

# Distribution of computation

## **client-server models**

# Introduction to threads

## Basic idea

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

## Contexts

**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

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

# Why use threads

## Some simple reasons

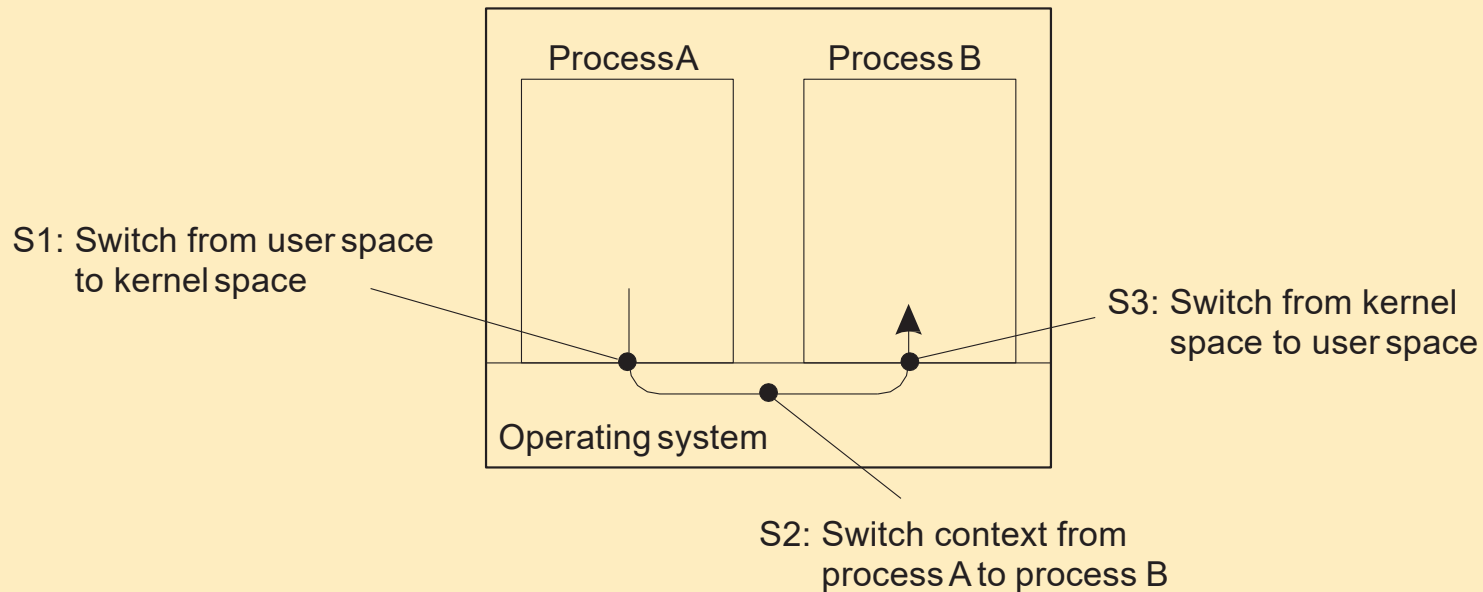
**Avoid needless blocking:** a single-threaded process will **block** when doing I/O; in a multi-threaded process, the operating system can switch the CPU to another thread in that process.

**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

## Avoid expensive context switching



## Trade-offs

Threads use the same address space: more prone to errors

No support from OS/HW to protect threads using each other's memory

Thread context switching may be faster than process context switching

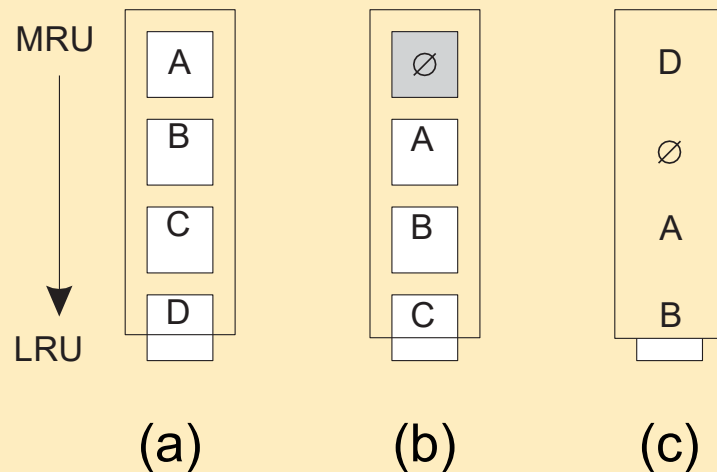
# The cost of a context switch

Consider a simple clock-interrupt handler

**direct costs:** actual switch and executing code of the handler

**indirect costs:** other costs, notably caused by messing up the cache

What a context switch may cause: indirect costs



(a) before the context switch

(b) after the context switch

(c) after accessing block *D*.

# Threads and operating systems

## Main issue

Should an OS kernel provide threads, or should they be implemented as user-level packages?

## User-space solution

All operations can be completely handled **within a single process**  $\Rightarrow$  implementations can be extremely efficient.

**All** services provided by the kernel are done **on behalf of the process in which a thread resides**  $\Rightarrow$  if the kernel decides to block a thread, the entire process will be blocked.

Threads are used when there are lots of external events: **threads block on a per-event basis**  $\Rightarrow$  if the kernel can't distinguish threads, how can it support signaling events to them?



# Threads and operating systems

## Kernel solution

The whole idea is to have the kernel contain the implementation of a thread package. This means that **all** operations return as system calls:

Operations that block a thread are no longer a problem: the **kernel schedules another available thread** within the same process.

handling external events is simple: the **kernel** (which catches all events) **schedules the thread associated with the event**.

The problem is (or used to be) the **loss of efficiency** due to the fact that each thread operation requires a trap to the kernel.

