Duke Systems

Services and Scale

Jeff Chase

Duke University

A simple, familiar example

request
"GET /images/fish.gif HTTP/1.1"

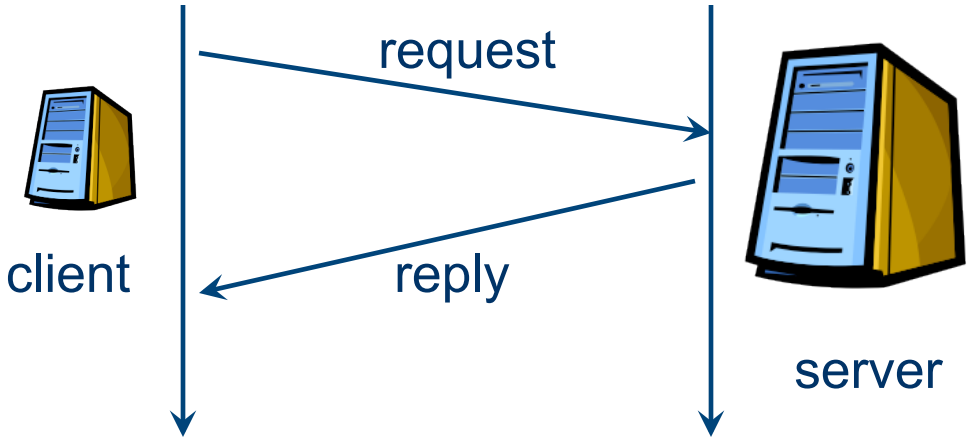
client (initiator)

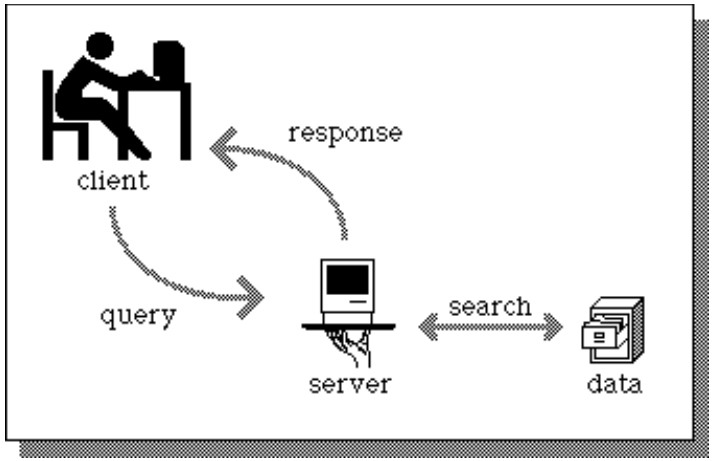
server

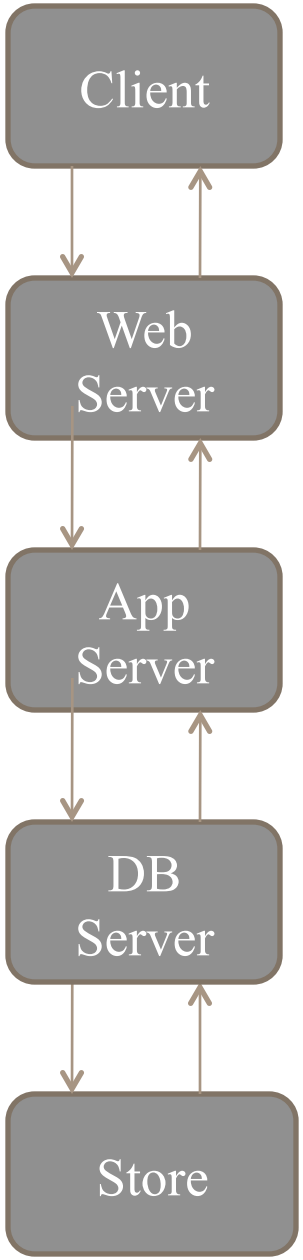
```
sd = socket(…);
connect(sd, name);
write(sd, request…);
read(sd, reply…);
close(sd);
```

```
s = socket(…);
bind(s, name);
sd = accept(s);
read(sd, request…);
write(sd, reply…);
close(sd);
```

A service


request
client
reply
server


response
client
query
search
server
data


Client
Web Server
App Server
DB Server
Store

The Steve Yegge rant, part 1
Products vs. Platforms

Selectively quoted/clarified from http://steverant.pen.io/, emphasis added. This is an internal google memorandum that ”escaped”. Yegge had moved to Google from Amazon. His goal was to promote service-oriented software structures within Google.

So one day Jeff Bezos [CEO of Amazon] issued a mandate....[to the developers in his company]:

His **Big Mandate** went something along these lines:

1) **All teams will henceforth expose their data and functionality through service interfaces.**

2) Teams must communicate with each other through these interfaces.

3) **There will be no other form of interprocess communication allowed**: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

The Steve Yegge rant, part 2
Products vs. Platforms

4) It doesn't matter what technology they use. HTTP, Corba, PubSub, custom protocols -- doesn't matter. Bezos doesn't care.

5) **All service interfaces, without exception, must be** designed from the ground up to be **externalizable**. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

6) **Anyone who doesn't do this will be fired**.

7) **Thank you; have a nice day!**

SaaS platforms

New!
$10!

Web/SaaS/cloud
http://saasbook.info

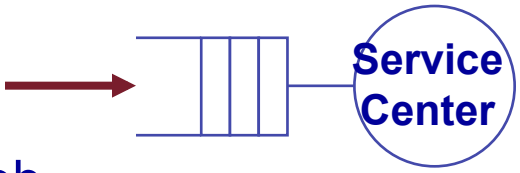
- **A study of SaaS application frameworks is a topic in itself.**
- **Rests on material in this course**
- **We'll cover the basics**
  - **Internet/web systems and core distributed systems material**
- **But we skip the practical details on specific frameworks.**
  - **Ruby on Rails, Django, etc.**
- **Recommended: Berkeley MOOC**
  - **Fundamentals of Web systems and cloud-based service deployment.**
  - **Examples with Ruby on Rails**

Server performance

- How many clients can the server handle?
- What happens to performance as we increase the number of clients?
- What do we do when there are too many clients?

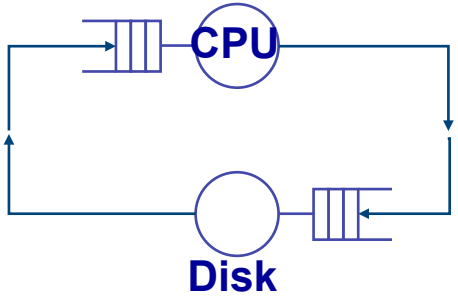
Understanding performance: queues

offered load
request stream @
arrival rate λ
Request == task == job



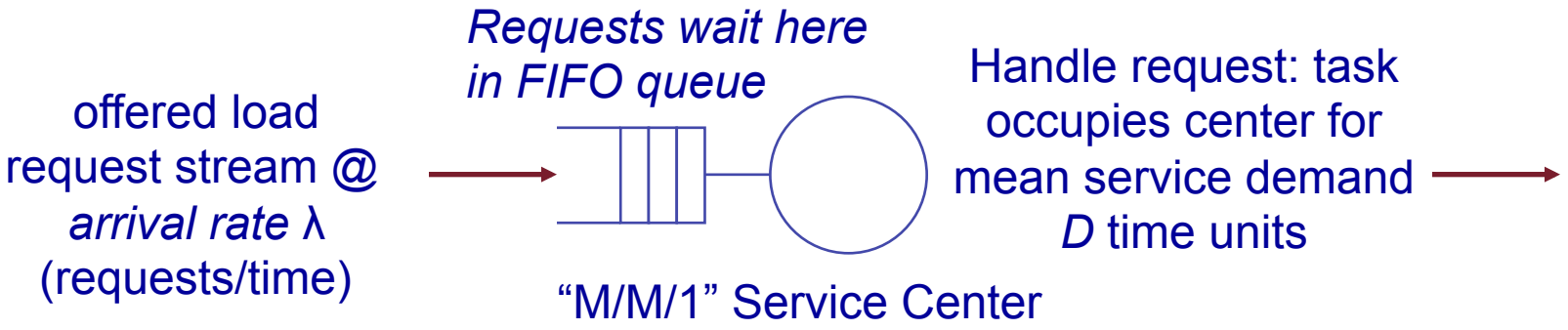
Handle request: task occupies center for *D* time units (its **service demand**).

Note: real systems are **networks** of centers and queues. To maximize overall utilization/throughput, we must think about how the centers interact. (For example, go back and look again at multi-level feedback queue with priority boosts for I/O bound jobs.)



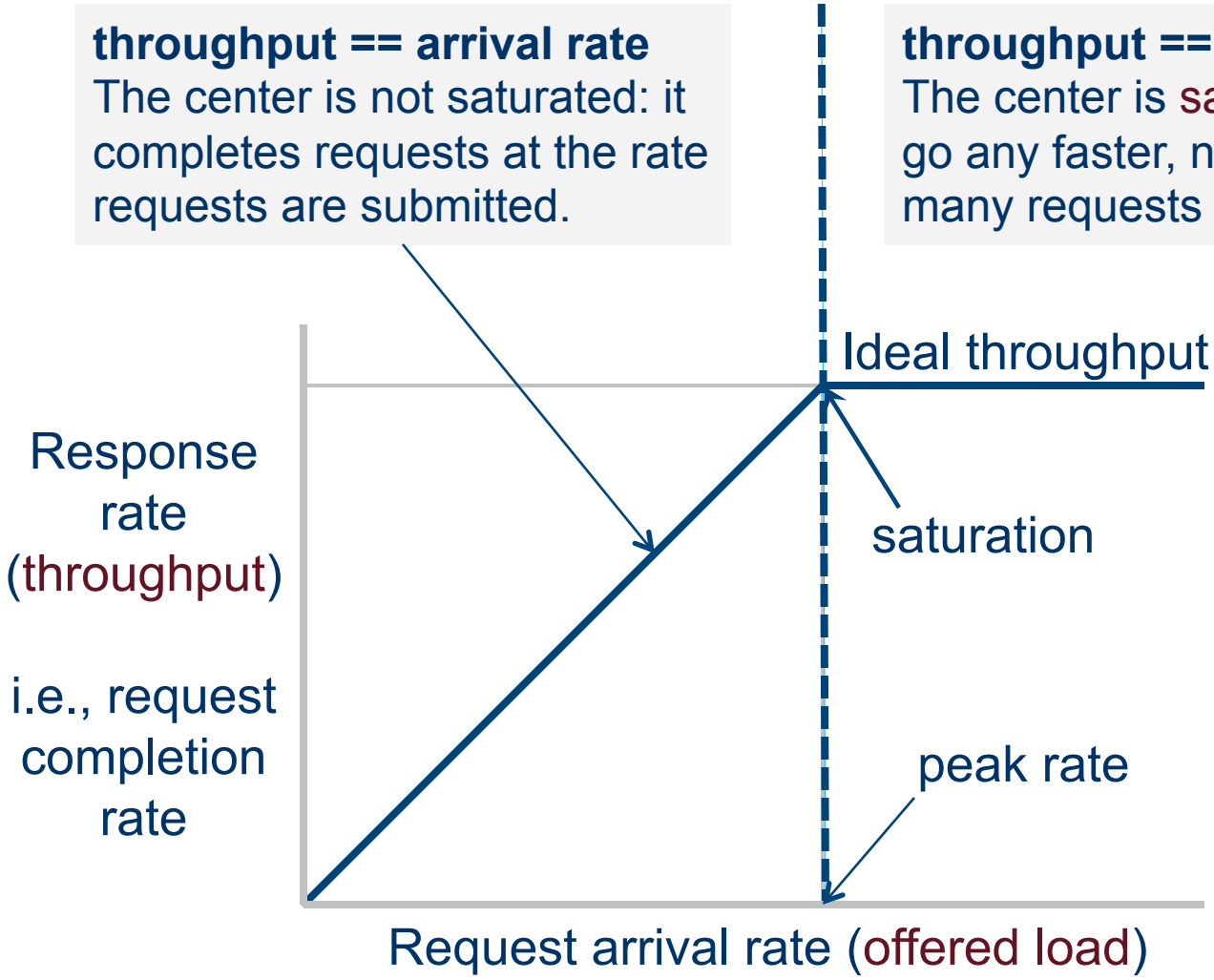
But we can also "squint" and think of the entire network as a single queueing center (e.g., a server), and we won't go too far astray.

Queuing Theory for Busy People



- **Big Assumptions** (at least for this summary)
  - Single service center (e.g., one core), with no concurrency.
  - Queue is First-Come-First-Served (FIFO, FCFS).
  - Independent request arrivals at mean rate λ (**poisson arrivals**).
  - Requests have independent service demands at the center.
  - i.e., arrival interval (1/λ) and service demand (D) are exponentially distributed (noted as "M") around their means.
  - These assumptions are rarely exactly true for real systems, but they give a rough ("back of napkin") understanding of queue behavior.

Ideal throughput: cartoon version



This graph shows throughput (e.g., of a server) as a function of offered load. It is idealized: your mileage may vary.

Throughput: reality

Thrashing, also called congestion collapse
Real servers/devices often have some pathological behaviors at saturation. E.g., they abort requests after investing work in them (thrashing), which wastes work, reducing throughput.

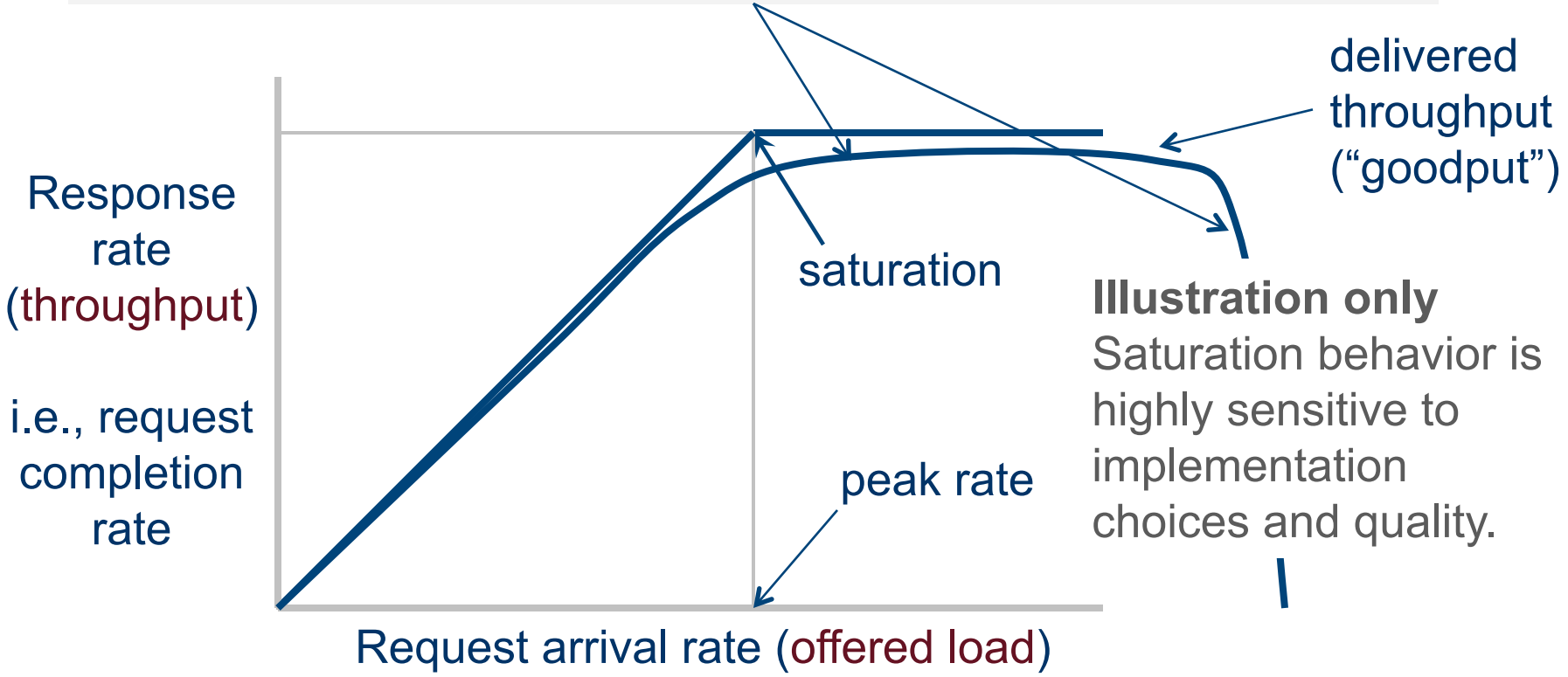


Illustration only
Saturation behavior is highly sensitive to implementation choices and quality.

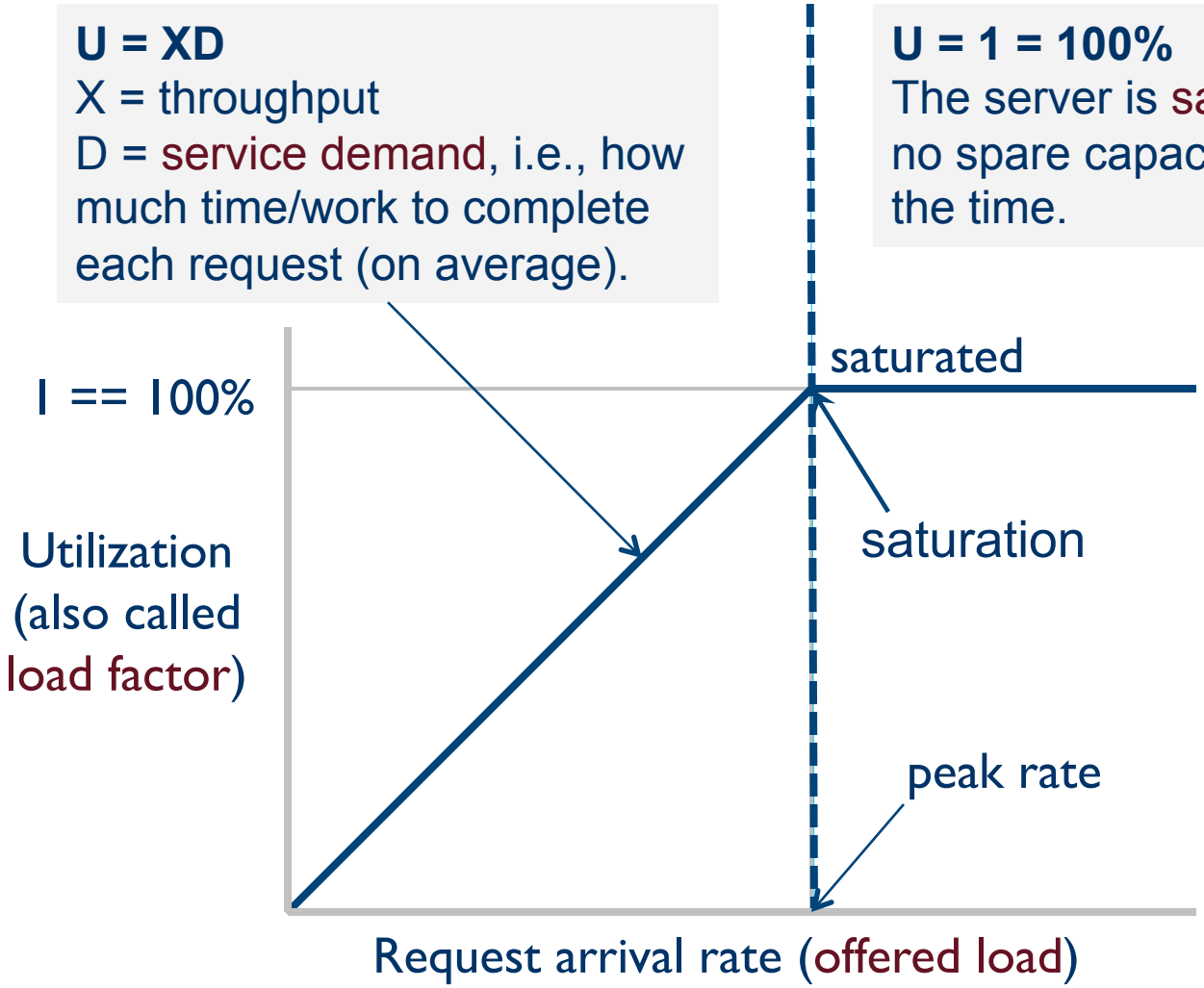
Utilization

- What is the probability that the center is busy?
  - Answer: some number between 0 and 1.
- What percentage of the time is the center busy?
  - Answer: some number between 0 and 100
- These are interchangeable: called **utilization** U
- The probability that the service center is **idle** is 1-U

Utilization: cartoon version

U = XD
X = throughput
D = service demand, i.e., how much time/work to complete each request (on average).

