Processes

Thomas Schwarz, SJ

Overview

- Process: A program in execution
  - OS-Level: Management and Scheduling of Processes
  - Distributed Systems: Other issues become more important

1. Threads
2. Virtualization
3. Client Server Organization
4. Process Migration

Threads

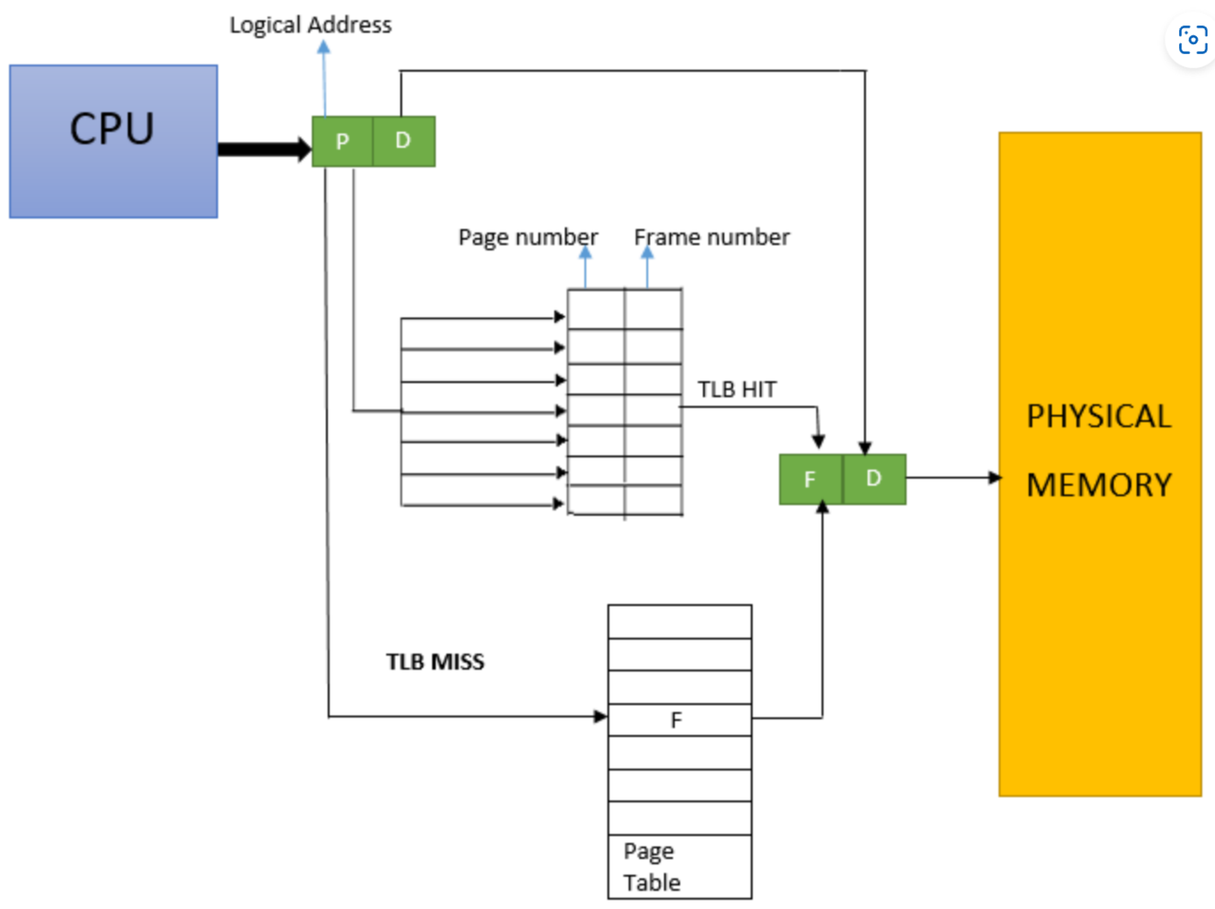
- Provide finer granularity for processes
    - Use case: Server for a client-server system
        - One listening thread:
            - Receives incoming request
            - Dispatches request to a number of worker requests
        - Worker threads:
            - Generated by listening thread or reside in a thread pool

Threads

- To execute a program:
  - OS generates a number of ***virtual processors***
    - Each one for running a different program
  - Tracked in a ***process table***
    - with CPU register values, memory maps, file handlers, privileges, accounting information, …
    - Entries jointly form a ***process content***
  - Process content is the software analogue of a processor context, needed to handle an interrupt

Threads

- OS provides process isolation
  - Despite sharing of CPUs, cores, … by several processes
  - OS create completely independent address space
  - Necessitates ***Memory Management Unit*** and ***Translation Lookaside Buffer***



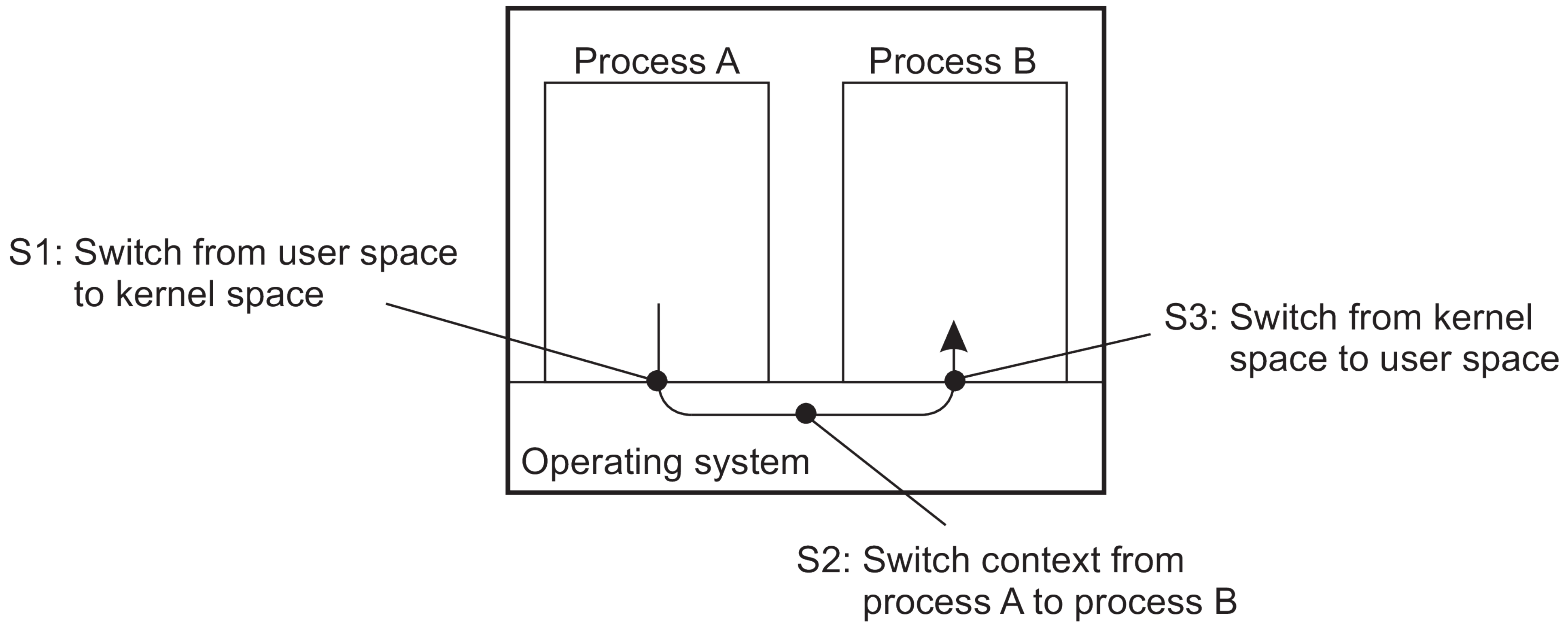
Threads

- Threads share the process’s memory space
- ***Thread context*** is small:
    - Processor context plus thread management
        - E.g. blocked by a mutex variable

Threads

- Advantages of threads in nondistributed systems
  - Deal with blocking system calls
  - Exploit parallelism
  - Large applications use ***Inter-Process Communication***s (IPC)
    - pipes, message queues, shared memory segments
    - Threads are more light-weight

Threads


Process A
Process B
S1: Switch from user space to kernel space
S3: Switch from kernel space to user space
Operating system
S2: Switch context from process A to process B

- 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

Threads

- Context Switch Costs:
    - Direct costs: switch and accessing handler
    - Indirect costs: Caches are messed up

Threads

- Threading and multiprocessing in Python
    - Multiprocessing is dependent on platform

```
import threading
import time
import random as rd

def to_str(my_time):
    return f'{my_time.tm_min}:{my_time.tm_sec:02d}'

def sleeper(name):
    while True:
        t = time.gmtime()
        s = rd.randint(1,5)
        print(f'{to_str(t)}  {name} is going to sleep for {s} 
seconds')
        time.sleep(s)
        t = time.gmtime()
        print(f'{to_str(t)}  {name} has woken up')
        time.sleep(rd.randint(1,5))
```

```
if __name__ == '__main__':
    x = threading.Thread(target=sleeper, args=('Bob',))
    y = threading.Thread(target=sleeper, args=('Alice',))
    x.start(); time.sleep(0.1); y.start()
    x.join(); y.join()
```

```
def to_str(my_time):
    return f'{my_time.tm_min}:{my_time.tm_sec:02d}'

shared_variable = rd.randint(0,1000)

def sleeper(name):
    while True:
        t = time.gmtime()
        s = rd.randint(1,5)
        print(f'{to_str(t)}  {name} is going to sleep for {s}
seconds')
        time.sleep(s)
        t = time.gmtime()
        print(f'{to_str(t)}  {name} has woken up, sees
{shared_variable}')
        time.sleep(rd.randint(1,5))
```

```
def changer():
    while True:
        print('changer')
        global shared_variable
        time.sleep(rd.randint(1,5))
        shared_variable = (shared_variable + rd.randint(1,50))%1000

if __name__ == '__main__':
    x = threading.Thread(target=sleeper, args=('Bob',))
    y = threading.Thread(target=sleeper, args=('Alice',))
    c = threading.Thread(target=changer, args=())
    c.start()
    x.start(); time.sleep(0.1); y.start()
    x.join(); y.join(); c.join()
```

Threads

- Threading in C++

```
#include <thread>

int main(int argc, const char * argv[]) {
    for(int i=0; i<NRTHREADS; i++){
        my_threads[i] = std::thread(insert_,
                                    i*NRELEMENTS/NRTHREADS,
                                    (i+1)*NRELEMENTS/NRTHREADS-1,
                                    std::ref(my_SH));
    }
    for(int i=0; i<NRTHREADS; i++){
        my_threads[i].join();
    }
}
```

Threads

- Use locks

```
class Bucket{
protected:
    unsigned int bucket_no;
    std::vector<key_value_pair> my_bucket;
    mutable std::shared_mutex bucket_guard;
}

Bucket::look_up(const unsigned int& key)
{
    std::shared_lock lock(bucket_guard);
    for (auto& element: my_bucket)
    {
        if(element.key == key) {
            return &element;
       }
    }
    return nullptr;
}
```

Threads

- Implemented by packages
- **Option 1: User-level threads**

  + Cheap to create a new thread

  - Basically, allocation of a stack for each thread

  + Context switches are simple

  - Only CPU registers
  - No need to change memory maps, flush the TLB, do CPU accounting

Threads

- Implemented by packages
- **Option 1: User-level threads**

  - Many-to-one threading model:

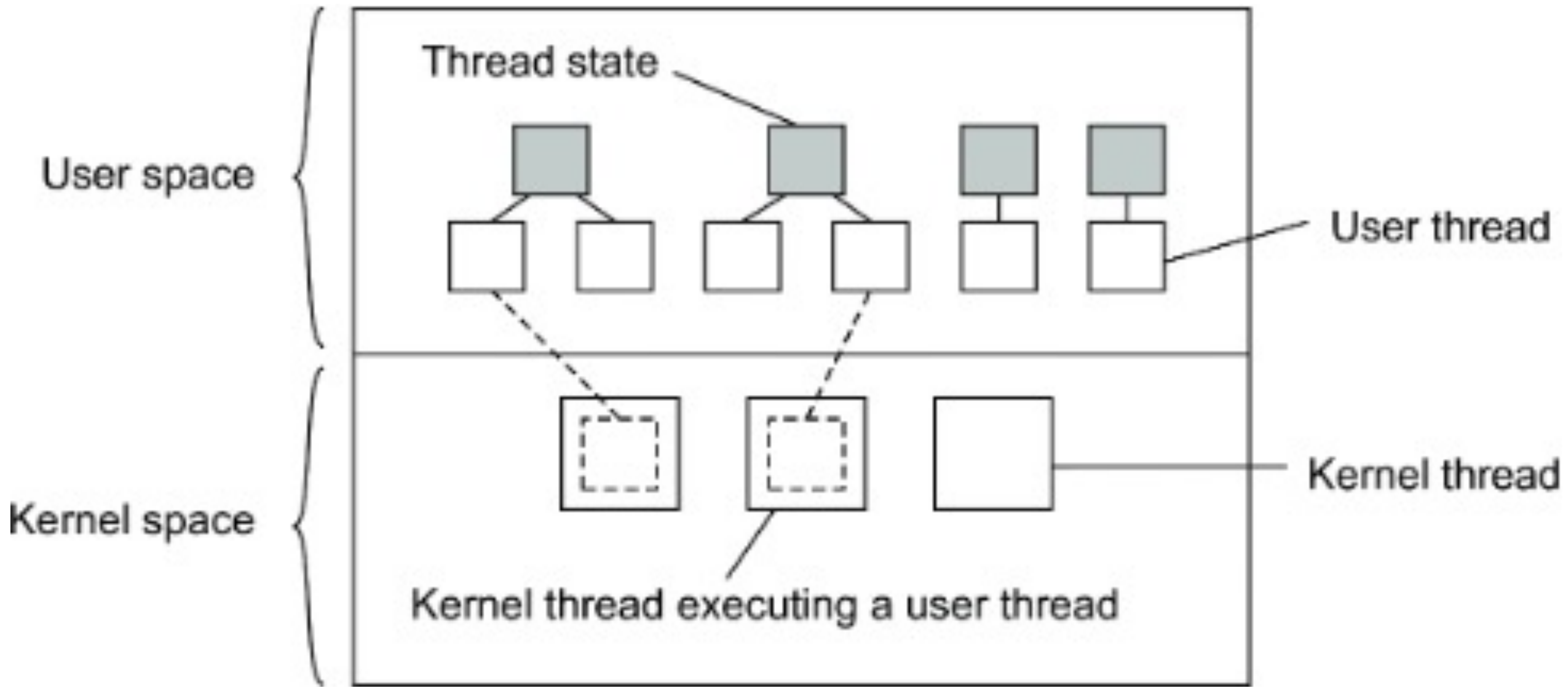
  - A single schedulable unit:
    - If a system call blocks, all threads are blocked

Threads

- **Option 2: Kernel-level Threads**
  - **One-to-one threading model**:
    - Each thread can be scheduled
  - Switching thread contexts may now become as expensive as switching process contexts.
    - In practice: costs of context switch are determined mainly by effectives on caches
    - Most OS offer this model

Threads

- **Option 3: Many-to-many threading model**
    - Introduce a two-level threading approach: kernel threads that can execute user-level threads.



Threads

- 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.

-

Threads

- 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.
-

Threads

- Threads in distributed systems
    - Using threads at the client side
        - E.g.: 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.

Threads

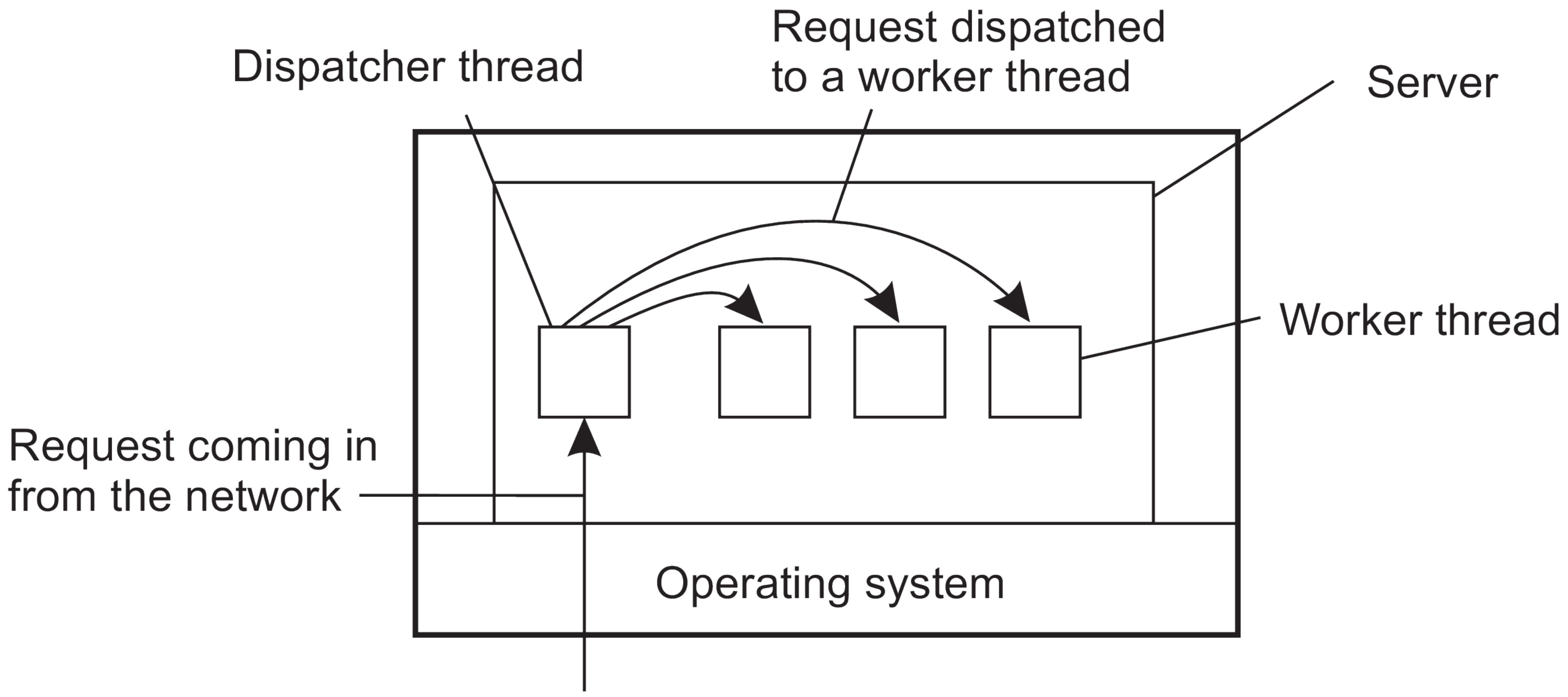
- Threads in distributed systems
    - Using threads at the client side
        - E.g.: Multiple request-response calls to other machines (RPC)
            - A client does several calls at the same time, each one by a different thread.
            - It then waits until all results have been returned.
        - Note: if calls are to different servers, we may have a linear speed-up.