## Conclusion – but

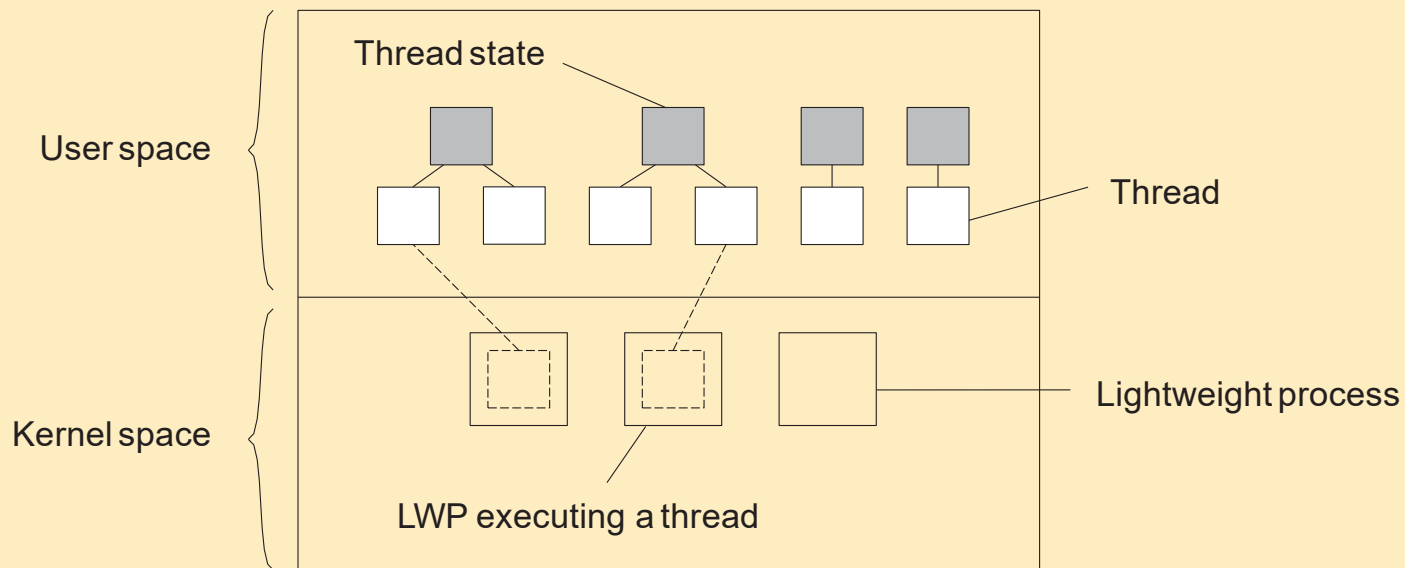
Try to mix user-level and kernel-level threads into a single concept, however, performance gain has not turned out to outweigh the increased complexity.

# Lightweight processes

## Basic idea

Introduce a two-level threading approach: **lightweight processes** that can execute user-level threads.

but the actual execution of the thread is done by the kernel



# Lightweight processes

prima quando volevo piu istanze dello stesso processo avevo tanti processi che eseguivano lo stesso codice, ora abbiamo un unico processo che lancia diversi thread che eseguono lo stesso codice . quindi prima molti processi , ora un processo con molti threads

## Principle operation

il kernel non è consapevole di cio sara lo user che gli dice di eseguire il threadma per il kernel un thread vale l'altro

User-level thread does system call  $\Rightarrow$  **the LWP that is executing that thread, blocks**. The thread remains **bound** to the LWP.

The kernel can **schedule another LWP having a runnable thread bound to it**. Note: this thread can switch to **any** other runnable thread currently in user space.

A thread calls a blocking user-level operation  $\Rightarrow$  do context switch to a runnable thread, (then bound to the same LWP).

When there are no threads to schedule, an LWP may remain idle, and may even be removed (destroyed) by the kernel.

## Note

This concept has been virtually abandoned – it's just either user-level or kernel-level threads.

# Using processes and threads at the client side

## Multithreaded web client

Hiding network latencies:

Web browser scans an incoming HTML page, and finds that **more files need to be fetched**.

**Each file is fetched by a separate thread**, each doing a (blocking) HTTP request.

As files come in, the browser displays them.

## 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: does it help?

## Thread-level parallelism: TLP

Let  $c_i$  denote the fraction of time that exactly  $i$  threads are being executed simultaneously.

$$TLP = \frac{\sum_{i=1}^N i \cdot c_i}{1 - c_0}$$

with  $N$  the maximum number of threads that (can) execute at the same time.

## Practical measurements

A typical Web browser has a TLP value between 1.5 and 2.5  $\Rightarrow$  threads are primarily used for [logically organizing](#) browsers.

# Using processes and threads at the server side

## 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.

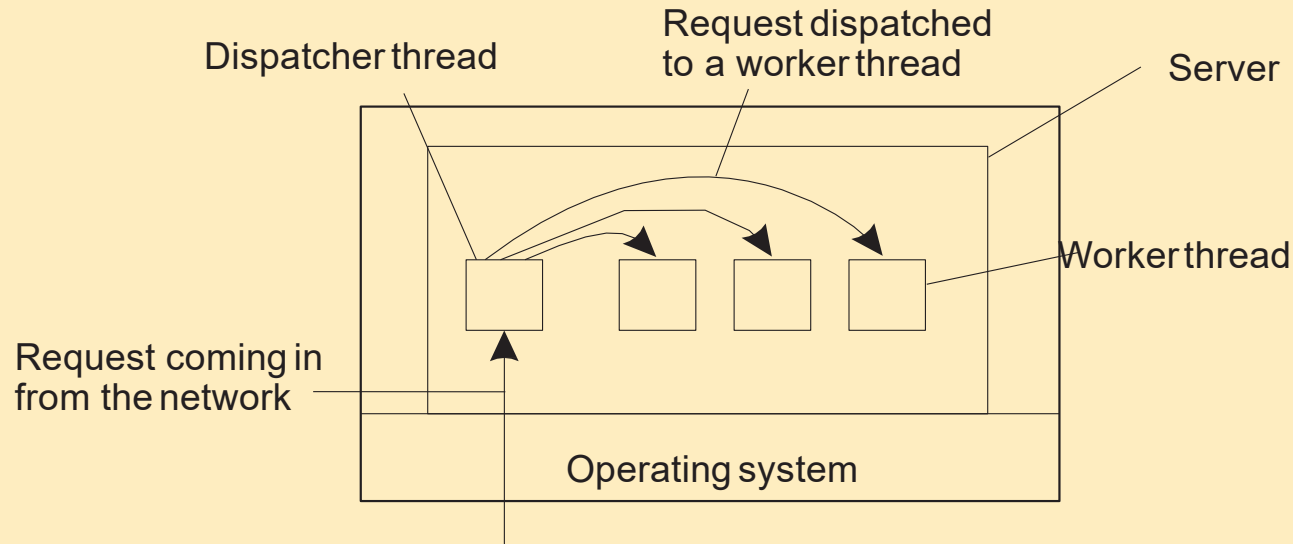
## Better structure

Most servers have high I/O demands. Using simple, **well-understood blocking calls** simplifies the overall structure.

