Distributed Systems
Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science
Room R4.20, steen@cs.vu.nl

Chapter 03: Processes

Version: November 2, 2009

vrije Universiteit amsterdam

Contents

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. (hard. registers)

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. (subset hard. registers + private stack)

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.(a lot of state - Process Control Block)

Context Switching

|processor context| < |thread context| < |process context|

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).

PCB

usually a private execution stack per thread

Context Switching

for user-space threads, kernel-space threads need intervention of OS, more on this below…

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 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.

Threads and Operating Systems

e.g., original Linux and Java (green-) threads were implemented in user-space

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 lots of 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 big problem is the loss of efficiency due to the fact that each thread operation requires a trap to the kernel.

Current thread library implementations adhere to the Native POSIX Thread Library (NPTL) specification (API); the API does not specify type of thread *but*
e.g., Linux clib uses kernel-threads
e.g., Java threads are implemented as kernel threads

Threads and Distributed Systems

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.

Threads and Distributed Systems

Improve performance

- Starting a thread is much 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.

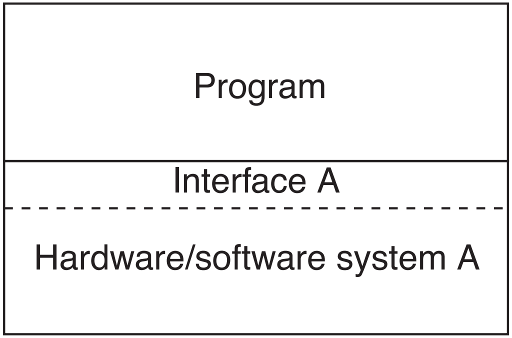
Virtualization

this was back in 2009, it is fundamental now!

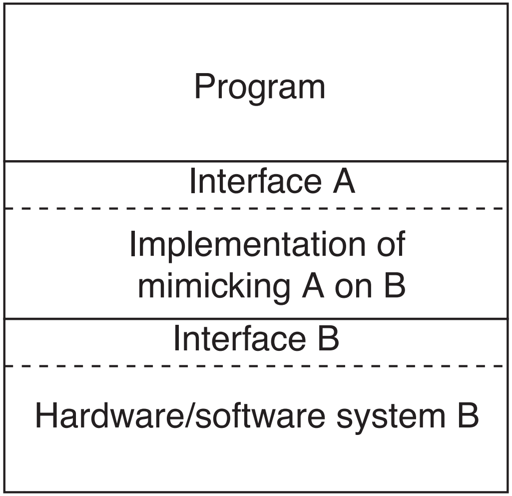
Observation

Virtualization is becoming increasingly important:

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



(a) bare metal configuration

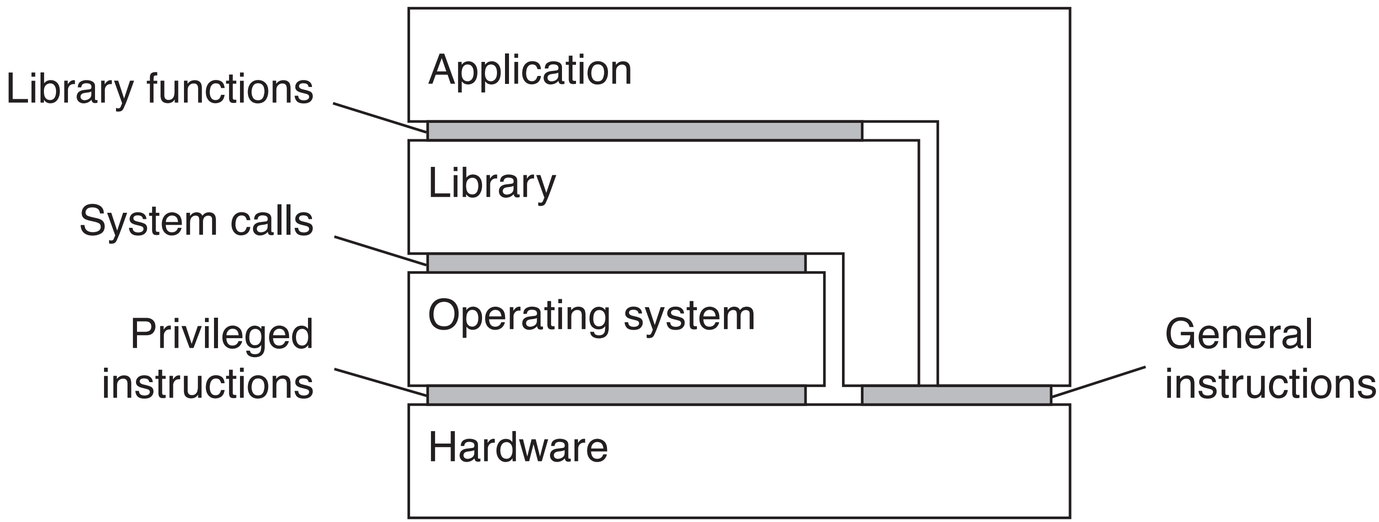


host OS configuration (b)

Architecture of VMs

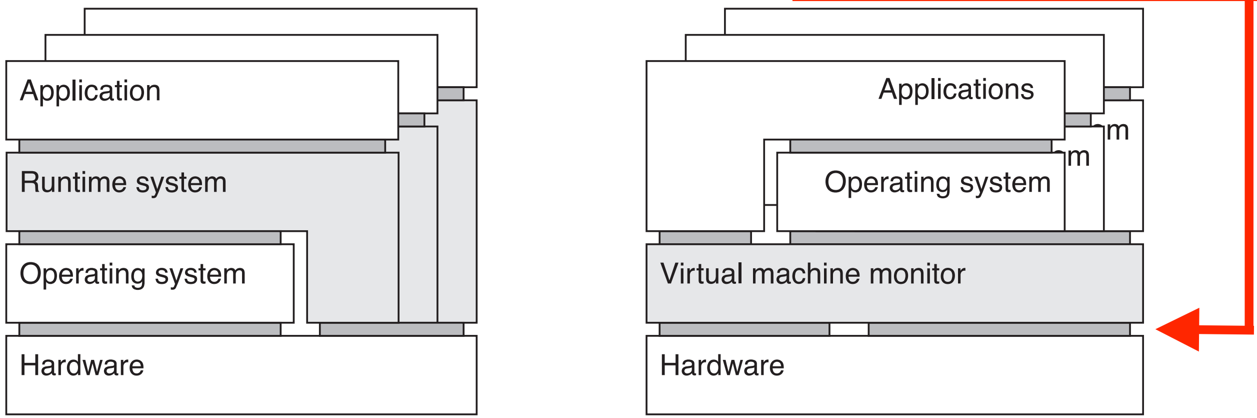
Observation

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



Process VMs versus VM Monitors

usually the VMM runs on top of a host OS



(a) (b)

- Process VM: A program is compiled to intermediate (portable) code, which is then executed by a runtime system (Example: Java VM).
- VM Monitor: A separate software layer mimics the instruction set of hardware $\Rightarrow$ a complete operating system and its applications can be supported (Example: VMware, VirtualBox).