Threads

- Multi-threaded clients: Does it help?
  - **T**hread-**L**evel **P**arallelism (TLP)
    - Let c_i be the proportion of time that exactly i threads are running
    - $$\text{TLP} = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}$$
    - Web browser: TLP between 1.5 and 2.5
    - So, should use 2 or 3 cores

Threads

- Multi-threaded servers
    - Dispatch — Worker Thread Model


Dispatcher thread
Request dispatched to a worker thread
Server
Worker thread
Request coming in from the network
Operating system

Threads

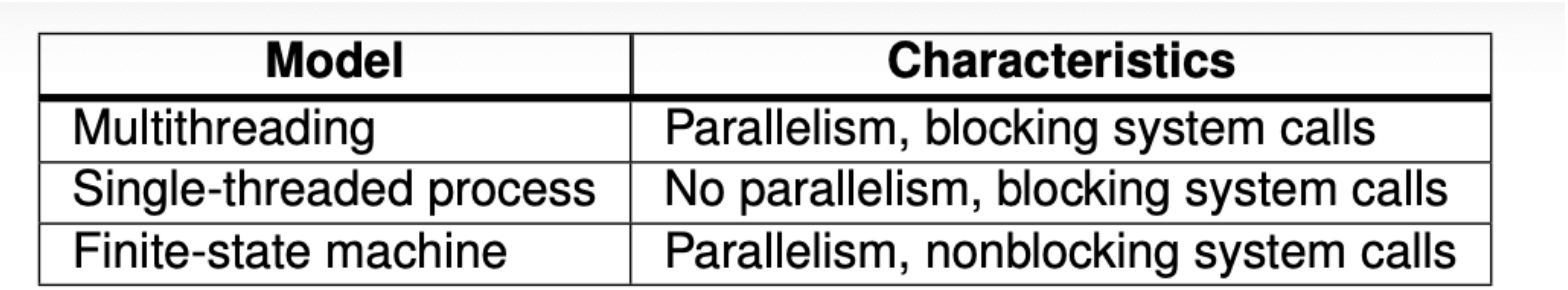
- Example: File server that occasionally needs to wait for a block to be provided
    - Can be organized as above
    - Can be organized as a single thread
        - The main loop of the file server
            - gets a request
            - examines it,
            - carries it out to completion
            - before getting the next one.

Threads

- Third option: run the server as a big single-threaded finite-state machine
  - Disk operations are non-blocking (asynchronous)
    - When terminated, OS informs process
    - Server maintains information about each disk operation
    - Once disk operation has finished, server looks up information on the operation and proceeds
  - Basically, a finite state machine simulating threads

Threads

- File Server Construction



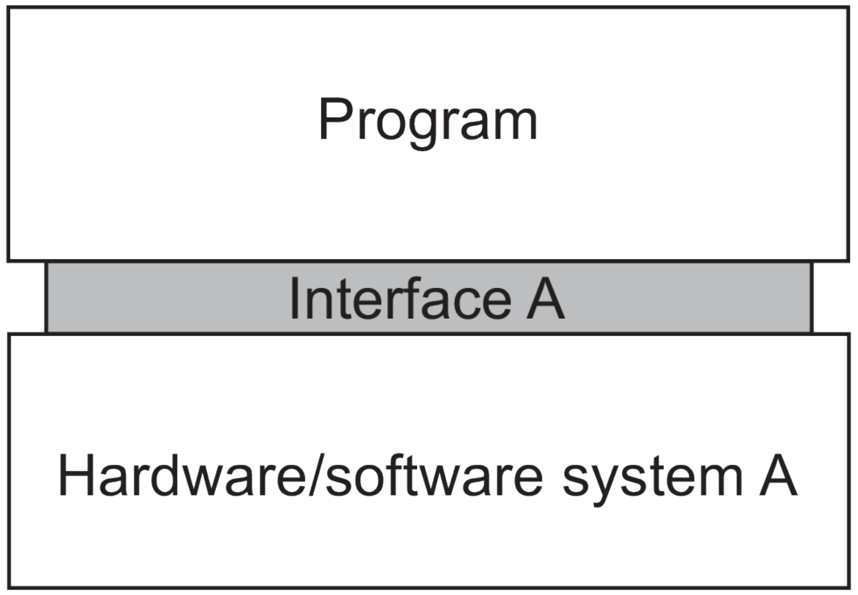
Model	Characteristics
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

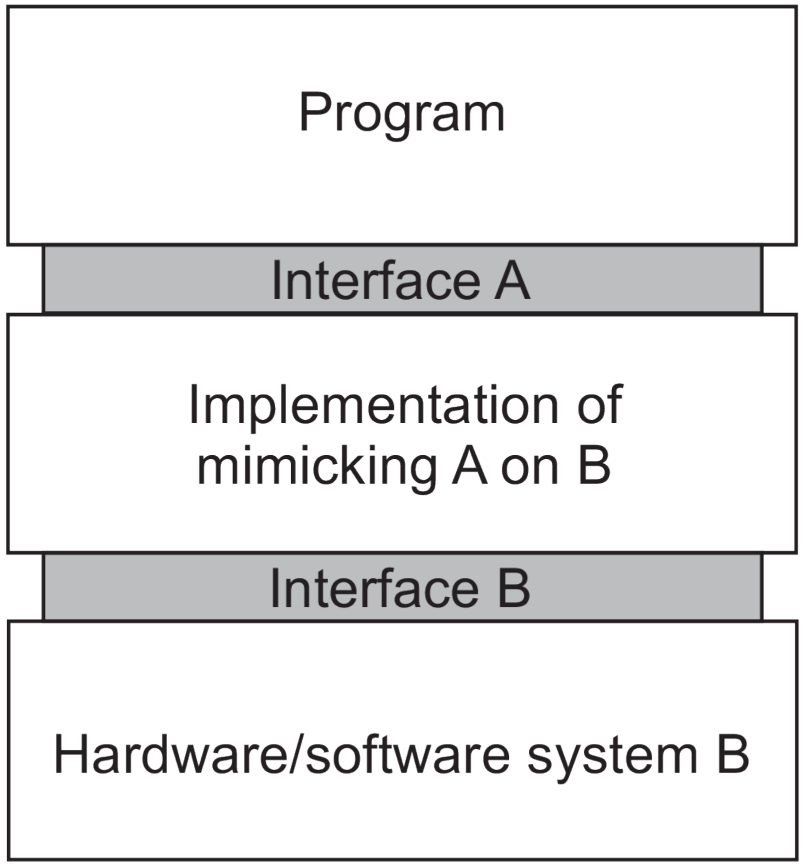
Virtualization

- Threads appear to be parallel even on a single-threaded architecture
- Virtual memory gives impression of a very large memory system
- These are instance of ***resource virtualization***
- Virtualization allows application software to outlive systems software and hardware

Virtualization

- IBM System/370:
  - Machine Virtualization: multiple “concurrent” use
  - Hypervisor / Virtual Machine Monitor (VMM):
    - Allocates and manages hardware resources



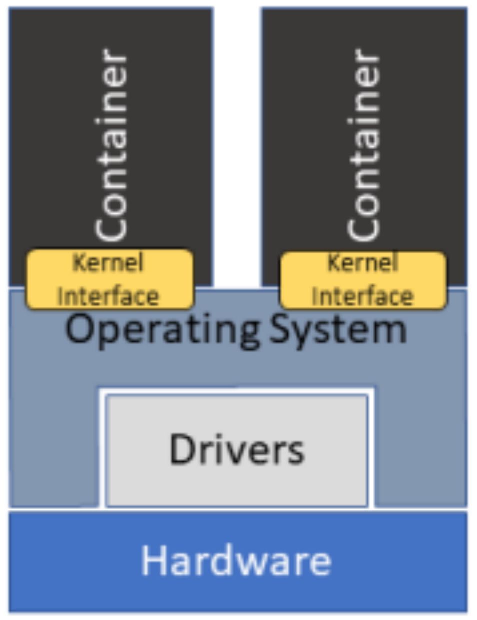

Program
Interface A
Implementation of mimicking A on B
Interface B
Hardware/software system B

Virtualization

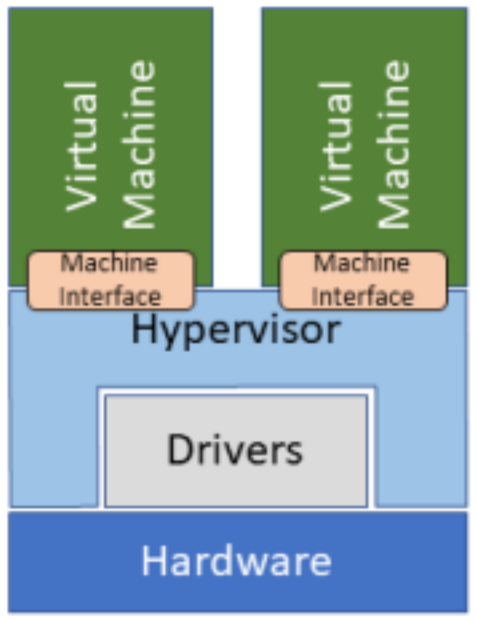
- Hypervisor implemented in software only
  - Needed tricks to interact with x86 hardware
  - Some tricks use a lot of time
    - Modify OS: *paravirtualization*
    - Guest OS is now aware of running in a hypervisor
    - Hardware vendors use hardware extensions in order to avoid the need for paravirtualization
    - But can still be used for better performance

Virtualization

- Operating System Virtualization
  - Multiple instances of an OS environment run simultaneously
    - BSD Jail, Cells, LXC, LPAR
    - Kernel provides basic services



OS virtualization

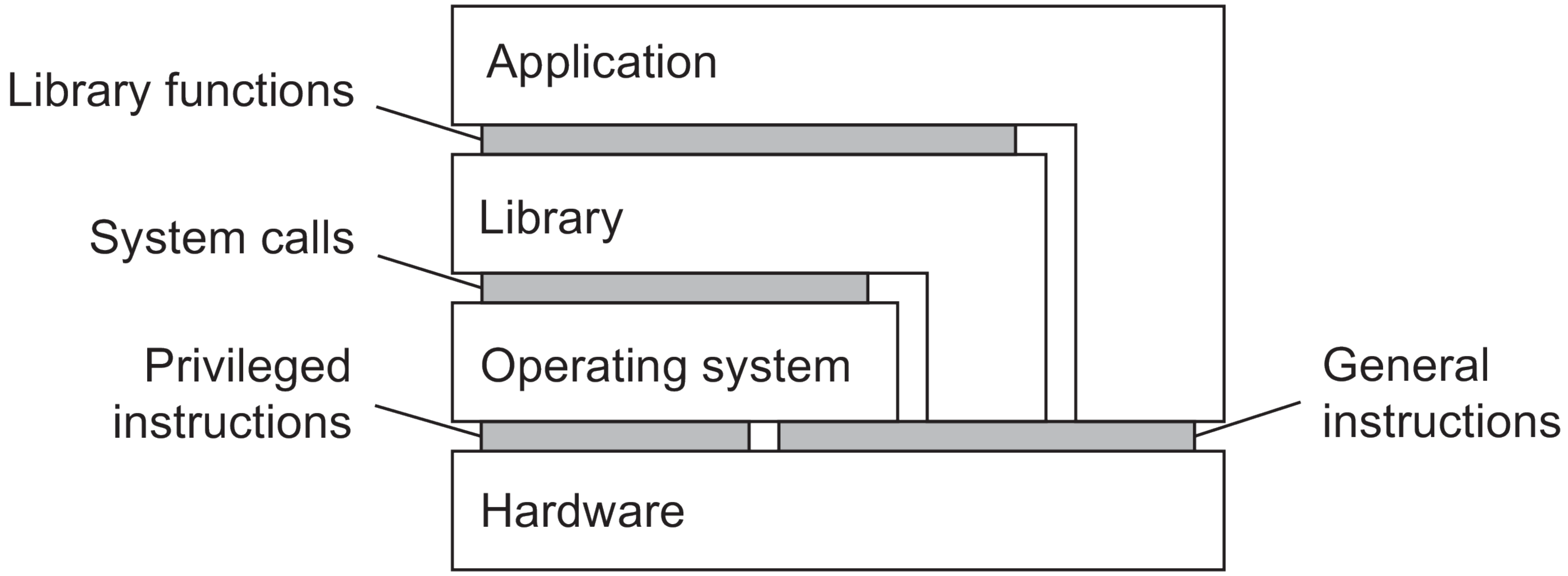


Hardware virtualization

Virtualization

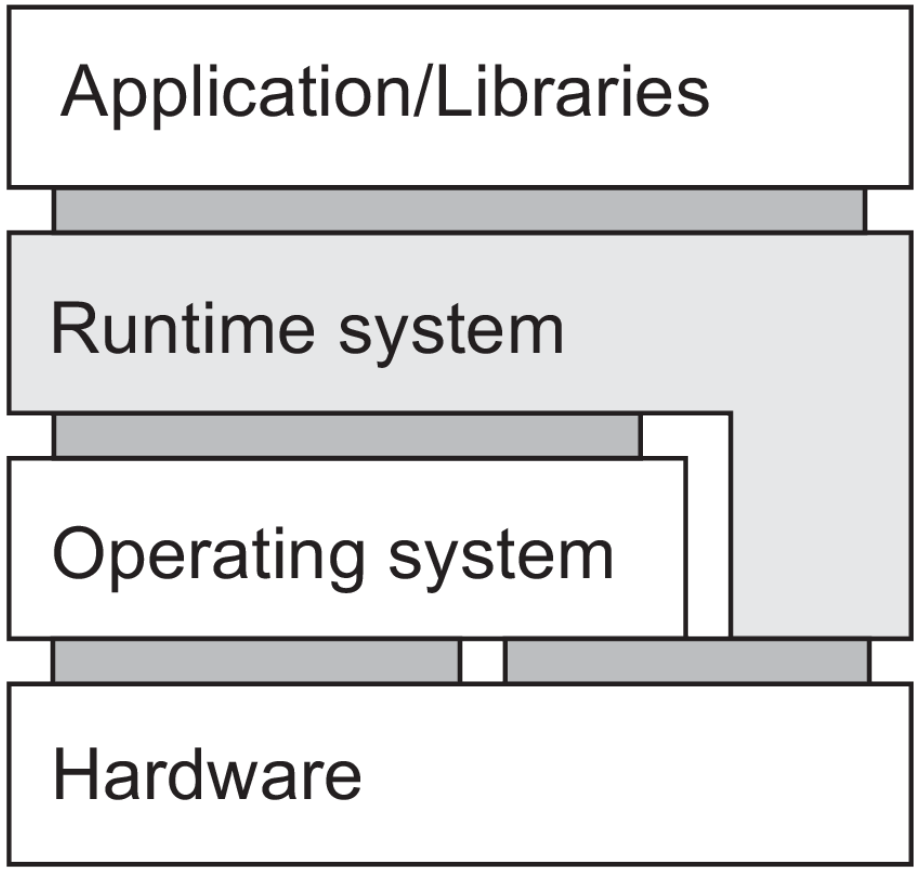
- Interfaces
    - Hardware <—> Software
        - **I**nstruction **S**et **A**rchitecture **(ISA)**
            - Privileged instructions (only available to kernel)
            - General instructions (can be executed by any programs)
    - An interface consisting of **system calls** offered by OS
    - An interface of library calls (**API**)

Virtualization



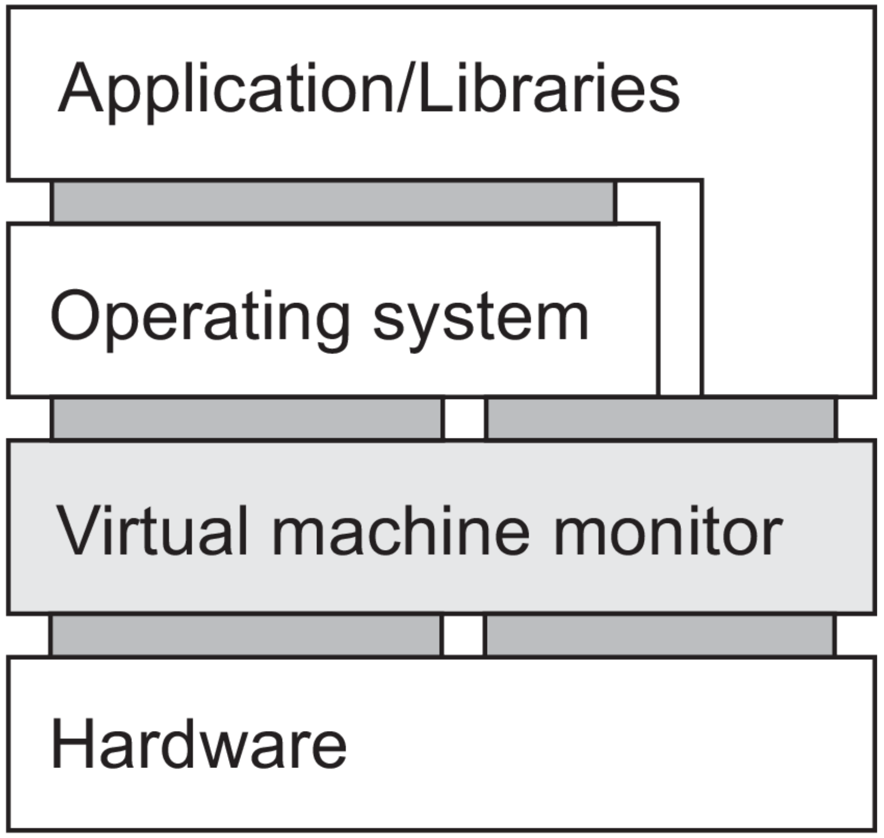
Virtualization

- Process Virtual Machine
  - provides an abstract instruction set
  - instructions can be interpreted
    - Java Virtual Machine
  - instructions can be emulated
    - e.g. running windows applications on Unix