Multithreaded programs tend to be **smaller and easier to understand** due to **simplified flow of control**.

# Why multithreading is popular: organization

## Dispatcher/worker model

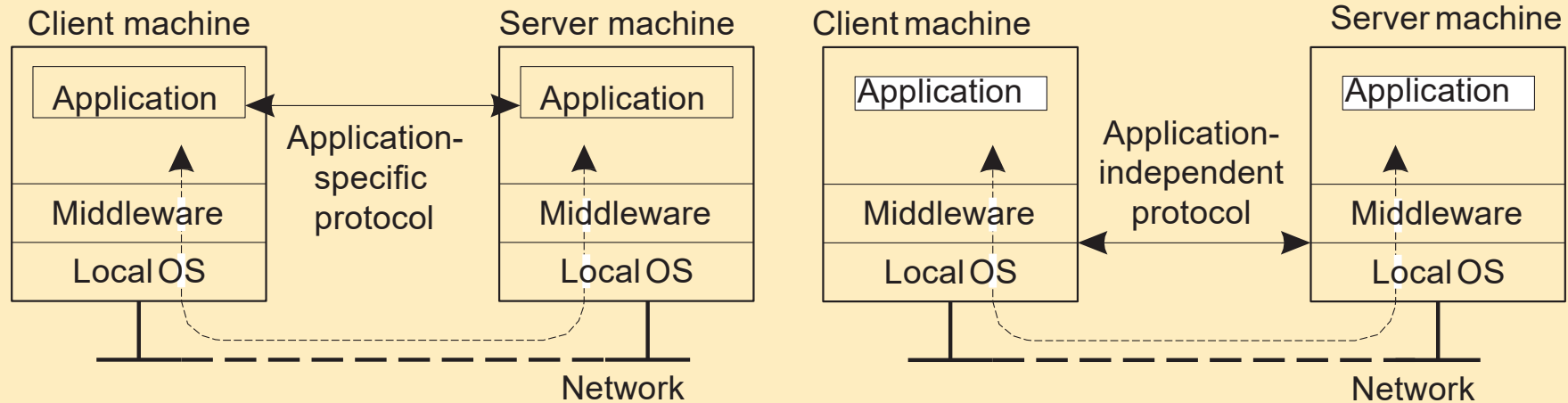


## Overview

Model	Characteristics
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

# Client-server interaction

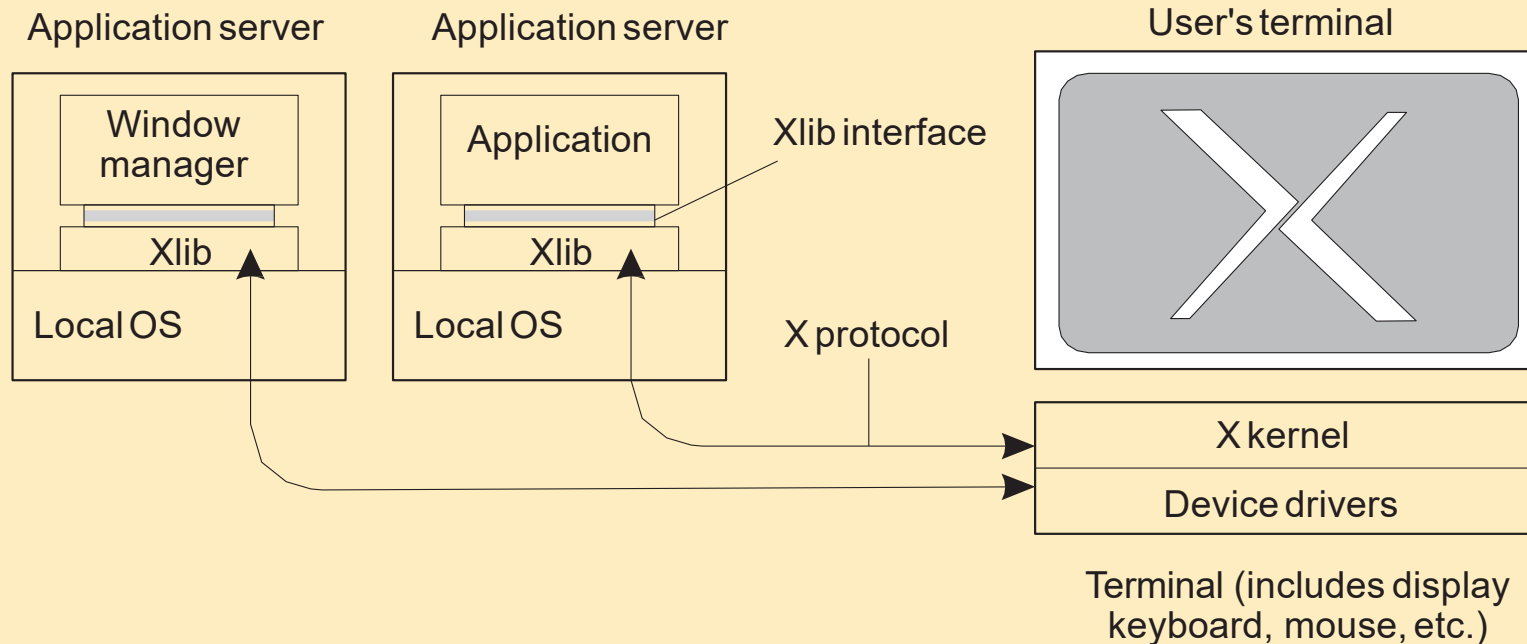
## Distinguish application-level and middleware-level solutions





# Example: The X Window system

## Basic organization



## X client and server

The application acts as a **client** to the X-kernel, the latter running as a **server** on the client's machine.