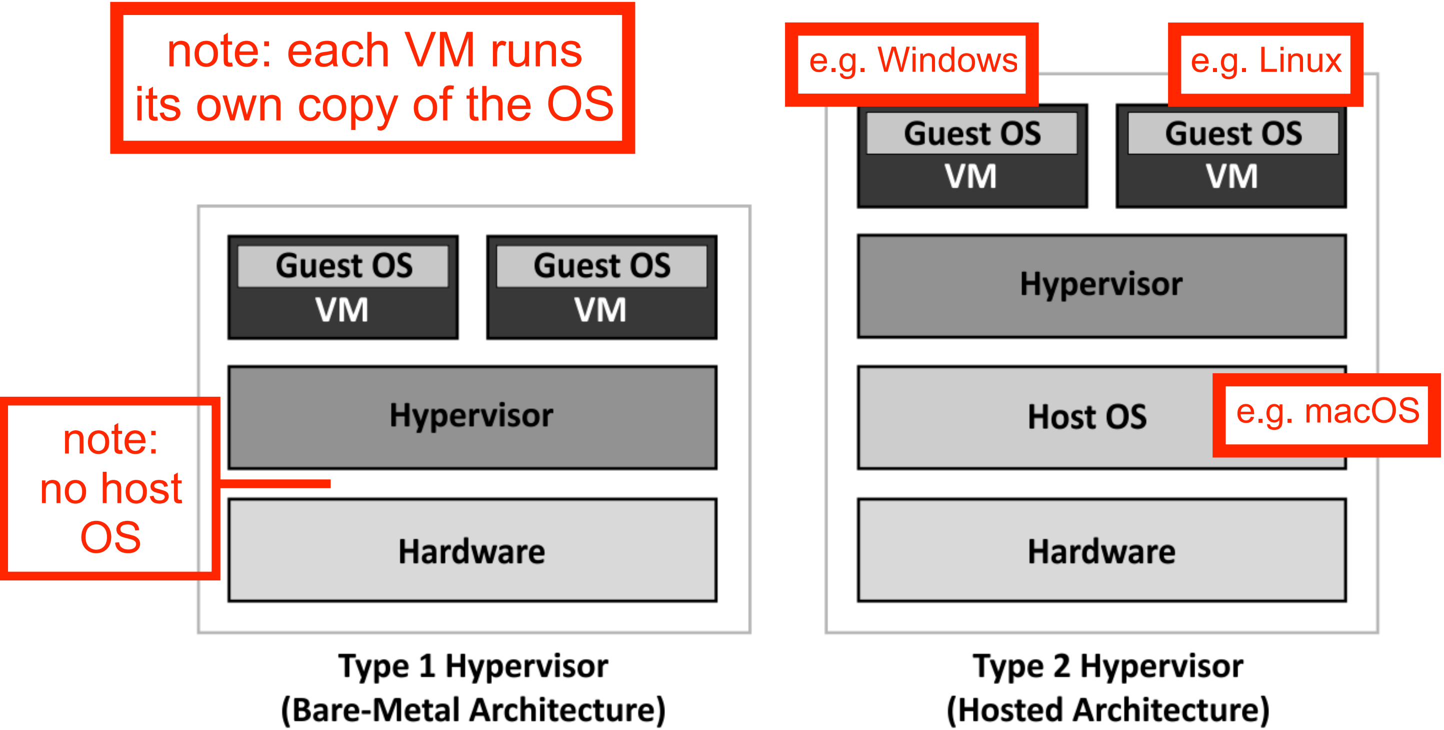
VM Monitors on operating systems

Practice

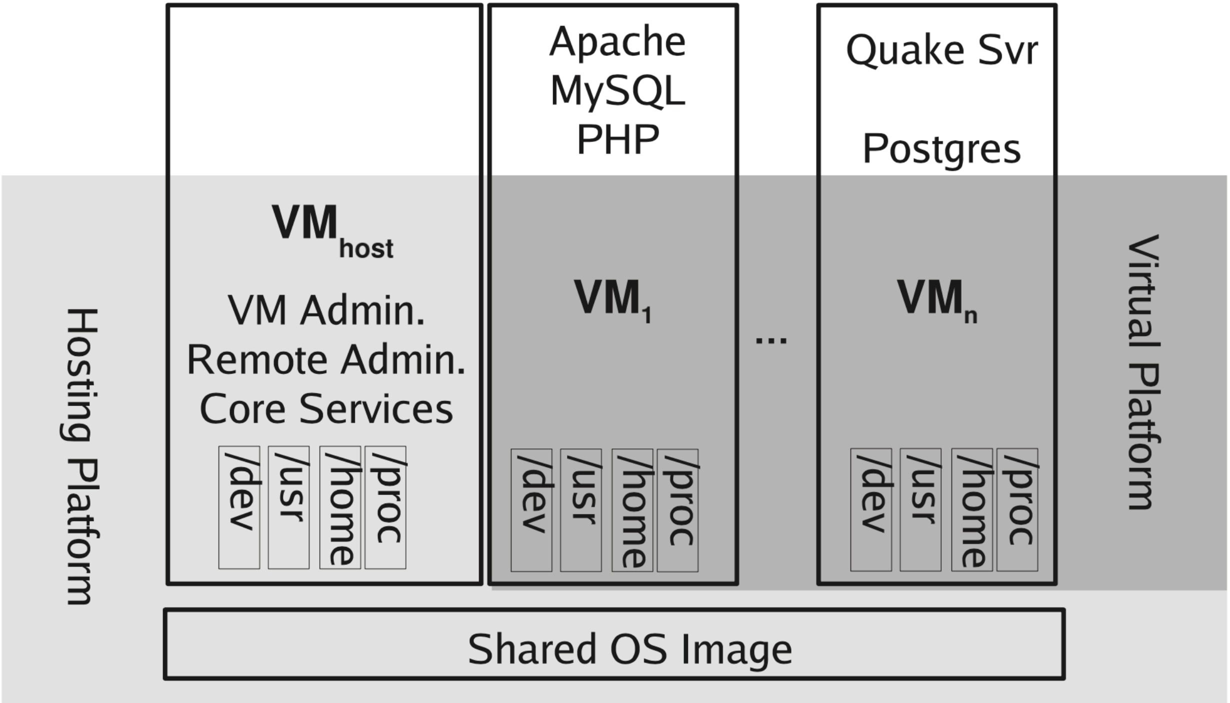
We're seeing VMMs run on top of existing operating systems.

- Perform binary translation: while executing an application or operating system, translate instructions to that of the underlying machine.
- Distinguish sensitive instructions: traps to the orginal kernel (think of system calls, or privileged instructions).
- Sensitive instructions are replaced with calls to the VMM.

