Scalability

Wednesday, February 03, 2010
3:59 PM

The commandments of cloud computing

- So far, we've constructed cloud applications according to a (so far) unjustified set of "**commandments**: "

a. Thou shalt write thy client to **contact only one designated (logical) server.**
b. Thou shalt **store all persistent data in distributed objects**, not on the server or client!
c. Thou shalt **exchange information** with other application instances **only through use of distributed objects,** and not via direct communications between instances.
d. Thou shalt use **strong consistency** of the distributed datastore to insure that application instances **see the same view of data.**

Basic theme of this lecture

Wednesday, February 09, 2011
4:33 PM

So far, you've been taught to write killer apps through killer programs of high complexity.
The lesson of the cloud: simple programs always win.
I will emphasize how to fit a program into the cloud, by **simple application** of internally complex **black boxes**.
This means: the real complexity of programming the cloud is in dealing with complex **interfaces**.

Justifying the commandments

Thursday, February 04, 2010
1:22 PM

The justification for these rules includes concerns of **deployment**, **demand**, and **response time**.

Deployment: the act of making an application available to users.

Provisioning: assigning servers to an application.

Demand: rate of request arrivals from users.

Response time: time between receipt of a request and the response.

Some vocabulary

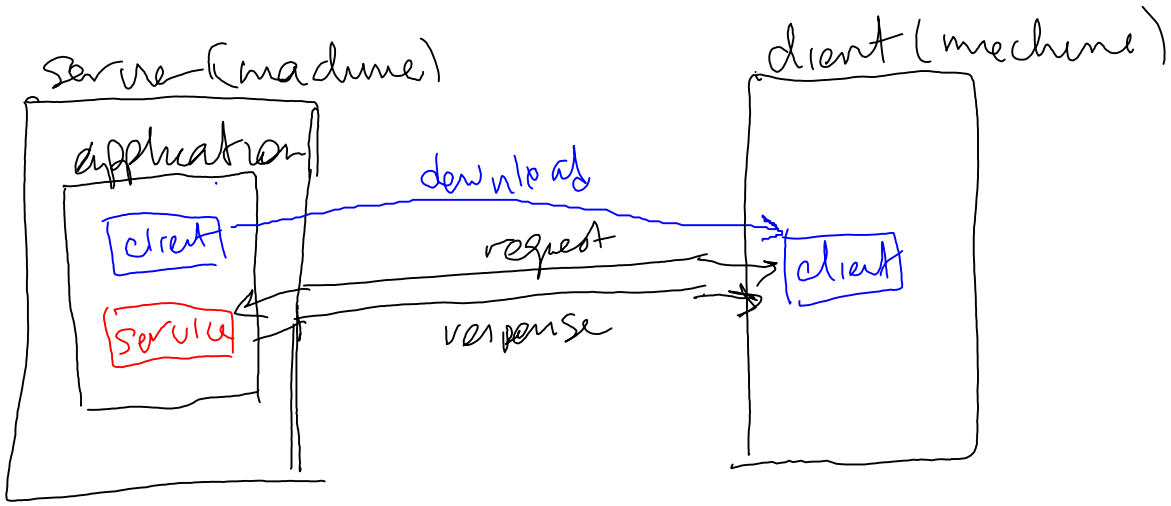
Thursday, February 04, 2010
3:29 PM

Some vocabulary

An **application** is the combination of client-side and server-side code that results from a project.

A server is a **physical or virtual machine** that runs applications.

A service is a **specific mechanism** for responding to requests from within an application.



Inadequacies of language

Thursday, February 04, 2010
3:40 PM

Alas, this language is inadequate

- We want to be able to distinguish between
  - A blueprint for a server
    - … and a machine that follows the blueprint.
  - An application as a **blueprint for execution**
    - … and a running application that **follows the blueprint.**
  - A service as a **blueprint for behavior**
    - … and a running service that **has that behavior.**

Object-oriented language

Thursday, February 04, 2010
3:43 PM

One potential help is object-oriented language

A **class is a blueprint** for creating an object.

A **class instance is an object** made according to the blueprint.

We can use this to distinguish between a thing and its deployment.

Thursday, February 04, 2010
3:44 PM

Objects and instances

A **server** (class, configuration) is a blueprint for **building a server**.

A **server instance** is a machine built (deployed) to those specifications.

An **application** (class, deployable) is a blueprint for **interactions** with the cloud.

An **application instance** is an executing (deployed) program that satisfies that blueprint.

A **service** (class) is a blueprint for **behavior** in responding to clients.

A **service instance** is a running (deployed) service that satisfies that blueprint.