U = 1 = 100%
The server is saturated. It has no spare capacity. It is busy all the time.



This graph shows utilization (e.g., of a server) as a function of offered load. It is idealized: each request works for D time units on a single service center (e.g., a single CPU core).

The Utilization “Law”

- If the center is not saturated then:
  - $U = \lambda D$ = (arrivals/time) * service demand
- **Reminder**: that’s a rough average estimate for a mix of arrivals with average service demand D.
- If you actually measure utilization at the center, it may vary from this estimate.
  - But not by much.

It just makes sense

The thing about all these laws is that they just make sense. So you can always let your intuition guide you by working a simple example.

If it takes 0.1 seconds for a center to handle a request, then peak throughput is 10 requests per second. So let's say the offered load λ is 5 requests per second.

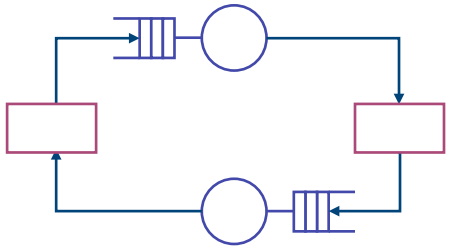
Then $U = \lambda * D = 5 * 0.1 = 0.5 = 50\%$.

It just makes sense: the center is busy half the time (on average) because it is servicing requests at half its peak rate. It spends the other half of its time twiddling its thumbs. The probability that it is busy at any random moment is 0.5.

Note that the key is to **choose units that are compatible**. If I had said it takes 100 milliseconds to handle a request, it changes nothing. But U = 5*100 = 500 is not meaningful as a percentage or a probability. U is a number between 0 and 1. So you have to do what makes sense. Our treatment of the topic in this class is all about formalizing the intuition you have anyway because it just makes sense. Try it yourself for other values of λ and D.

Understanding utilization and throughput

- Throughput/utilization are “easy” to understand for a single service center that stays busy whenever there is work to do.
- It is more complex for a network of centers/queues that interact, and where each task/job/request visits multiple centers.
- **And that’s what real computer systems look like.**
  - E.g., CPU, disk, network, and mutexes…
  - Other synchronization objects



- The centers can service requests **concurrently**!
- Some may be slower than others; any bottlenecks limit overall throughput. If there is a bottleneck, then other centers are underutilized even if the overall system is saturated.

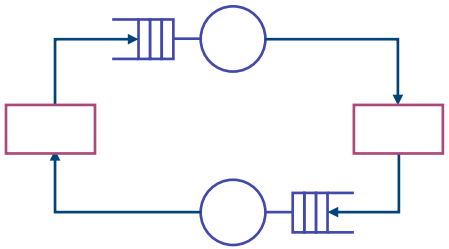
Understanding utilization and throughput

Is high utilization good or bad?

Good. We don't want to pay $$$ for resources and then leave them idle. Especially if there is useful work for them to do!

Bad. We want to serve any given workload as efficiently as possible. And we want resources to be ready for use when we need them.

Utilization ←→ contention



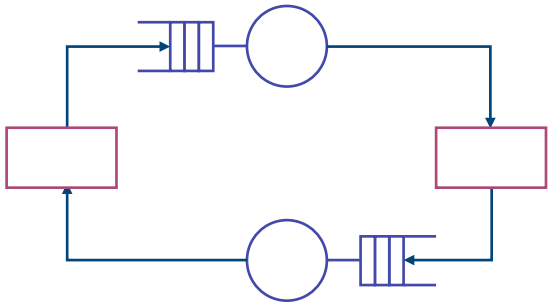
Understanding bottlenecks

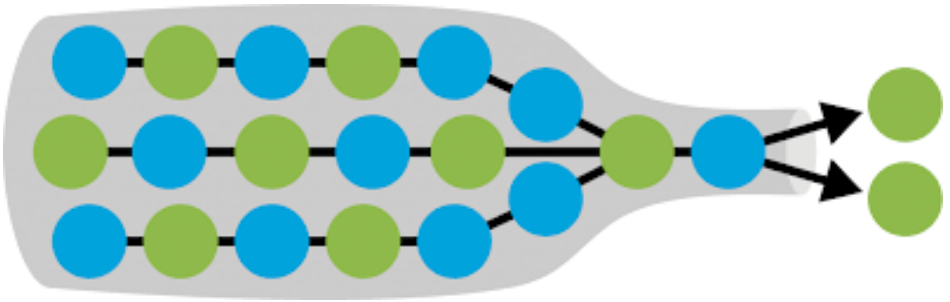
In a multi-center queue system, performance is limited by the center with the **highest utilization** for any workload.

That's the center that saturates first: the **bottleneck.**

Always optimize for the bottleneck.

E.g., it's easy to know if your service is "CPU-limited" or "I/O limited" by running it at saturation and looking at the CPU utilization. (e.g., "top").





Mean response time (R) for a center

R == D
The server is idle. The response time of a request is just the time to service the request (do requested work).

R = D + queuing delay (DN)
As the server approaches saturation, the queue of waiting requests (size N) grows without bound. (We will see why in a moment.)

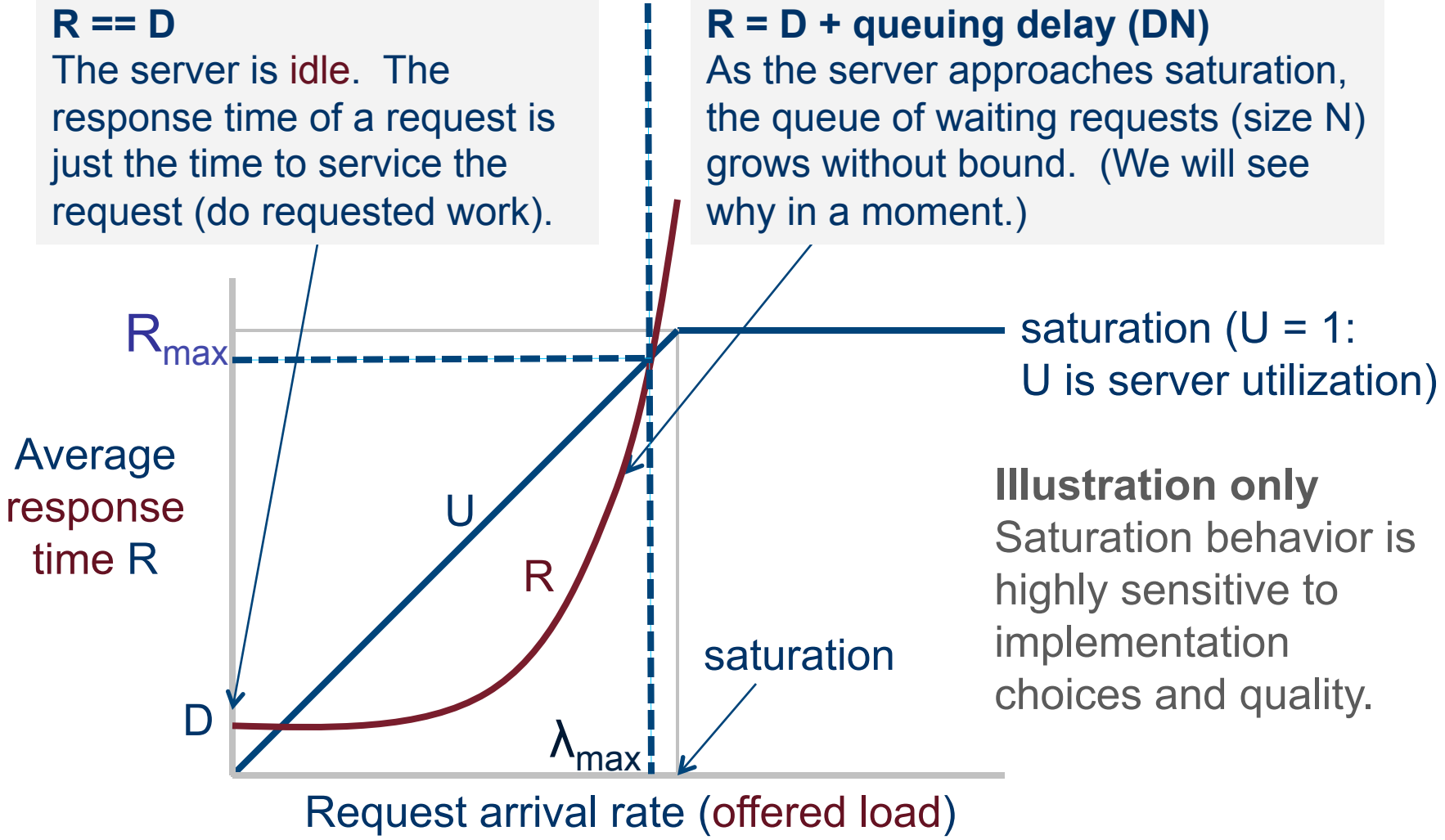


Illustration only
Saturation behavior is highly sensitive to implementation choices and quality.

Little's Law

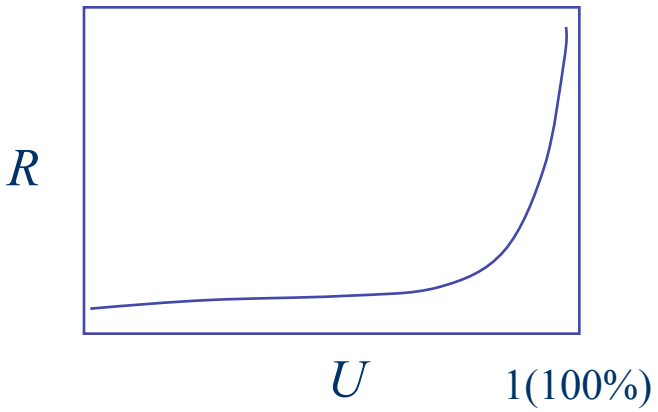
For a FIFO/FCFS queue in steady state, **mean response time R** and **mean queue length N** are governed by:

Little's Law: $N = \lambda R$

Why?

- Suppose a task T is in the system for R time units.
- During that time:
  - λR new tasks arrive (on average)
  - N tasks depart (all the tasks ahead of T, on average).
- But in **steady state**, the flow in balances flow out.
  - "Obviously": throughput $X = \lambda$ in steady state. Otherwise requests "bottle up" in the server -- not a steady state.

Inverse Idle Time "Law"



Service center *saturates* as 1/ λ approaches *D*: small increases in λ cause large increases in the expected response time *R*. At saturation *R* is unbounded (divide by zero: no idle time at saturation == 100% utilization).

Little's Law gives **mean response time** $R = D/(1 - U)$.
("Service demand over the idle time")

Intuitively, an average task ***T***'s response time $R = D + DN$.
(Serve *T* at cost D, and N other tasks ahead of *T* in queue.)
Substituting λR for *N* (by Little's Law): $R = D + D\,\lambda R$
Substituting *U* for λD (by Utilization Law): $R = D + UR$
$R - UR = D \rightarrow R(1 - U) = D \rightarrow R = D/(1 - U)$

Why Little's Law is important

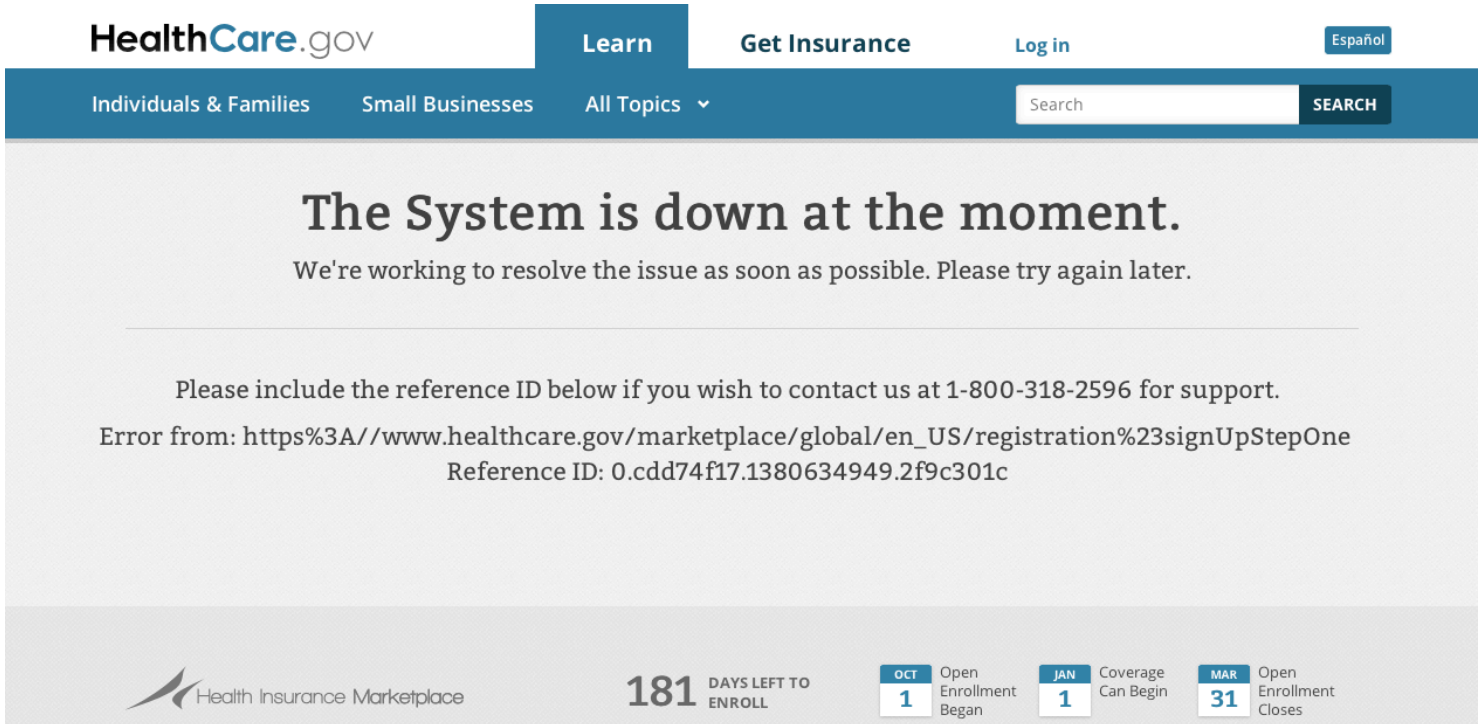
1. Intuitive understanding of FCFS queue behavior.
   - Compute response time from demand parameters (λ, D).
   - Compute N: how much storage is needed for the queue.
2. Notion of a saturated service center.
   - Response times rise rapidly with load and are unbounded.
     - At 50% utilization, a 10% increase in load increases R by 10%.
     - At 90% utilization, a 10% increase in load increases R by 10x.
3. Basis for predicting performance of queuing networks.
   - Cheap and easy "back of napkin" (rough) estimates of system performance based on observed behavior and proposed changes, e.g., capacity planning, "what if" questions.
   - Guides intuition even in scenarios where the assumptions of the theory are not (exactly) met.

The problem of volume continues to be a top concern for the administration, Zients said. Right now, HealthCare.gov can comfortably handle between 20,000 and 25,000 users at a time. But at "peak volumes, some users still experience slower response times," he said.

Officials are also expecting traffic to spike at the end of the month and onward. So this weekend, the administration is adding more servers and data storage to help handle any additional load.

The goal is "to maintain good speed and response times at higher volumes," Zients said. "This is a key focus of our work now."

The New York Times



Part 2

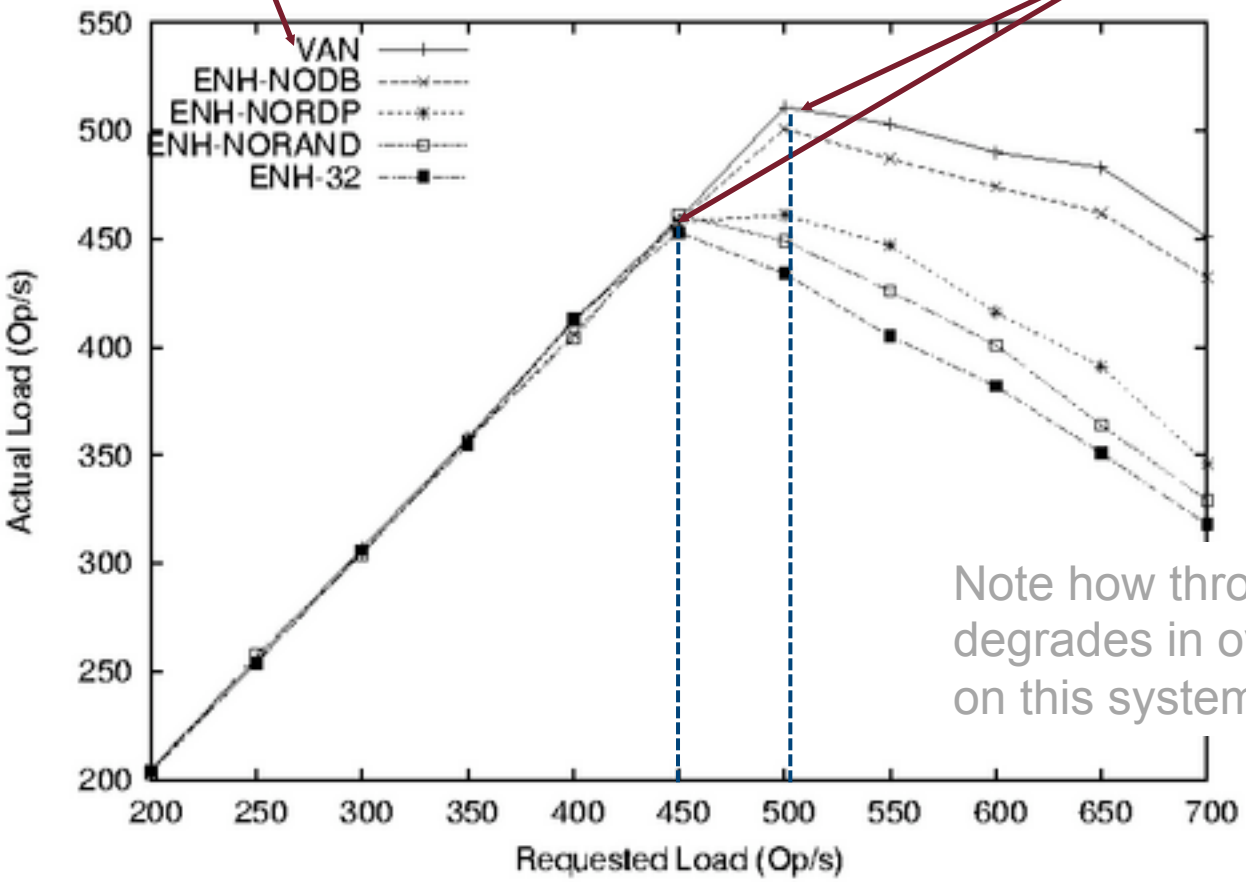
MANAGING SCALABLE PERFORMANCE

Improving performance (X and R)

1. **Make the service center faster. ("scale up")**
   - Upgrade the hardware, spend more $$$
2. **Reduce the work required per request (D).**
   - More/smarter caching, code path optimizations, use smarter disk layout.
3. **Add service centers, expand capacity. ("scale out")**
   - RAIDs, blades, clusters, elastic provisioning
   - N centers improves throughput by a factor of N: **iff** we can partition the workload evenly across the centers!
   - **Note**: the math is different for multiple service centers, and there are various ways to distribute work among them, but we can "squint" and model a balanced aggregate roughly as a single service center: the cartoon graphs still work.