Hypervisor-based Virtualization

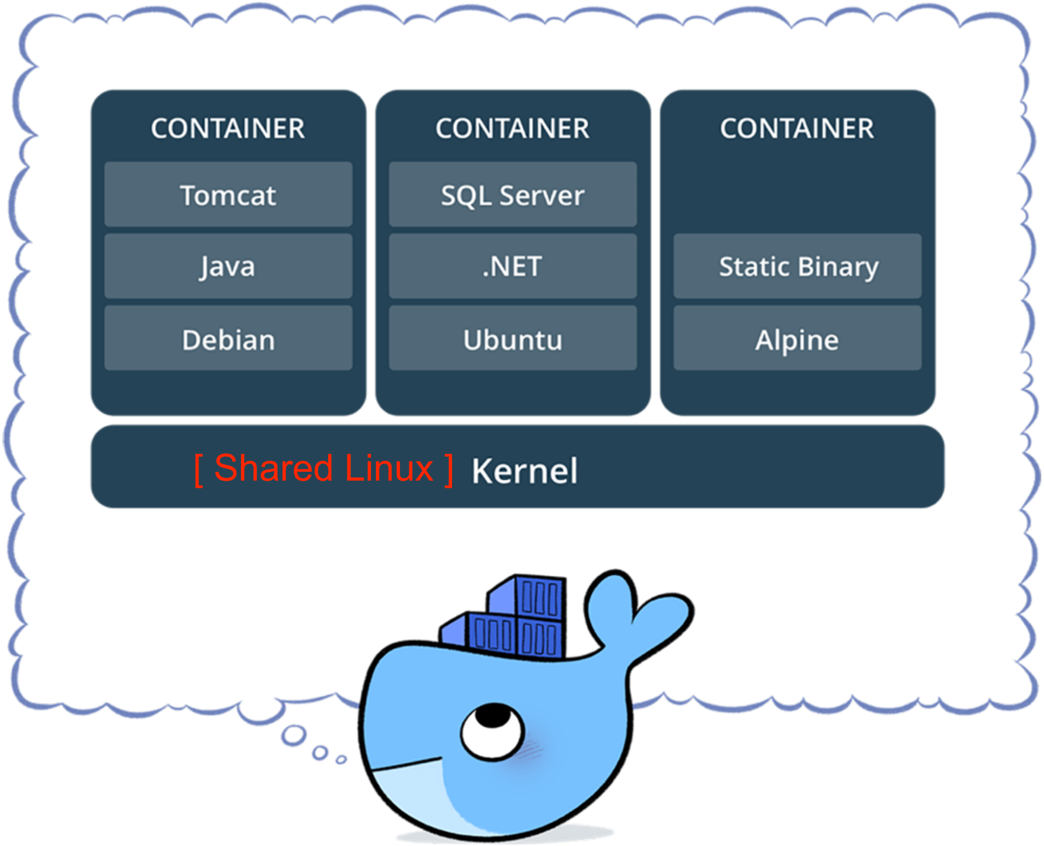


https://www.casesup.com/files/infrastructure/hypervisors_type.png note: containers share a copy of the OS



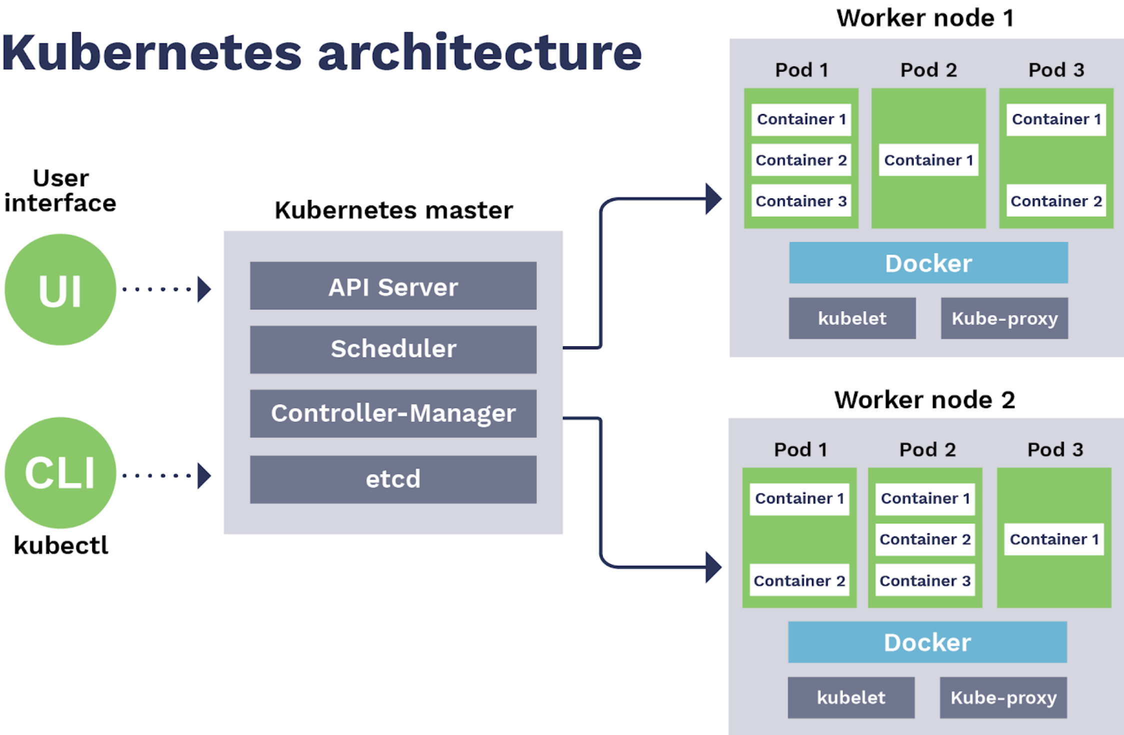
Container-based Virtualization

from Stephen Soltesz et al. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors software tools like Docker and Kubernetes are used to manage containers


