# Distributed Systems

(4th edition, version 01)

## Chapter 03: Processes





## 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

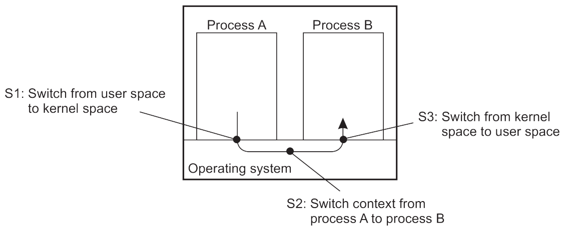
- Avoid needless blocking: a single-threaded process will block when doing I/O; in a multithreaded process, the operating system can switch the CPU to another thread in that process.
- Exploit parallelism: the threads in a multithreaded process can be scheduled to run in parallel on a multiprocessor 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

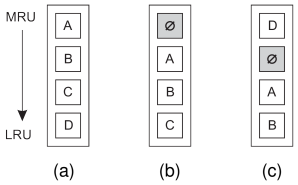
Processes Threads

## 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*.


Introduction to threads 7 / 54


Processes Threads

## A simple example in Python

```
from multiprocessing import Process
from time import *
from random import *

def sleeper(name):
    t = gmtime()
    s = randint(1,20)
    txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' is going to sleep for '+str(s)+' seconds'
    print(txt)
    sleep(s)
    t = gmtime()
    txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' has woken up'
    print(txt)

if __name__ == '__main__':
    p = Process(target=sleeper, args=('eve',))
    q = Process(target=sleeper, args=('bob',))
    p.start(); q.start()
    p.join(); q.join()
```

```
40:23 eve is going to sleep for 14 seconds
40:23 bob is going to sleep for 4 seconds
40:27 bob has woken up
40:37 eve has woken up
```


Introduction to threads 8 / 54


Processes Threads

## A simple example in Python

```
from multiprocessing import Process
from threading import Thread

shared_x = randint(10,99)

def sleeping(name):
    global shared_x
    t = gmtime(); s = randint(1,20)
    txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' is going to sleep for '+str(s)+' seconds'
    print(txt)
    sleep(s)
    t = gmtime(); shared_x = shared_x + 1
    txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' has woken up, seeing shared x being '
    print(txt+str(shared_x) )

def sleeper(name):
    sleeplist = list()
    print(name, 'sees shared x being', shared_x)
    for i in range(3):
        subsleeper = Thread(target=sleeping, args=(name+' '+str(i),))
        sleeplist.append(subsleeper)

    for s in sleeplist: s.start(); for s in sleeplist: s.join()
    print(name, 'sees shared x being', shared_x)

if __name__ == '__main__':
    p = Process(target=sleeper, args=('eve',))
    q = Process(target=sleeper, args=('bob',))
    p.start(); q.start()
    p.join(); q.join()
```


Introduction to threads 9 / 54

## A simple example in Python

```
eve sees shared x being 71
53:21 eve 0 is going to sleep for 20 seconds
bob sees shared x being 84
53:21 eve 1 is going to sleep for 15 seconds
53:21 eve 2 is going to sleep for 3 seconds
53:21 bob 0 is going to sleep for 8 seconds
53:21 bob 1 is going to sleep for 16 seconds
53:21 bob 2 is going to sleep for 8 seconds
53:24 eve 2 has woken up, seeing shared x being 72
53:29 bob 0 has woken up, seeing shared x being 85
53:29 bob 2 has woken up, seeing shared x being 86
53:36 eve 1 has woken up, seeing shared x being 73
53:37 bob 1 has woken up, seeing shared x being 87
bob sees shared x being 87
53:41 eve 0 has woken up, seeing shared x being 74
eve sees shared x being 74
```





## 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 ⇒ implementations can be extremely efficient.
- All services provided by the kernel are done on behalf of the process in which a thread resides ⇒ if the kernel decides to block a thread, the entire process will be blocked.
- Threads are used when there are many external events: threads block on a per-event basis ⇒ 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:

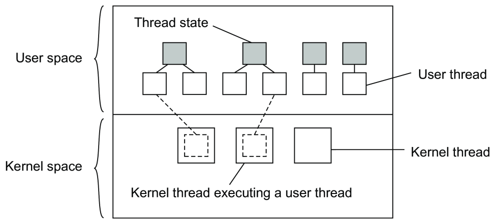
- Operations that block a thread are no longer a problem: the kernel schedules another available thread within the same process.
- handling external events is simple: the kernel (which catches all events) schedules the thread associated with the event.
- The problem is (or used to be) the loss of efficiency because 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 generally outweigh the increased complexity.