This graph shows how certain design alternatives under study impact a server's throughput. The alternatives reduce per-request work(D or overhead) and/or improve load balancing. (This is a graph from a random research paper: the **design alternatives** themselves are not important to us.)

saturation



Measured throughput ("goodput") **Higher numbers are better.**

Note how throughput degrades in overload on this system.

Offered load (requests/sec)

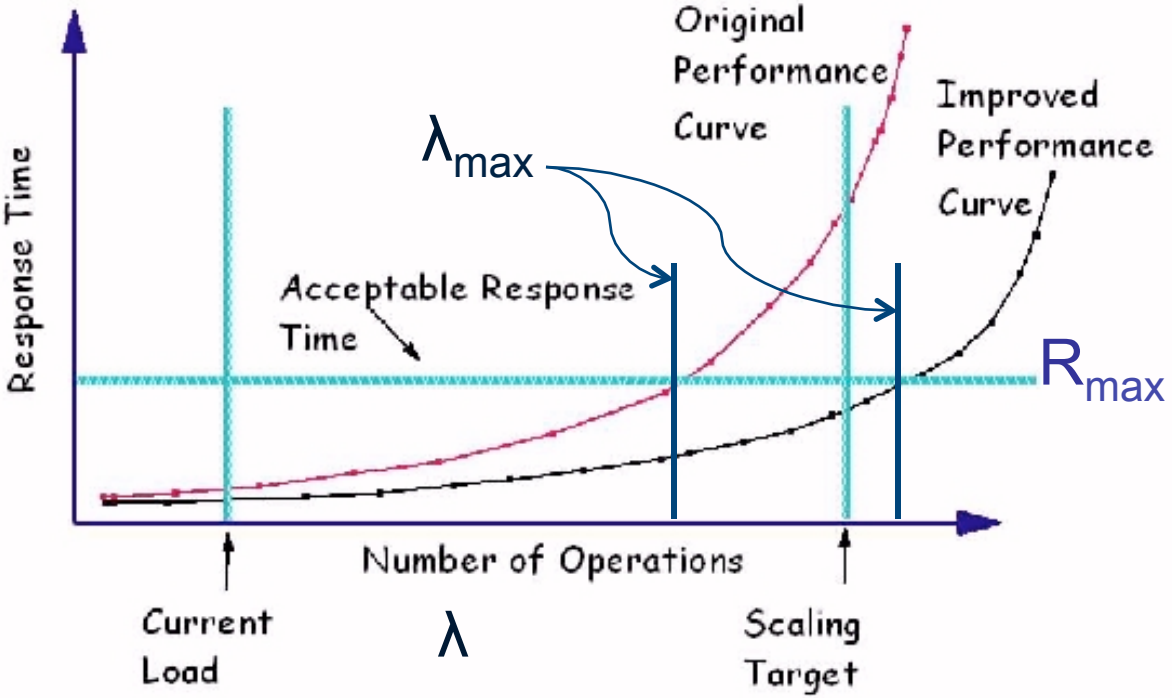
Saturation and response time

In the real world we don't want to saturate our systems.

We want systems to be responsive, and saturated systems aren't responsive.

How to measure maximum capacity of a server?
Characterize **max request rate** λ_{max} this way:

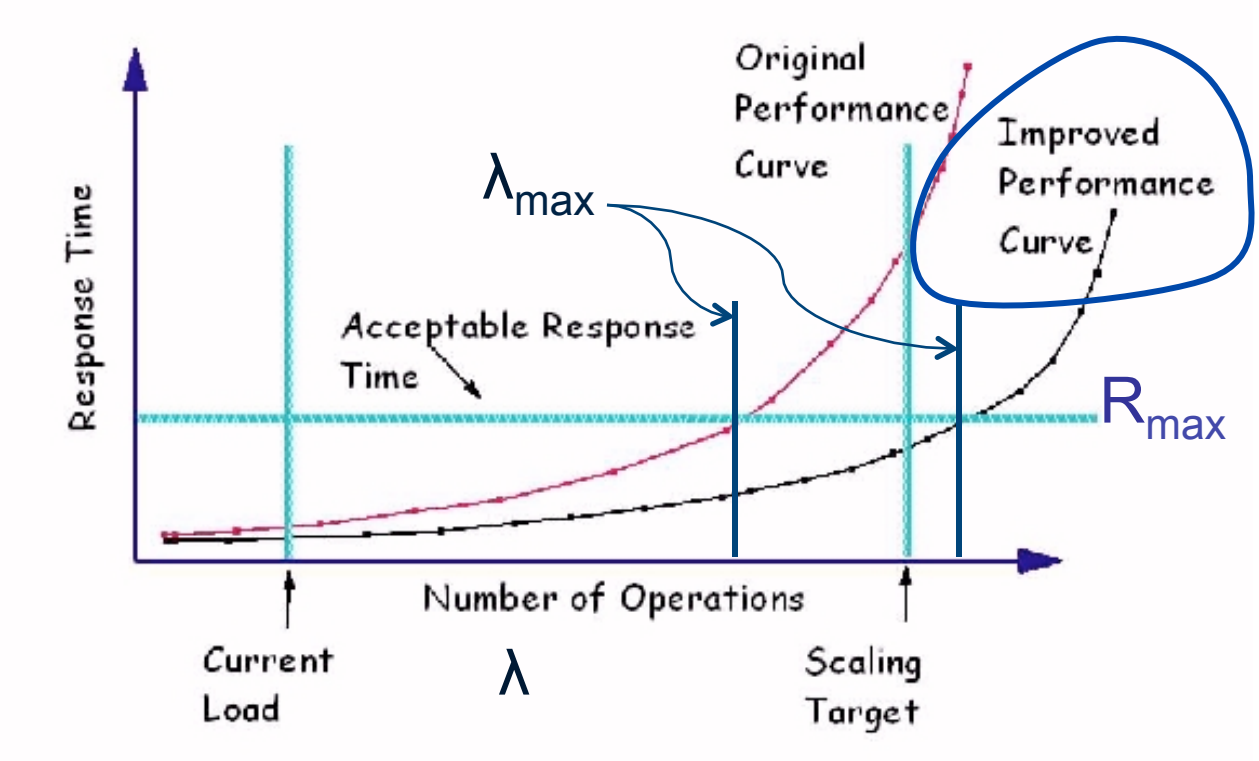
1. Define a response time objective: maximum acceptable response time (R_{max}): a simple form of **Service Level Objective** (SLO).
2. Increase λ until system response time surpasses R_{max} : that is λ_{max}.



[graphic from IBM.com]

Improving response time

If we improve the service for "higher capacity" by any means, the effect is to push the response time curve out to the right.



[graphic from IBM.com]

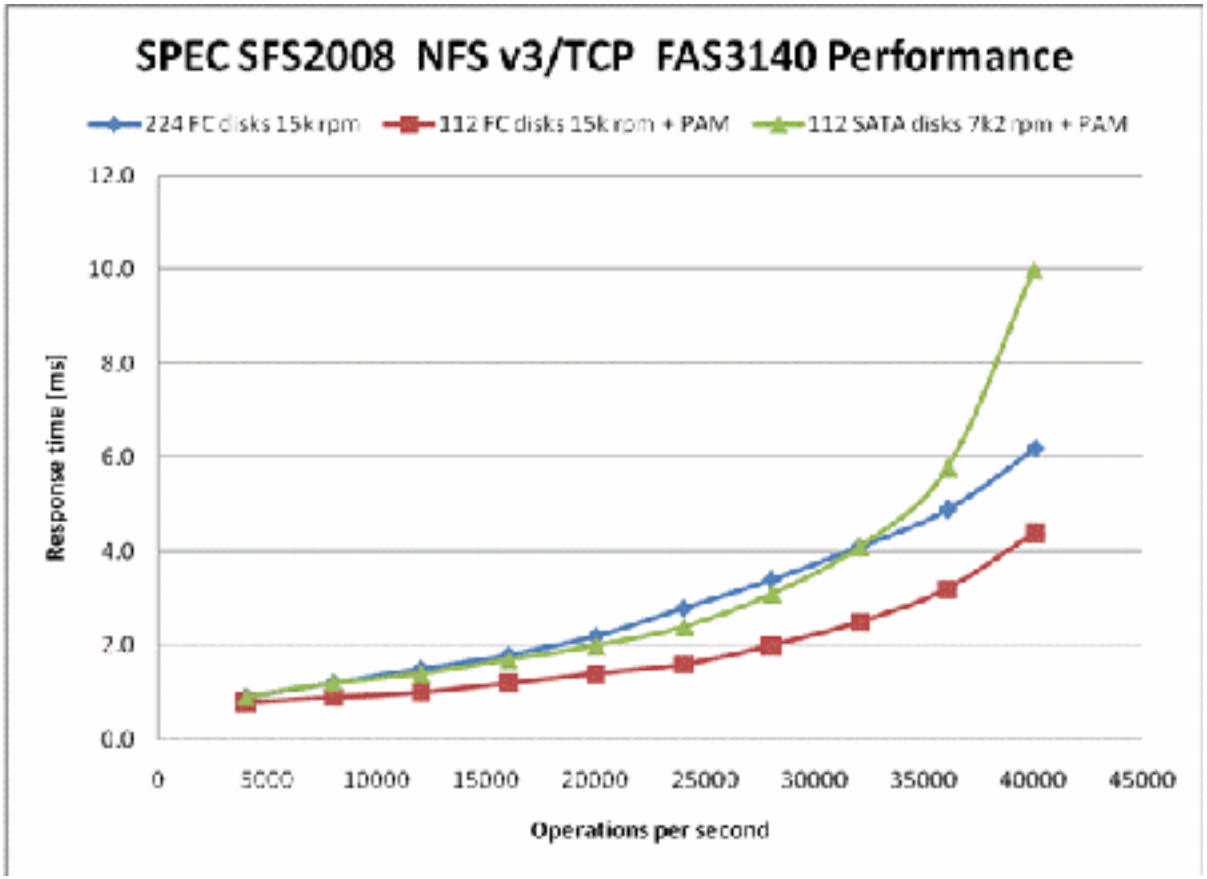
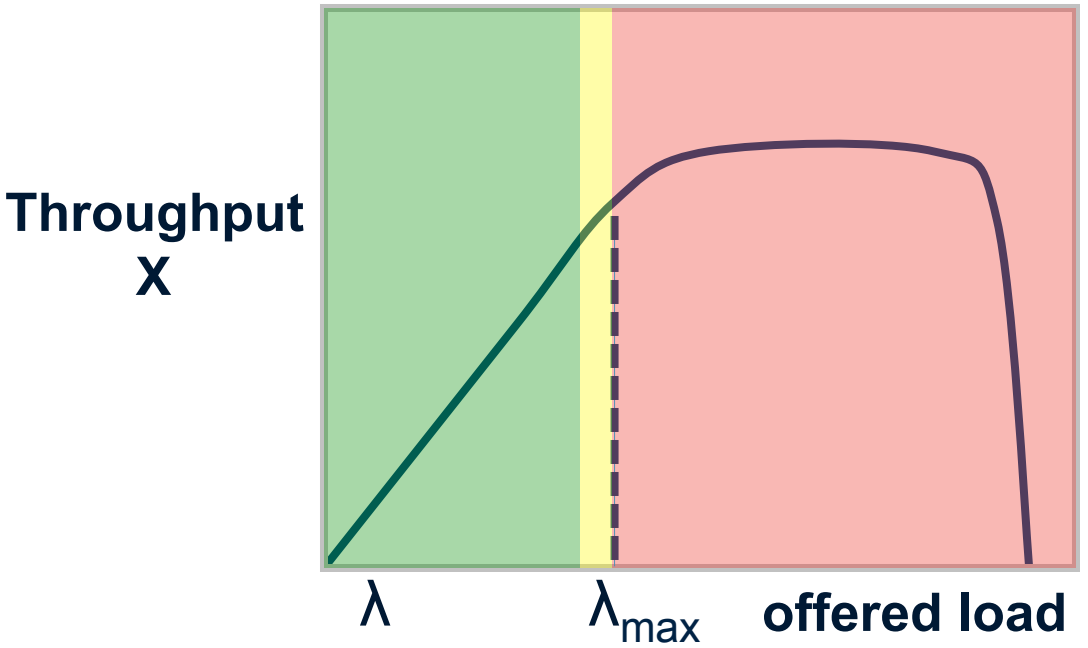


Illustration: if we improve/expand the service by any means, the effect is to push the R curve out to the right. **Roughly.**

Managing overload

What should we do when a service is in overload?

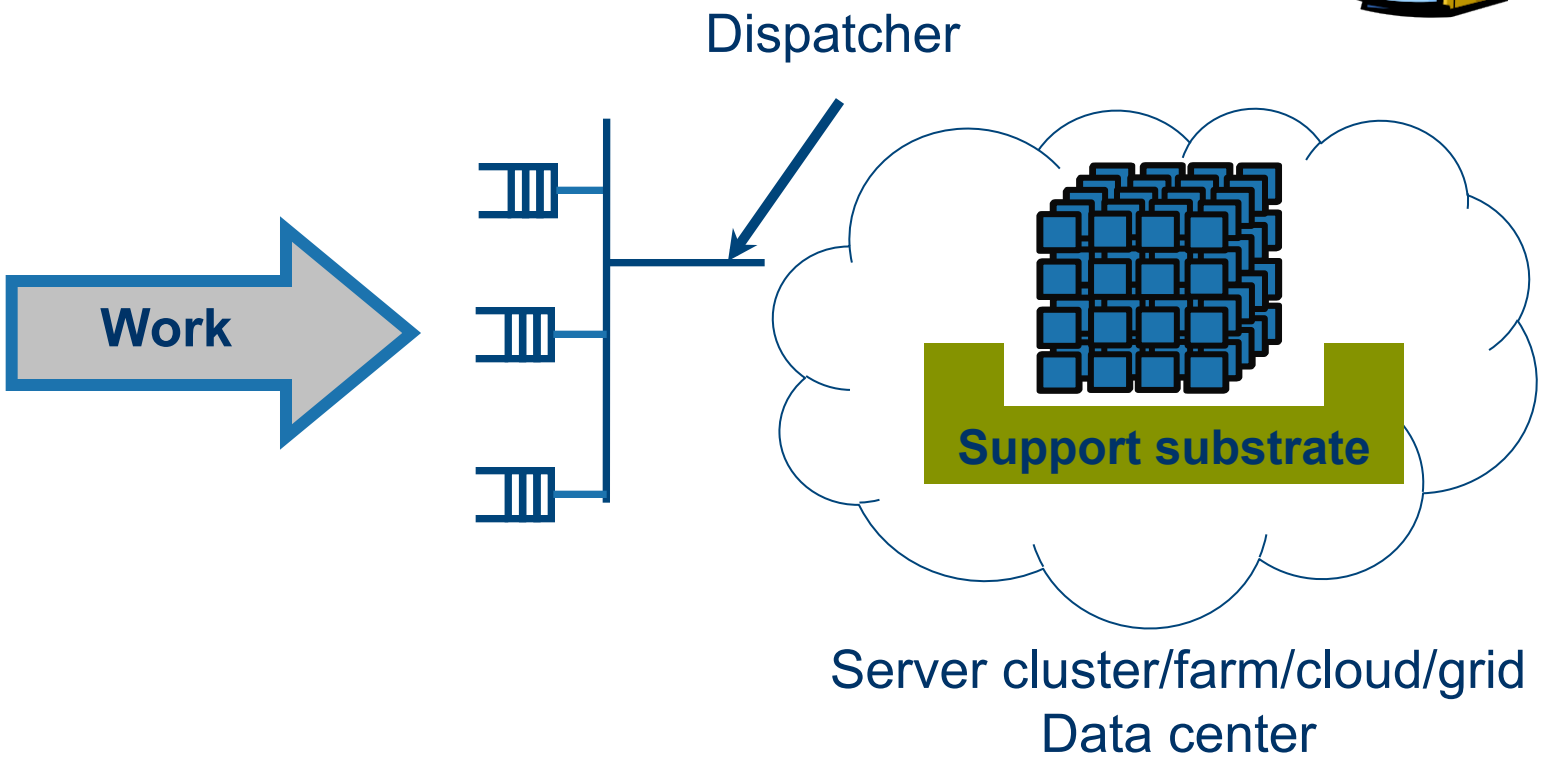
- **Overload**: service is close to saturation. $\lambda > \lambda_{max}$
- Overload → work queues grow without bound, increasing memory consumption and response time.



