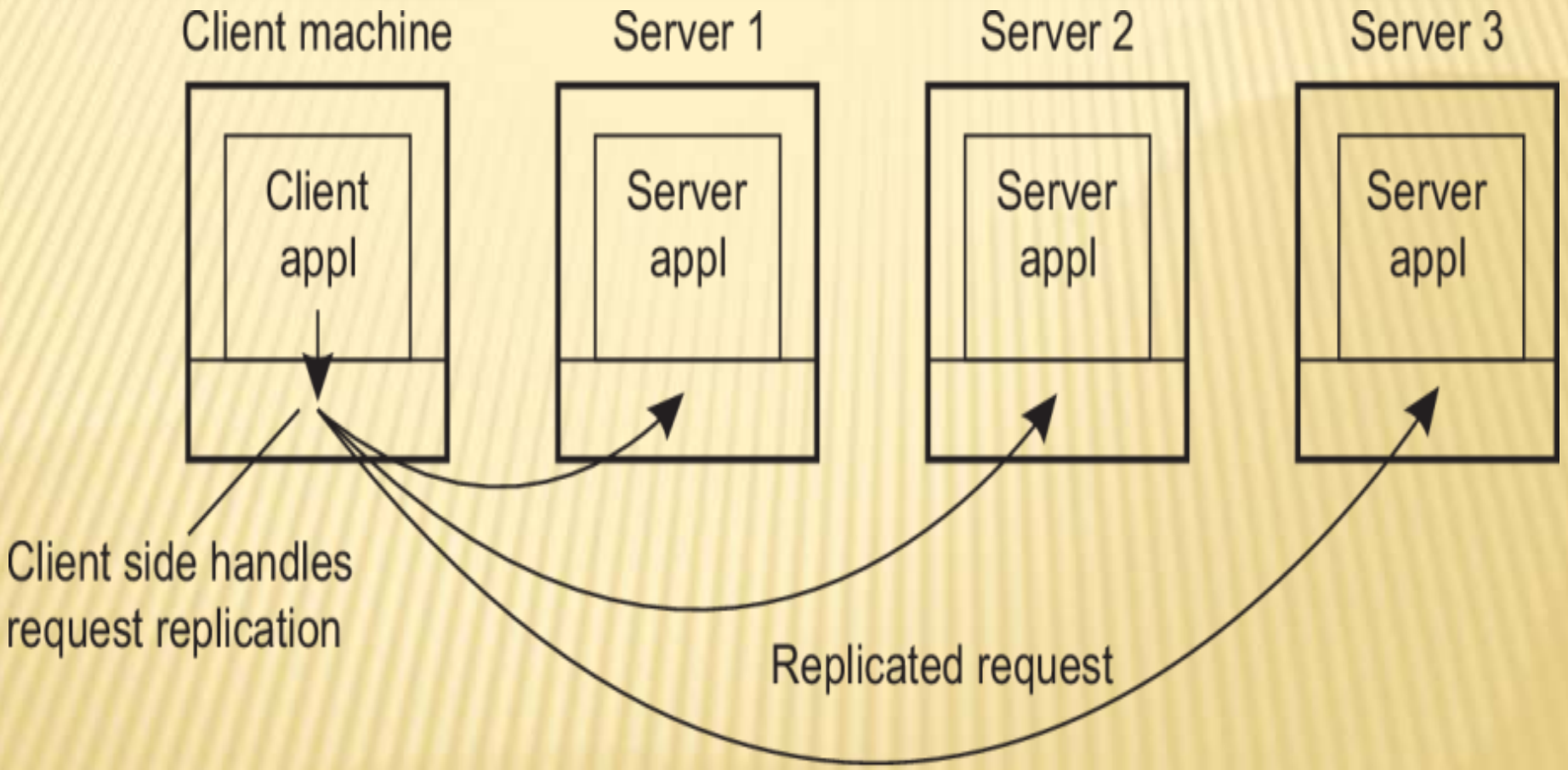
Distinguish application-level and middleware-level solutions



Client-Side Software

- Access transparency: client-side stubs for RPCs
- 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 (Stubs)



Servers: General Organization

Basic model:

A process implementing a specific service on behalf of a collection of clients. 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.