Thursday, February 04, 2010
1:56 PM

Deployment, objects, and instances

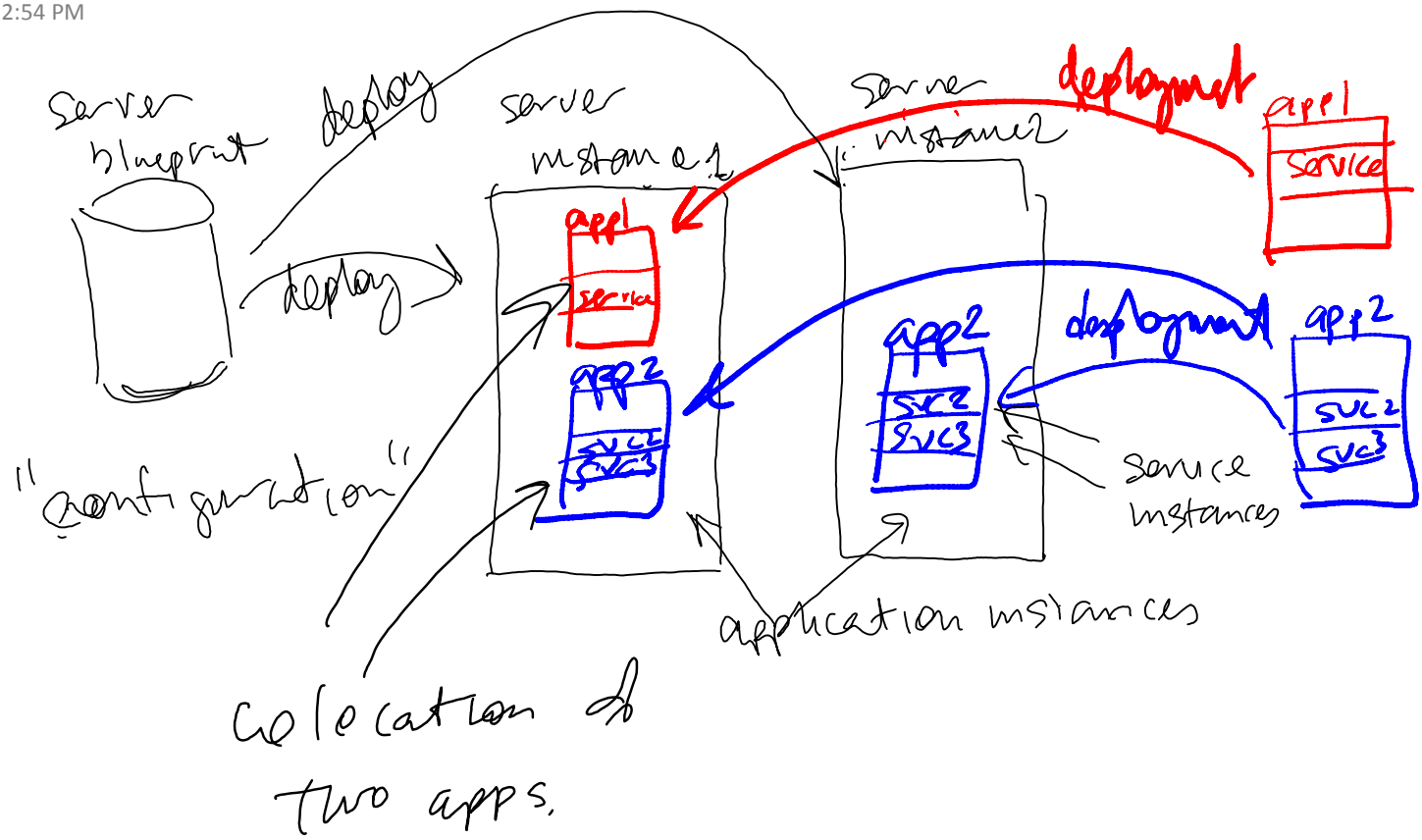
An application is **deployed** as a set of **application instances.**

Each **application instance** is assigned to and executes on a (unique) **server instance.**

Each **application instance** consists of one or more **service instances,** e.g., google AppEngine servlets.

Server instances and application instances

Thursday, February 04, 2010
2:54 PM



Thursday, February 04, 2010
3:06 PM

Applications versus application instances

An **application** is like a program: it has the potential to run and provide services. It is also like a class, in the sense that it has predefined properties.

An **application instance** is analogous to a **running program (process)**; it is providing services. It has the character of a class instance.

Server versus server instance

Thursday, February 04, 2010
3:19 PM

There is an analogous relationship between server classes and server instances

A **server class** refers to a blueprint for configuration of hardware and software, while
A **server instance** is a machine configured according to that blueprint.
A **machine** becomes a **server instance** by conforming to requirements for the instance.

(End of lecture on 2/7/2011)

Programs versus Specifications

Wednesday, February 09, 2011
9:29 AM

Programs versus specifications

We are used to thinking of a program as "something that is **executed** to give a specific result".

In cloud computing, it can be better to think of a program as "a **specification** of what needs to be computed."

The simpler the specification is, the more options we have to compute the thing.

Simplicity is equivalent with **flexibility of response**.