Options for overload

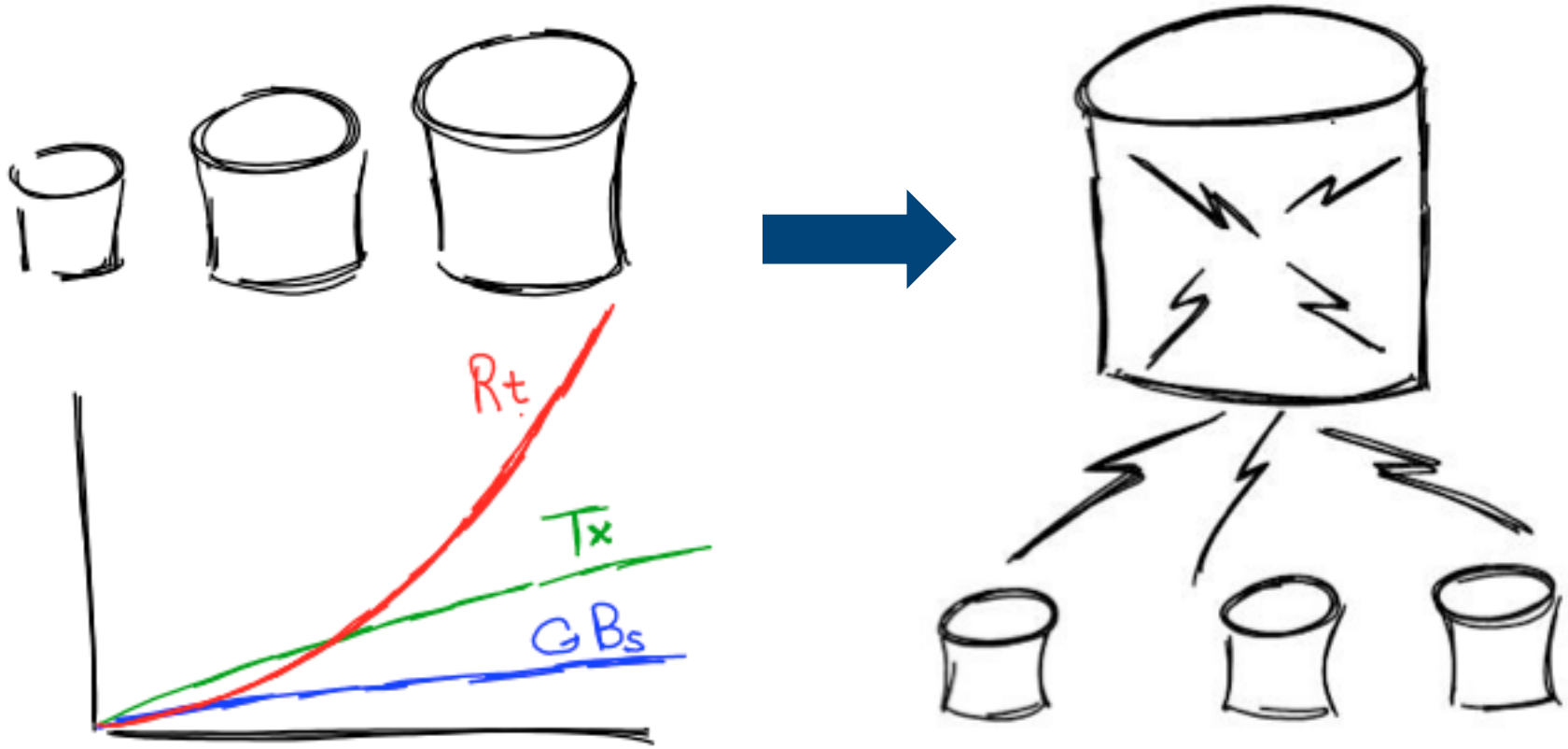
1. **Thrashing**
   - Keep trying and hope things get better. Accept each request and inject it into the system. Then drop requests at random if some queue overflows its memory bound. **Note**: leads to dropping requests after work has been invested, wasting work and reducing throughput (e.g., “congestion collapse”).
2. **Admission control** or **load conditioning**
   - Reject requests as needed to keep system healthy. Reject them early, before they incur processing costs. Choose your victims carefully, e.g., prefer “gold” customers, or reject the most expensive requests.
3. **Dynamic provisioning or elastic scaling**
   - E.g., acquire new capacity “on the fly” (e.g., from a cloud provider), and shift load over to the new capacity.

Scaling a service



Incremental scalability. Add servers or "bricks" for scale and robustness. Issues: state storage, server selection, request routing, etc.

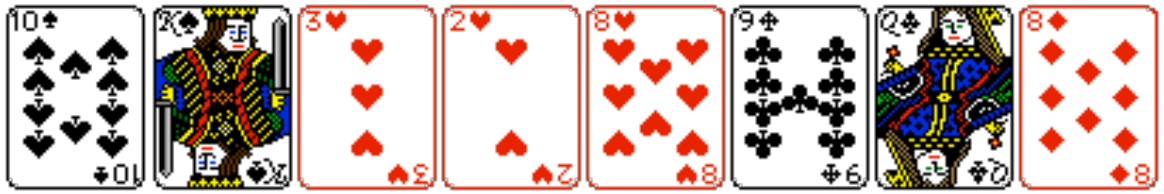
Scale-up vs. scale-out



http://dbshards.com/dbshards/database-sharding-white-paper/

Concept: load spreading

- Spread (“deal”) the data across a set of storage units.
  - Make it “look like one big unit”, e.g., “one big disk”.
  - Redirect requests for a data item to the right unit.
- The concept appears in many different settings/contexts.
  - We can spread load across many servers too, to make a **server cluster** look like “one big server”.
  - We can spread out different data items: objects, records, blocks, chunks, tables, buckets, keys….
  - Keep track using maps or a deterministic function (e.g., a hash).
- Also called sharding, declustering, striping, “bricks”.

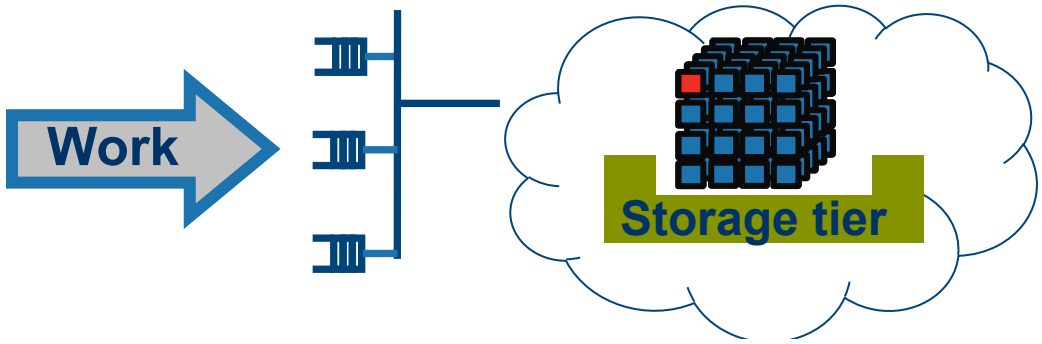


Service scaling and bottlenecks

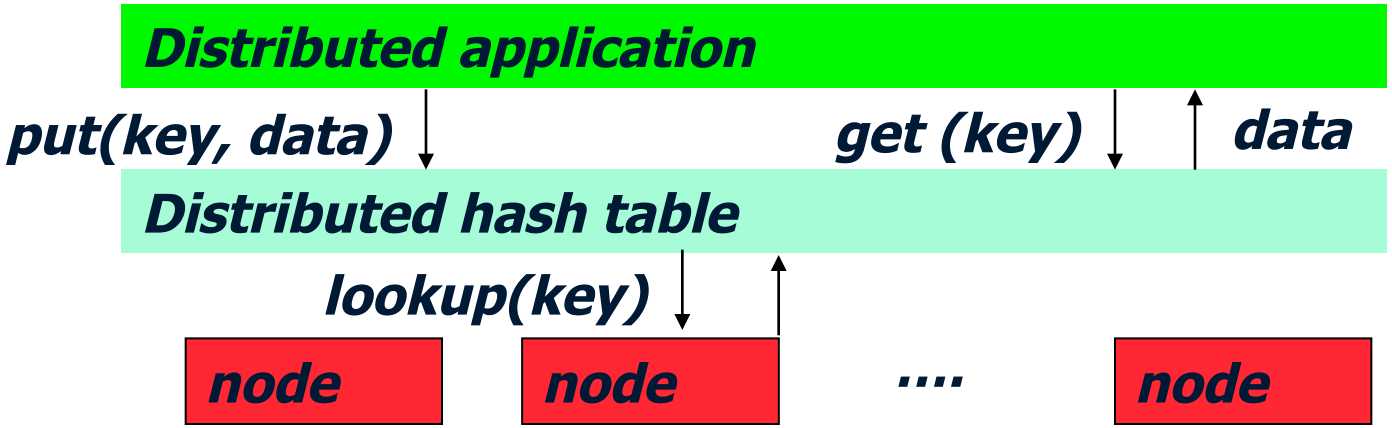
Scale up by adding capacity incrementally?

- "Just add bricks/blades/units/elements/cores"...but that presumes we can **parallelize** the workload.
- **"Service workloads parallelize easily."**
  - Many independent requests: spread requests across multiple units.
  - Problem: some requests use shared data. Partition data into chunks and spread them across the units: be sure to read/write a common copy.
- Load must be evenly distributed, or else some unit saturates before the others (**bottleneck** or **hot spot**).

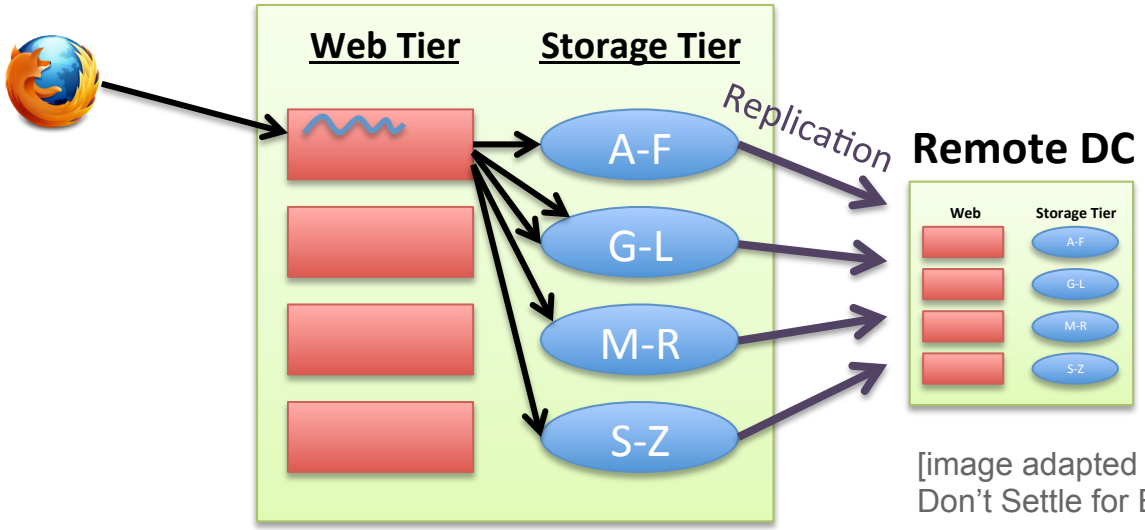
A bottleneck limits throughput and/or may increase response time for some class of requests.



Storage tier: key-value stores



[image adapted from Morris, Stoica, Shenker, etc.]



Example of how to scale the storage tier.

Incrementally scalable?
Balanced load?

[image adapted from Lloyd, etc., Don't Settle for Eventual]

Bottlenecks and hot spots: analysis

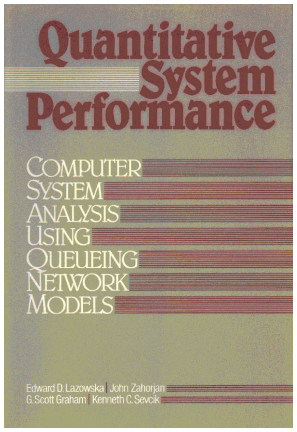
1. Suppose requests are divided evenly among N servers. Mean per-request processing time is D, and also each request reads data from a storage tier at mean cost 2D.

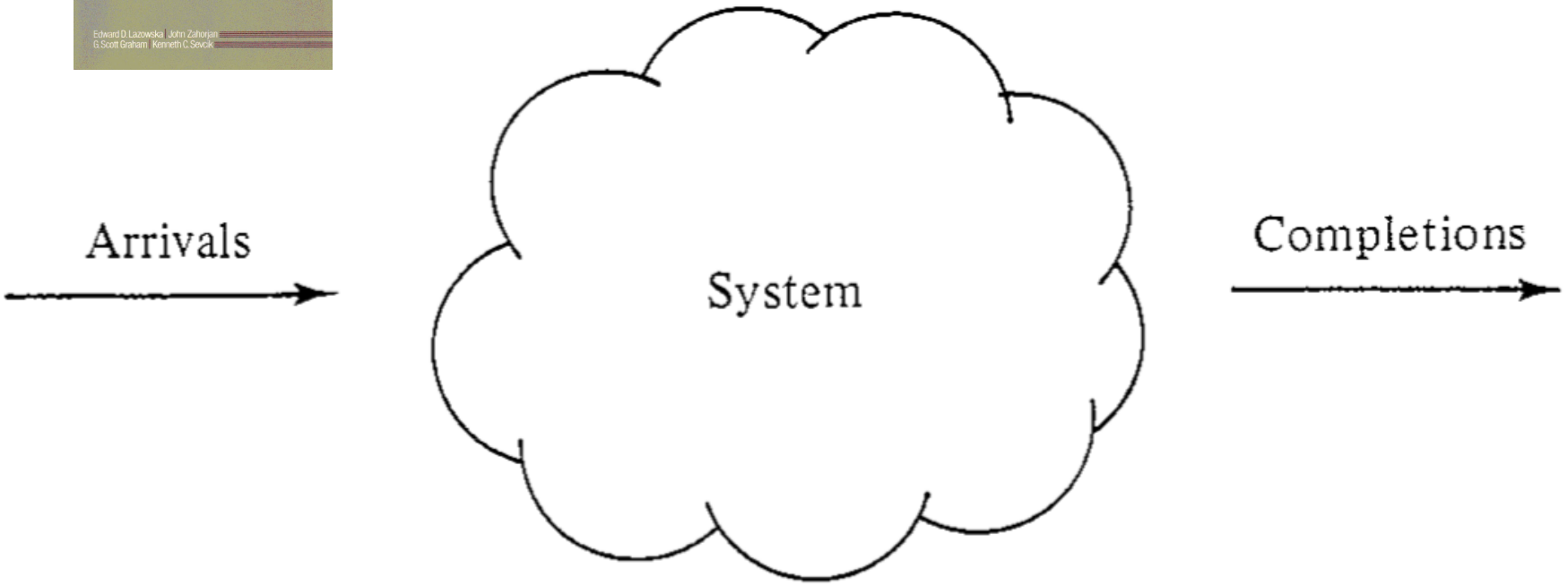
- Simplistic assumption (for now): all nodes are single-threaded.
- If there are N servers in the storage tier, what is the maximum throughput of the system? What is the utilization of the first tier?
- How should we provision capacity to “fix it”?

2. Suppose one of the N servers takes 2D per request.

- What is the impact on **throughput**?
- What is the impact on **response time**?
- Is the effect equivalent if the server has demand D but receives requests at double the rate of the others? How is it different?

3. Suppose the request rate doubles? What then?



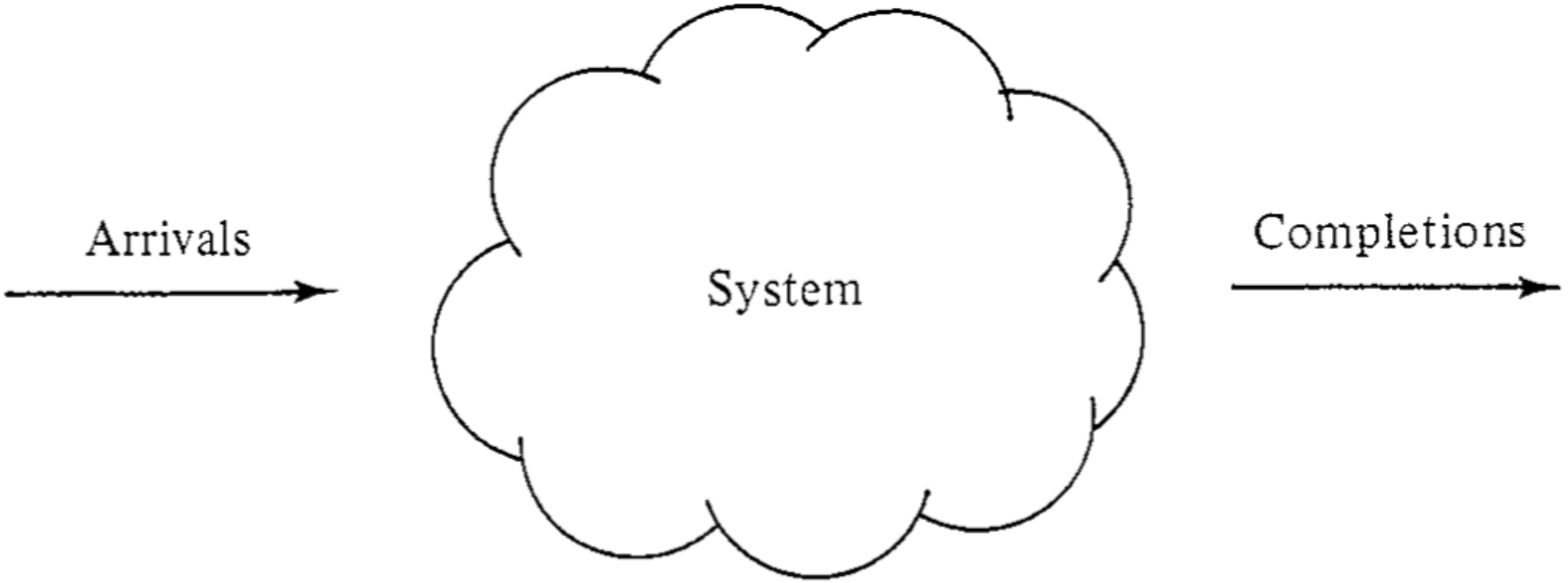


Varying workload

Fixed system

Varying performance

The math also works in the other direction….