# Improving X

## Practical observations

There is often no clear separation between application logic and user-interface commands

Applications tend to operate in a tightly synchronous manner with an X kernel

## Alternative approaches

Let applications control the display **completely**, up to the pixel level (e.g., **VNC, Windows Remote desktop**)

Provide only a few high-level display operations (dependent on local video drivers), allowing more efficient display operations.

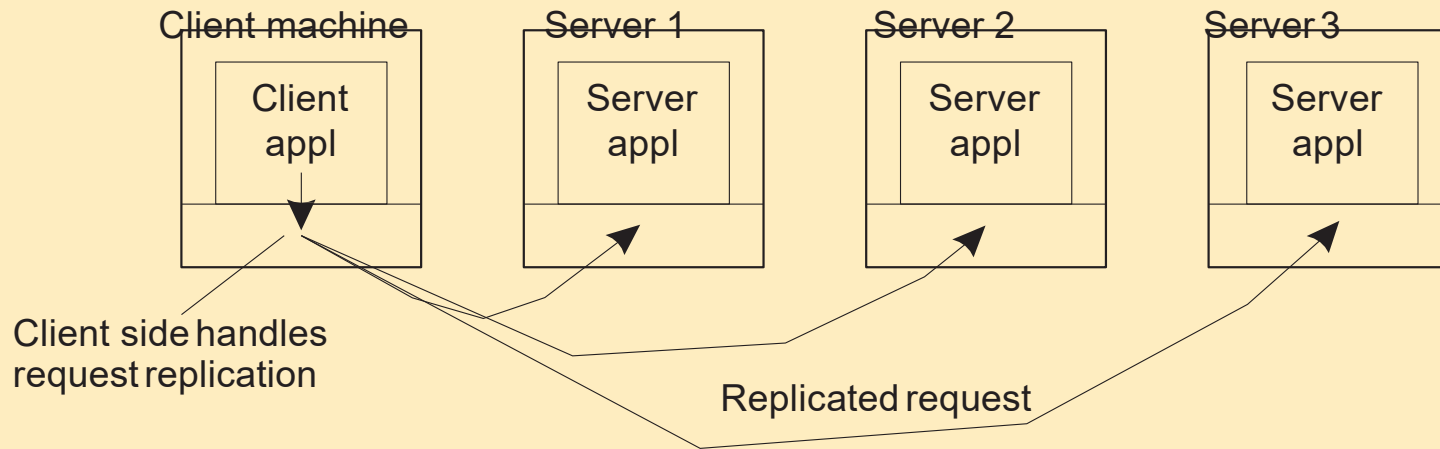
# Client-side software

Generally tailored for distribution transparency **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).

# Servers: concurrent servers

## 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.

## 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

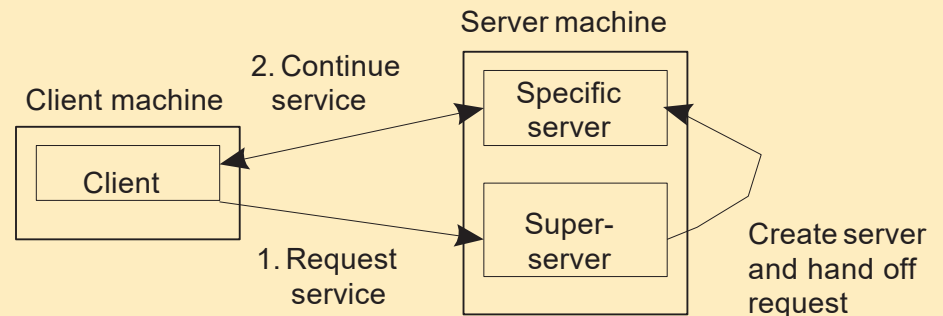
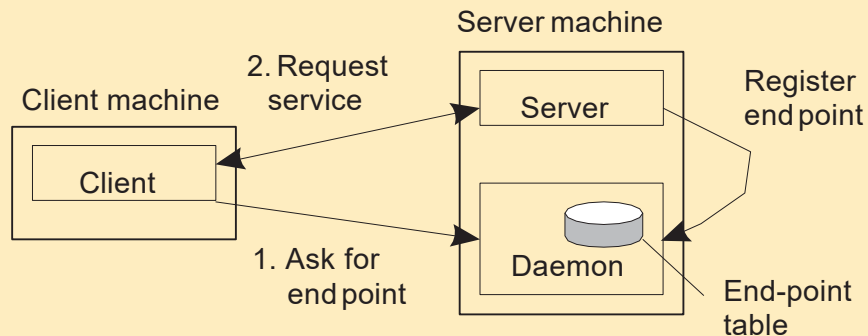
Concurrent servers are the norm: they can easily handle multiple requests, notably in the presence of blocking operations (to disks or other servers).

# 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



# Out-of-band communication

## Issue

Is it possible to **interrupt** a server once it has accepted (or is in the process of accepting) a service request?

## Solution 1: Use a separate port for urgent data

Server has a separate thread/process for urgent messages

Urgent message comes in  $\Rightarrow$  **associated request is put on hold**

Note: we require **OS supports priority-based scheduling**

## Solution 2: Use facilities of the transport layer

Example: TCP allows for urgent messages in same connection

Urgent messages can be caught using OS signaling techniques

# Servers and state

## Stateless servers

*è la solita roba*

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

## Consequences

Clients and servers are **completely independent**

**State inconsistencies** due to client or server crashes **are reduced**

Possible **loss of performance** because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

## Question

Does connection-oriented communication fit into a stateless design?