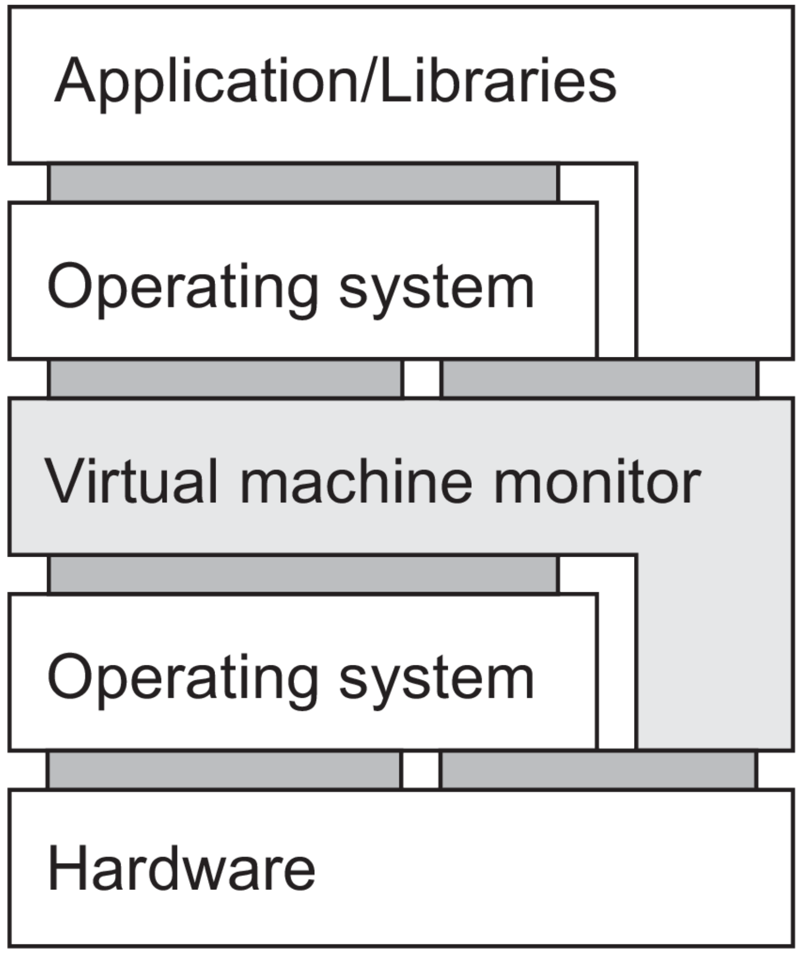
Virtualization

- Native virtual machine monitor
  - Implemented directly on hardware
  - Allows different OS to run
  - Has to provide and regulate access to various resources

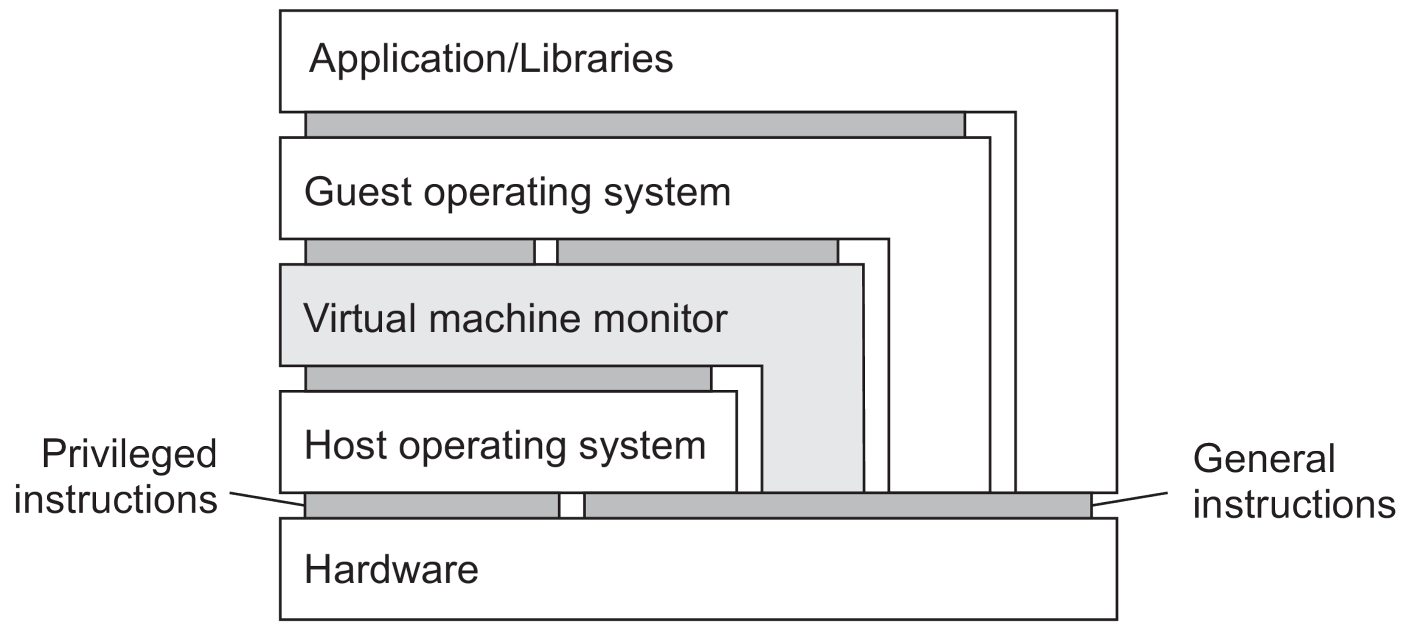


Virtualization

- **Hosted virtual machine monitor**
  - Instead of reimplementing device drivers etc. fresh
  - Use a host operating system



Virtualization



- Virtual machines perform in general well
- Large part of code runs directly on the hardware

Virtualization

- How to make virtualization fast:
  - Machine is either in user- or in system-mode
  - Some instructions are only available in system-mode
  - Memory addressing is relative
  - A ***privileged instruction*** if executed in user mode causes a ***trap*** to the OS

Virtualization

- How to make virtualization fast:
  - **Control-sensitive instruction** can affect the configuration of a machine
    - E.g. changes the memory offset, changes interrupt table, …
  - ***Behavior-sensitive instruction*** is an instruction whose behavior is determined by the context in which it is executed.
    - `POPF` instruction for Intel x86
    - sets an interrupt enabled flag, but only if executed in system mode

Virtualization

- How to make virtualization fast
  - *For any conventional computer, a virtual machine monitor can be constructed if the set of sensitive instructions is a subset of the set of privileged instructions*

  - As long as sensitive instructions are caught when executed in user mode, all non-sensitive instructions can run natively on the underlying hardware

Virtualization

- How to make virtualization fast
  - Intel x86 instruction set has 17 sensitive instructions that are not privileged
  - First solution: Emulate all instructions
    - Too slow
    - VMWare: scan the executable and insert code around the non-privileged sensitive instructions to divert control to the virtual machine monitor

Virtualization

- Second solution: ***Paravirtualization (XEN)***
  - Modifies guest OS
  - all side effects of running non-privileged sensitive instructions in user mode, which would normally be executed in system mode, are dealt with.

Virtualization / Containers

- Frequent scenario:
    - Applications are stable, but need their own environment to run
- ***Containers*** allow that

Virtualization / Containers

- Naïve version
  - Copies the entire environment
  - Install it in a subdirectory of the root file system
  - Use `chroot` to divert user into the subdirectory and run one or more applications
- Does not isolate applications running in different containers
- Copying is inefficient since many libraries, etc. are shared
- Host OS needs some control over resource usage

Virtualization / Containers

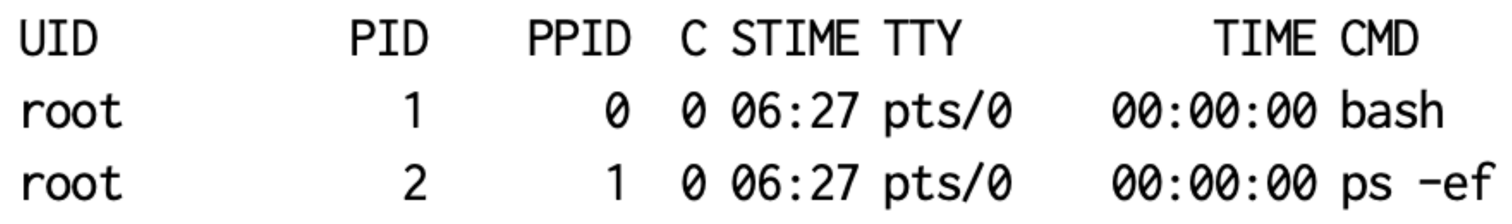
- Mechanism used:
  - Namespaces: a collection of processes in a container is given their own view of identifiers
  - Union file system: combine several file systems into a layered fashion with only the highest layer allowing for write operations (and the one being part of a container).
  - Control groups: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

Virtualization / Containers

- Namespaces
  - Give a process running inside the container the illusion that they are running on their own
    - E.g. PID
      - Every machine should have a single init process with PID 1
      - Can be done with Unix unshare command:
        ```
        unshare --pid --fork --mount-proc
        bash
        ```

Virtualization / Containers

- brings the calling process into a new shell in which the command ps -ef yields:



```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 06:27 pts/0    00:00:00 bash
root           2       1  0 06:27 pts/0    00:00:00 ps -ef
```

- New collection of processes with just one with PID 1

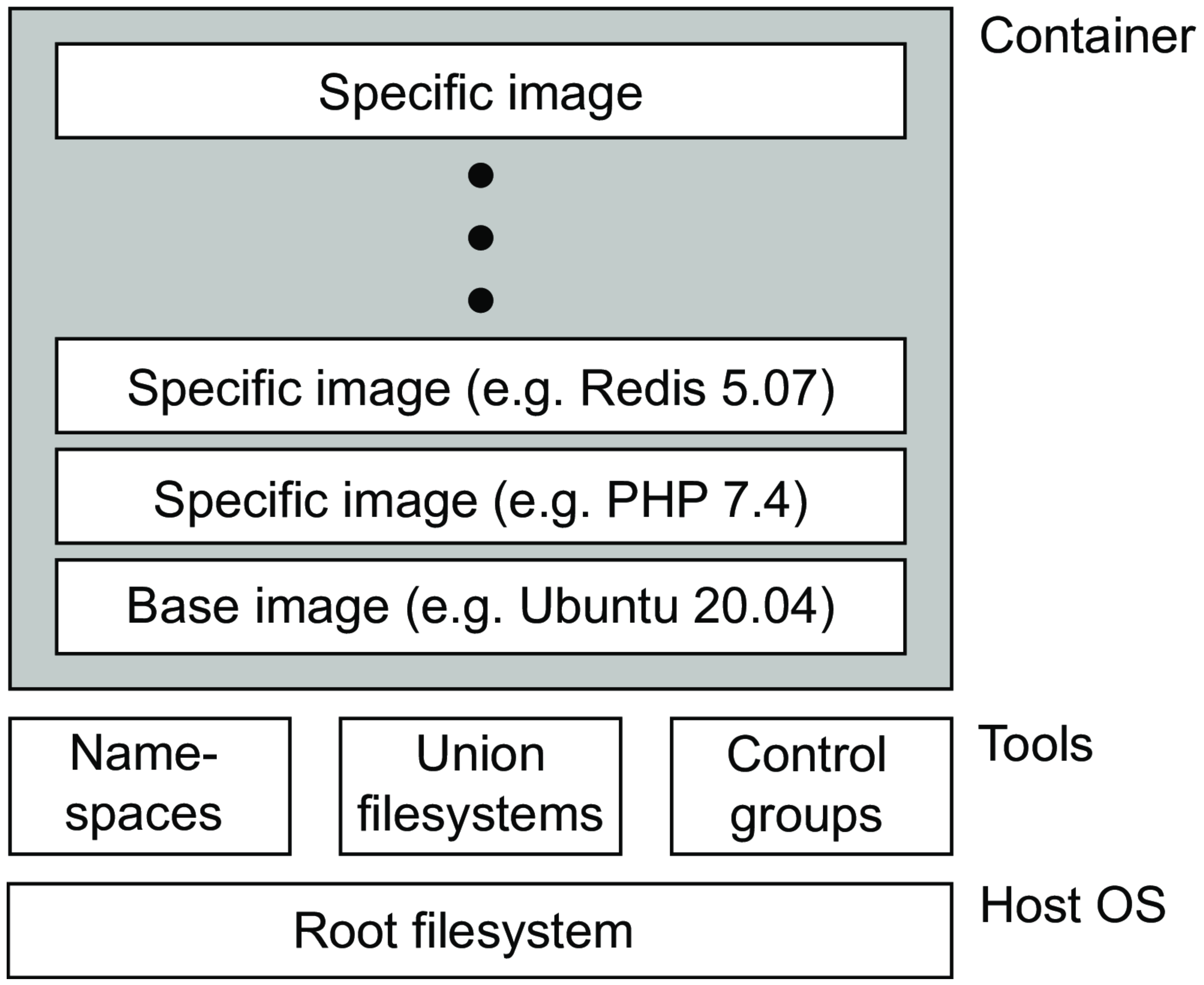
Virtualization / Containers

- Efficient sharing of existing file systems
  - Many containers have a common instance of an OS, e.g. Ubuntu 20.4
  - Use this as a base layer and stack other parts on top of it
  - For example, **replace** all the subdirectories of PHP7.4 by an older version of PHP

Virtualization / Containers

- **cgroups** control what a container uses
  - the collection of processes running in a cgroup is restricted to the amount of main memory that they can use, the priority when it comes to using the CPU, …
  - Host OS can prevent a single container from using too much of the resources

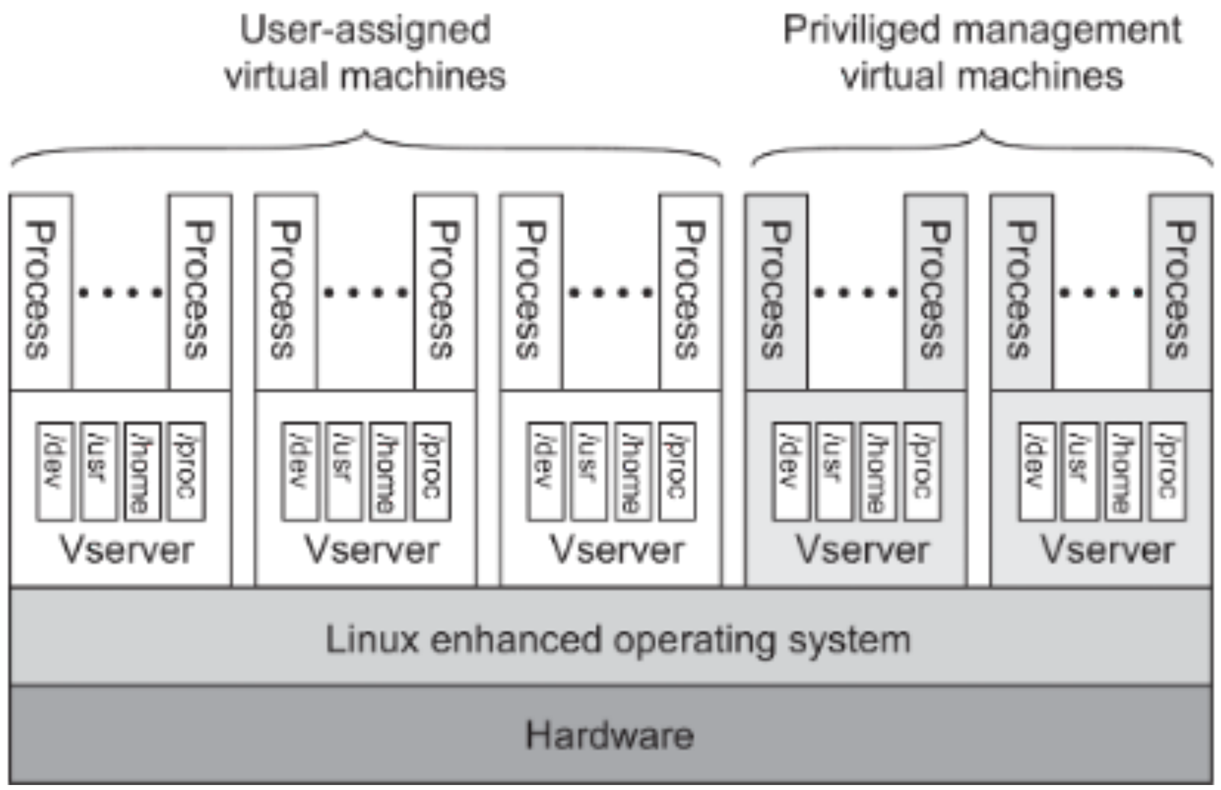
Virtualization / Containers



Example: PlanetLab

- PlanetLab was a collaborative distributed system
    - different organizations each donated one or more computers
    - adding up to a total of hundreds of nodes
    - these computers formed a 1-tier server cluster, where access, processing, and storage could all take place on each node individually.
    - Management of PlanetLab was by necessity almost entirely distributed.
    - The project closed down in 2020.

Example: PlanetLab


User-assigned virtual machines
Priviliged management virtual machines
Process
Process
/dev
/usr
/home
/proc
Vserver
Linux enhanced operating system
Hardware

- Independent and protected environment with its own libraries, server versions, and so on.