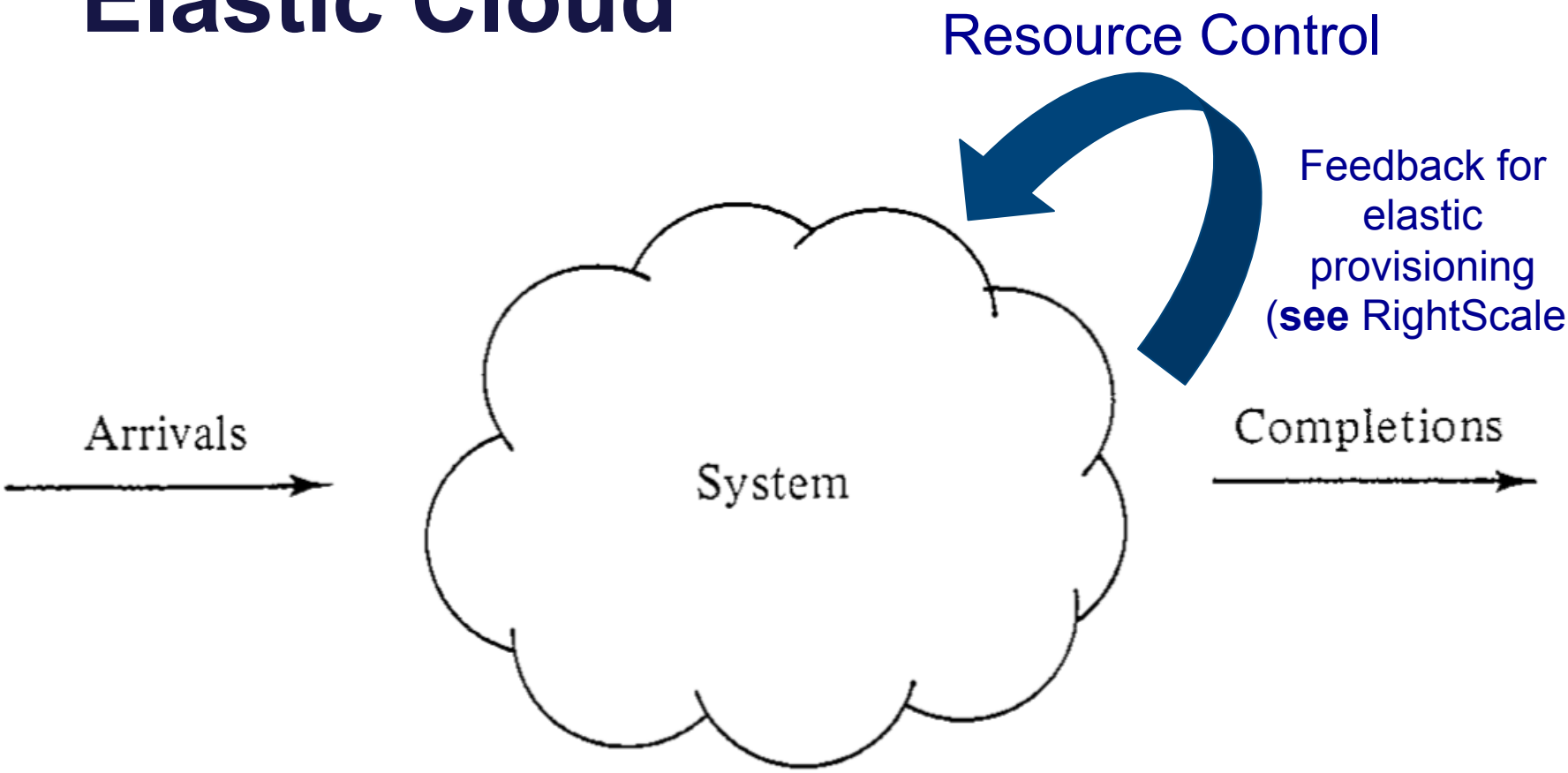


Varying workload

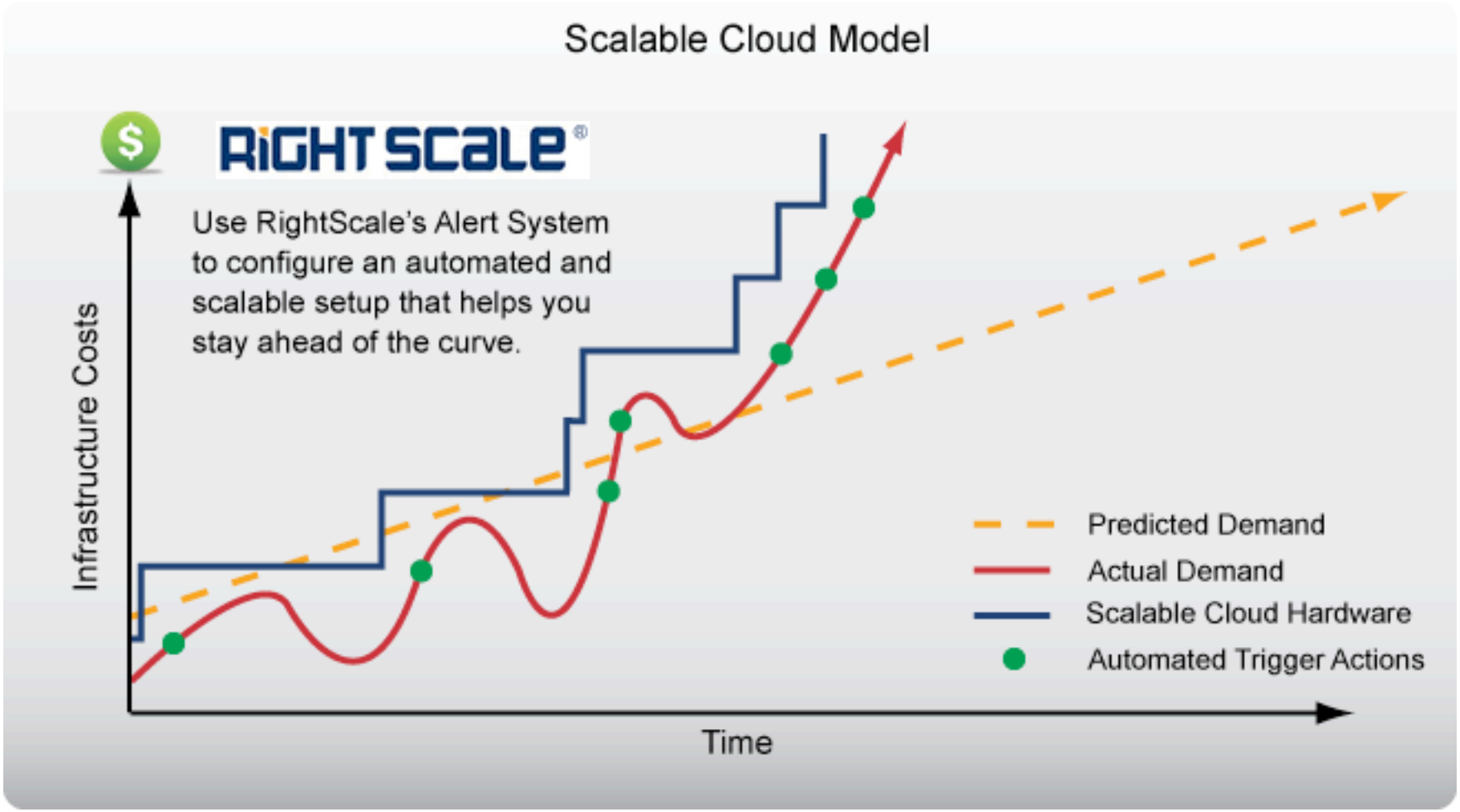
Varying system (use elastic scaling)

Fixed performance (meet SLO)

“Elastic Cloud”



Elastic scaling: “pay as you grow”



Elastic scaling: points

- What are the “automated triggers” that drive scaling?
  - Monitor system measures: N, R, U, X (from previous class)
  - Use models to derive the capacity needed to meet targets
    - Service Level Objectives or SLO for response time
    - target average utilization
- How to adapt when system is under/overloaded?
  - Obtain capacity as needed, e.g., from cloud (“pay as you grow”).
  - Direct traffic to spread workload across your capacity (servers) as evenly and reliably as you can. (Use some replication.)
  - Rebalance on failures or other changes in capacity.
  - Leave some capacity “headroom” for sudden load spikes.
  - Watch out for bottlenecks! But how to address them?

SEDA: An architecture for well-conditioned scalable internet services

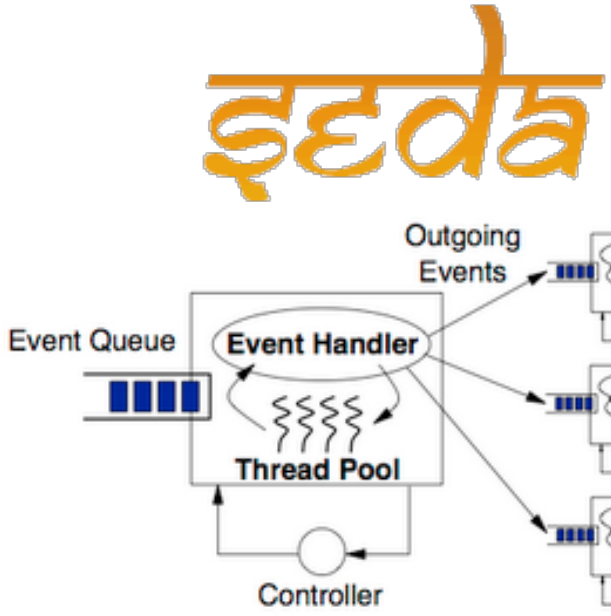
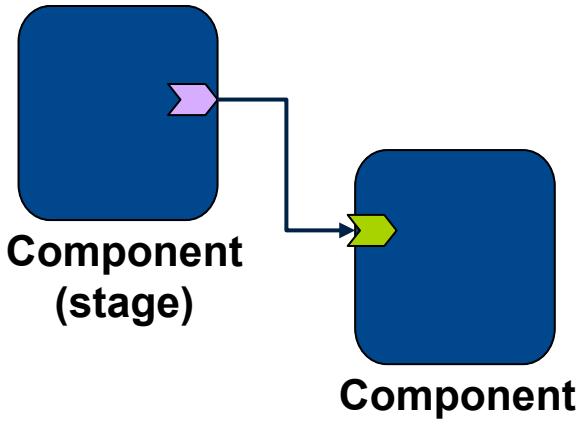
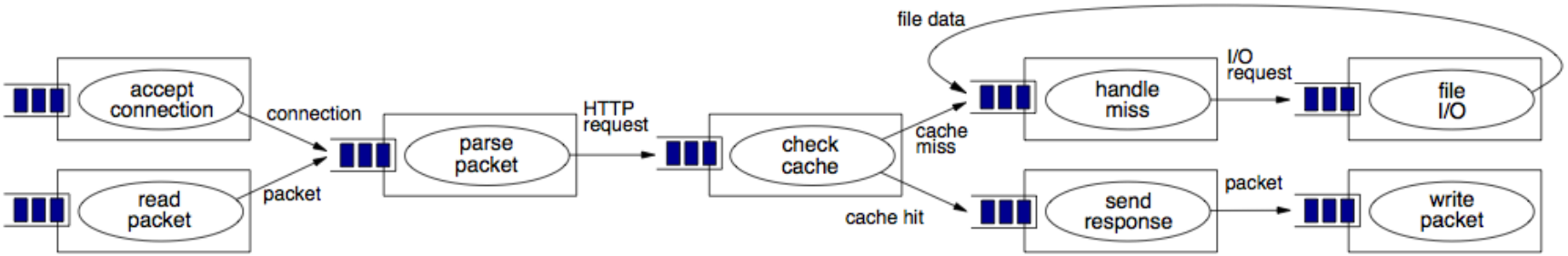


Figure 6: **A SEDA Stage:** *A stage consists of an* incoming event queue, *a* thread pool, *and an application-supplied* event handler. *The stage's operation is managed by the* controller, *which adjusts resource allocations and scheduling dynamically.*

- A 2001 paper, mentioned here because it offers basic insight into server structure and performance.
- Internally, server software is "like" server hardware: requests "flow through" a graph of processing **stages**.
- SEDA is a software architecture to manage this flow explicitly.
- We can control how much processing power to give to each stage by changing the number of servers, or threads dedicated to it (SEDA on a single server).
- We can identify bottlenecks by observing queue lengths. If we must drop a request, we can pick which queue to drop it from.



Staged Event-Driven Architecture (SEDA)



Decompose service into *stages* separated by *queues*

- Each stage performs a subset of request processing
- Stages internally event-driven, typically nonblocking
- Queues introduce execution boundary for isolation and conditioning

Each stage contains a *thread pool* to drive stage execution

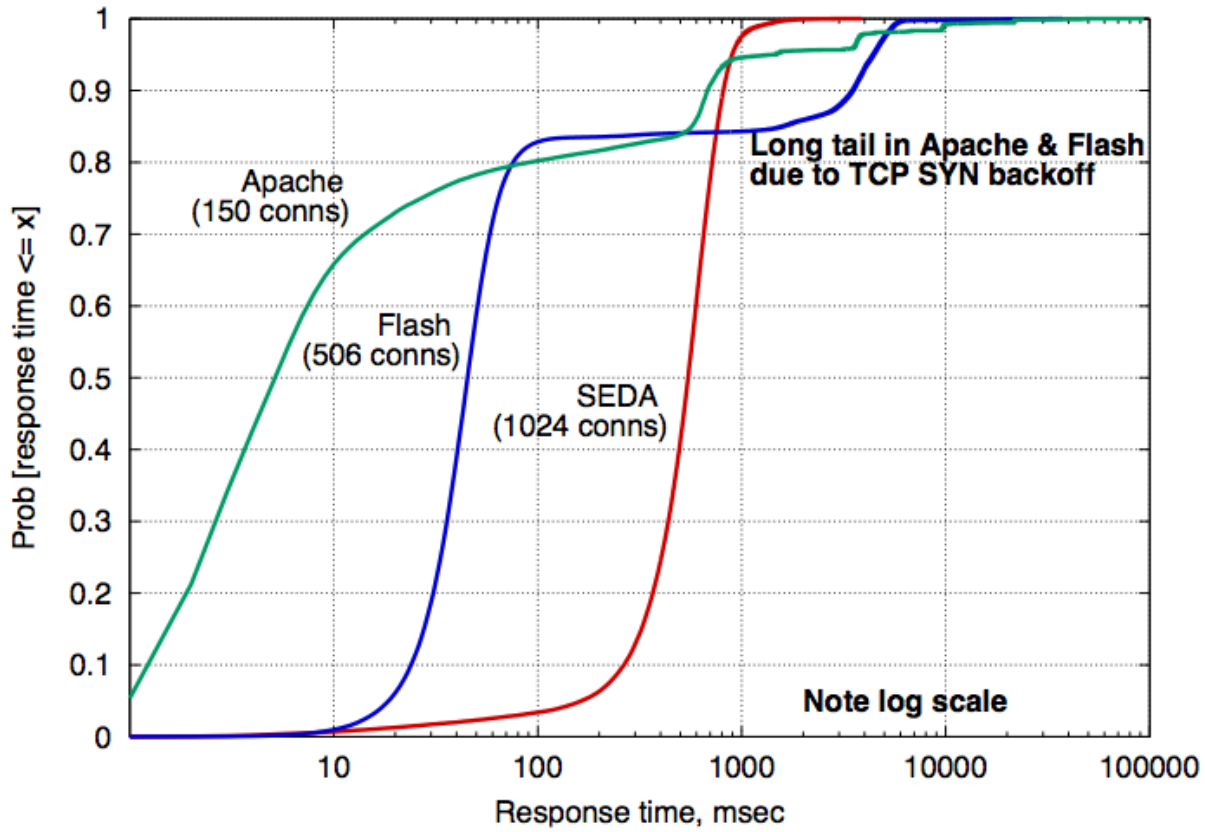
- However, threads are not exposed to applications
- Dynamic control grows/shrinks thread pools with demand
  - *Stages may block if necessary*

Compare to our earlier treatment of event-driven models and thread pools.

Best of both threads and events:

- Programmability of threads with explicit flow of events

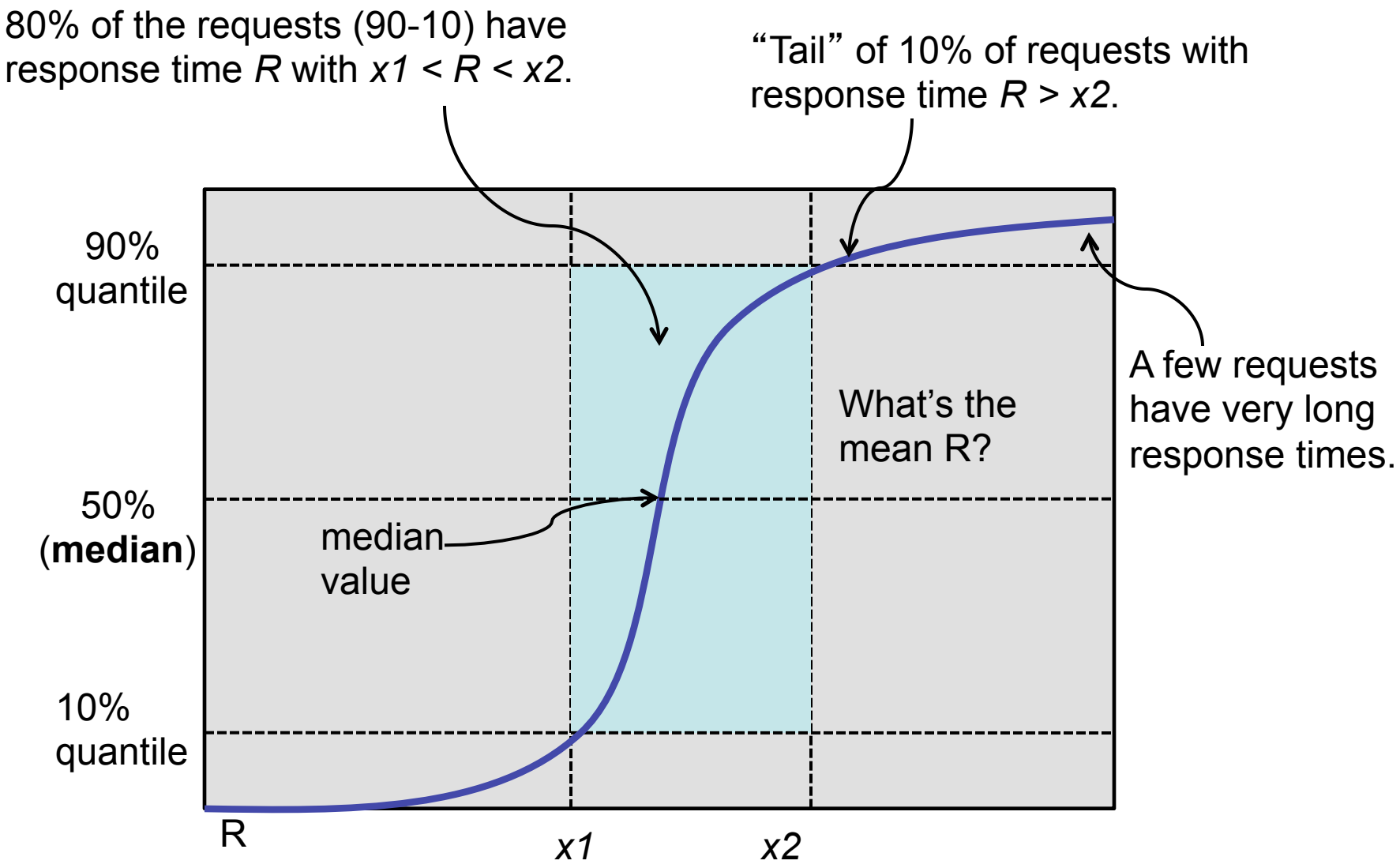
Response Time Distribution - 1024 Clients



	SEDA	Flash	Apache
Mean RT	547 ms	665 ms	475 ms
Max RT	3.8 sec	37 sec	1.7 minutes

- SEDA yields **predictable performance** - Apache and Flash are very **unfair**
  - *''Unlucky'' clients see long TCP retransmit backoff times*
  - *Everyone is ''unlucky'': multiple HTTP requests to load one page!*

Cumulative Distribution Function (CDF)



Understand how/why the mean (average) response time can be misleading.

SEDA Lessons

- Mean/average values are often not useful to capture system behavior, esp. for bursty/irregular measures like response time.
  - You have to look at the actual **distribution** of the values to understand what is happening, or at least the **quantiles**.
- Long response time tails can occur under overload, because (some) queues (may) GROW, leading to (some) very long response times.
  - E.g., consider the “hot spot” example earlier.
- A staged structure (multiple components/stages separated by queues) can help manage performance.
  - Provision resources (e.g., threads) for each stage independently.
  - Monitor the queues for bottlenecks: underprovisioned stages have longer queues.
  - Choose which requests to drop, e.g., drop from the longest queues.
- **Note**: staged structure can also help simplify concurrency/locking.
  - SEDA stages have no shared state. Each thread runs within one stage.