Concurrent Servers

Two basic types

- **Iterative server:** Server handles the request before attending a next request.
- **Concurrent server:** Uses a dispatcher, which picks up an incoming request that is then passed on to a separate thread/process.

Observation

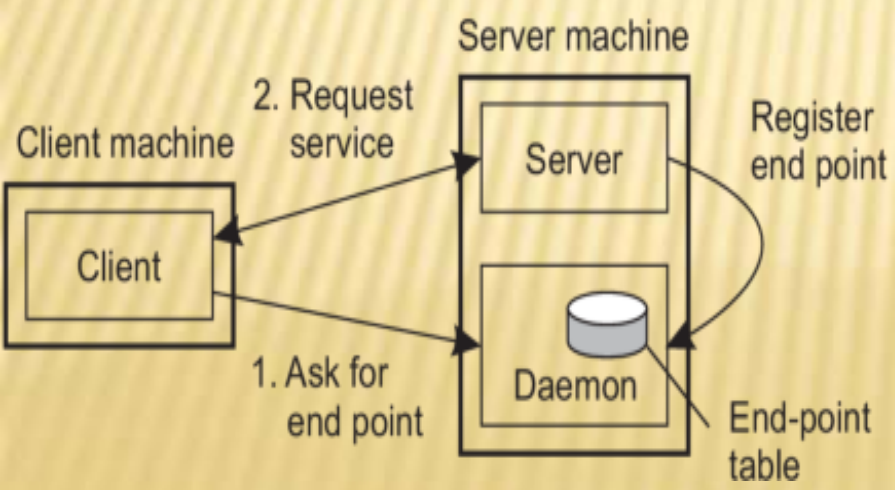
Concurrent servers are the norm: they can easily handle multiple requests, notably in the presence of blocking operations (to disks or other servers).

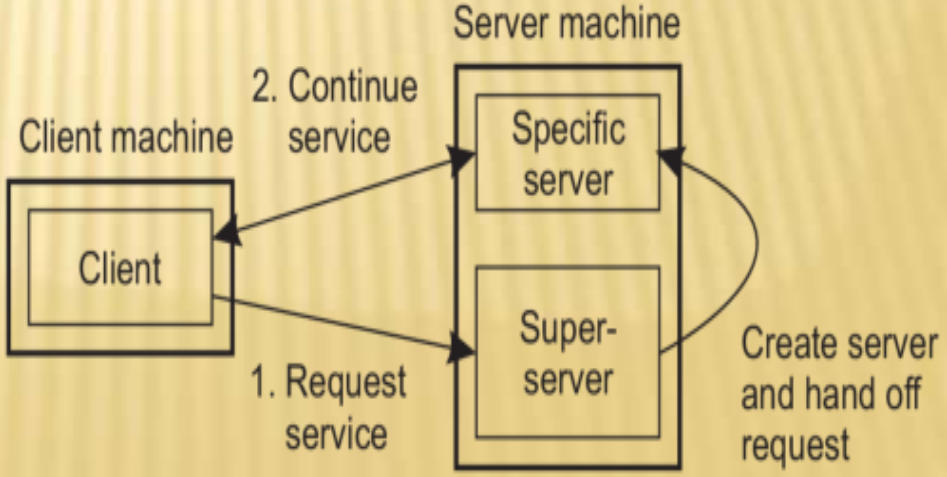
Contacting a Server

Observation: most services are tied to a specific port

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
smtp	25	Simple Mail Transfer
www	80	Web (HTTP)

Dynamically assigning an end point





Servers and State

Stateless servers:

Never keep accurate information about the status of a client after having handled a request:

- Don't record whether a file has been opened (simply close it again after access)
- Don't promise to invalidate a client's cache
- Don't keep track of your clients