Example: PlanetLab

- **VMM**: Virtual Machine Monitor
  - enhanced Linux OS
  - Capable of supporting containers
  - Supports **Vservers**
- **Vserver** : container in execution
- VMM ensures that Vservers are separated

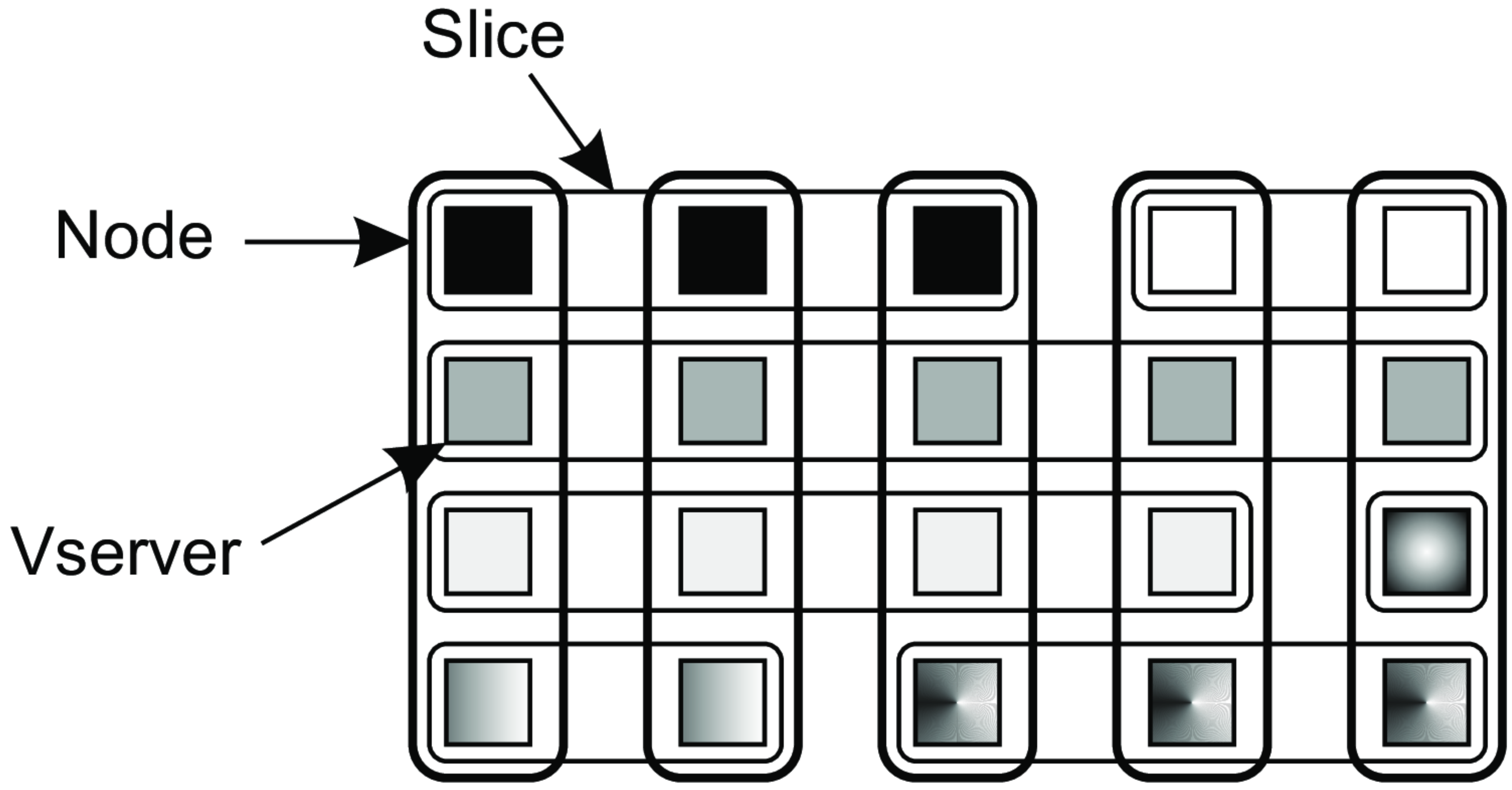
Example: PlanetLab

- PlanetLab **slices**
  - Each slice is a set of Vservers running on different nodes

Example: PlanetLab

- **Node Manager**:
  - Each node has a Vserver that creates other Vservers and monitors resource use
- **Slice Creation Service:**
  - contacts node managers to ask for a Vserver
  - only **slice authorities** could do so
  - In practice, there was only one

Example: PlanetLab



Example: PlanetLab

- PlanetLab replaced by EdgeNet
    - Uses Docker with Kybernetes instead

Virtualization / Containers

- Comparing virtual machines with containers
  - Sharma et al 2016: LXC - Linux containers vs. KVM hypervisor

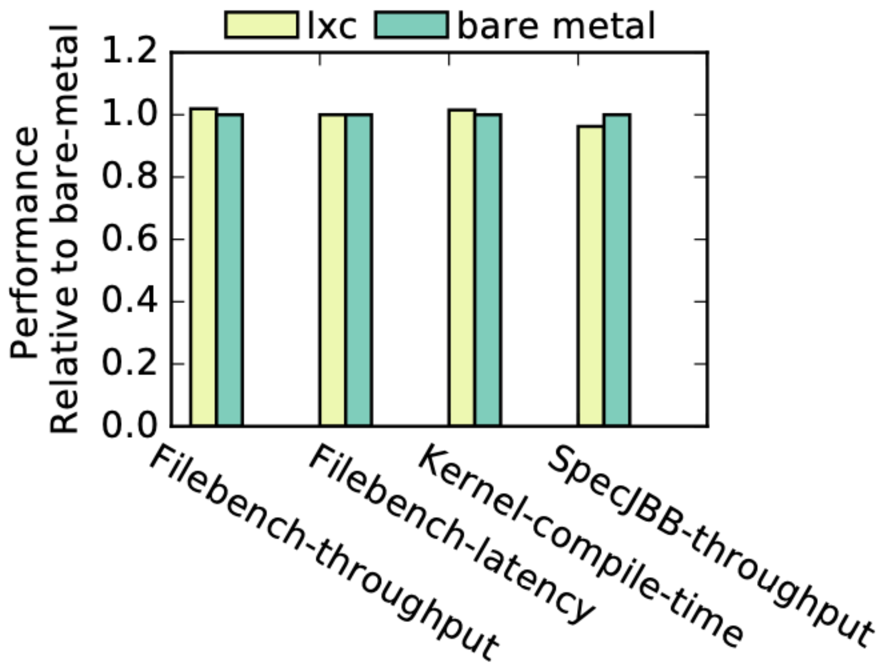
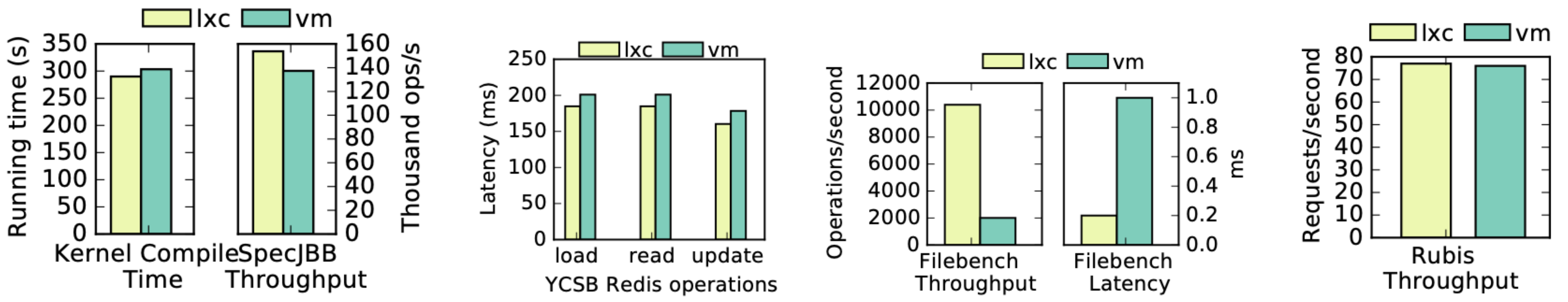


Figure 3: LXC performance relative to bare metal is within 2%.

Virtualization / Containers



(a) CPU intensive (b) Memory intensive (c) Disk intensive (d) Network intensive

Figure 4: Performance overhead of KVM is negligible for our CPU and memory intensive workloads, but high in case of I/O intensive applications. Unlike CPU and memory operations, I/O operations go through the hypervisor—contributing to their high overhead.

Virtualization / Containers

- The performance overhead of hardware virtualization is low when the application does not have to go through the hypervisor, as is the case of CPU and memory operations. Throughput and latency of I/O intensive applications can suffer even with paravirtualized I/O.

Virtualization / Containers

- van Rijn et al (2021):
  - No performance overhead for CPU-bound workloads
  - Sometimes overhead for memory-bound workloads
  - Network throughput is usually lower
  - Startup times differ

Application: Elastic Compute Cloud

- EC2:
    - Can rent Amazon Machine Images (AMI)
        - Preconfigured machine images
        - E.g. LAMP (Linux-Apache-MySql-PHP)
    - AMI is launched: Gives EC2 instance
        - In one geographical region
    -

Application: Elastic Compute Cloud

- CPU: allows selecting the number and type of core, including GPUs
- Memory: defines how much main memory is allocated to an instance
- Storage: defines how much local storage is allocated
- Platform: distinguishes between 32-bit or 64-bit architectures
- Networking: sets the bandwidth capacity that can be used

Application: Elastic Compute Cloud

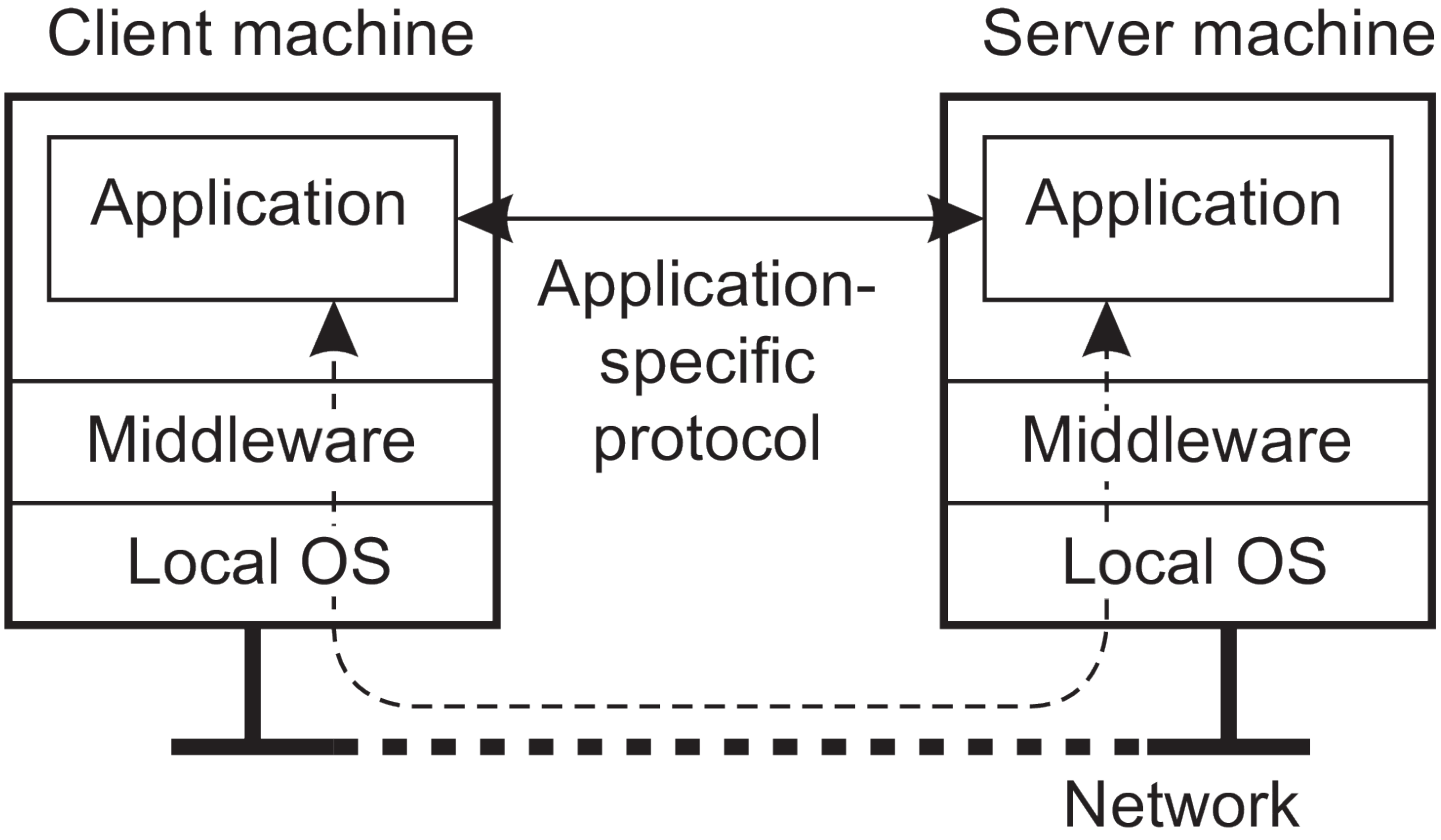
- AMI storage is transient
  - Needs to use S3 in order to store data permanently
  - Or use an Elastic Block Store (EBS)

Clients

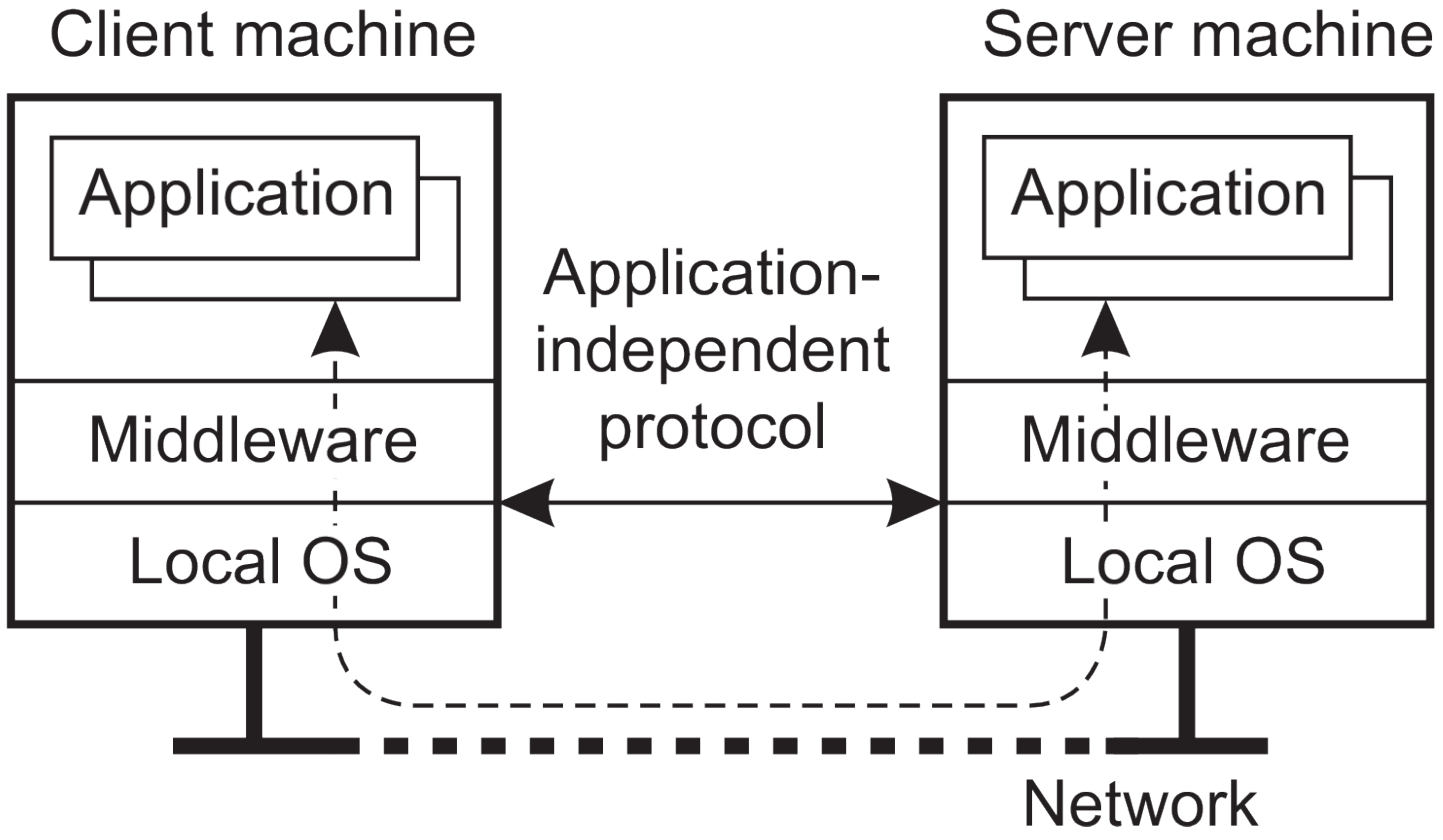
Client - Server Anatomy

- Networked user interfaces
  1. Client has a counterpart for each service
  2. Client is offered only a user interface
     - So that client is like a terminal