Part 3

LIMITS OF SCALABLE PERFORMANCE

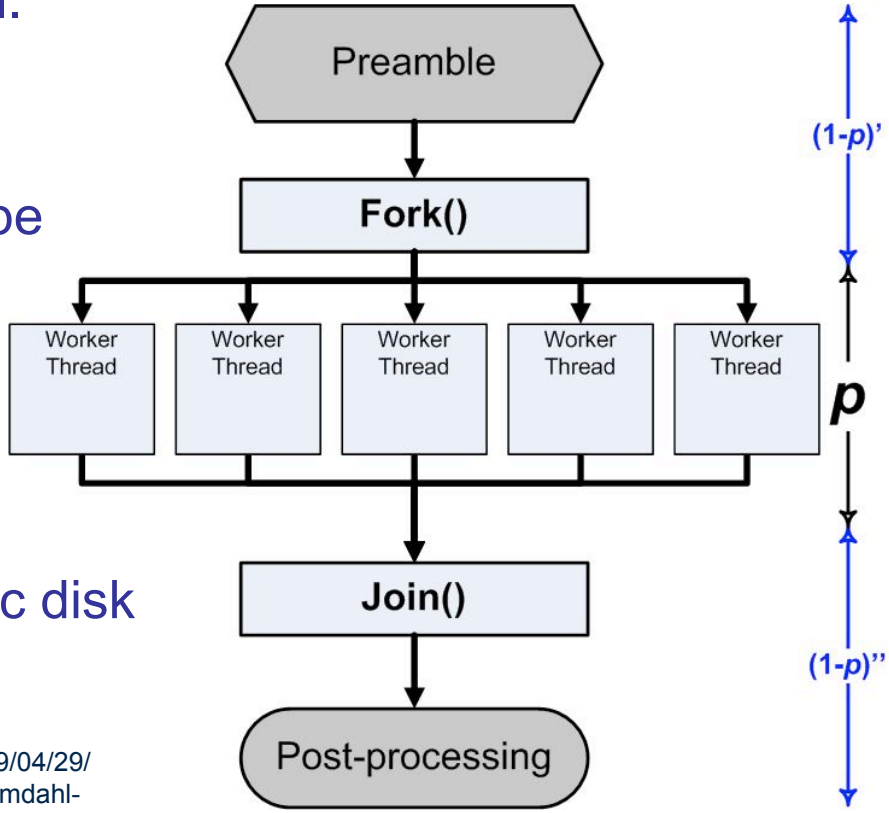
Parallelization

A simple treatment

A program has some work to do. We want to do it fast. How?
Do it on multiple computers/cores in parallel.

But we won't be able to do **all** of the work in parallel.

Some portion will be **serialized.**
E.g.:
startup,
locking
combining results
access to a specific disk



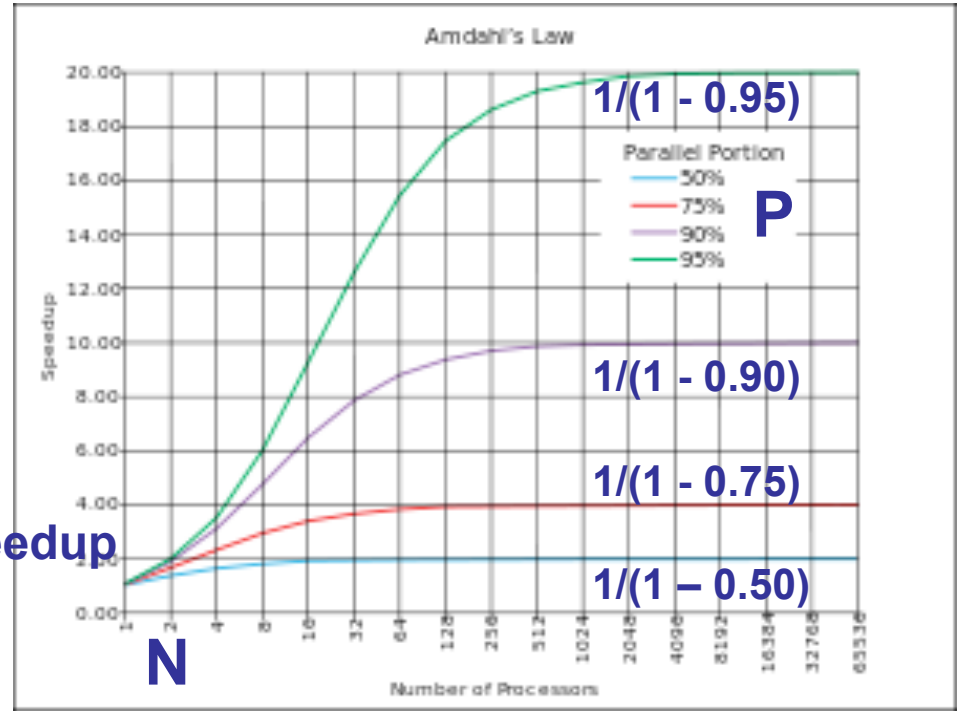
Suppose some portion **p** of the work can be done in parallel.

Then a portion **1-p** is serial.

How much does that help?

http://blogs.msdn.com/b/ddperf/archive/2009/04/29/parallel-scalability-isn-t-child-s-play-part-2-amdahl-s-law-vs-gunther-s-law.aspx

Amdahl's Law



Law of Diminishing Returns

"Optimize for the primary bottleneck."

Normalize runtime = 1
(On a single core.)
Now **parallelize**:

Parallel portion: **P** ($0 \leq P \leq 1$)
Serial portion: **1-P**

N-way parallelism (N cores)
Runtime is now:
P/N + (1-P)

Even if "infinite parallelism", runtime is 1-P in the limit. It is determined by the serial portion.
Bottleneck: limits performance.

Speedup = before/after
Bounded by 1/(1-P)

Amdahl's Law

In the case of parallelization, Amdahl's law states that if *P* is the proportion of a program that can be made parallel (i.e., benefit from parallelization), and (1 – *P*) is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using *N* processors is

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

In the limit, as *N* tends to infinity, the maximum speedup tends to 1 / (1 – *P*). In practice, performance to price ratio falls rapidly as *N* is increased once there is even a small component of (1 – *P*).

As an example, if *P* is 90%, then (1 – *P*) is 10%, and the problem can be sped up by a maximum of a factor of 10, no matter how large the value of *N* used. For this reason, parallel computing is only useful for either small numbers of processors, or problems with very high values of *P*: so-called embarrassingly parallel problems. A great part of the craft of parallel programming consists of attempting to reduce the component (1 – *P*) to the smallest possible value.

What is the "serial portion" that "cannot be parallelized"?

- Mutexes/critical sections
- Combining results from parallel portions (e.g., "reducers")
- …

“Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”

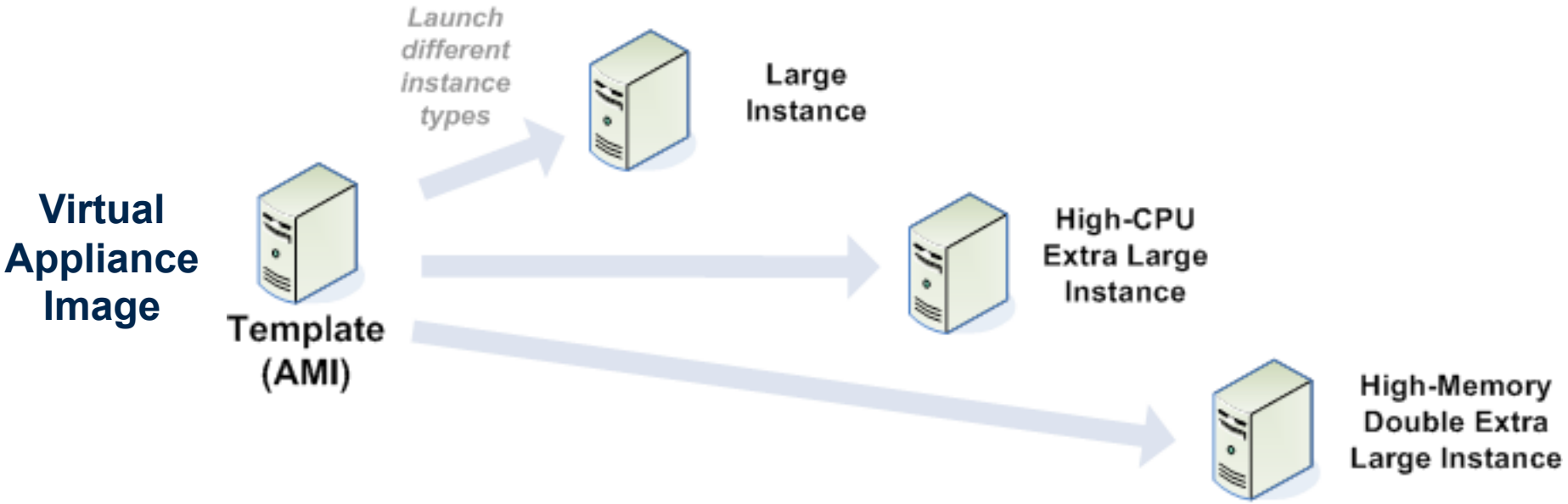
- US National Institute for Standards and Technology

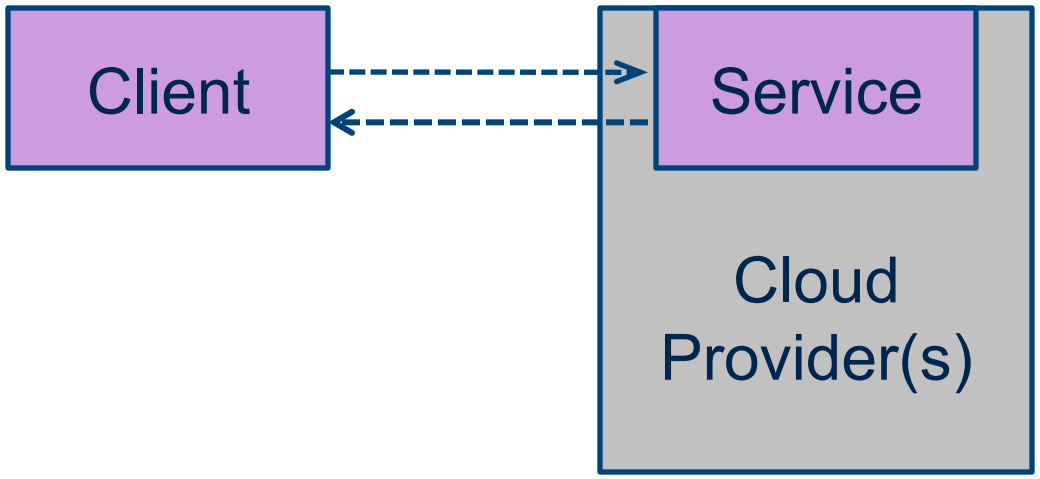
http://www.csrc.nist.gov/groups/SNS/cloud-computing/

Part 4

VIRTUAL CLOUD HOSTING

EC2 Elastic Compute Cloud
The canonical public cloud



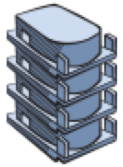


AWS Free Tier

Launch new applications, test existing applications in the cloud, or simply gain hands-on experience with AWS.

Get started with AWS for Free »

Compute
Amazon EC2
750 hours/month*



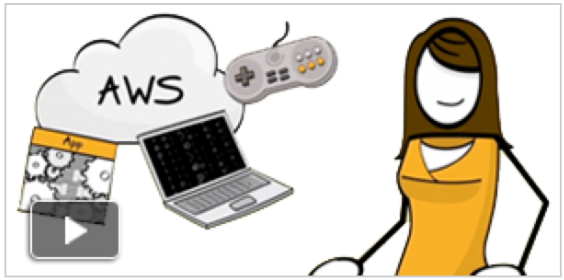
Storage
Amazon S3
5 GB*

Database
DynamoDB
100 MB of
SSD-backed storage*

Get Started for Free » **Launch virtual machines and apps in minutes.**

WHAT IS CLOUD COMPUTING?

Learn the benefits of Cloud Computing with AWS



WHAT IS AMAZON WEB SERVICES?

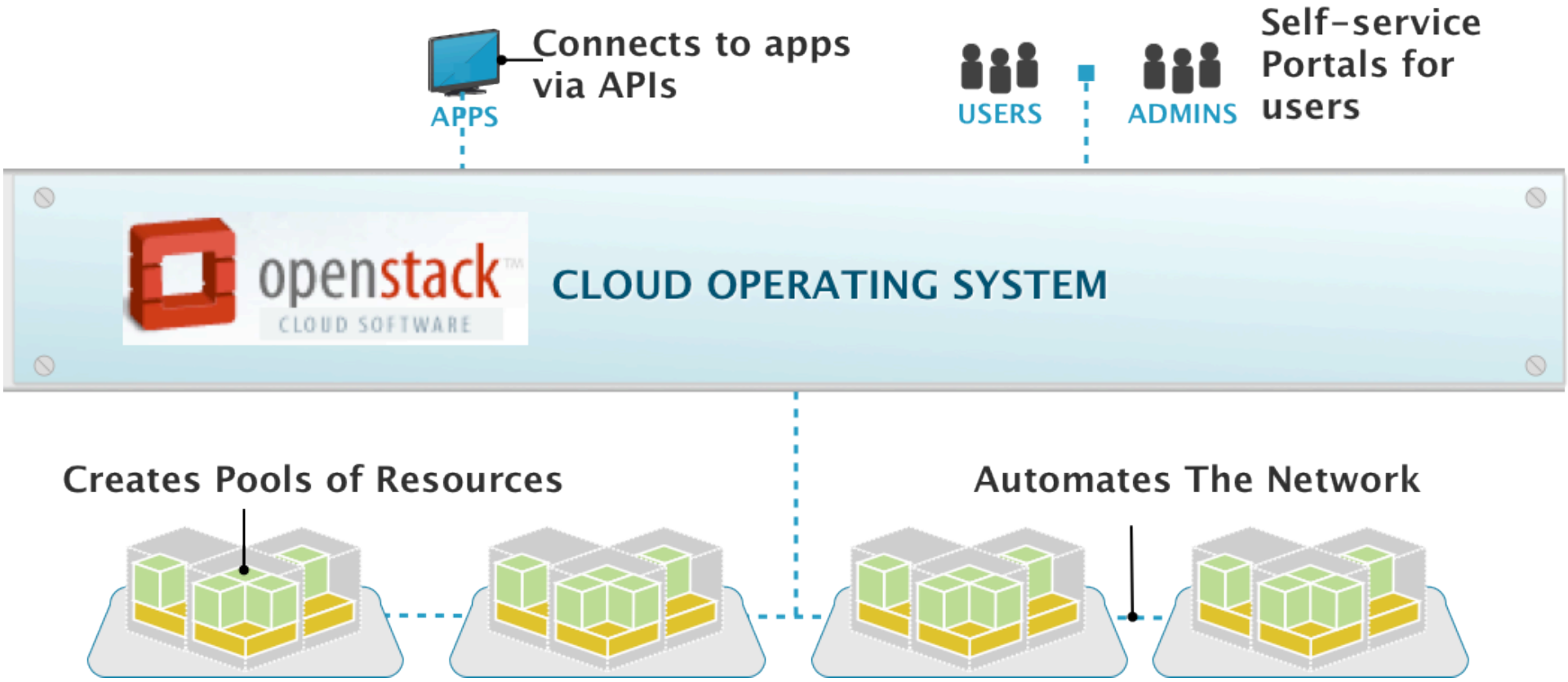
Learn about the AWS platform, products and services

GET STARTED WITH AWS

Start using AWS in under 15 minutes

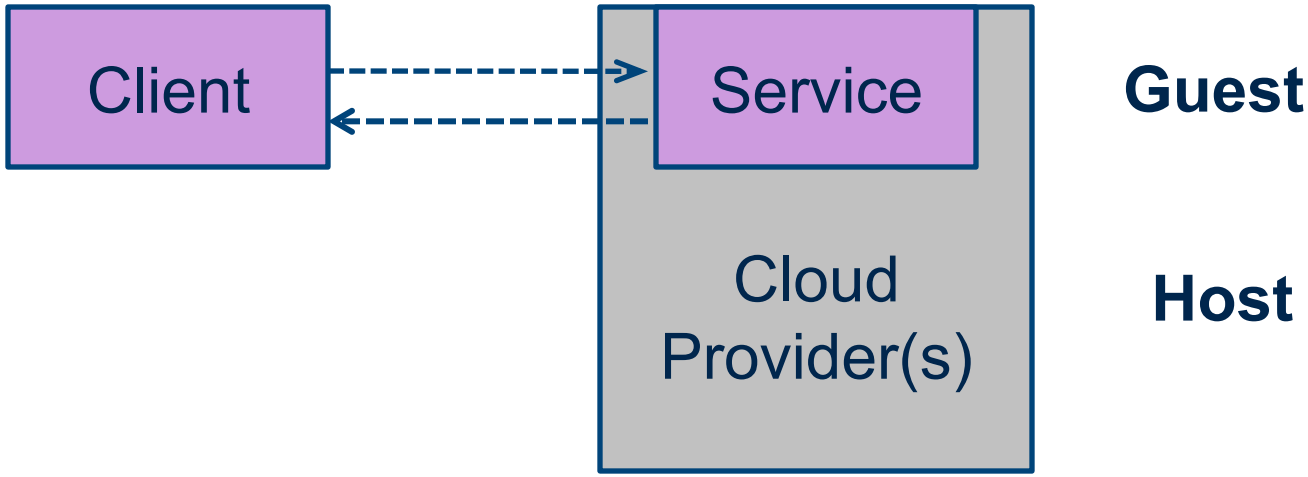
OpenStack, the Cloud Operating System

Management Layer That Adds Automation & Control



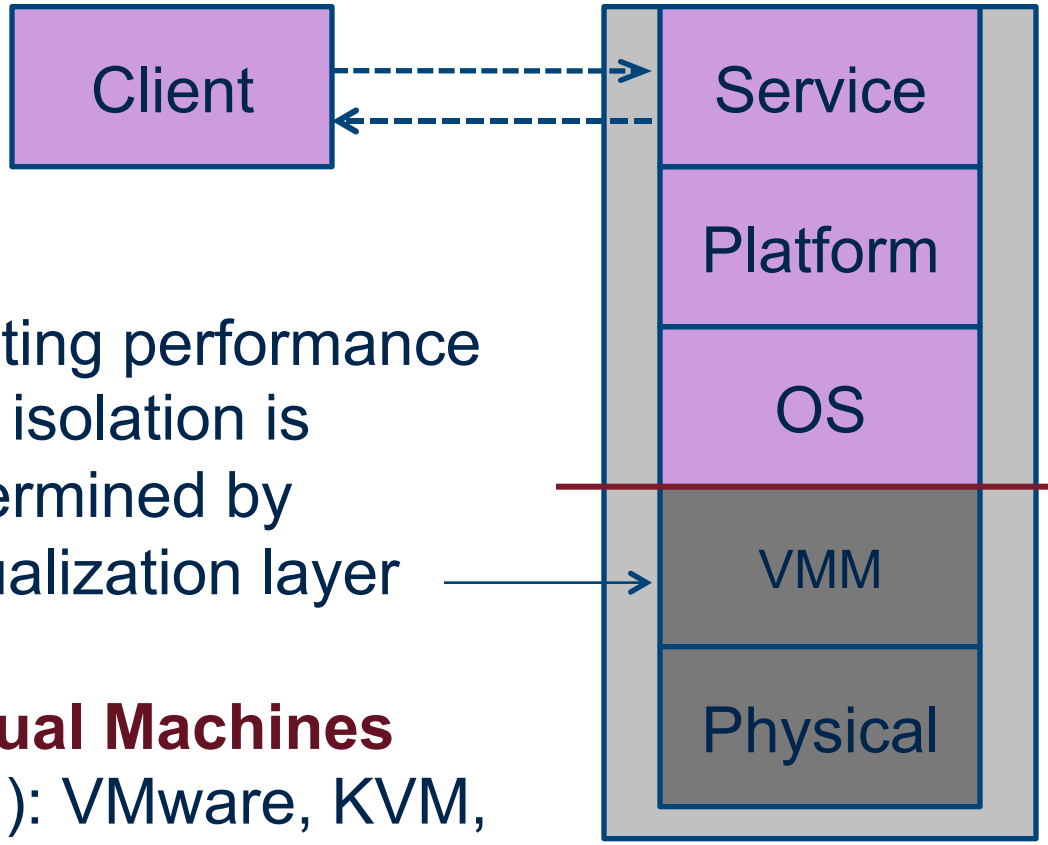
[Anthony Young @ Rackspace]

Host/guest model



- **Service is hosted by a third party.**
  - **flexible programming model**
  - **cloud APIs for service to allocate/link resources**
  - **on-demand: pay as you grow**

IaaS: Infrastructure as a Service



EC2 is a public IaaS cloud (fee-for-service).