## Combining user-level and kernel-level threads

### Basic idea

Introduce a two-level threading approach: kernel threads that can execute user-level threads.








## User and kernel threads combined

### Principle operation

- User thread does system call ⇒ the kernel thread that is executing that user thread, blocks. The user thread remains bound to the kernel thread.
- The kernel can schedule another kernel thread having a runnable user thread bound to it. Note: this user thread can switch to any other runnable user thread currently in user space.
- A user thread calls a blocking user-level operation ⇒ do context switch to a runnable user thread, (then bound to the same kernel thread).
- When there are no user threads to schedule, a kernel thread may remain idle, and may even be removed (destroyed) by the kernel.





## Using 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.

Processes Threads

# 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 ⇒ threads are primarily used for logically organizing browsers.


Threads in distributed systems 16 / 54


Processes Threads

# Using 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

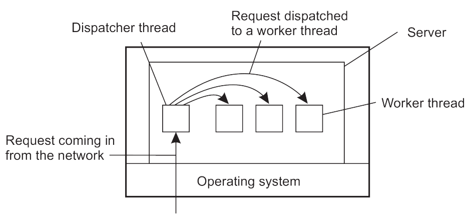
- Most servers have high I/O demands. Using simple, well-understood blocking calls simplifies the structure.
- Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control.


Threads in distributed systems 17 / 54


Processes Threads

# 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 |


Threads in distributed systems 18 / 54

Processes Virtualization

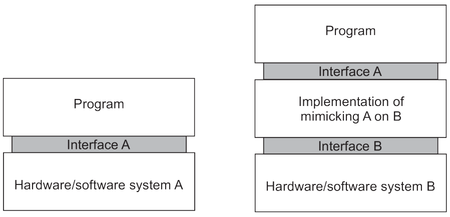
## Virtualization

### Observation

Virtualization is important:

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

### Principle: mimicking interfaces





Principle of virtualization 19 / 54


Processes Virtualization

## Mimicking interfaces

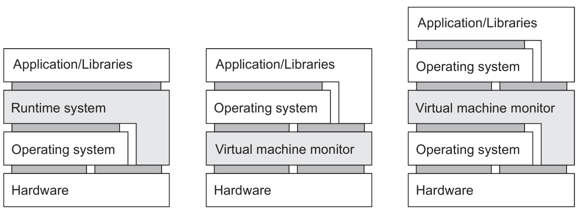
### Four types of interfaces at three different levels

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)


Principle of virtualization 20 / 54


Processes Virtualization

## Ways of virtualization




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

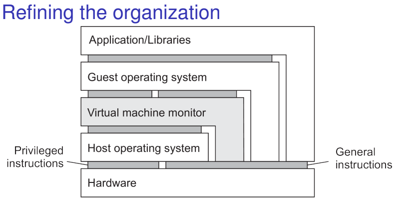
### Differences

(a) Separate set of instructions, an interpreter/emulator, running atop an OS.
(b) Low-level instructions, along with bare-bones minimal operating system
(c) Low-level instructions, but delegating most work to a full-fledged OS.


Principle of virtualization 21 / 54

## Zooming into VMs: performance

### 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

### Special instructions

- Control-sensitive instruction: may affect configuration of a machine (e.g., one affecting relocation register or interrupt table).
- Behavior-sensitive instruction: effect is partially determined by context (e.g., `POPF` sets an interrupt-enabled flag, but only in system mode).



Processes Virtualization

Principle of virtualization 22 / 54



## Condition for virtualization

### Necessary condition

*For any conventional computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

### Problem: condition is not always satisfied

There may be sensitive instructions that are executed in user mode without causing a trap to the operating system.

