Can "scale" cloud applications "on the edge" by adding server instances. (So far, haven't considered scaling the interior of the cloud).

Recall: where we are

Distributed services form an inner core (JDO)

App servers form an outer edge
The safe operating bounds of the edge nodes of the cloud are based upon:

- how demand changes over time
- how quickly you can re-allocate server power to the edge nodes.

Key tool: virtualization.

Assumptions for today's discussion:

- Flowless switching between edge nodes.
- Response time $P = t_{server} + t_{network} + t_{cloud}$.
- $t_{network}$ is just client-to-edge and edge-to-client.
- $t_{cloud}$ includes networking inside the cloud.
- Both of these are subject to their own SLAs.

So we concentrate on keeping $t_{server}$ within reasonable bounds.
Virtualization
So far: we know how to throw server power at an application to react to demand and **increase throughput**. Next step: how to switch a server from one role to another. Key tool: **server virtualization**
What is virtualization?

In its simplest form, creating a fictional version of a thing that acts like the real thing. This is called "virtualizing a thing". Many possible things...
Several kinds of virtualization:

**Machine virtualization**: thinking of a physical machine as "hosting" one or more virtual machines.

**I/O virtualization**: sharing an I/O device with several running operating systems, perhaps even on different physical machines.

**Filesystem virtualization**: making one instance of an operating system behave as if it were multiple, distinct instances by controlling filesystem access.

**Program virtualization**: program is compiled to run on a virtual machine, e.g., the Java Virtual Machine (JVM).
Mostly, we'll be concerned with machine (hardware) virtualization:

- make **one physical machine** act like **many virtual machines**.
- "host" various OS instances (either similar or different) on different virtual machines.
- Turn OS instances on or off as needed.
Why we need machine virtualization

Repurposing a server can require rebuilding the whole operating system. This can take (relatively) a lot of time.

Virtualizing the machine allows two or more operating systems to share it. (One can be running normally even while another is being built.)

Also, whole operating system instances can be prebuilt and latent, ready to start running when needed.
Most things about machine virtualization are easy.
   Running two operating systems is like running two "programs".
     Separate sections of memory.
     Separate disk.
Hard part: sharing resources and devices.
Devices have state.
   A disk drive doesn't take well to being asked to read something during a write.
   An I/O driver is a piece of software designed to maintain proper device state.
Therefore, most of the work of machine virtualization is I/O virtualization.
Some common attributes of all I/O virtualization strategies.

Each device has **one real driver.**
Other drivers may seem to contact the device directly, but in fact, they talk through that one "master" driver.
Main difference between strategies: **where that master driver is located.**
The hypervisor

Best thought of as a **miniature operating system** for the machine
Runs instances of operating systems as if they were **application programs**!
Each instance
  is (theoretically!) **unaware** of the other instances.
  has its **own memory and disk**.
  Is **scheduled** to use the CPU(s) by the **hypervisor**.
Two virtualization approaches

VMWare: closed-source hypervisor

http://www.vmware.com

Xen: open-source hypervisor

Domain 0 guest: Does I/O
Domain U guest: Regular OS
I/O done outside hypervisor.
Two keys to machine virtualization

**Meta-scheduling**: giving each virtual machine a time slice of the real machine.

**I/O virtualization**: making sure that each machine gets a consistent view of each I/O device.
Meta-scheduling

Give each OS a chance to run in turn.
Round-robin, no priority.
"Slows down" each OS by the same amount.
Can give more "busy" guest OS's more time.
But...

In this course we are not really concerned about how virtualization works
Rather, we are concerned with how to use it to obtain cloud-like behavior.
Arguments for machine virtualization

Latency hiding: OS and application instances spend much of their time "waiting" for data.

Server consolidation: easier to manage one physical server instead of multiple ones.

Conflict avoidance: application instances can require conflicting versions of OS instances.

Online build: can build OS instances while others are running.

Quick deployment: can hold OS instances "in reserve" and deploy them very quickly.