Deployment of private clouds is growing rapidly w/ open IaaS cloud software.

Hosting performance and isolation is determined by virtualization layer

Virtual Machines (VM): VMware, KVM, etc.

XCAT

openstack

The New York Times

December 3, 2013

Google Joins a Heavyweight Competition in Cloud Computing

By QUENTIN HARDY

MOUNTAIN VIEW, Calif. — Google already runs much of the digital lives of consumers through email, Internet searches and YouTube videos. Now it wants the corporations, too.

The search giant has for years been evasive about its plans for a so-called public cloud of computers and data storage that is rented to individuals and businesses. On Tuesday, however, it will announce pricing, features and performance guarantees aimed at companies ranging from start-ups to multinationals.

It is the latest salvo in an escalating battle among some of the most influential companies in technology to control corporate and government computing through public clouds. That battle, which is expected to last years and cost the competitors billions of dollars annually in material and talent, already includes Microsoft, IBM and Amazon.

As businesses move from owning their own computers to renting data-crunching power and software over the Internet, this resource-rich foursome is making big promises about computing clouds. Supercomputing-based research, for example, won't be limited to organizations that can afford supercomputers. And tech companies with a hot idea will be able to get big fast because they won't have to build their own computer networks.

Over the last several years, each of the big cloud providers has built a global network of over a million computer servers. In the process, the companies are rethinking almost every step to maximize efficiency and power. Intel, the world's largest semiconductor maker, now has six salespeople assigned full time to Amazon, feeding a continuous appetite for new computers.

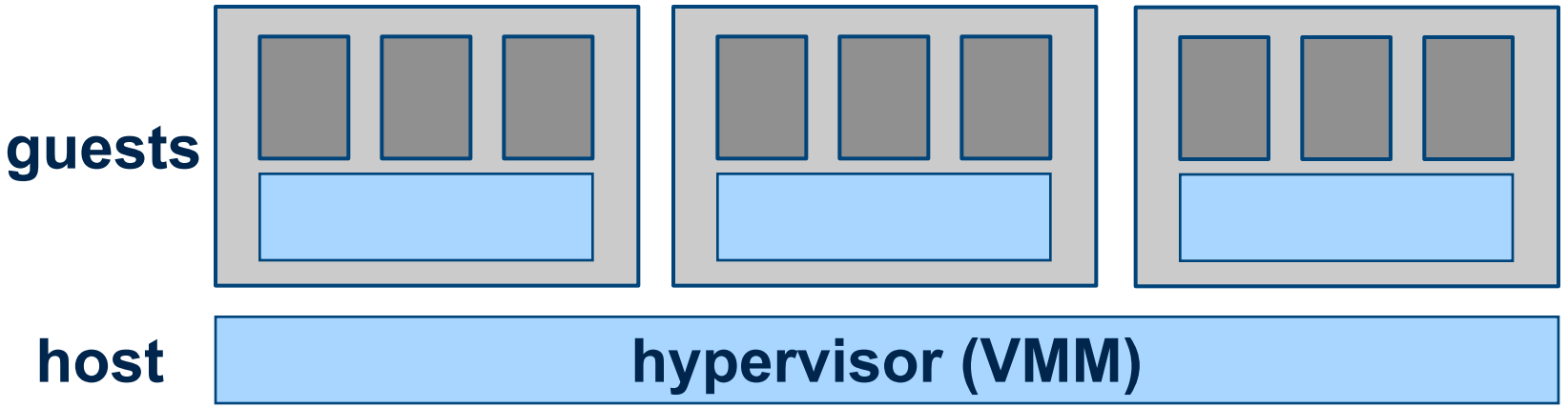
The biggest promise of these clouds is their ability to make it easy to do things that would have cost millions of dollars in hardware just a few years ago.

That is, unless you want to build your own public cloud. Executives at all four public cloud competitors say there is no college course or professional training for running computers at this scale: The only real way to learn how to do it is by working at the handful of companies with the resources to pull it off.

"We're giving people the same services we rely on to run Google," said Mr. DeMichillie. "I wouldn't say spending billions of dollars doesn't matter, but there is a learning by doing in this, too; hard information problems we've tackled."

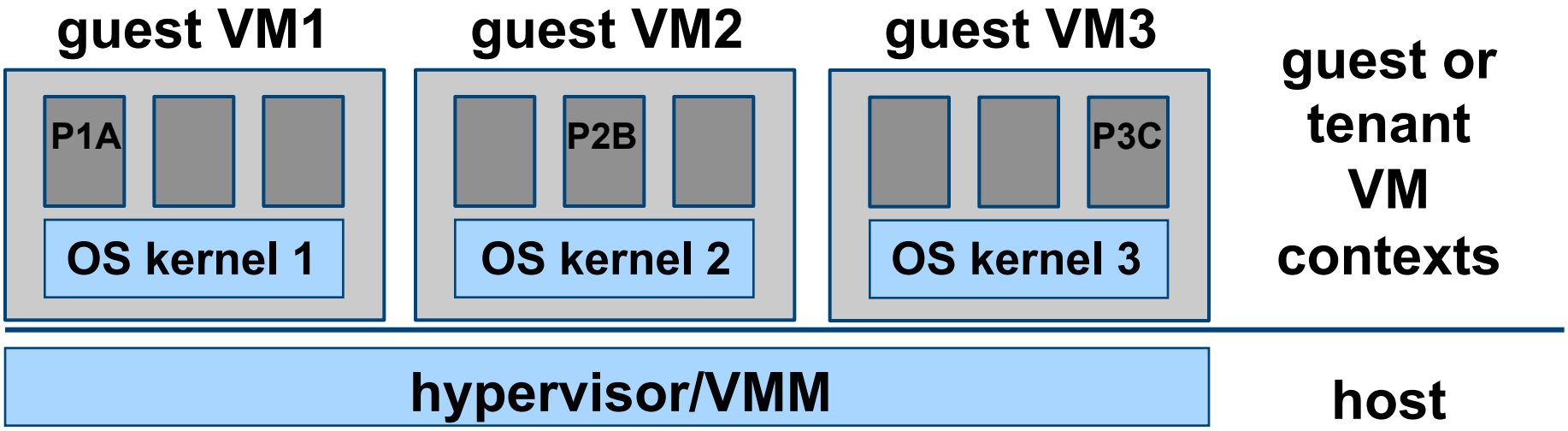
Native virtual machines (VMs)

- Slide a **hypervisor** underneath the kernel.
  - New OS layer: also called **virtual machine monitor (VMM)**.
- Kernel and processes run in a **virtual machine (VM)**.
  - The VM "looks the same" to the OS as a physical machine.
  - The VM is a sandboxed/isolated context for an entire OS.
- Can run multiple VM **instances** on a shared computer.




vmware


Xen


guest VM1
guest VM2
guest VM3
guest or tenant VM contexts
P1A
P2B
P3C
OS kernel 1
OS kernel 2
OS kernel 3
hypervisor/VMM
host


KVM

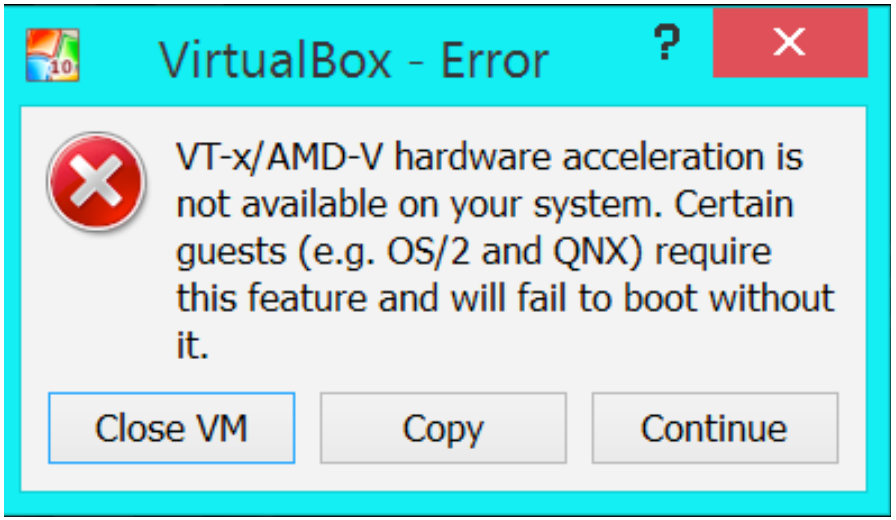

Sun
VirtualBox


Parallels

Virtualization support: VT

- These VMs can run a full OS with a kernel and multiple processes with **direct execution**: they are not interpreted!
- Kernel, process, and hypervisor all run on the same cores, at full speed. (Note: distinct from Java JVM.)
- VMs used to be implemented in all sorts of goofy ways.
- Since 2007-2010 chip vendors offer hardware support.

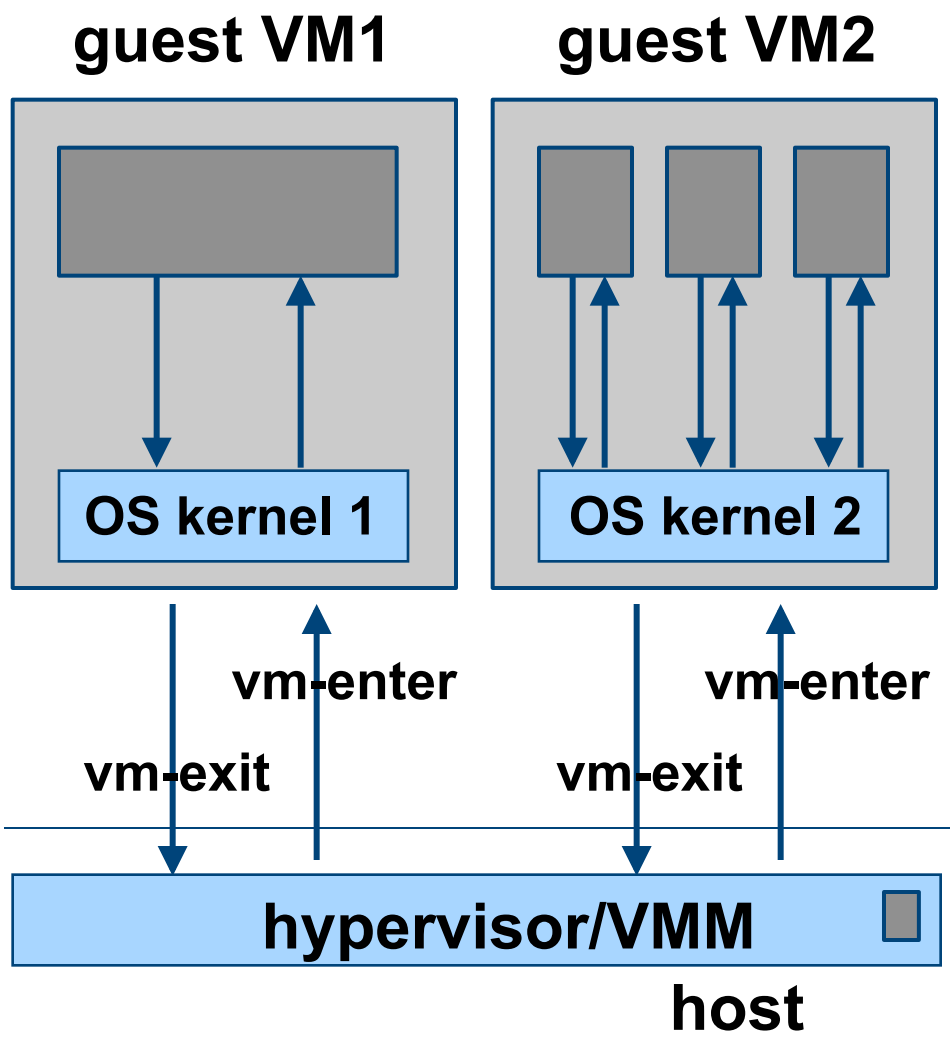
Intel VT and VT-d, AMD-V
+ new CPU modes
+ new CPU events/transitions
+ a new level of VA translation
  Extended Page Tables (EPT)



VT in a Nutshell

- **New VM mode bit**
  - **Orthogonal to CPL (e.g., kernel/user mode)**
- **If VM mode is off → host mode**
  - Machine "looks just like it always did" ("**VMX root**")
- **If VM bit is on → guest mode**
  - Machine is running a guest VM: "**VMX non-root mode**"
  - Machine "looks just like it always did" to the guest, BUT:
  - Various events trigger gated entry to hypervisor (in VMX root)
  - A "**virtualization intercept**": exit VM mode to VMM (**VM Exit**)
  - Hypervisor (VMM) can control which events cause intercepts
  - Hypervisor can examine/manipulate guest VM state and return to VM (**VM Entry**)

VT: core modes



VMX non-root mode
"guest mode"
(user or kernel)

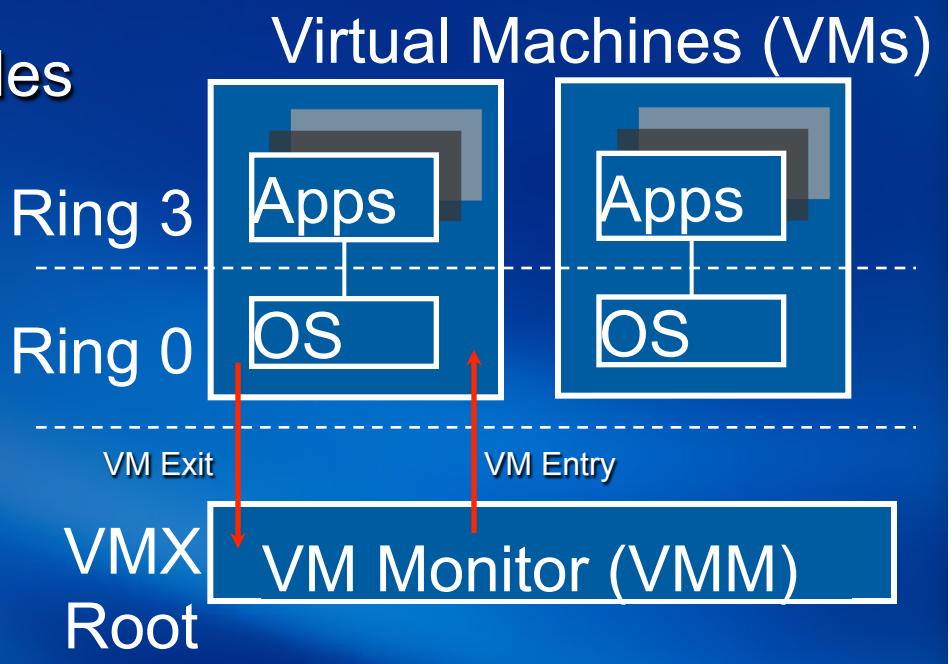
CPU events
trap, fault, interrupt
and **return**

VMExit and **VMEnter**
intercept and return

VMX root mode
"host mode"
(user or kernel)

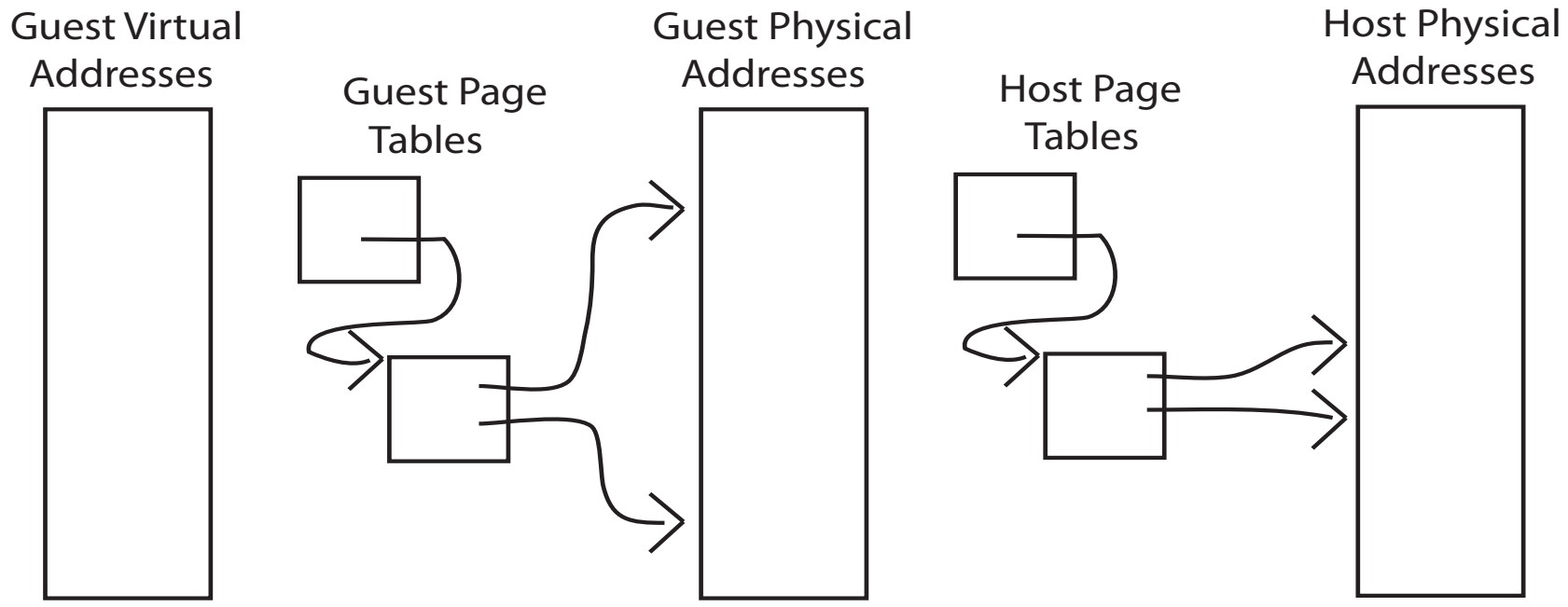
CPU Virtualization With VT-x

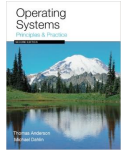
- Two new VT-x operating modes
  - Less-privileged mode (VMX non-root) for guest OSes
  - More-privileged mode (VMX root) for VMM
- Two new transitions
  - VM entry to non-root operation
  - VM exit to root operation