### Solutions

- Emulate all instructions
- Wrap nonprivileged sensitive instructions to divert control to VMM
- Paravirtualization: modify guest OS, either by preventing nonprivileged sensitive instructions, or making them nonsensitive (i.e., changing the context).

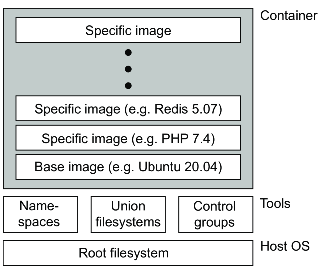


Processes Virtualization

Principle of virtualization 23 / 54



## Containers




- Namespaces: a collection of processes in a container is given their own view of identifiers
- Union file system: combine several file systems into a layered fashion with only the highest layer allowing for `write` operations (and the one being part of a container).
- Control groups: resource restrictions can be imposed upon a collection of processes.



Processes Virtualization

Containers 24 / 54

## Example: PlanetLab

### Essence
Different organizations contribute machines, which they subsequently share for various experiments.

### Problem
We need to ensure that different distributed applications do not get into each other's way ⇒ virtualization

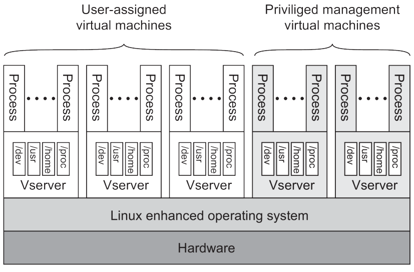








## PlanetLab basic organization




### Vserver
Independent and protected environment with its own libraries, server versions, and so on. Distributed applications are assigned a collection of vservers distributed across multiple machines





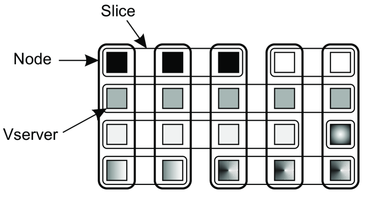




## PlanetLab Vservers and slices

### Essence
- Each Vserver operates in its own environment (cf. `chroot`).
- Linux enhancements include proper adjustment of process IDs (e.g., `init` having ID 0).
- Two processes in different Vservers may have same user ID, but does not imply the same user.

### Separation leads to slices



Processes Virtualization

## 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

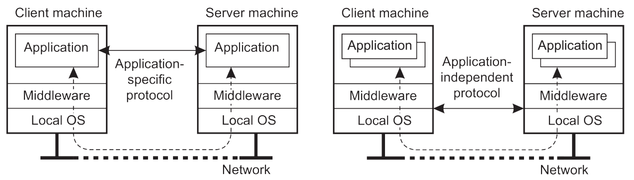
Instead of renting out a physical machine, a cloud provider will rent out a VM (or VMM) that may be sharing a physical machine with other customers ⇒ almost complete isolation between customers (although performance isolation may not be reached).


Application of virtual machines to distributed systems 28 / 54


Processes Clients

## Client-server interaction

### Distinguish application-level and middleware-level solutions



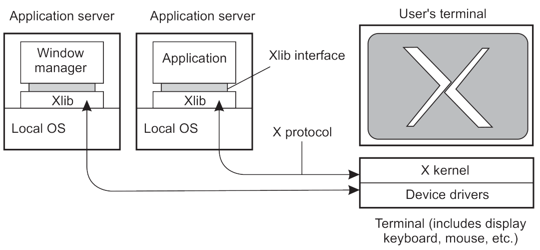


Networked user interfaces 29 / 54


Processes Clients

## 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.


Networked user interfaces 30 / 54

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





## Virtual desktop environment

### Logical development

With an increasing number of cloud-based applications, the question is how to use those applications from a user's premise?