CONTAINER
Tomcat
Java
Debian
CONTAINER
SQL Server
.NET
Ubuntu
CONTAINER
Static Binary
Alpine
[Shared Linux] Kernel

kubernetes

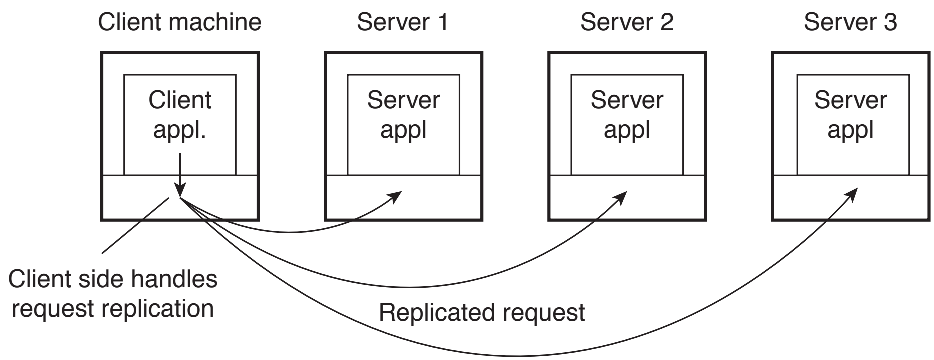
Kubernetes architecture



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: General organization

Basic model

A server is a process that waits for incoming service requests at a specific transport address. In practice, there is a one-to-one mapping between a port and a service.

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
	24	any private mail system
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol
sunrpc	111	SUN RPC (portmapper)
courier	530	Xerox RPC

Servers: General organization

Type of servers

Superservers: Servers that listen to several ports, i.e., provide several independent services. In practice, when a service request comes in, they start a subprocess to handle the request (UNIX *inetd*)

Iterative vs. concurrent servers: Iterative servers can handle only one client at a time, in contrast to concurrent servers

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)

Servers and state

Stateful servers

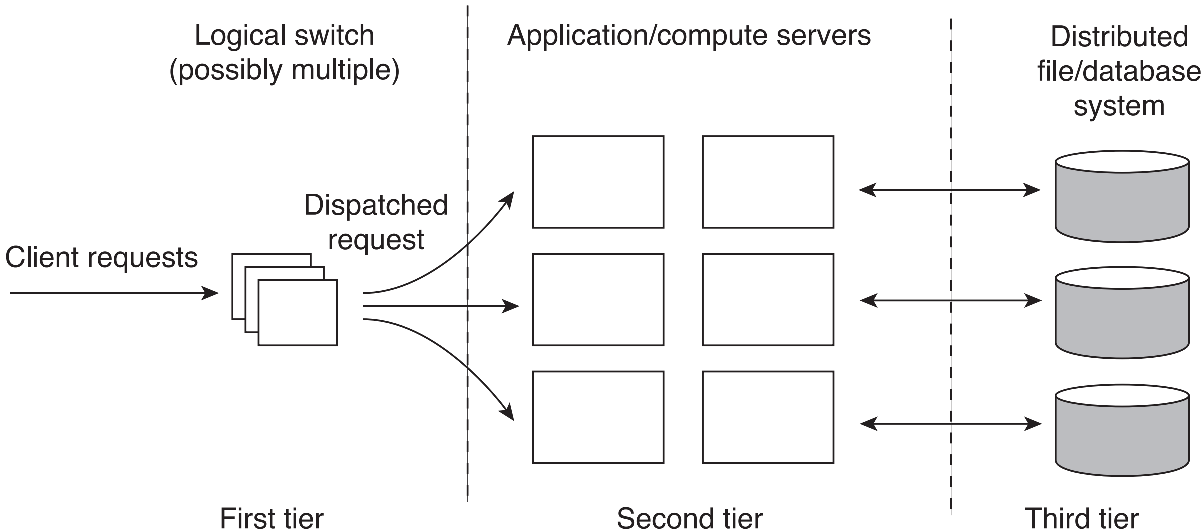
Keeps track of the status of its clients:

- 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 not a major problem.