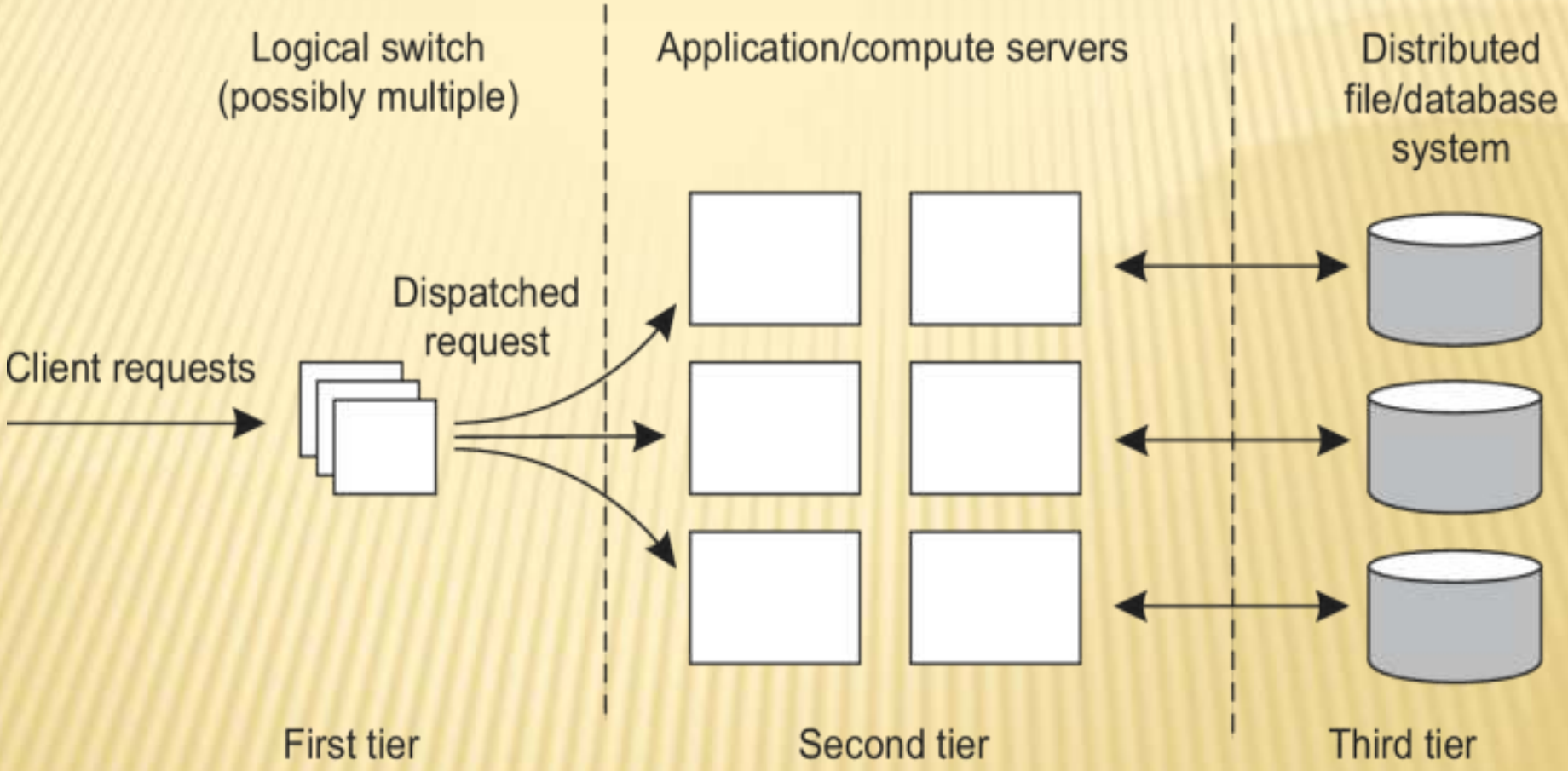
Servers and State

Stateful servers

Keeps track of the status of its clients:

- Record that a file has been opened, so that prefetching can be done
- Knows which data a client has cached, and allows clients to keep local copies of shared data

Server Cluster: Three Different Tiers



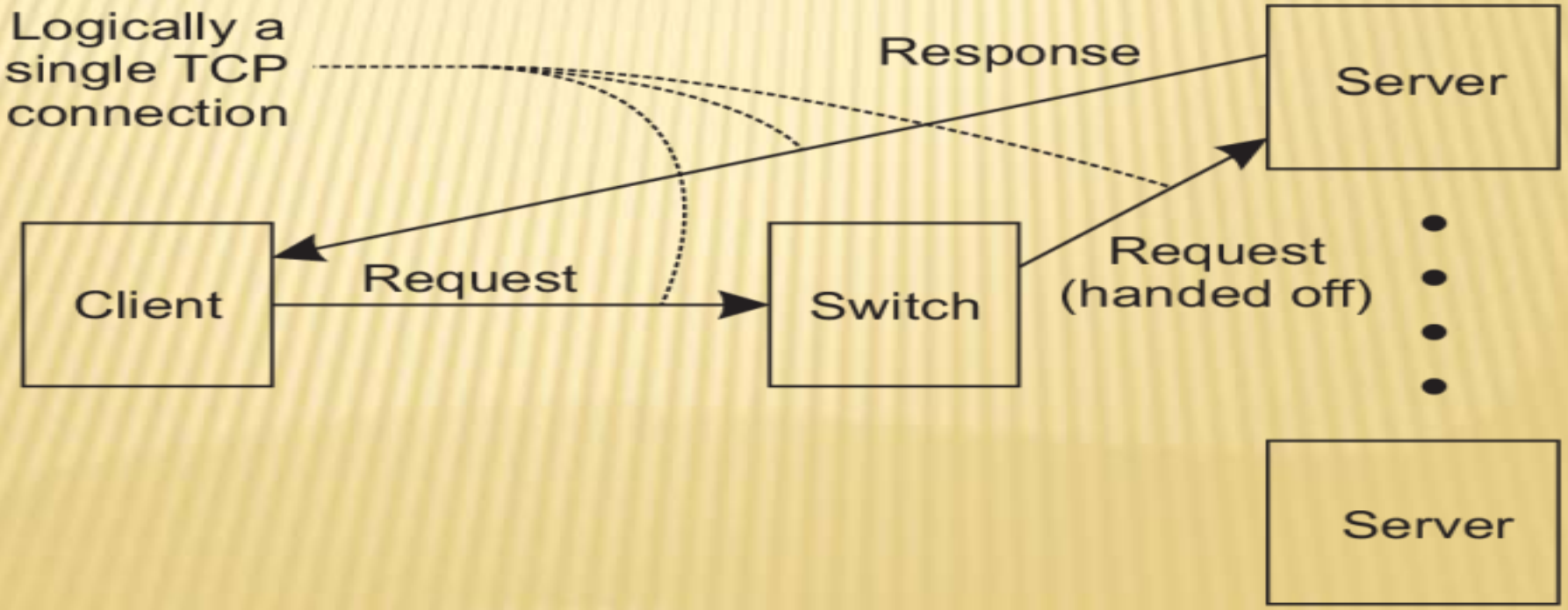
The first tier is generally responsible for passing requests to an appropriate server: request dispatching

Request Handling

Observation

Having the first tier handle all communication from/to the cluster may lead to a bottleneck.