Client - Server Anatomy

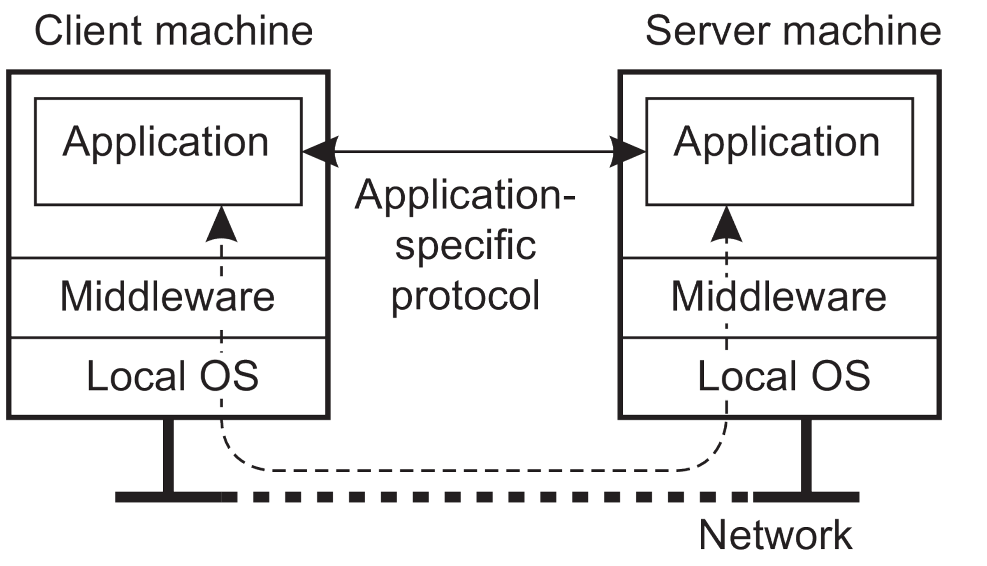


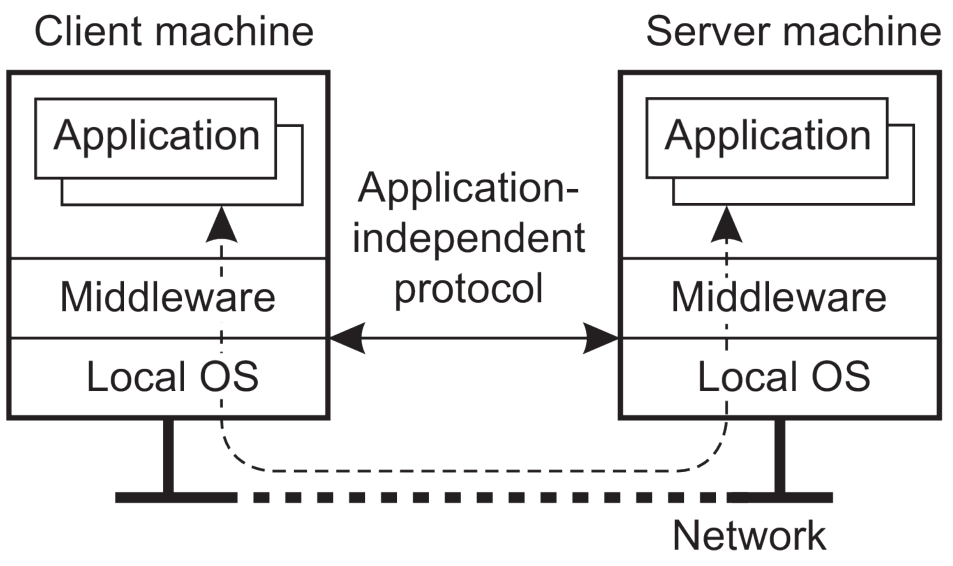
Client - Server Anatomy



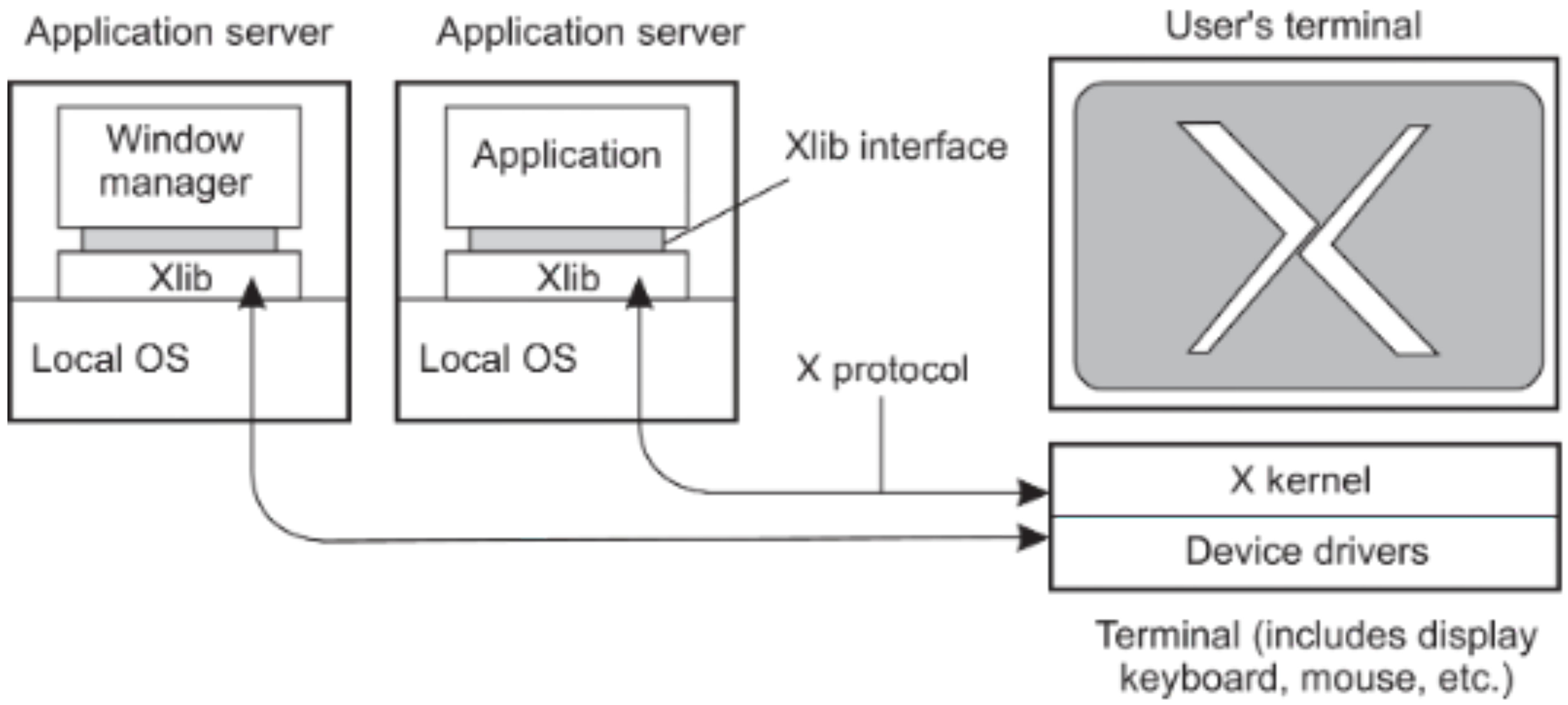
Client - Server Anatomy

- Application-level vs middleware level

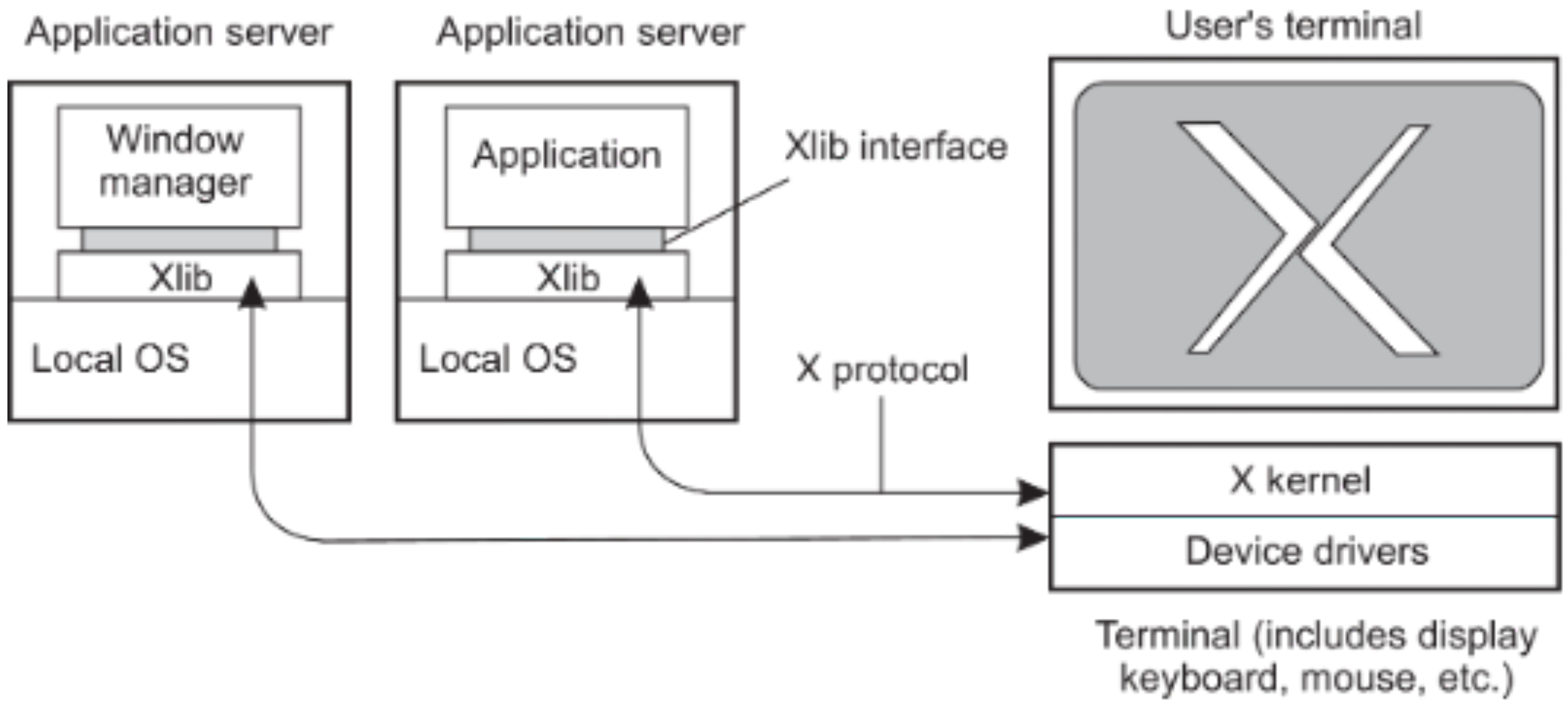




Example: X-Window system



Example: X-Window system



- 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.

Example: X-Window system

- X-kernel
  - Contains all terminal-specific device drivers
  - Interface available to applications via `Xlib`
    - which usually use toolkits on top of `Xlib`
- X-protocol
  - X-kernel and X-applications are not necessarily on the same machine
  - provides application-level protocol for exchanging data and events with the X kernel

Example: X-Window system

- 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

Example: X-Window system

- Several applications can communicate simultaneously with the X kernel
- Window manager has special rights to dictate “look and feel”
- X-kernel acts as a server - applications are clients

Example: X-Window system

- 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.

Example: Virtual Desk Top Environments

- 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

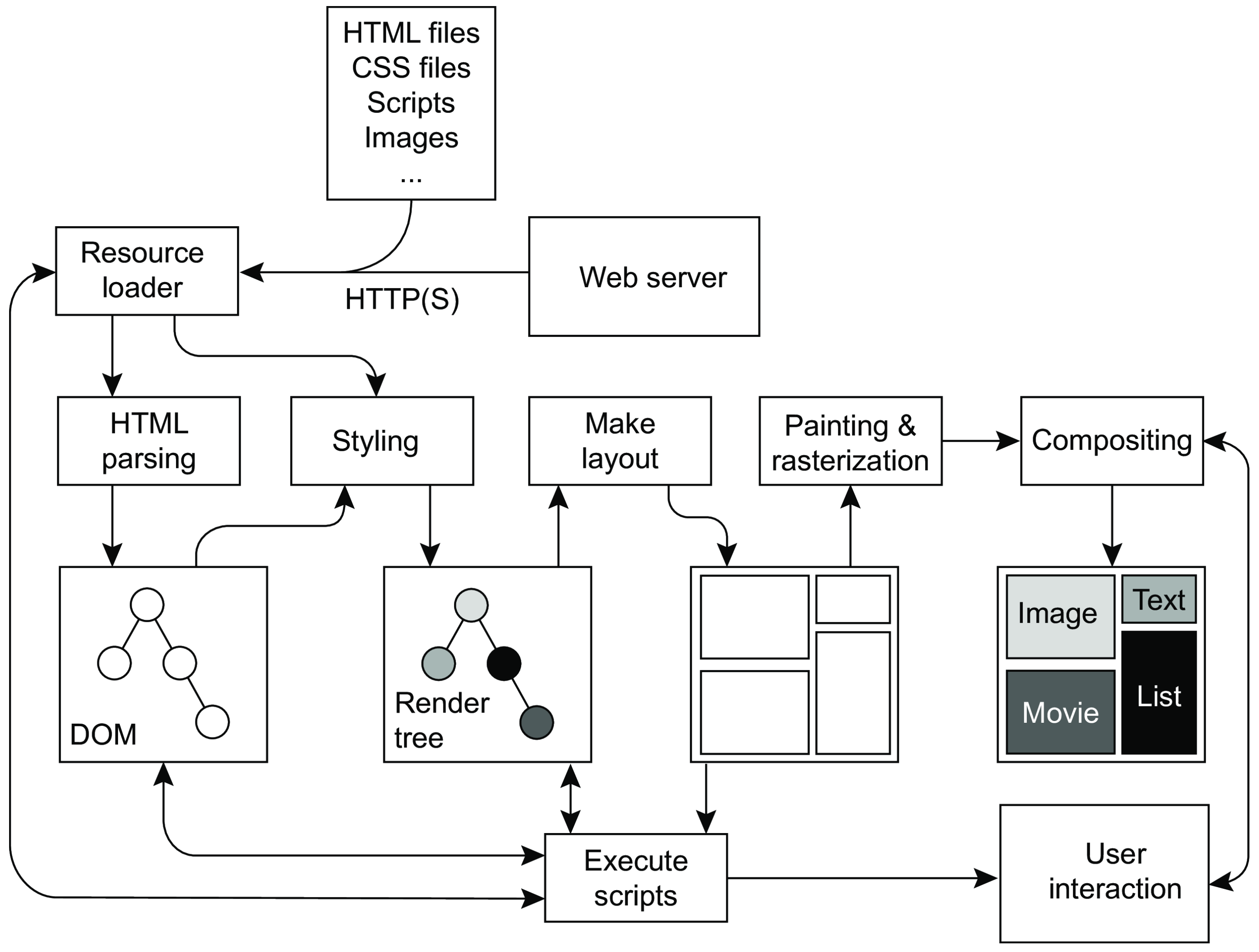
The Google Chromebook

Example: Virtual Desk Top Environments

- Google Chrome OS
  - Web-browser is the interface to applications in the cloud
  - Multiple stand-alone applications
    - Each operates with a cloud-based counterpart.
- Has been extended with Android and Chrome applications

Example: Virtual Desk Top Environments

- Chrome Browser
  - 25,000,000 lines of code



Example: Virtual Desk Top Environments

- Fetch HTML file
  - Set up parallel connections to other material referenced in this page. (Threading!)
  - Parse with **Document Object Model** (DOM)
    - Gives structure of the html file
    - Styling information is provided by a separate document
  - Now we have the layout
    - Painting component uses paint operations to process layout instruction
  - Rasterization process

Example: Virtual Desk Top Environments

- Each browser has separate components for handling scripts
  - JavaScript, WebAssembly codes

Example: Virtual Desk Top Environments

- A lot can happen in parallel
    - and even on GPU
- Code is executed in a sandbox
    - No direct communication with the underlying OS

Example: Virtual Desk Top Environments

- Browser can be sufficient for offering a virtual desktop environment
- But usually have to interact with local resources:
    - Media: camera, microphone, speakers, …

Example: Virtual Desk Top Environments

- Applications can run natively on the computer hosting the client-side desktop
  - E.g. Smartphones and mobile apps
    - Can be used offline
- **Progressive Web Apps (PWA)**
  - Use browser as their hosting environment
  - Yet appear as ordinary mobile app
    - Move much of server-side content not dependent on network connectivity to the client, where it is cached

Example: Virtual Desk Top Environments

- Current Trend:
  - Moving from very thin clients to hosts with
    - much more functionality
    - communication over Internet is optimized
- Compared to fat clients:
  - browser based apps are either to manage as servers can simply upload new parts

Client-Side Software for Distribution Transparency

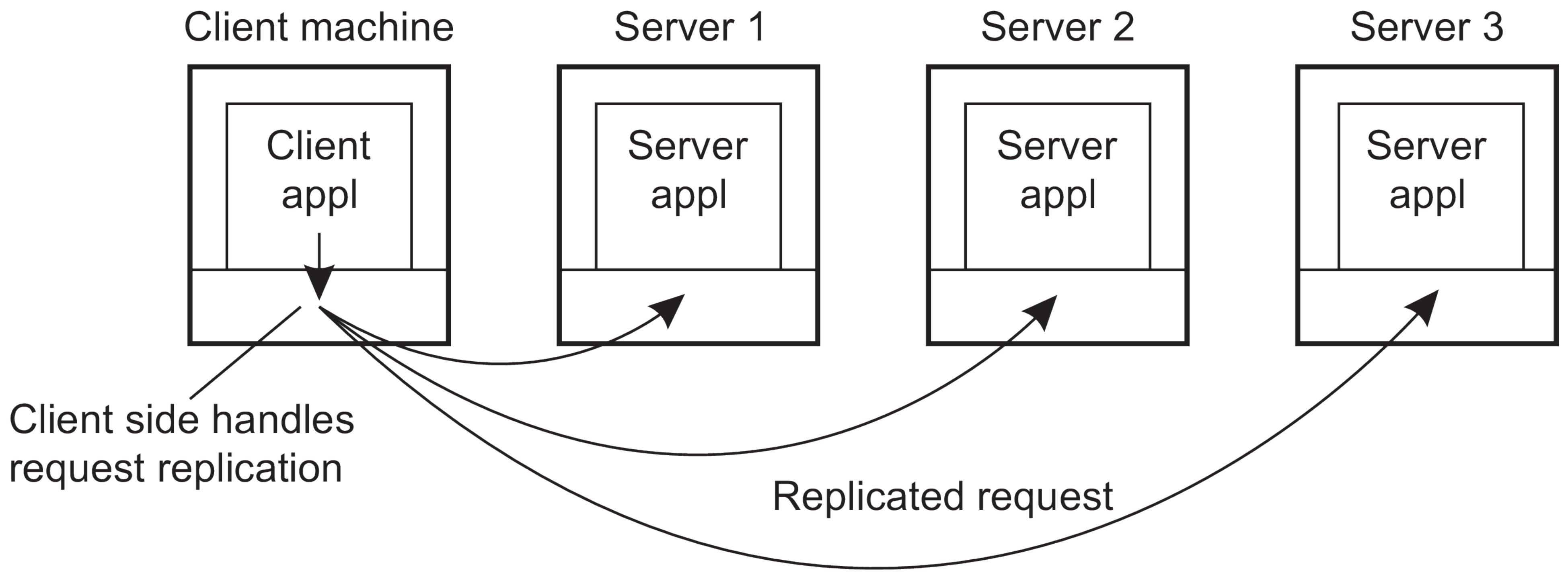
- Client software is more than just user interfaces
    - Processing and Data Level are also executed on the client side
        - E.g.: Embedded Client Software:
            - ATM, cash register, barcode reader, TV set-top box
            - User interface is a small part compared to local processing and communicaton

Client-Side Software for Distribution Transparency

- Client software has components to achieve **distribution transparency**
  - **Access Transparency:**
    - Generate a client stub from an interface definition of what the server has to offer
  - **Location, Migration, and Relocation Transparency:**
    - Multitude of architectures
    - Naming and cooperation with client-side software

Client-Side Software for Distribution Transparency

- **Replication Transparency**



Transparent replication of a server using a client-side solution

Client-Side Software for Distribution Transparency

- **Failure Transparency**
  - client middleware
- **Concurrency Transparency**
  - Special intermediate servers
    - transaction monitors

Servers

- 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.