Quick "sleep" and "wake": can boot an instance and then turn it off (sleep) until needed.
Two requests don't take 2x CPU time
Cloud apps spend much of their time "waiting for the cloud" to return data.

Example: read **blocks** and **waits** until target data is known to be consistent.

So, multiple cloud apps can run on the **same hardware** and use the time that other apps **spend waiting**. This is called **latency hiding**.
Deployment time

It takes time from deciding to deploy a new server instance to time when it is available for switching.

On physical hardware:
- **can take 30 minutes** to build a server instance.

On virtual machine
- **Can build a server instance in advance** and keep it available and unused.
- **Can share memory** between (exclusively activated) instances.
- **Disk can't be shared (easily).**

In either case,
- **60 seconds** to boot a server instance.
- **10 seconds** to wake up a sleeping instance.
Why is deployment time important?

Demand changes over time.
SLA limits are constant over time.
Response time increases with demand.
Speed with which one can deploy a server determines safety margin for avoiding SLA violations.
SLA violations cost money!
Scalability and Elasticity

**Scalability**: a property of the application that allows deployment to **scale with demand** to preserve response time.

An application is **scalable** if there is no limit to the number of application instances that can handle requests, so that demand is met with instances.

**Elasticity**: a property of the **environment** in which an application is deployed, that allows the environment to **react to demand** by re-deploying the application to preserve response time.

An environment is **elastic** if infrastructure is present to allow dynamic demand-based increase or decrease in active application instances.
The "reason" for elasticity is SLAs

When one is nearing an SLA violation, "deploy more server instances" to cope with demand.
When demand is low, "decommission some server instances" to save operational costs.
Why virtualization is important to elasticity

Avoiding SLA violations requires rapid deployment
Safe zone for performance depends upon relationship between maximum load growth rate and speed of deployment.
Some notation:

\( P = \text{performance} = \text{time spent between request and response.} \)

\( P_{\text{minimum}} = \text{minimum theoretical response time.} \)

\( P_{\text{SLA}} = \text{SLA upper bound on response time.} \)

We want \( P_{\text{minimum}} \leq P \leq P_{\text{SLA}} \)

But remembering our notation from last time:

\( t_{\text{server}} = \text{time spent in server to be scaled.} \)

\( t_{\text{network}} = \text{time spent in network communications.} \)

\( t_{\text{cloud}} = \text{time spent waiting for cloud.} \)

\( t_{\text{minimum}} = \text{theoretical minimum server time.} \)

\( t_{\text{SLA}} = \text{maximum time that can be spent on server to comply with SLA.} \)

We want \( P_{\text{minimum}} = t_{\text{minimum}} + t_{\text{network}} + t_{\text{cloud}} \)

\( \leq P = t_{\text{server}} + t_{\text{network}} + t_{\text{cloud}} \)

\( \leq P_{\text{SLA}} = t_{\text{SLA}} + t_{\text{network}} + t_{\text{cloud}} \)

\( P_{\text{minimum}} \leq P \leq P_{\text{SLA}} \text{ whenever } t_{\text{minimum}} \leq t_{\text{server}} \leq t_{\text{SLA}}. \)

(Here, and from now on, \( t_{\text{cloud}} \) includes cloud communication time and \( t_{\text{network}} \) is just from client to edge and back.)
For simplicity,

Assume $t_{\text{cloud}}$, $t_{\text{network}}$ are (relatively) constant.
Then what varies is $t_{\text{server}}$.
Assume we're talking (for today) about "average" behavior.
The naïve approach:

If $P \ "near" \ P_{SLA}$, add servers.
If $P \ "near" \ P_{minimum}$, remove servers.
Problems with the naïve approach

When you add servers, you are already in violation!

How close is "close enough" to $P_{\text{minimum}}$ to remove servers?
A typical strategy for SLA compliance

Four watermarks

- $t_{\text{minimum}}$: theoretical minimum server time, leads to $P_{\text{minimum}}$: minimum response time.
- $t_{\text{sufficient}}$: sufficient server response time, leads to $P_{\text{sufficient}}$: sufficient overall response time.
- $t_{\text{safe}}$: worst acceptable server response time, leads to $P_{\text{safe}}$: worst acceptable overall response time.
- $t_{\text{SLA}}$: SLA violation boundary, leads to $P_{\text{SLA}}$: worst penalty-free overall response time.
Typical strategy for avoiding SLA violations:

If $P > P_{\text{safe}}$, add servers.