# 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

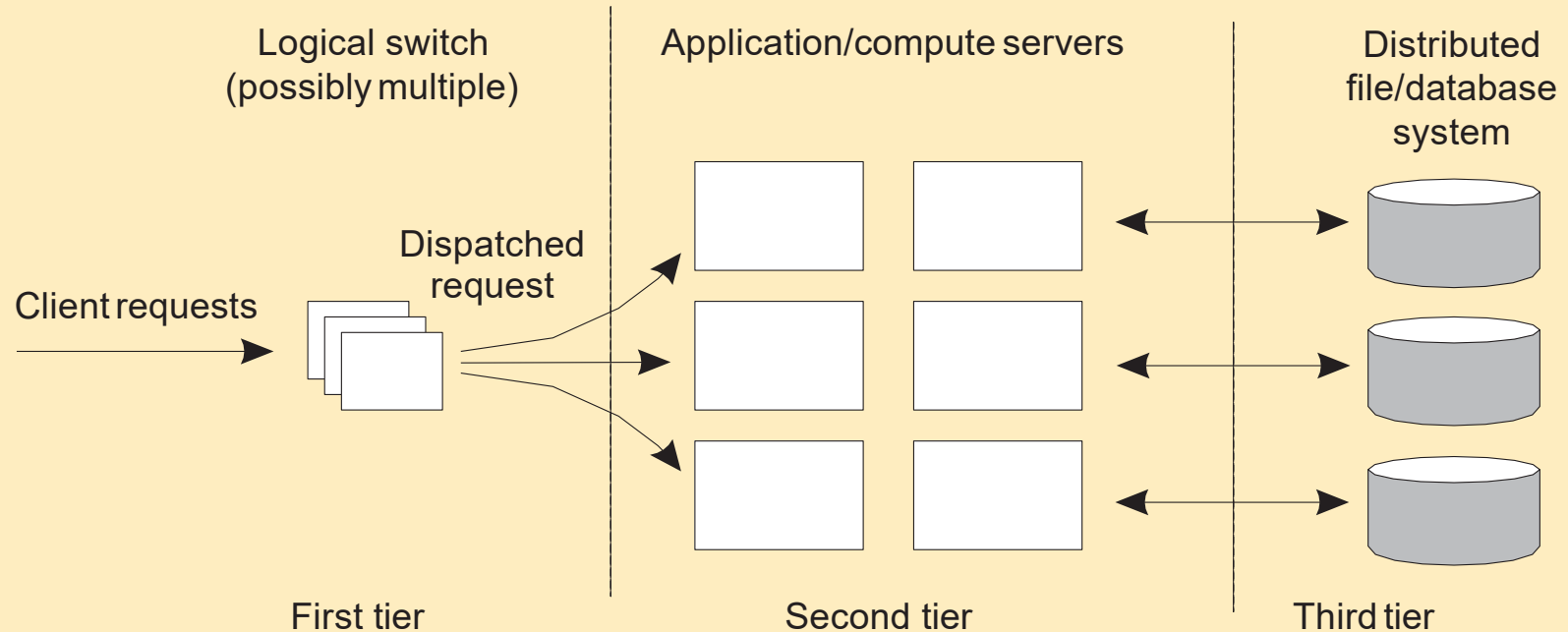
## Observation

The performance of stateful servers can be extremely high, provided clients are allowed to keep local copies. As it turns out, reliability is often not a major problem.