Servers

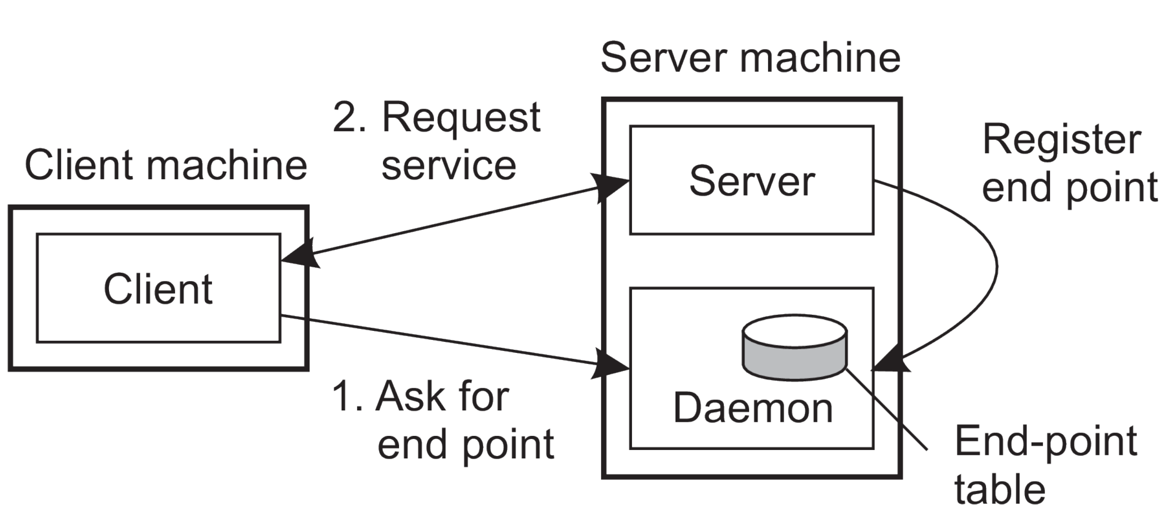
- Concurrent versus iterative servers
    - Iterative server: Server handles the request before attending a next request.
    - Concurrent server: Uses a dispatcher, which picks up an incoming request that is then passed on to a separate thread/process.
        - multi-threading
        - forking a new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

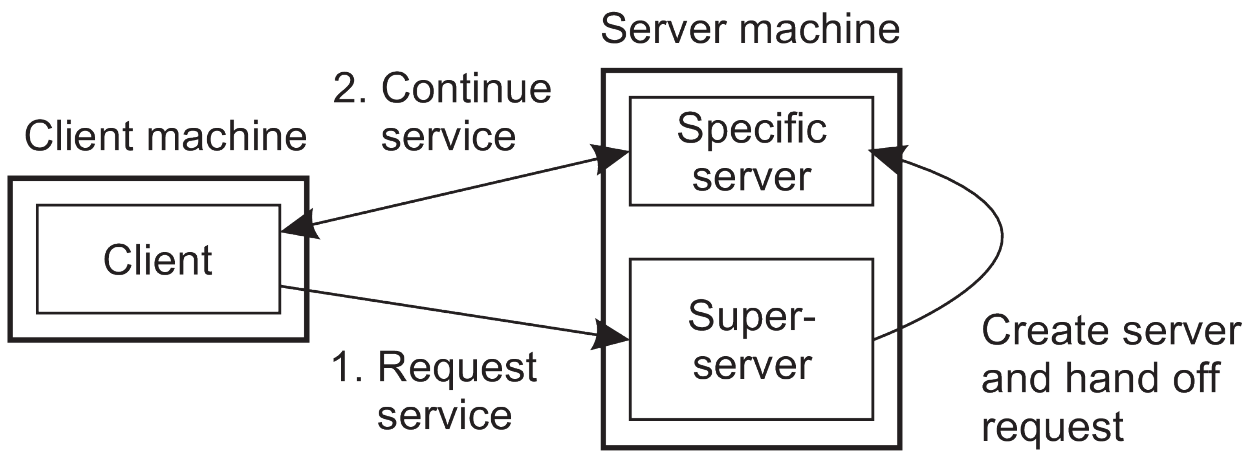
Servers

- Contacting a server: End Points / Ports
  - Each server listens to a specific end point
  - How to find it?
    - Most services tied to a specific port
      - ftp-data 20
      - ftp 21
      - telent 23
      - smtp 25
      - www 80

Servers

- Dynamically assigning an end point





Servers

- `inetd` (internet service daemon) is a super-server daemon
    - Listens on designated ports
    - Requests are (usually) served by spawning a process
    - Configured in /etc/inetd.conf
- Replaced by xinetd, …

Servers

- 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

Servers

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

- Issue
    - Stateless versus stateful servers

- 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

- Stateless servers
- Never keep accurate information about the status of a client after having handled a request:
- 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)

Servers

- **Soft State**
  - Server promises to maintain state for a limited time
  - Afterwards server falls back on default behavior

Servers

- 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

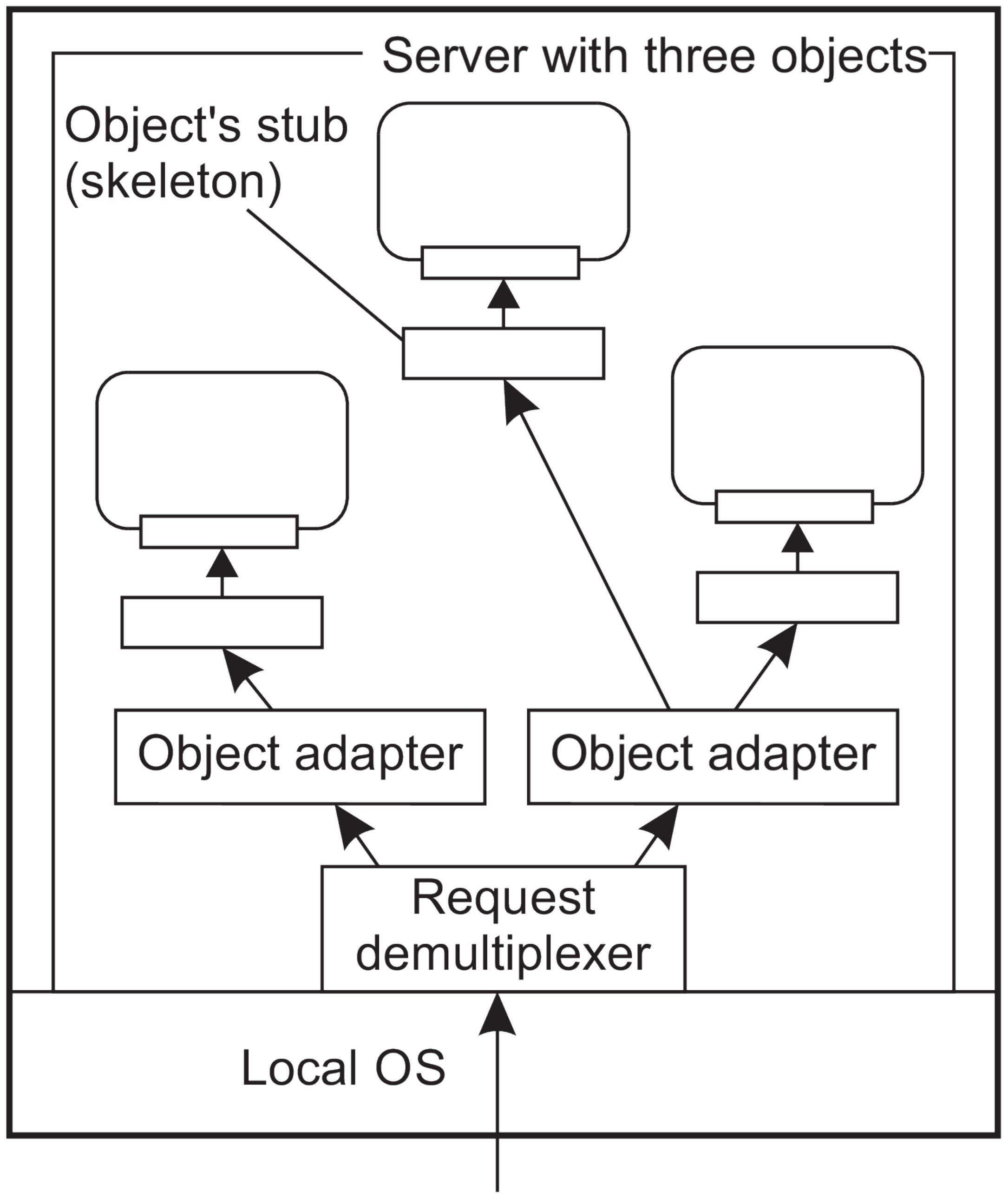
Servers

- Stateful Servers
- 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.
- Problem:
  - If a server crashes
    - Needs to recover its entire state

Object Servers

- Distributed objects need object servers
  - Object servers are a place where objects live
  - Object server does not offer service, only allows objects to offer them
- Objects are made up of data (its state) and code
  - Object servers can be multi-threaded and each object can be assigned a different thread or a separate thread may be used for each invocation request

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

- Invocation
- Simple approach:
    - All objects look alike and there is only one way to invoke an object
- A better approach: a **transient** object:
    - Object exists only as long as the server exists or even shorter
        - Advantage: Only active objects consume resources
- A better approach: Each object is places in a memory segment of its own
    - Objects do not share data or code
    - Objects share their code

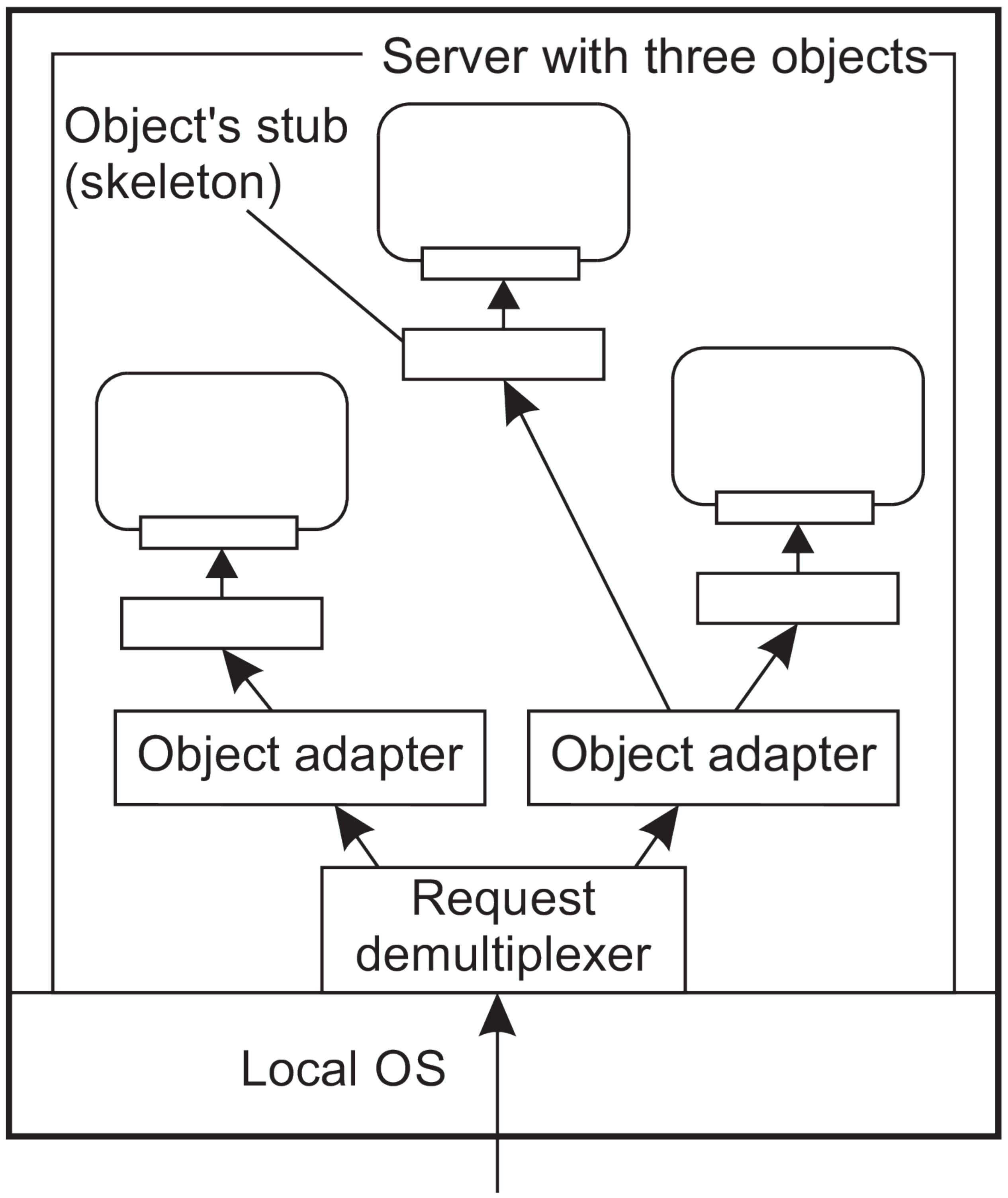
Object Servers

- Threading policies
  - Simplest: Server has only a single thread
  - Server has different thread for each object
  - Use thread for each invocation request

Object Servers

- **Activation policies**
  - Often the object itself must be brought into the server’s address space (activation)
  - Need to group objects per policy:
  - **Object adapter (wrapper)**
    - Software implementing a specific activation policy
    - Object adapter are unaware of the specific interfaces of the objects they control
    - When an invocation arrives, request is dispatched to the appropriate object adapter

Object Servers



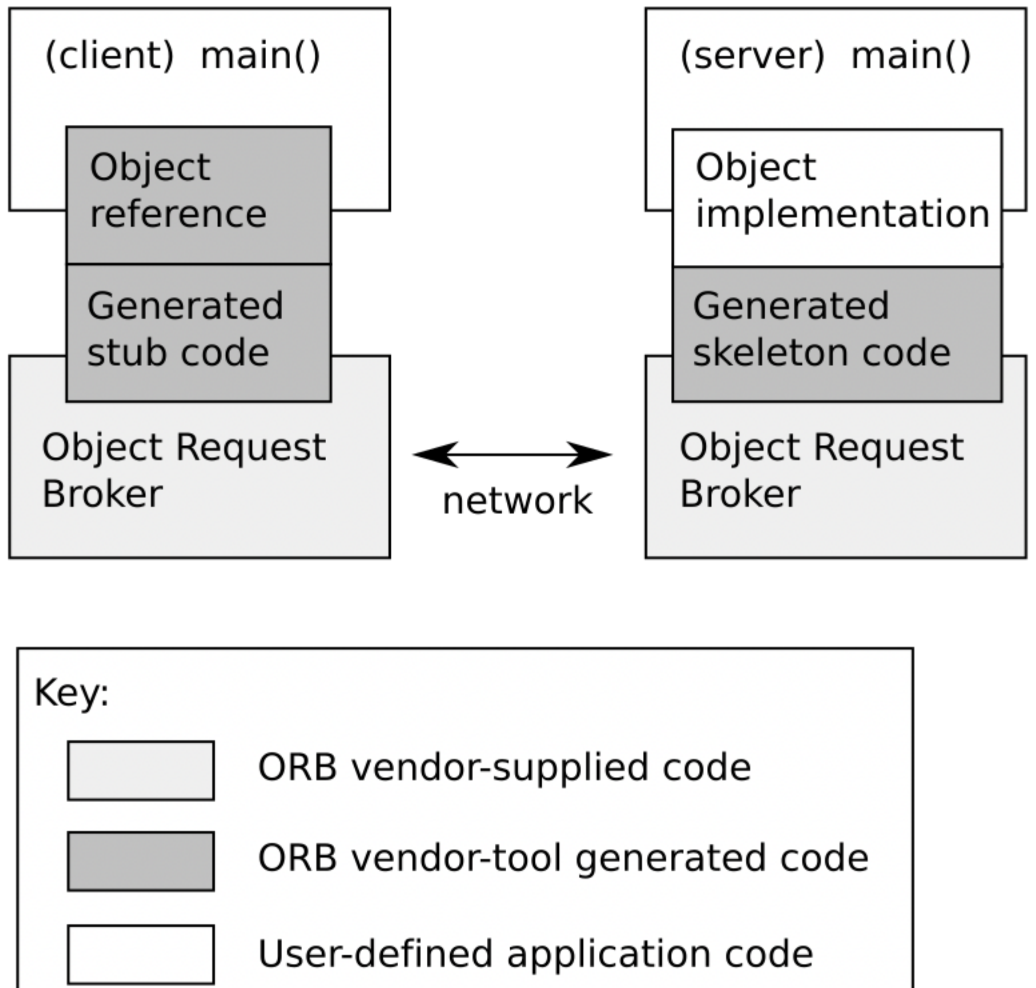
An object server supporting three different activation policies