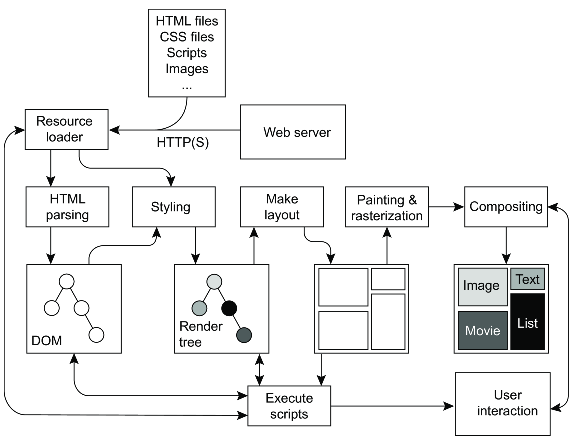
- Issue: develop the ultimate networked user interface
- Answer: use a Web browser to establish a seamless experience

The Google Chromebook





## The anatomy of a Web browser

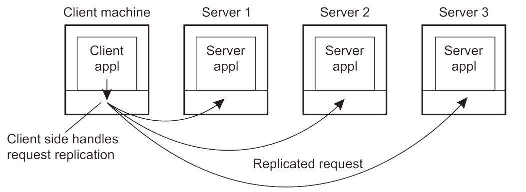


Processes Clients

## 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).


Client-side software for distribution transparency 34 / 54


Processes Servers

## 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.

### 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).


General design issues 35 / 54

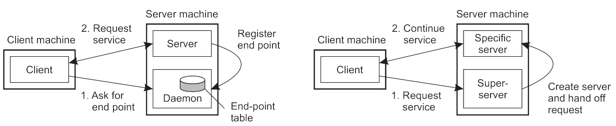

Processes Servers

## 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: two approaches





General design issues 36 / 54

## 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 ⇒ 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



Processes Servers

General design issues 37 / 54



## 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

### 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?



Processes Servers

General design issues 38 / 54



## 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.

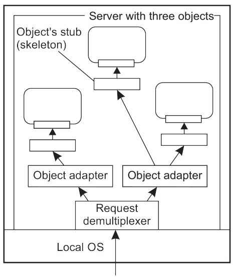


Processes Servers

General design issues 39 / 54

Processes Servers

## Object servers




- Activation policy: which actions to take when an invocation request comes in:
  - Where are code and data of the object?
  - Which threading model to use?
  - Keep modified state of object, if any?
- Object adapter: implements a specific activation policy


Object servers 40 / 54


Processes Servers

## Example: Ice runtime system – a server

```python
import sys, Ice
import Demo

class PrinterI(Demo.Printer):
    def __init__(self, t):
        self.t = t

    def printString(self, s, current=None):
        print(self.t, s)

communicator = Ice.initialize(sys.argv)

adapter = communicator.createObjectAdapterWithEndpoints("SimpleAdapter", "default -p 11000")
object1 = PrinterI("Object1 says:")
object2 = PrinterI("Object2 says:")
adapter.add(object1, communicator.stringToIdentity("SimplePrinter1"))
adapter.add(object2, communicator.stringToIdentity("SimplePrinter2"))
adapter.activate()

communicator.waitForShutdown()
```


Object servers 41 / 54


Processes Servers

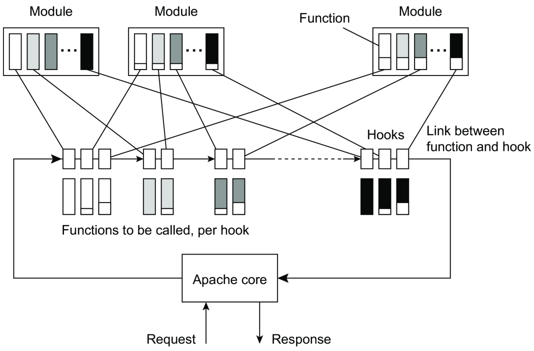
## Example: Ice runtime system – a client

```python
import sys, Ice
import Demo

communicator = Ice.initialize(sys.argv)

base1 = communicator.stringToProxy("SimplePrinter1:default -p 11000")
base2 = communicator.stringToProxy("SimplePrinter2:default -p 11000")
printer1 = Demo.PrinterPrx.checkedCast(base1)
printer2 = Demo.PrinterPrx.checkedCast(base2)
if (not printer1) or (not printer2):
    raise RuntimeError("Invalid proxy")

printer1.printString("Hello World from printer1!")
printer2.printString("Hello World from printer2!")

communicator.waitForShutdown()
```

```
Object1 says:  Hello World from printer1!
Object2 says:  Hello World from printer2!
```