Server clusters: three different tiers



Crucial element

The first tier is generally responsible for passing requests to an appropriate server.

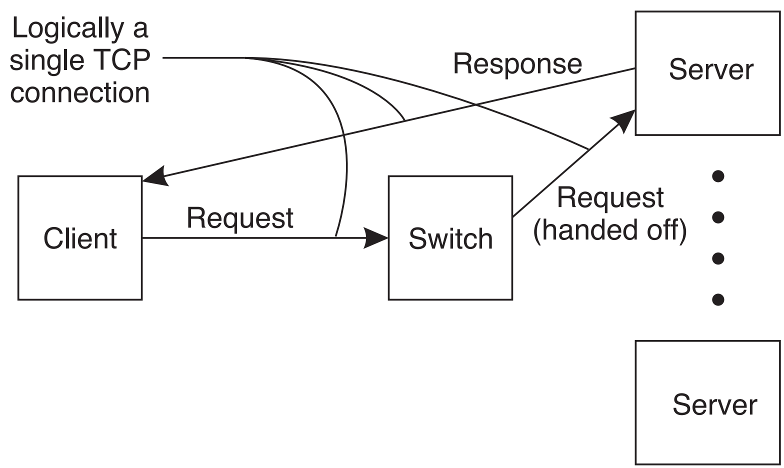
Request Handling

Observation

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

Solution

Various, but one popular one is TCP-handoff



Example: PlanetLab

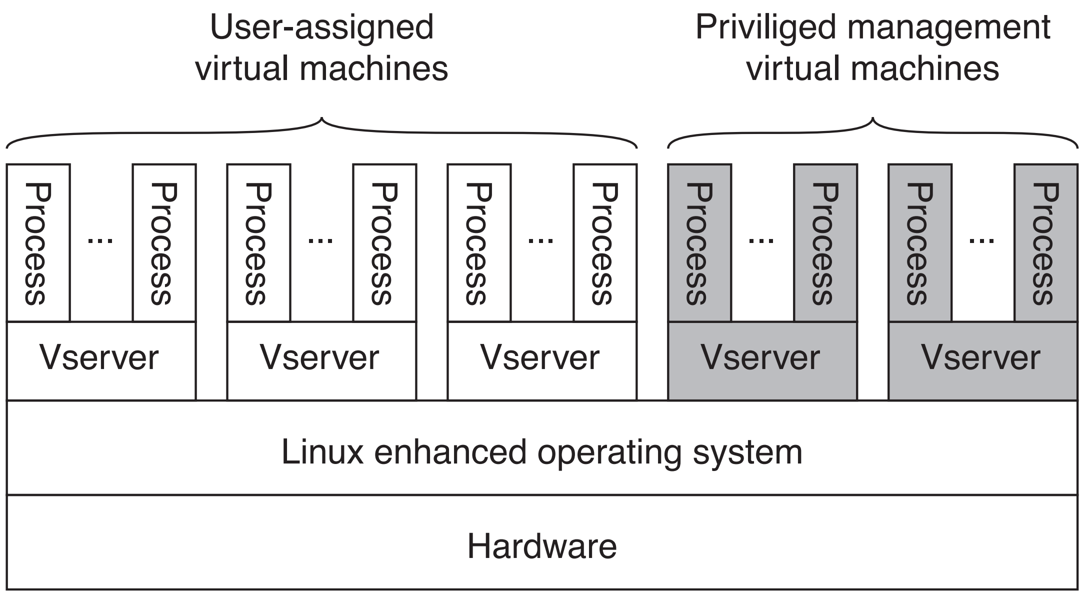
Essence

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 $\Rightarrow$ virtualization

Example: PlanetLab



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 (slice).

https://planetlab.cs.princeton.edu/