Object Servers

- Object adapter can support different activation policies by simply configuring it at runtime
  - In Corba, can decide:
    - should object continue to exist after its associated adapter has stopped?
    - adapter or application generate object identifiers?
    - operate as a single thread or with multiple threads

CORBA

- Common Object Request Broker Architecture (CORBA) (1991)
  - Enable communication between software written in different languages and running on different hardware platform
  - Uses an Interface Definition Language (IDL) and Object Request Brokers (ORB)



ICE runtime system

- Internet Communications Engine (ICE) implements object servers (2004)
  - ICE uses communicators:
    - Communicator manages basic resources:
      - Pool of threads, allocated DRAM, …
    - Configures environment
  - Use the communicator to create an object adapter

Ice runtime system

```
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 1100
object1 = PrinterI("Object1 says:")
object2 = PrinterI("Object2 says:")
adapter.add(object1, communicator.stringToIdentity("SimplePrinter1"))
adapter.add(object2, communicator.stringToIdentity("SimplePrinter2"))
adapter.activate()

communicator.waitForShutdown()
```

(a)

Create and initialize runtime
Returns communicator
Returns Object Adapter

Object Adapter listens on 11000

Can now create objects
Object Adapter will guide
communication to the correct object

```
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)
```

```
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 110
object1 = PrinterI("Object1 says:")
object2 = PrinterI("Object2 says:")
adapter.add(object1, communicator.stringToIdentity("SimplePrinter1"))
adapter.add(object2, communicator.stringToIdentity("SimplePrinter2"))
adapter.activate()

communicator.waitForShutdown()
```

Server code

(a)

```
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()
```

Client code

(b)

Ice runtime system

- Adapters can support many objects
  - Can dynamically load objects in memory when needed
    - This uses a *servant*, acting as a locator

Example: Apache Web Server

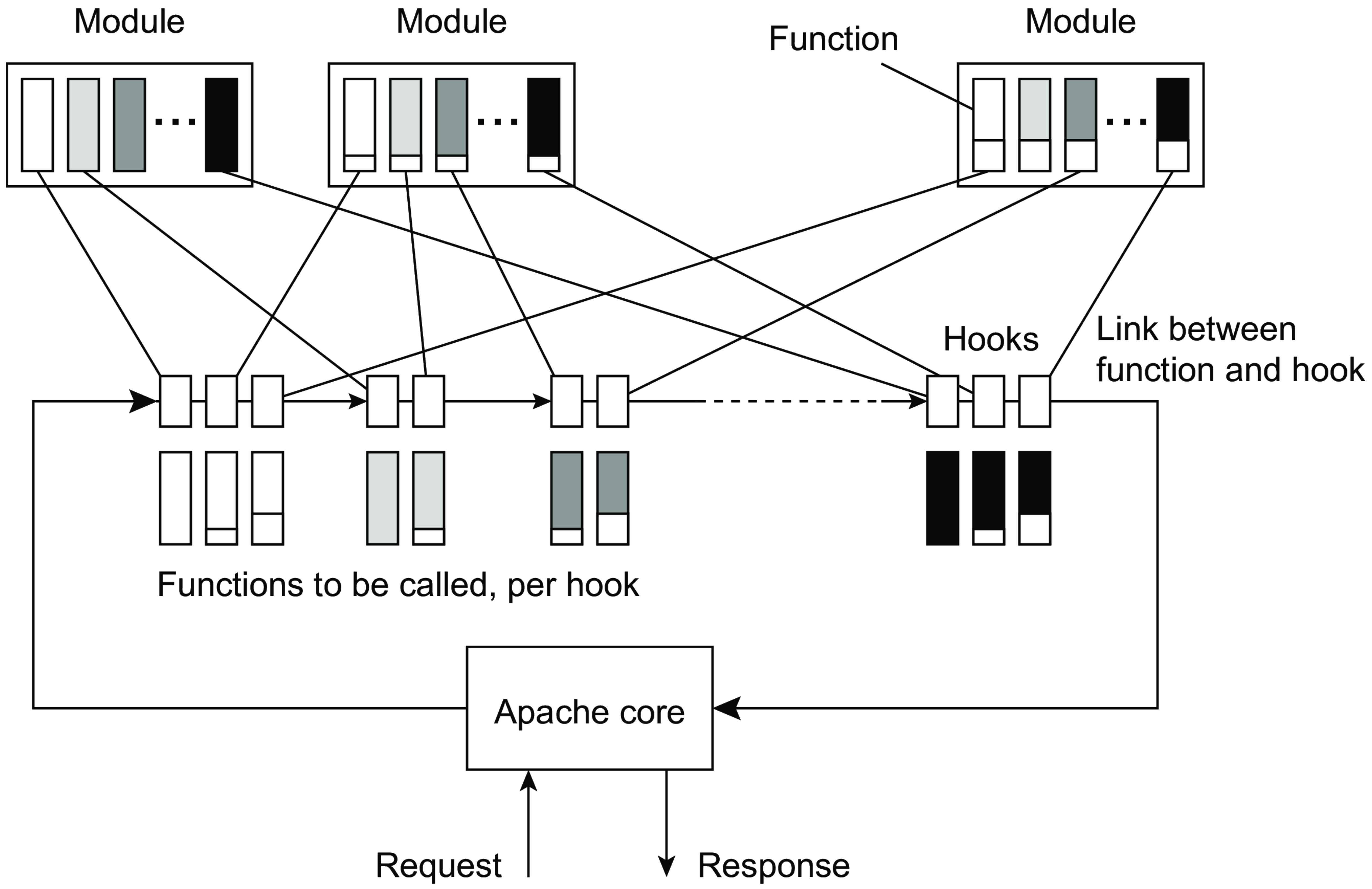
- Popular (21% market and going down)
  - Example of balancing separation of policies and mechanisms
    - Platform independent by building its own basic runtime environment
      - Which is then ported to the OS
      - **Apache Portable Runtime (APR)**
      - Library that provides
        - platform- independent interface for file handling, networking, locking, threads, …

Example: Apache Web Server

- AWS is a general server tailored for dealing with HTTP requests over TCP
- Organized in modules with several functions each:
    1. How do we ensure that a function is called in the first place?
    2. How do we ensure that a function is called at the right moment?
    3. How do we prevent that a function is called for a request it was not supposed to handle?

Example: Apache Web Server

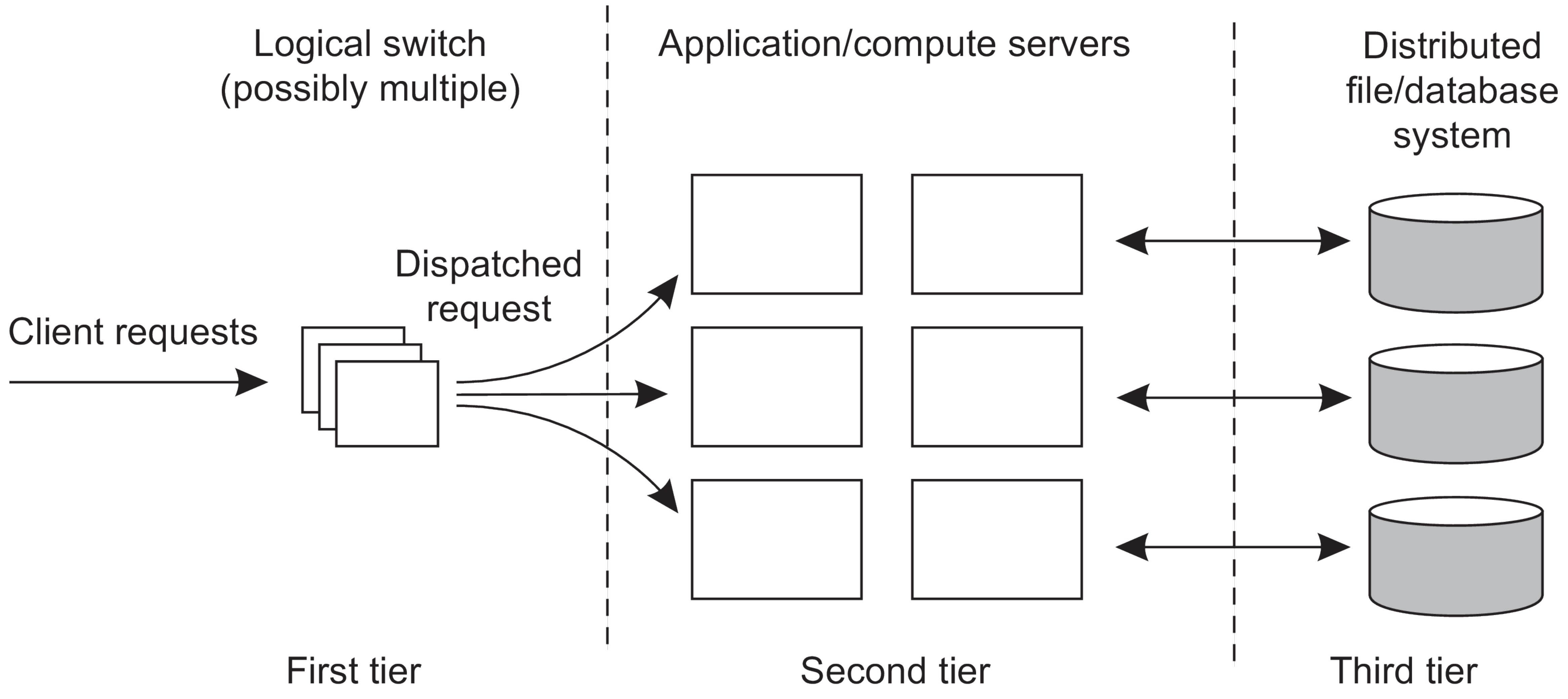
- 3. Functions can return a value DECLINED if the request was not meant for it
- 1. Uses **hook**s, placeholders for functions
  - Each module provides the core with a list of hooks
  - We statically or dynamically link modules to the Apache core
- 2. Because functions can sometimes depend on each other:
  - Can specify whether function should be called in the beginning, middle, or end phase
  - Can specify before and after of function calls


Module
Module
Function
Module
Hooks
Link between function and hook
Functions to be called, per hook
Apache core
Request
Response

Server Clusters

- Local Area Cluster
  - with high bandwidth and low latency
  - First Tier:
    - Consists of a switch to send requests to the second tier
      - E.g.:
        - Transport layer switch accepts TCP connections and passes requests to one of the servers
        - Web servers accepts HTTP requests, but passes partial requests to application servers, and later collect results in order to generate an HTTP response

Server Clusters



Server Clusters

- Architectures vary
  - E.g. streaming media use a two-tiered system
    - Each machine in second tier acts as a dedicated media server
  - E.g. Hardware can be tailored to types of requests

Server Clusters

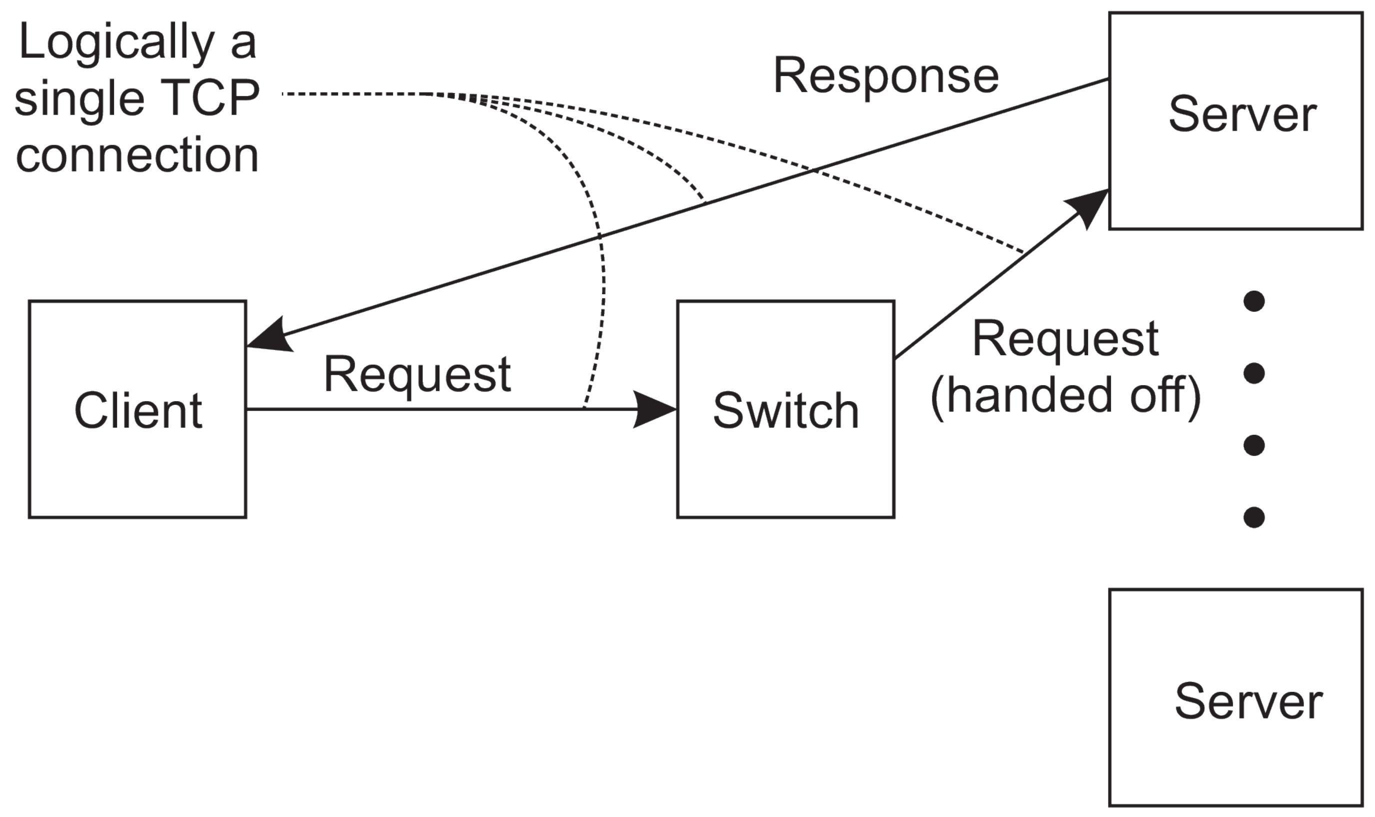
- Request dispatching (at the front end)
    - **Transport Layer Switches**
        - Accept TCP connection requests and hands them off to one of the servers
        - Server then stays in the middle of the TCP connection
            - Works like Network-Address Translation

Server Clusters

- Request dispatching (at the front end)
  - **Application Layer Switches**
    - E.g. Webserver inspects URL
      - Depending on URL, gates request to a specialized server
    - nginx servers can handle thousands of simultaneous connections
      - Works as a **reverse proxy**

Server Clusters

- TCP Handoff



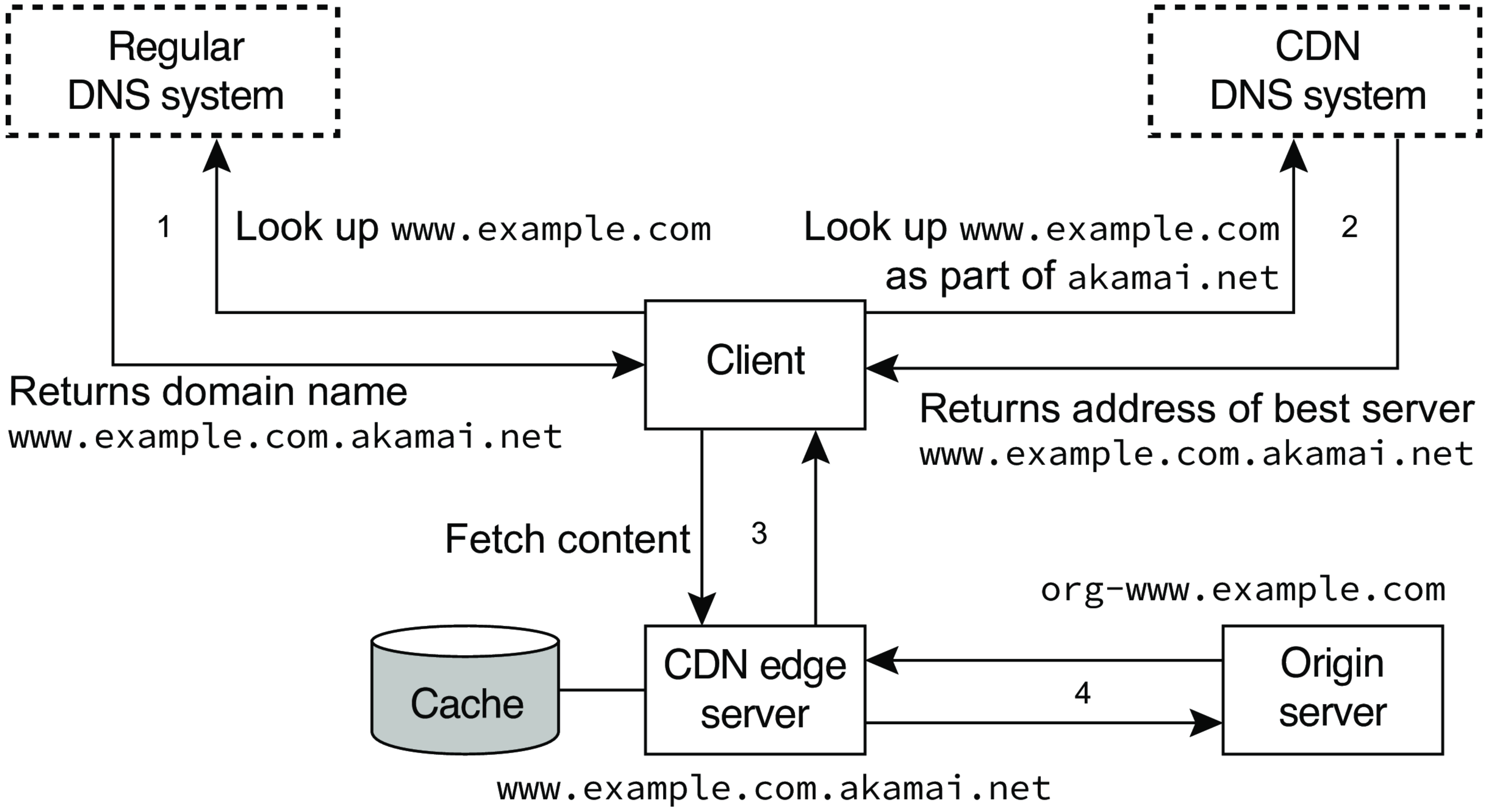
Server inserts switch's IP in the source field of the acknowledgment
Client sees communication only with switch