# Server clusters

## Common organization

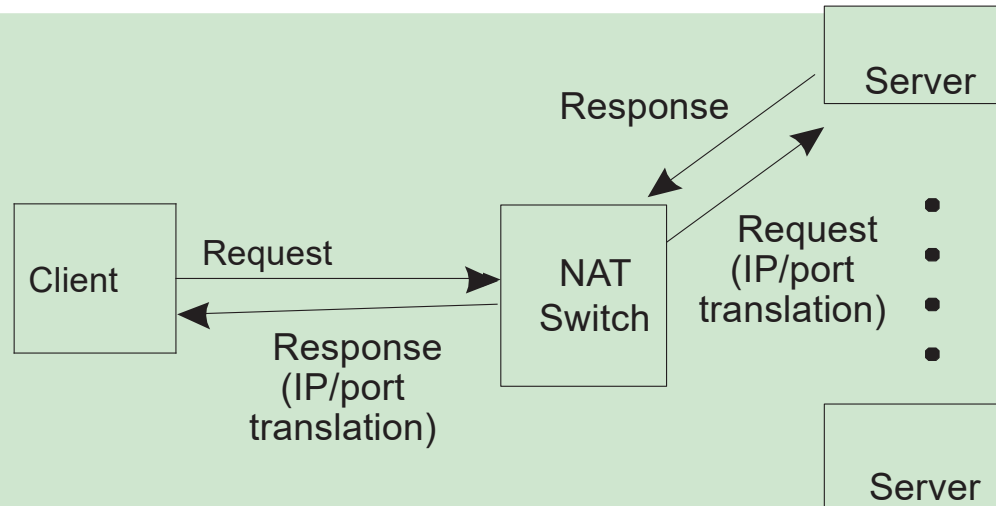


## Crucial element

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

# Request Handling

## First solution: NAT

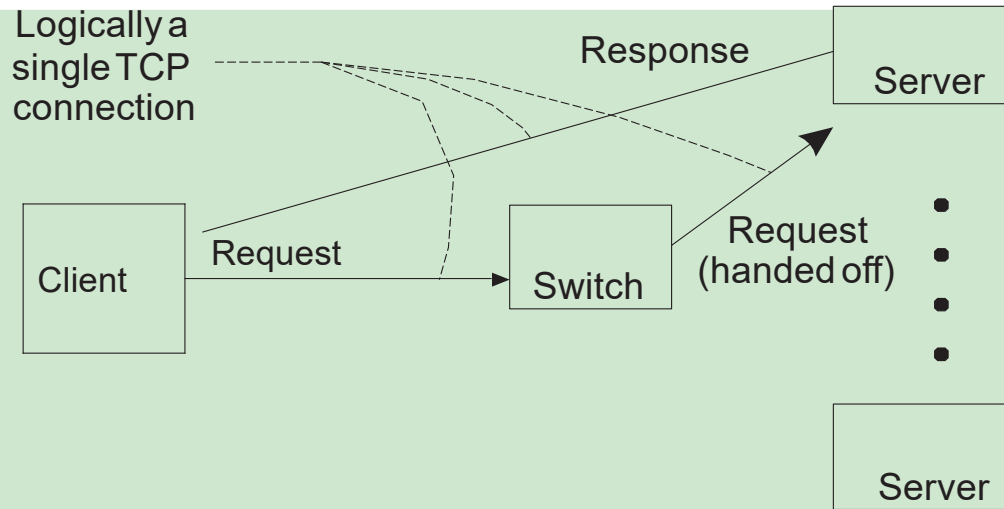


## Observation

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

# Request Handling

## Second solution: TCP handoff



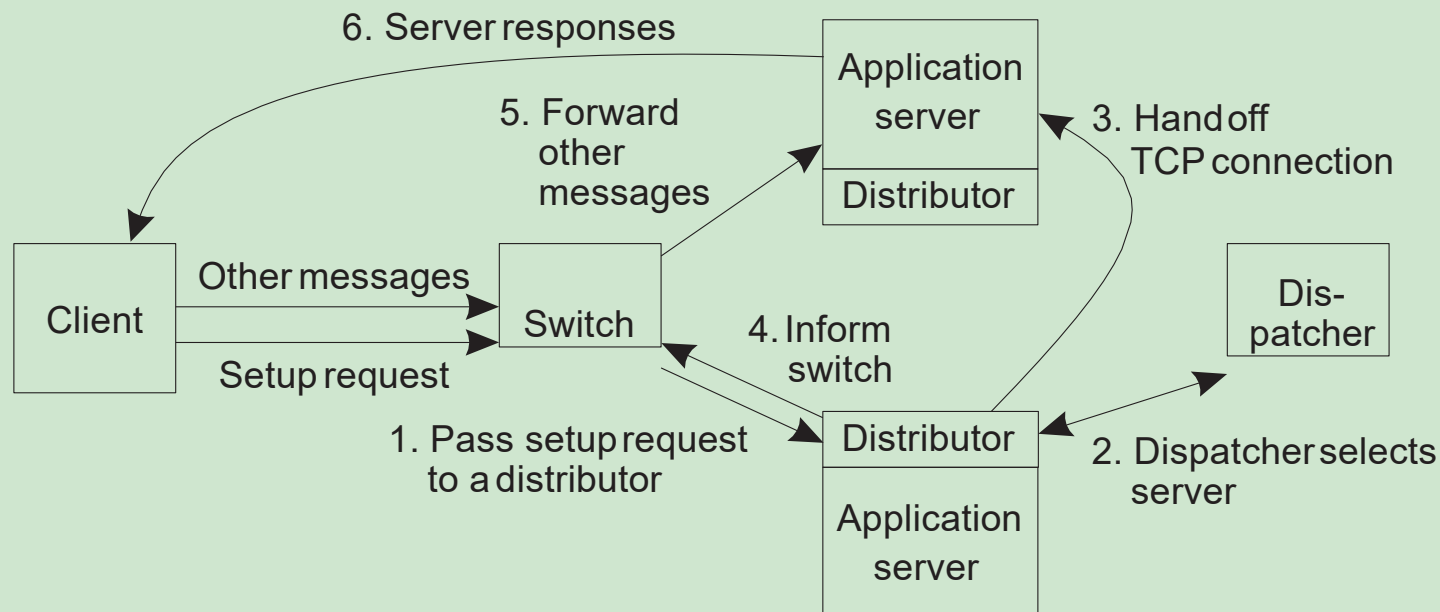
# Server clusters

The front end may easily get overloaded: special measures may be needed

**Transport-layer switching:** Front end simply passes the TCP request to one of the servers, taking some performance metric into account.

**Content-aware distribution:** Front end reads the content of the request and then selects the best server.

## Content aware distribution



# When servers are spread across the Internet

## Observation

Spreading servers across the Internet may introduce administrative problems. These can be largely circumvented by using data centers from a single cloud provider.

## Request dispatching: if locality is important

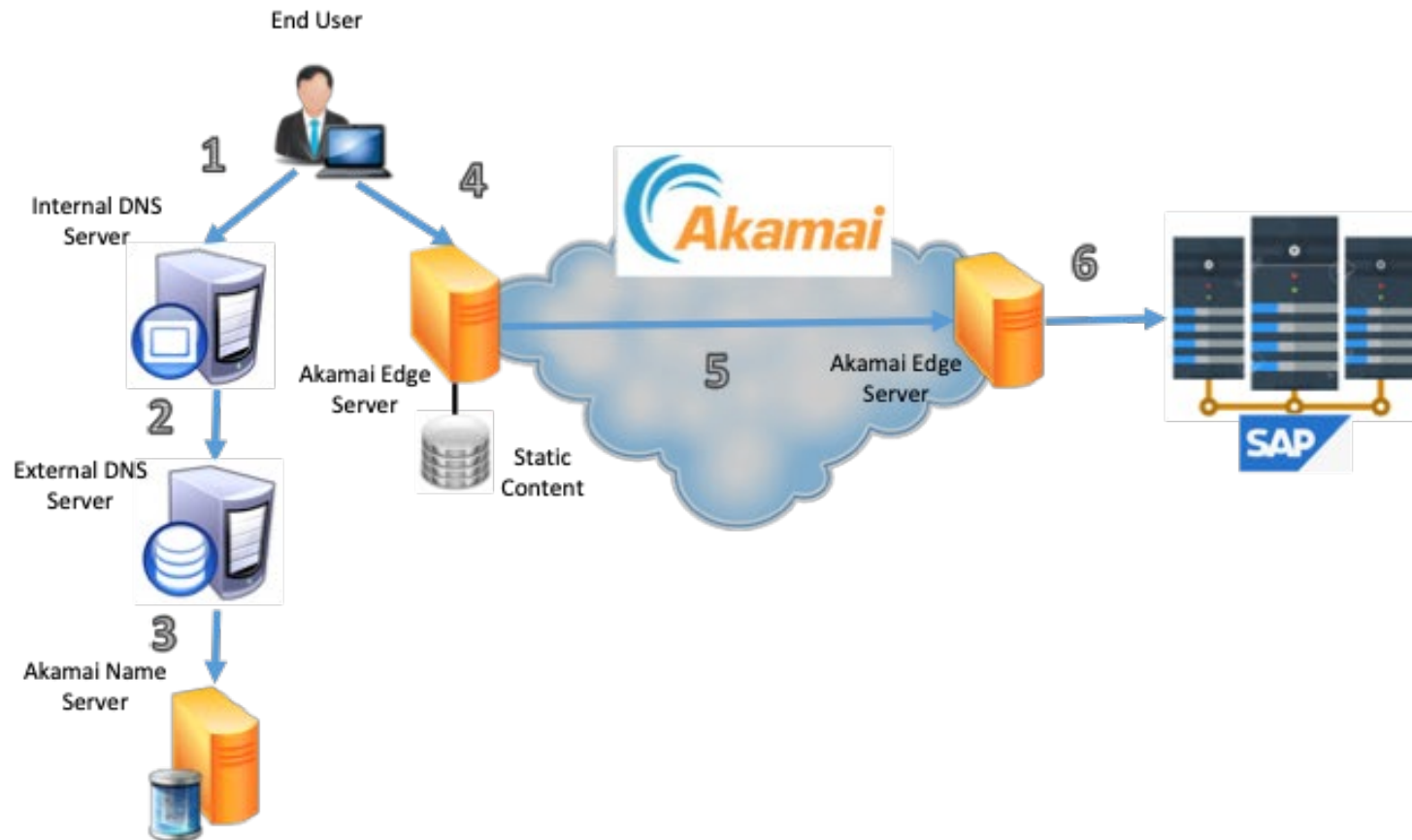
Common approach: use DNS:

- 1 Client looks up specific service through DNS - client's IP address is part of request
- 2 DNS server keeps track of replica servers for the requested service, and returns address of most local server.

## Client transparency

To keep client unaware of distribution, let DNS resolver act on behalf of client. Problem is that the resolver may actually be **far from local** to the actual client.

# How Akamai Works



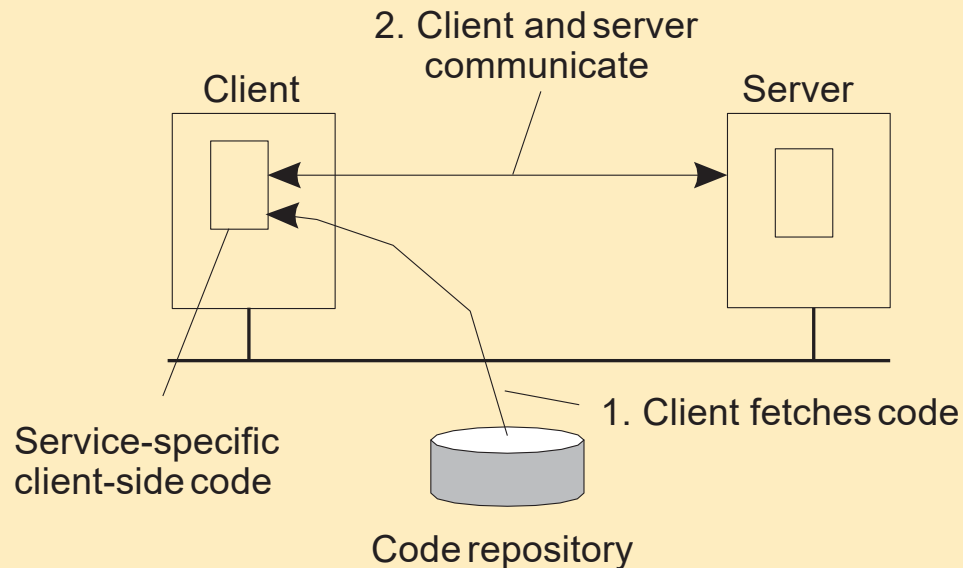
# CODE MIGRATION

## Reasons to migrate code: load distribution

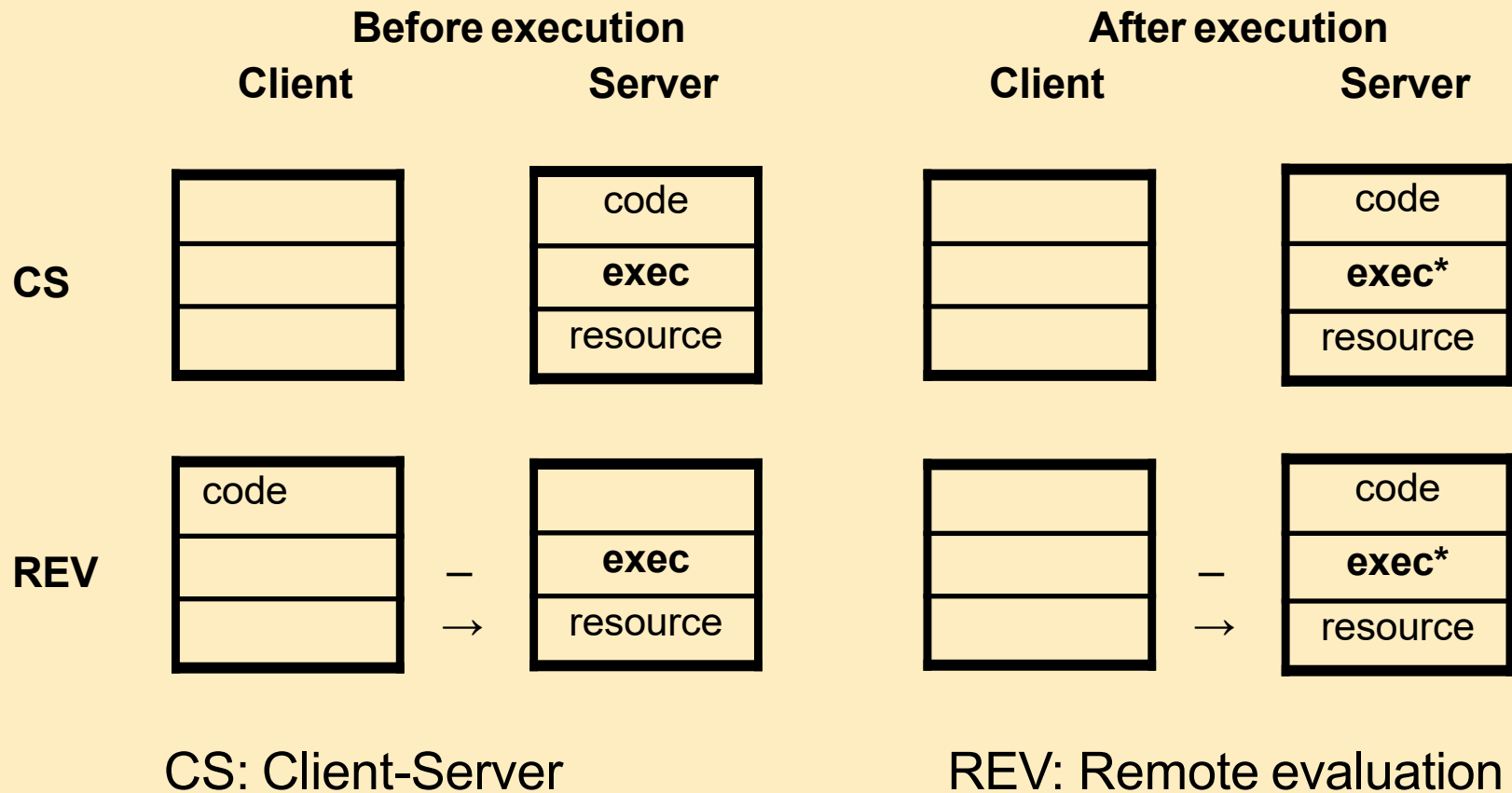
Ensuring that servers in a data center are **sufficiently** loaded (e.g., to prevent waste of energy)