Object servers 42 / 54

## Example: the Apache Web server





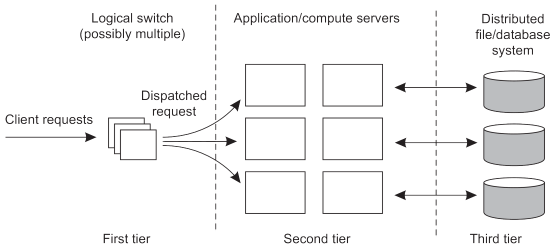



Example: The Apache Web server 43 / 54



## Three different tiers

### Common organization




### Crucial element

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

Processes Servers



Server clusters 44 / 54

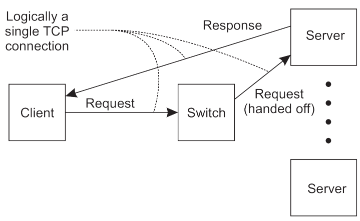


## Request Handling

### Observation

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

### A solution: TCP handoff




Processes Servers



Server clusters 45 / 54

## 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.

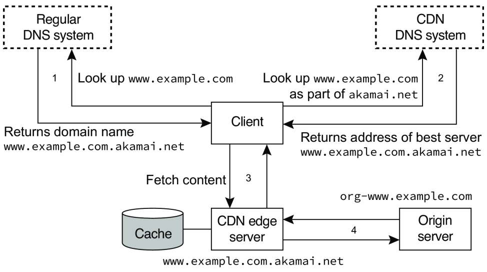
### Client transparency

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.





## A simplified version of the Akamai CDN




### Important note

The cache is often sophisticated enough to hold more than just passive data. Much of the application code of the origin server can be moved to the cache as well.



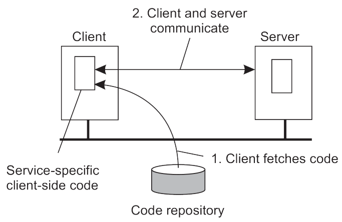


## Reasons to migrate code

### 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




Avoids pre-installing software and increases dynamic configuration.

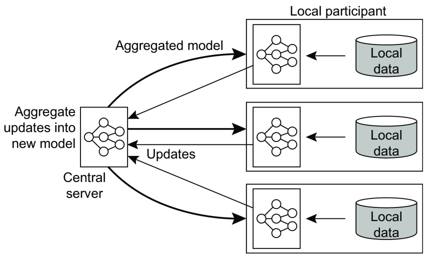
Processes Code migration

## Reasons to migrate code

### Privacy and security

In many cases, one cannot move data to another location, for whatever reason (often legal ones). Solution: move the code to the data.

### Example: federated machine learning

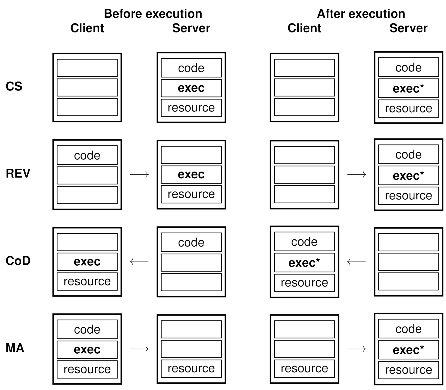




Reasons for migrating code 49 / 54


Processes Code migration

## Paradigms for code mobility




CS: Client-Server
CoD: Code-on-demand
REV: Remote evaluation
MA: Mobile agents


Models for code migration 50 / 54


Processes Code migration

## 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.


Models for code migration 51 / 54

## 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

### Observation

As containers are directly dependent on the underlying operating system, their migration in heterogeneous environments is far from trivial, to simply impractical, just as process migration is.

Processes Code migration



Migration in heterogeneous systems 52 / 54



## 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.

Processes Code migration



Migration in heterogeneous systems 53 / 54

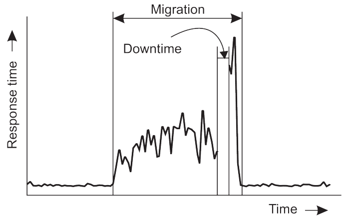


## Performance of migrating virtual machines

### 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




Processes Code migration



Migration in heterogeneous systems 54 / 54