Server Clusters

- Wide area clusters
  - **Content Delivery Networks** (CDN)
    - Let clients choose close servers
    - Use DNS

Example: Apache Web Server

- Akamei:


Regular DNS system
CDN DNS system
1
Look up www.example.com
Look up www.example.com as part of akamai.net
2
Client
Returns domain name www.example.com.akamai.net
Returns address of best server www.example.com.akamai.net
Fetch content
3
org-www.example.com
Cache
CDN edge server
4
Origin server
www.example.com.akamai.net

Example: Apache Web Server

- Akamai CDN:
  - There is an **origin server** accessible by the domain name
  - Domain name needs to be resolved via DNS

Example: Apache Web Server

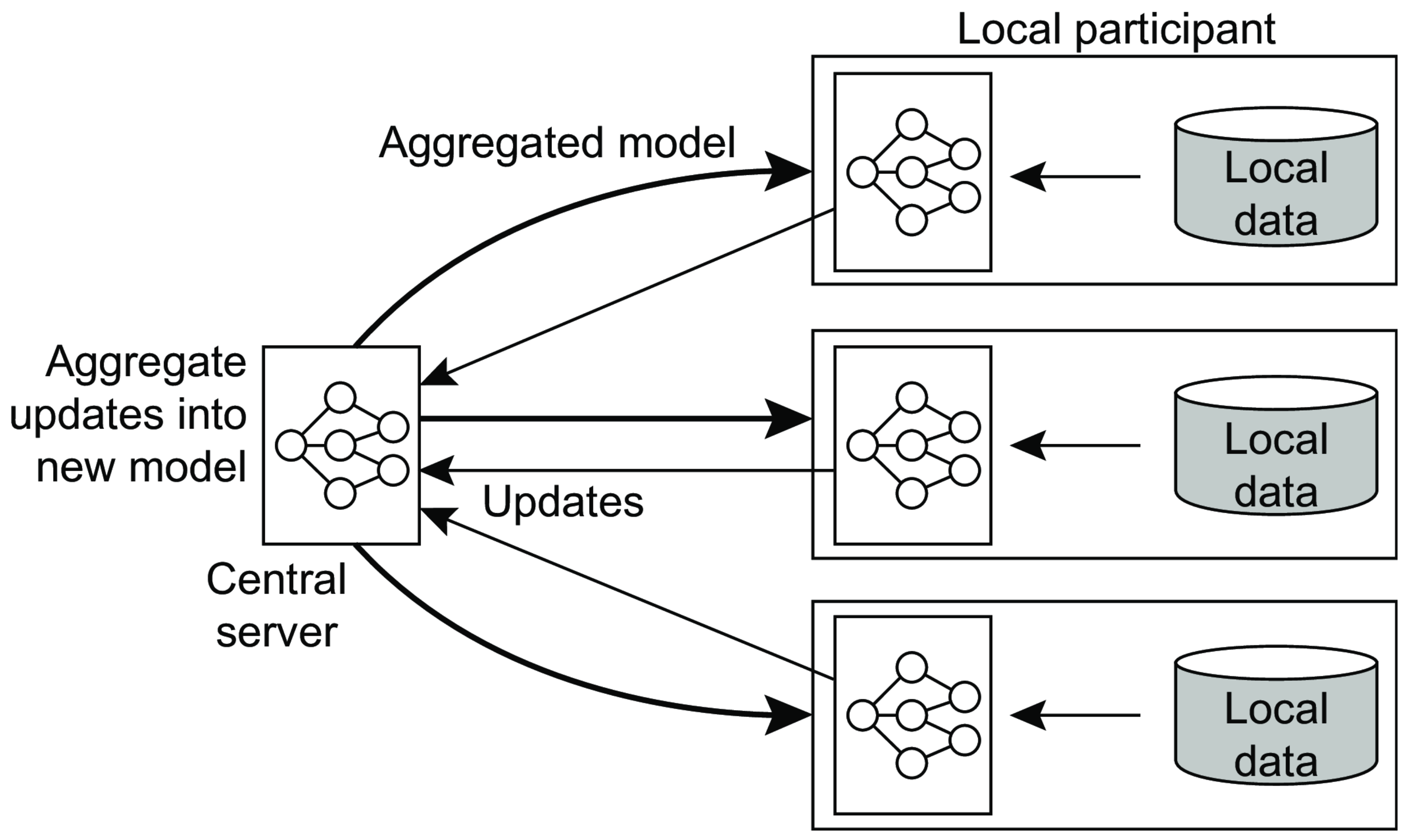
- Request dispatching
    - Redirection techniques
        - TCP handoff
        - DNS redirection
        - HTTP redirection:
            - Client is given an alternative URL in the HTTP response message

Code Migration

- Code migration:
  - Process migration: Entire process is moved from one machine to another
    - Very costly
  - Why do it?
    - Performance
      - Offload and sufficiently load machines
      - Move code close to the data
      - Move code to the client
      - Use a **mobile agent**

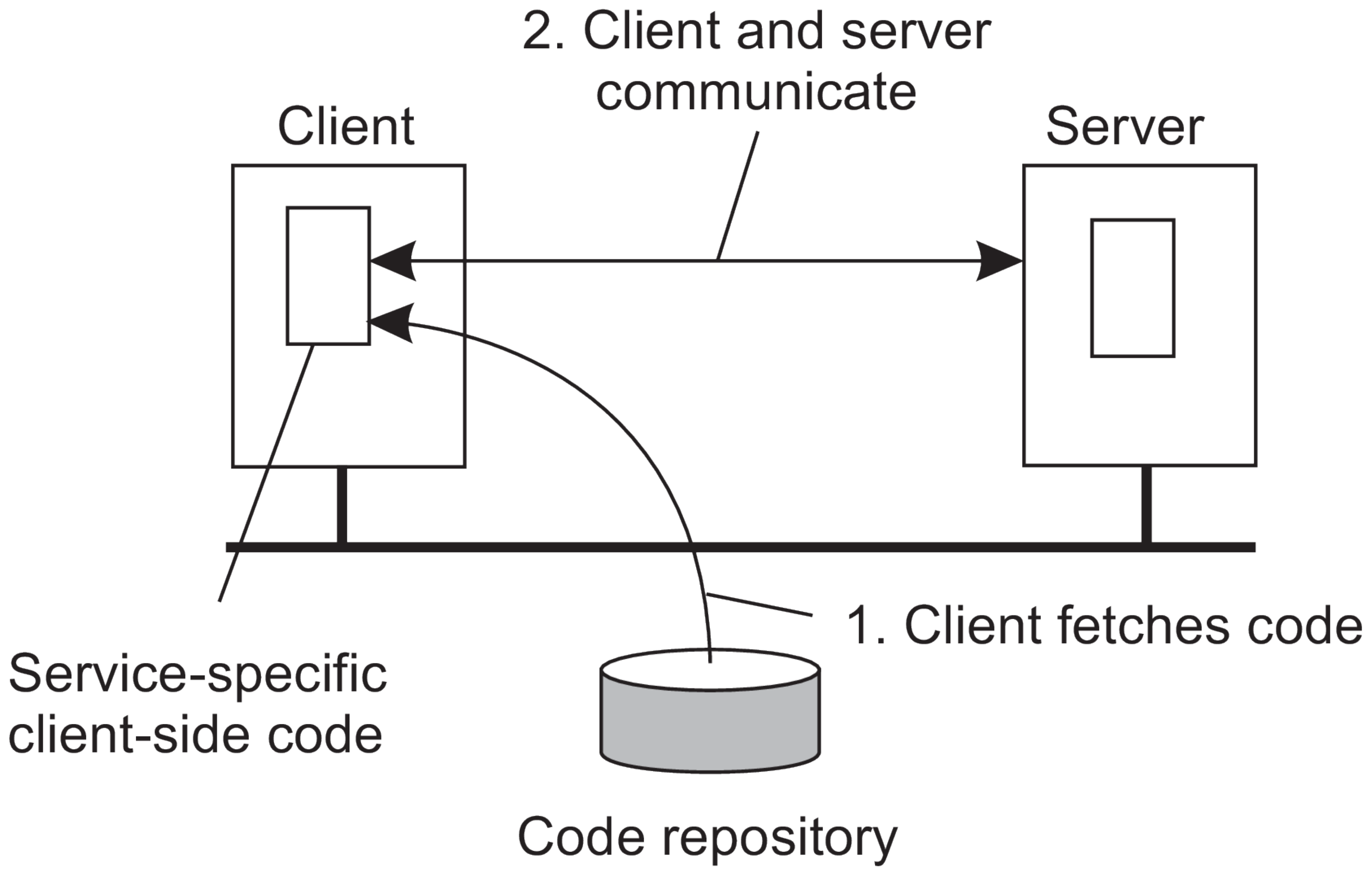
Code Migration

- Code migration:
  - Privacy and security
    - E.g. Federated Learning needs to move code to the data

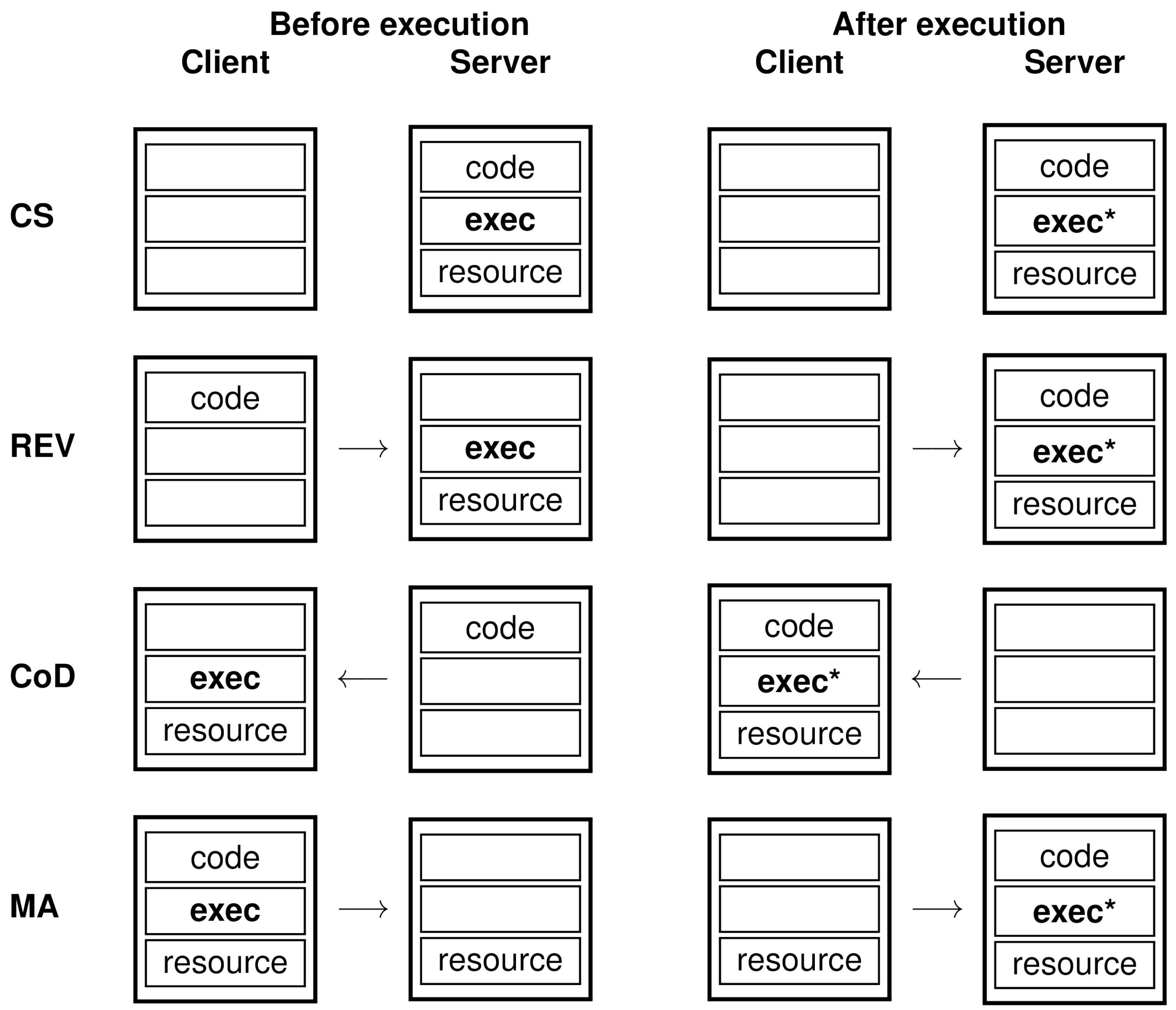


Code Migration

- Code migration
  - Flexibility by dynamically configuring distributed systems


2. Client and server communicate
Client
Server
1. Client fetches code
Service-specific client-side code
Code repository

Models for code migration



CS: Client-Server
CoD: Code-on-demand

REV: Remote evaluation
MA: Mobile agents

Code Migration

- **Strong Mobility**
  - The execution segment can be transferred as well
    - Process Migration
    - Remote cloning: create an exact copy of the original process

Code Migration

- Heterogeneous systems:
    - Pascal intermediate code
    - Java and Python Virtual Machines

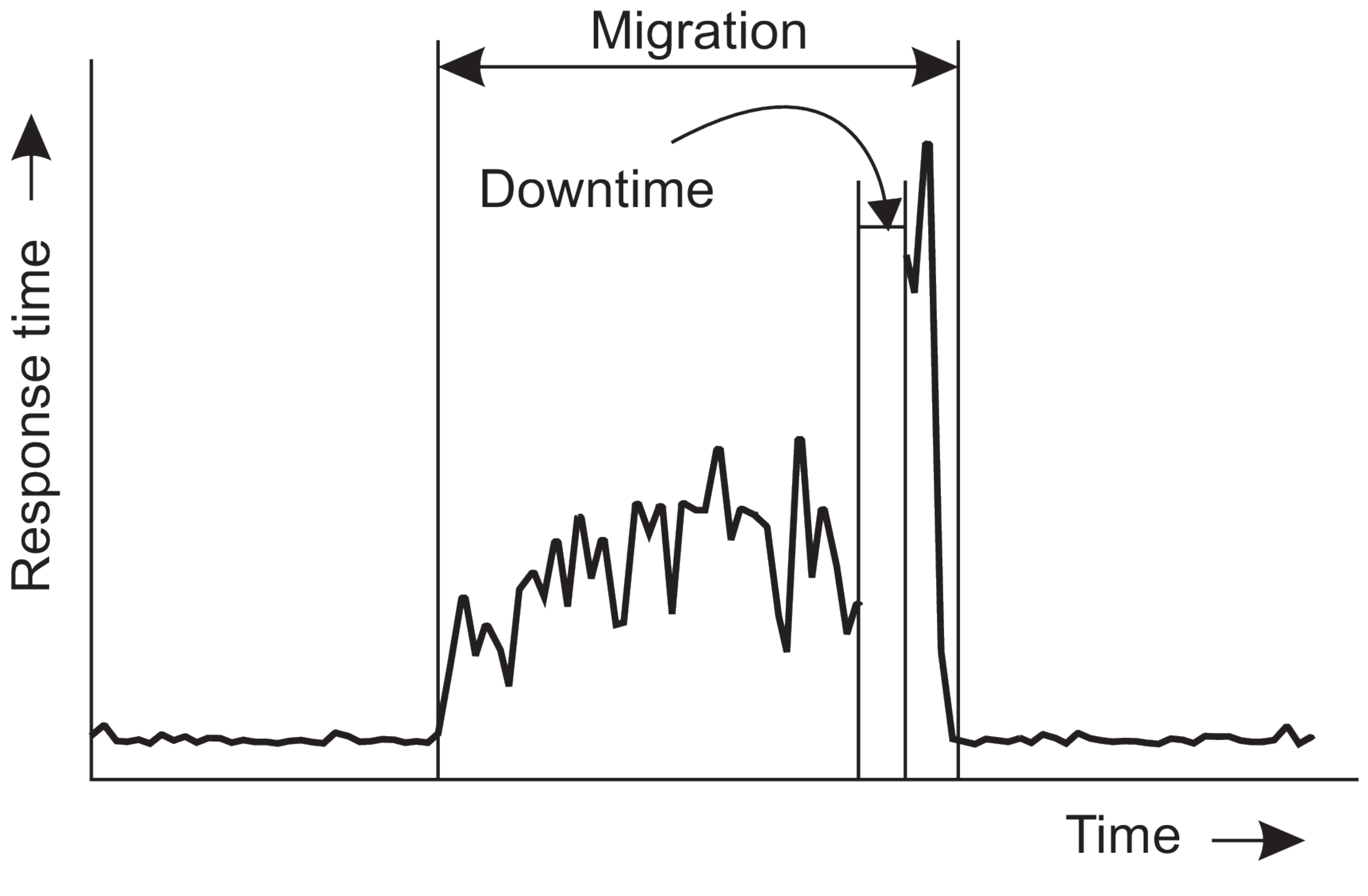
Code Migration

- Virtual Machine Migration
  - Migrating entire memory image
    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, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.
    - Pre-copy: Combine 1 with 2
  - Migrating bindings to local resources

Code Migration

- Migrating bindings
    - Relatively easy when migrating locally

Code Migration


Migration
Downtime
Response time →
Time →