A solution: TCP handoff



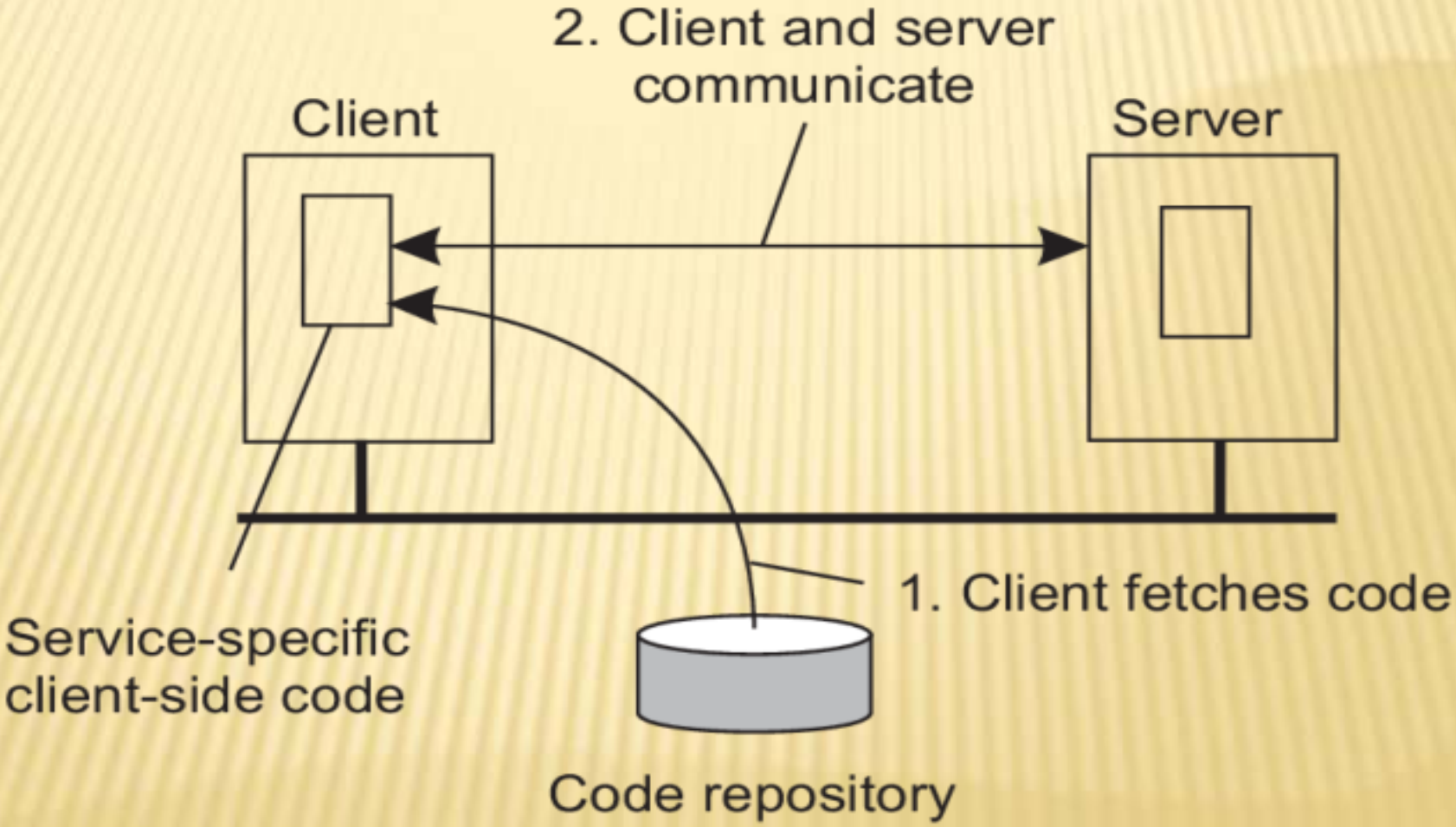
Reasons to Migrate Code

Load distribution

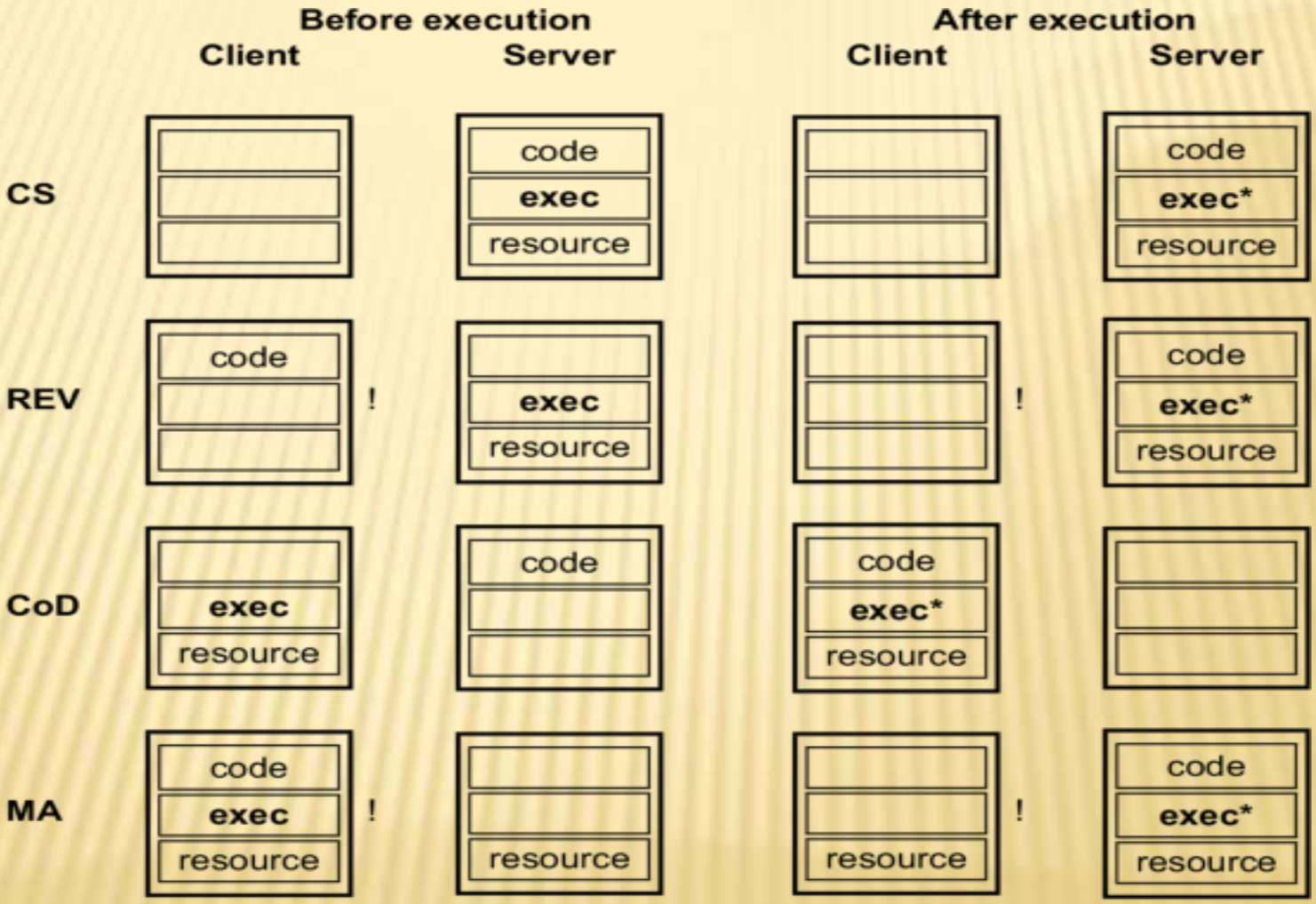
- Ensuring that servers in a data center are sufficiently loaded (e.g., to prevent waste of energy)
- Minimizing communication by ensuring that computations are close to where the data is (think of mobile computing).

Flexibility: moving code to a client when needed

Reasons to Migrate Code



Code Migration



Migration in Heterogeneous Systems

Main problem

- The target machine may not be suitable to execute the migrated code
- The definition of process/thread/processor context is highly dependent on local hardware, operating system

Only solution: abstract machine implemented on different platforms

- Interpreted languages, effectively having their own VM
- Virtual machine monitors

Migrating a Virtual Machine

Migrating images: three alternatives

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