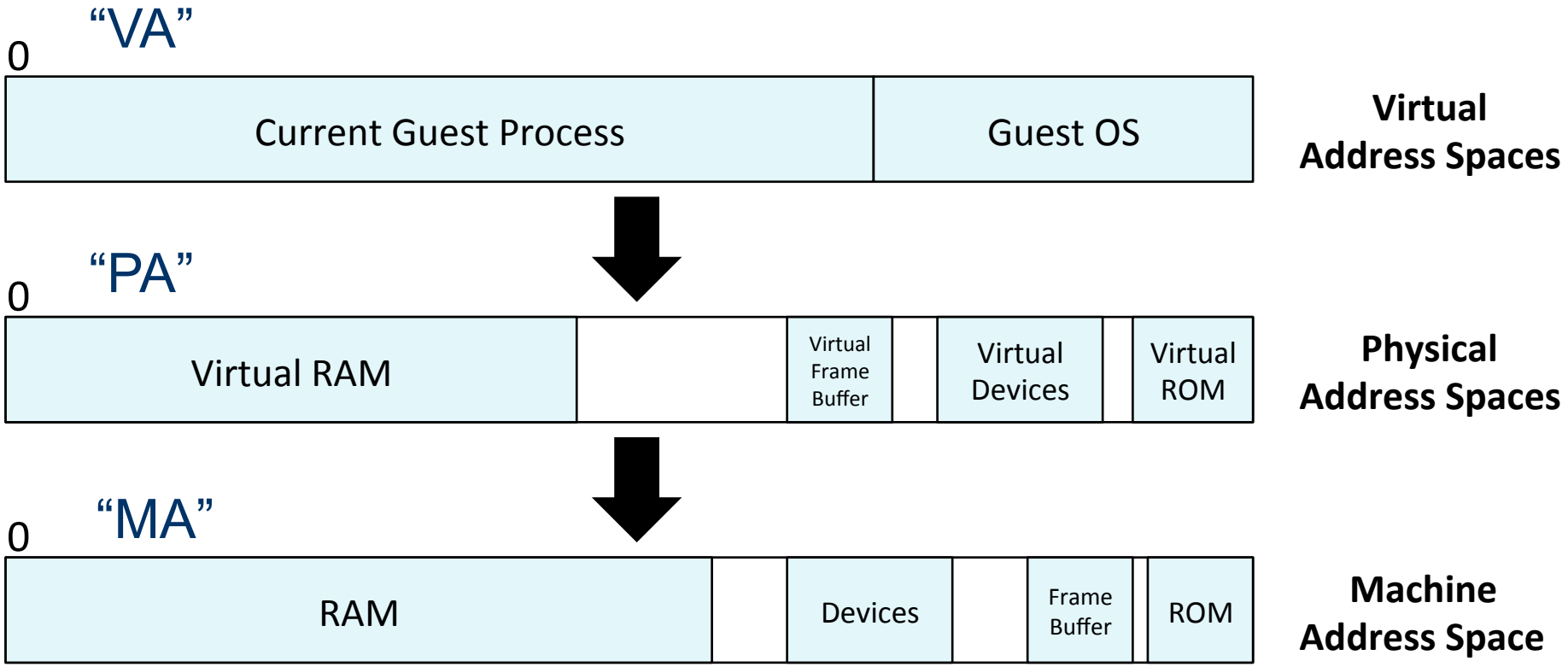
- Execution controls determine when exits occur
  - Access to privilege state, occurrence of exceptions, etc.
  - Flexibility provided to minimize unwanted exits
- VM Control Structure (VMCS) controls VT-x operation
  - Also holds guest and host state

Virtual Machines + Virtual Memory

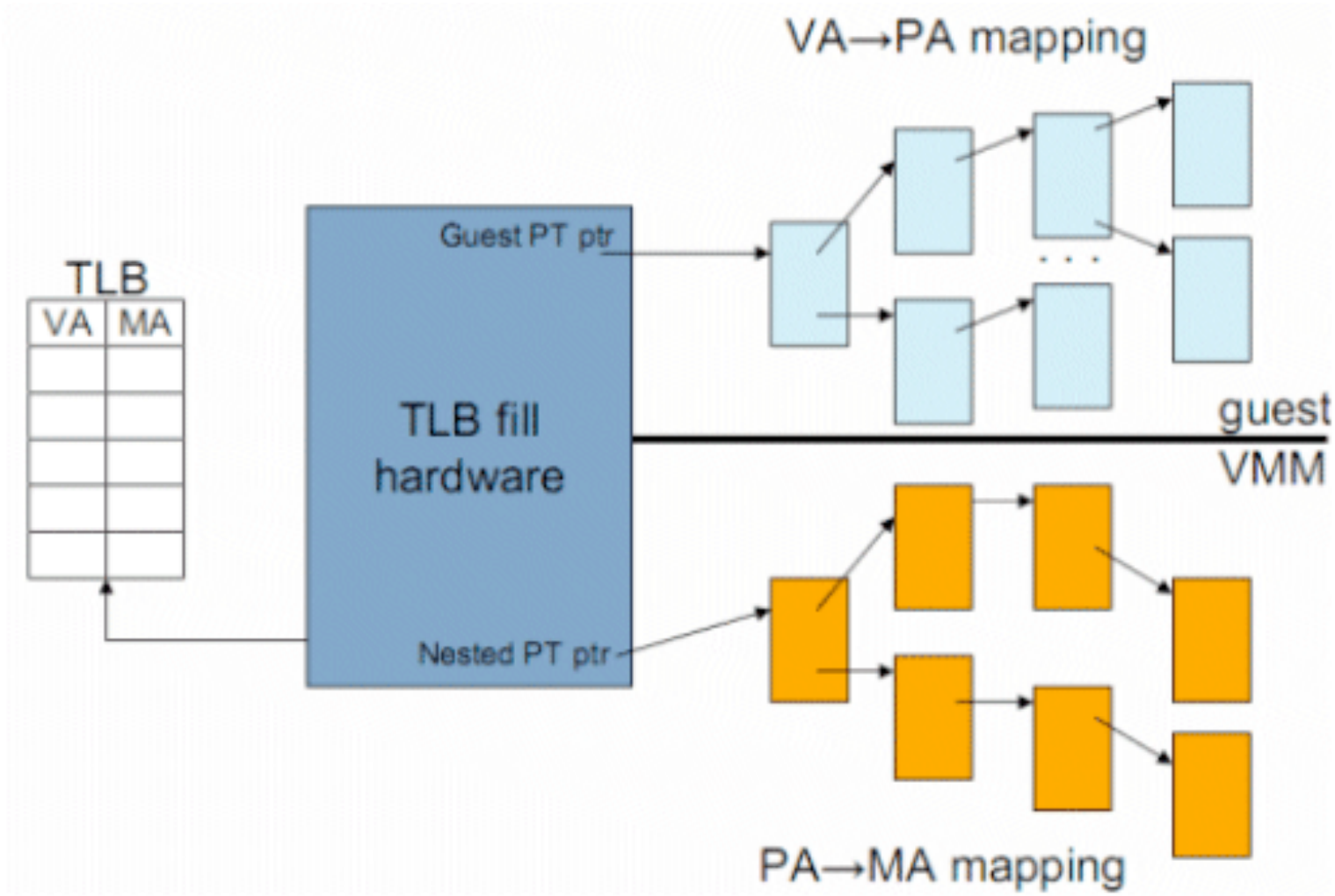




Three address spaces



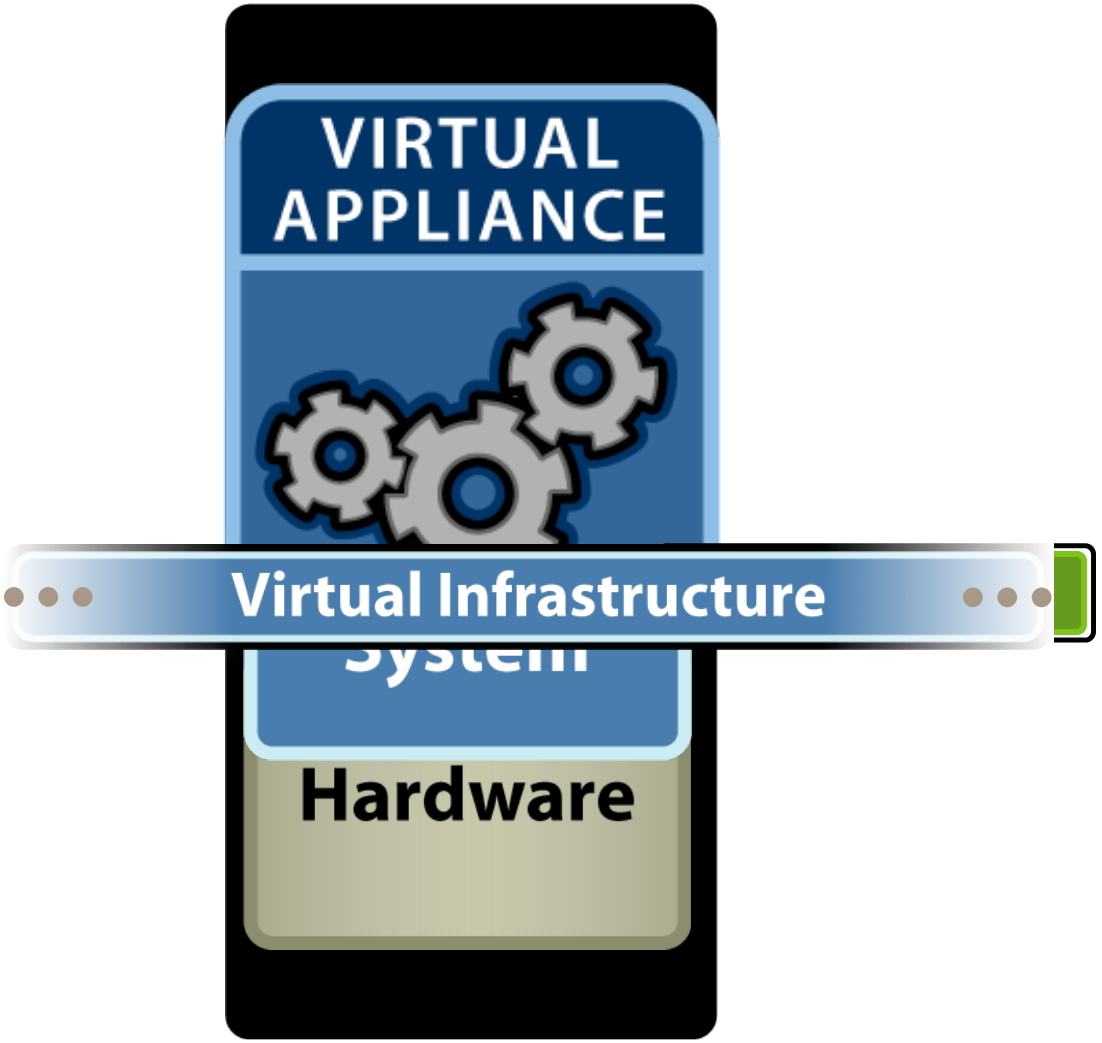
TLB caches VA->MA mappings



Image/Template/Virtual Appliance

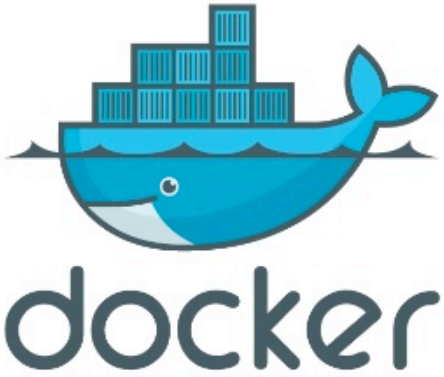
- A **virtual appliance** is a program for a virtual machine.
  - Sometimes called a **VM image** or **template**
- The image has everything needed to run a virtual server:
  - OS kernel program
  - file system
  - application programs
- The image can be instantiated as a VM on a cloud.
  - Not unlike running a program to instantiate it as a process

Thank you, VMware



Containers

- Note: lightweight **container** technologies offer a similar abstraction for software packaging and deployment, based on an extended process model.
  - E.g., Docker and Google Kubernetes



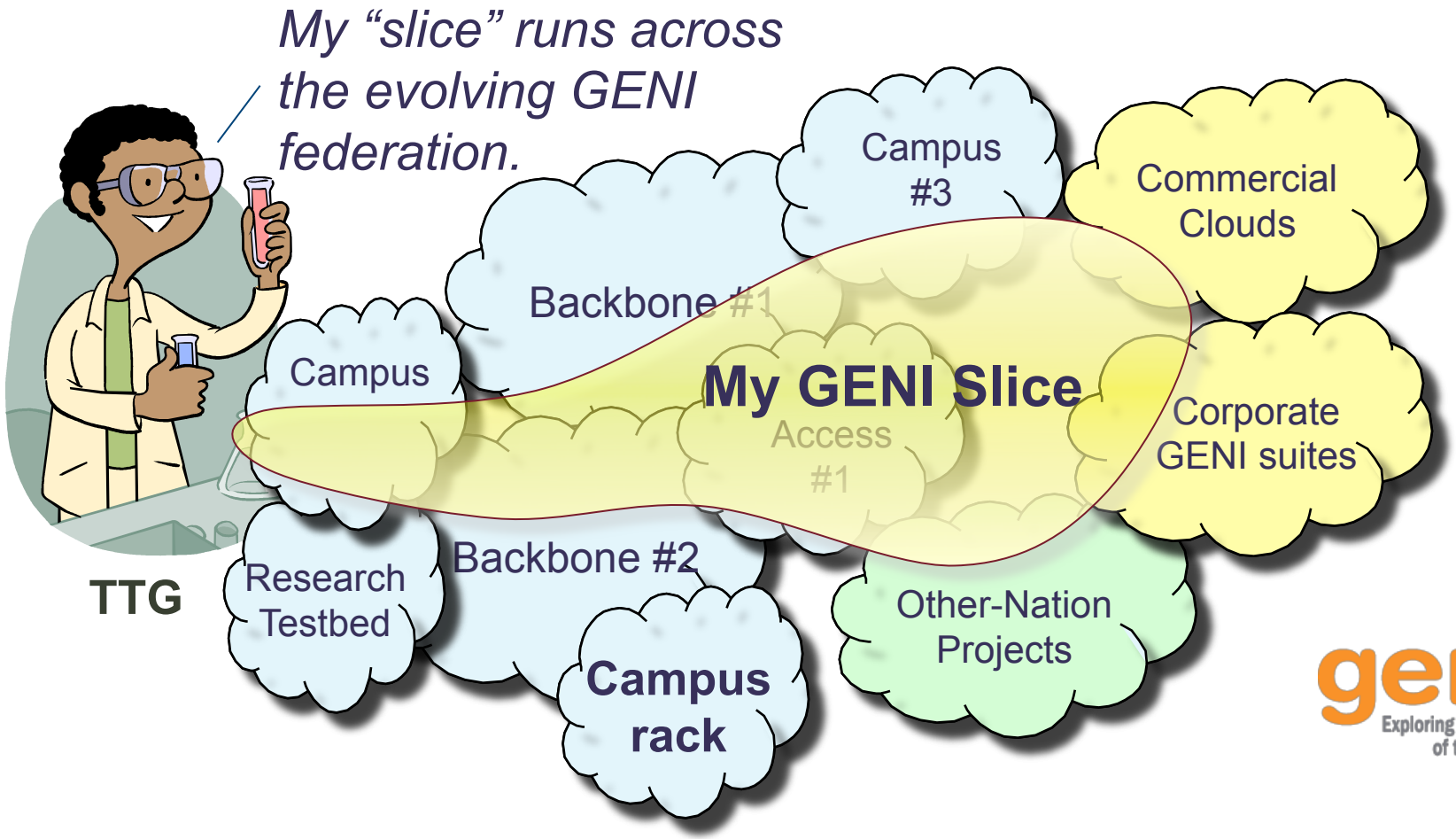
Docker, Containers, and the Future of Application Delivery

Part 5

NOTREACHED

Not to be tested.

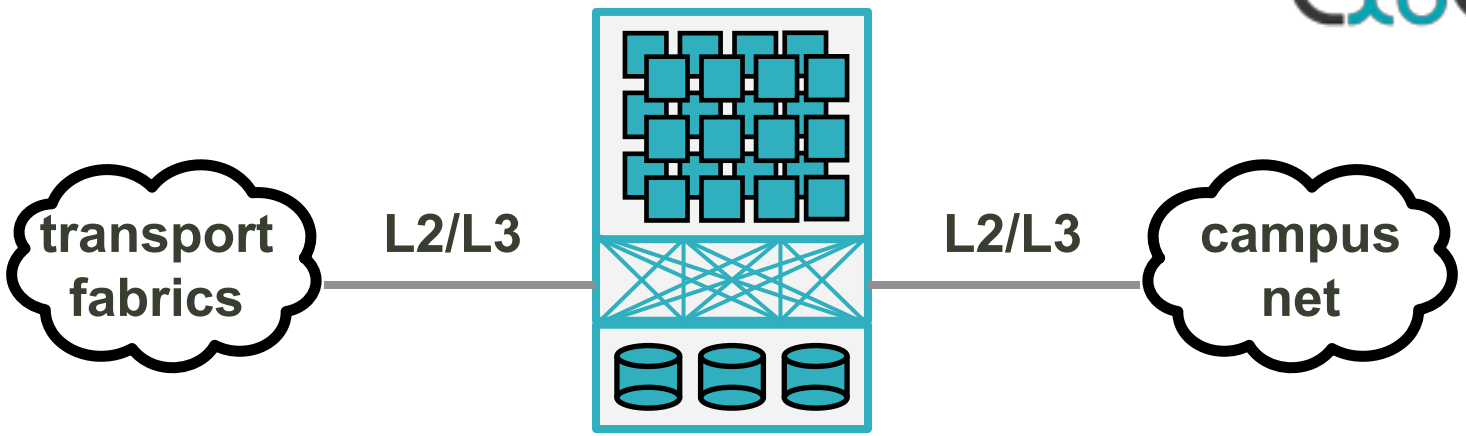
GENI: slices and federation



Slice: an end-to-end virtual network context spanning multiple sites, with configurable topology and properties, e.g., containment and isolation.

Not to be tested.

ExoGENI.net



ExoGENI Rack

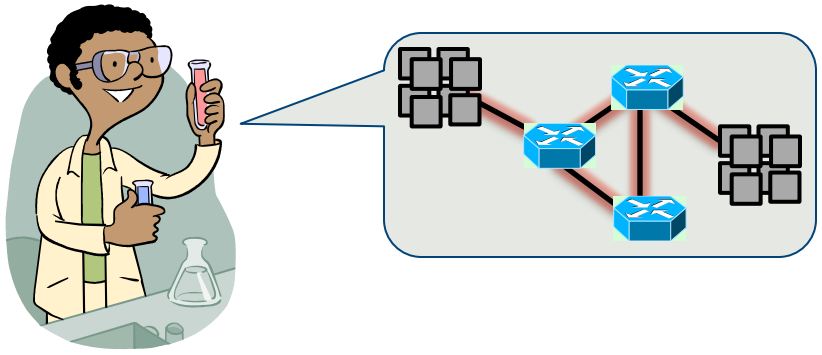
A packaged small-scale cloud site for a campus, lab, or PoP.

Linked to a federated hosting platform for tenant networks (slices).

Not to be tested.

ExoGENI.net: cartoon version

Not shown: dynamic slice adaptation under automated control



"Make my slice."

GENI control framework for federated orchestration
Open Resource Control Architecture

orca

*"**Instantiate** VMs and VLANs x, y, z."*

*"**Link** sites with circuits."*

*"**Enable** external SDN controller for x, y, z."*

Site A

openstack™

Site B

Network provider

OpenFlow