If $P < P_{\text{sufficient}}$, remove servers.
A simple calculation:

Let $P(t)$ be the *performance* of the application over time, e.g., average response time. We want small $P(t)$.

Let $R(t)$ be the *resources* allocated to the application, e.g., application instances deployed.

Let $L(t)$ be the *load* (demand) in requests per second.
Naïve physics of \( P \)

\( P(T) \) is a function of \( R(T) \) and \( L(T) \)!

As \( L(T) \) increases, \( P(T) \) increases.
As \( R(T) \) increases, \( P(T) \) decreases.

We want \( P(T) < P_{\text{SLA}} = \text{SLA limit} \).

So we need to **increase \( R \) as \( L \) increases** or --
equivalently -- **increase \( R \) as \( P \) increases**.
Suppose we're sampling performance at some fixed interval $\Delta t$. Suppose we know the maximum rate at which $P$ can increase; call that $M = \max(\Delta P/\Delta t)$.

Suppose that it takes us time $t_{\text{deploy}}$ to deploy a new instance. Let $N = \lceil t_{\text{deploy}}/\Delta t \rceil$ where $\lceil A \rceil$ is the least integer greater than $A$. This is the number of update cycles it takes to deploy an instance. $N$ can vary between 1 (pre-deployed virtual instance) and several hundred (need to install an instance).

Then it had better be that $P + NM < P_{\text{SLA}}$. Otherwise, the new instance has been deployed too late to help!
SLA violation avoidance:

Estimate $M = \max (\Delta P/\Delta t)$ = the amount that $P$ can grow by in one cycle.

Estimate $N = \lceil t_{\text{deploy}}/\Delta t \rceil$ = number of measurement cycles needed to deploy an instance.

If $P + NM > P_{\text{SLA}}$, we are risking a violation; add application instances to cope.

Thus $P < P_{\text{SLA}} - NM \equiv P_{\text{safe}}$

Defining $P_{\text{safe}} \equiv t_{\text{safe}} + t_{\text{network}} + t_{\text{cloud}}$, we get $t_{\text{safe}} = t_{\text{SLA}} - NM$

$$= t_{\text{SLA}} - \lceil t_{\text{deploy}}/\Delta t \rceil \max (\Delta P/\Delta t).$$
The margin of safety $t_{\text{safe}}$ is a function of deployment time $t_{\text{deploy}}$ and the projected maximum load increase rate $M$. In other words, **deployment time matters!**
Violation time is too late

Wednesday, February 10, 2010
10:37 AM

lots of SLA violations!

Response $P$

Deployment time

$P_{SLA}$

$P_{mm}$

time $T$
Ideal behavior

Wednesday, February 10, 2010
10:44 AM

[Diagram with handwritten notes]

New + or
server/application instance

Deployment

Deployment

Deployment
Actually two computed bounds:

- $t_{\text{sufficient}}$: how fast is fast enough.
- $t_{\text{safe}}$: how fast is safe in terms of avoiding SLA violations.
Basic elasticity calculation

If $P > P_{\text{safe}}$, **increase instances** to bring it below $P_{\text{safe}}$.
If $P < P_{\text{sufficient}}$, **decrease instances** to allow it to be larger.
Why virtualization is important in clouds
Quick deployment allows $t_{safe}$ nearer to $t_{SLA}$.
Which reduces the number of deployed instances to maintain $t_{safe}$.
Which saves power, cost, etc.
This is why virtualization is so important!
So far,

we know to keep $P_{\text{sufficient}} \leq P \leq P_{\text{safe}}$ by adding or removing instances.

This gives us **time to react** to changes in demand.

But **how many instances** should we add (or remove)?

That is a longer story (next time).