Minimizing communication by ensuring that computations are close to where the data is (think of mobile computing).

## Reasons to migrate code: flexibility, moving code to a client when needed

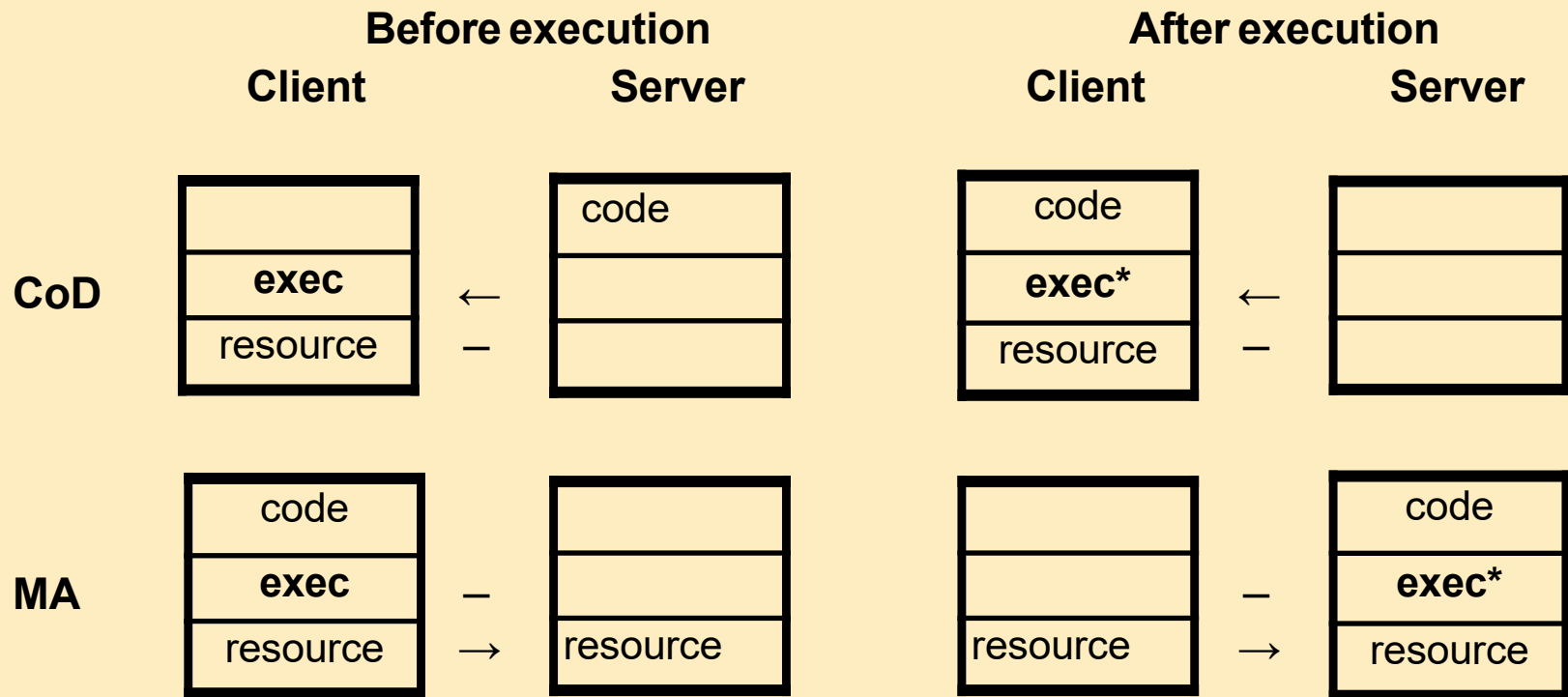


# Models for code migration





# Models for code migration



CoD: Code-on-demand

MA: Mobile agents

# Strong and weak mobility

## Object components

**Code segment:** contains the actual code

**Data segment:** contains the state

**Execution state:** contains context of thread executing the object's code

## Weak mobility: Move only code and data segment (and reboot execution)

Relatively simple, especially if code is portable

Distinguish **code shipping** (push) from **code fetching** (pull)

## Strong mobility: Move component, including execution state

**Migration:** move entire object from one machine to the other

**Cloning:** start a clone, and set it in the same execution state.

# 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 and runtime 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.

# Performance of migrating virtual machines

## Problem

A complete migration may actually take tens of seconds. We also need to realize that during the migration, a service will be completely unavailable for multiple seconds.

## Measurements regarding response times during VM migration