Aside: Turing computability and the cloud

Wednesday, February 09, 2011
3:35 PM

Aside: Turing computability and the cloud

- Any graduate of Comp170 knows that there are computer programs "complex enough" that the only way to generate their output is to execute them. E.g., due to
  - Church's thesis
  - Undecidability
- Comp170 also demonstrates that simpler programs can be analyzed and implemented flexibly, e.g., by a
  - State machine
  - Pushdown automaton (state machine with a stack)

Aside: Turing computability continued

Wednesday, February 09, 2011
3:40 PM

Complexity and the cloud

You might think that I'm going to teach you how to write very complex programs in the cloud.

In fact, I'm going to do the exact opposite: to show you how to keep programs simple by employing powerful abstractions.

The ideal cloud program is -- at some level -- conceptually a finite state machine (or, at worst, a pushdown automaton).

More complex program features are compartmentalized, interfaced, documented, and the internal details forgotten.

This will lead us -- eventually -- to Event/Condition/Action (ECA) programming

The ideal cloud program (ECA)

- Responds to **events** (E) in the cloud and on the client
- That satisfy certain **conditions** (C)
- With particular programmatic **actions** (A)
- (that are loop-free and based upon data transformations)
- (for more about why loops are to be avoided, see Comp105)

If we describe a program this way, the cloud has **maximum flexibility** in how it implements this specification.

More about this a lot later...

Thursday, February 04, 2010
4:40 PM

Limits of a single server

- No matter how much money you spend, a single server has a "speed limit" that doubles according to Moore's law every two years.
- Unfortunately, the number of potential users of a service can easily grow beyond the capacity of one server.
- One solution: **horizontal (edge) scaling**.

Different kinds of scaling work independently.

- Edge scaling.
- Core scaling.
- Network scaling.

Horizontal scaling

Thursday, February 04, 2010
3:07 PM

Horizontal (edge) scaling

Deploy **several application instances** of the same application.

Deploy a **service switch** that chooses which instance to use.

"Horizontal scaling" means using **peers** of a server at the same level.

"Vertical scaling" means using a **hierarchy** of servers.

"Core scaling" means expanding the infrastructure of the cloud itself.

Service switching

Thursday, February 04, 2010
1:36 PM

A service switch is a proxy network device that

- Receives all incoming requests for service.
- Forwards each one to an appropriate server for processing.
- (Optionally) receives the response to the request from the server, and forwards it to the end-user.

The simplest form of horizontal scalability

Thursday, February 04, 2010
1:38 PM

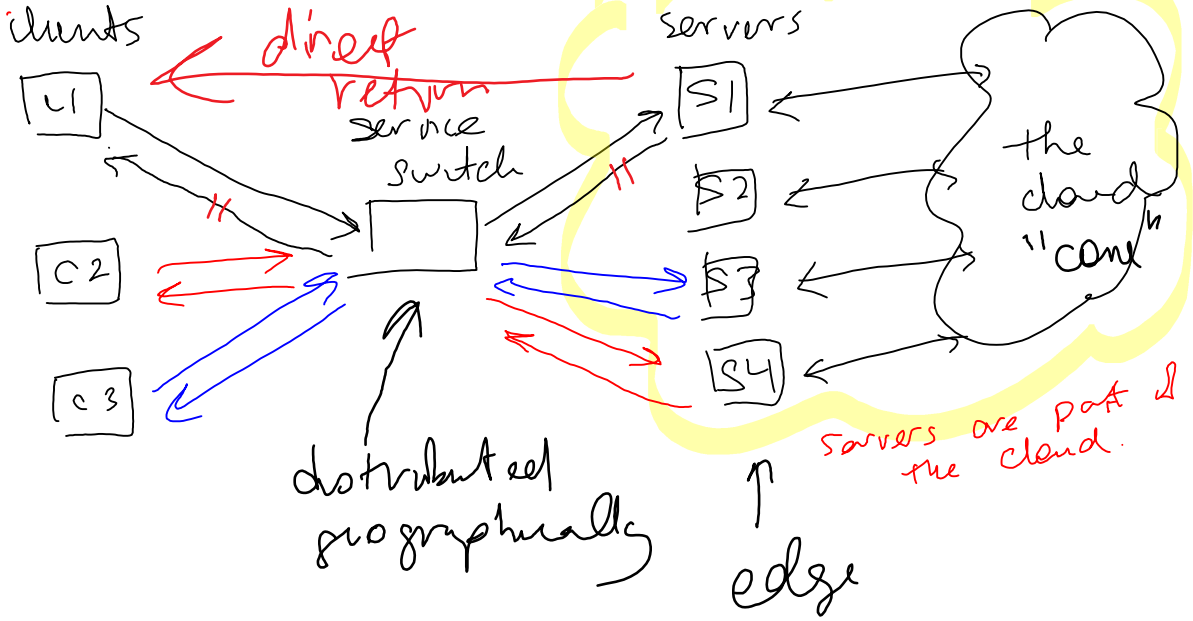
The simplest form of horizontal scaling

Service switch serves as gateway to N servers, all identical.

Each of N servers has the **same view of the cloud**.

Incoming requests are forwarded to the "**least busy server**"

Also sometimes called **load balancing**.

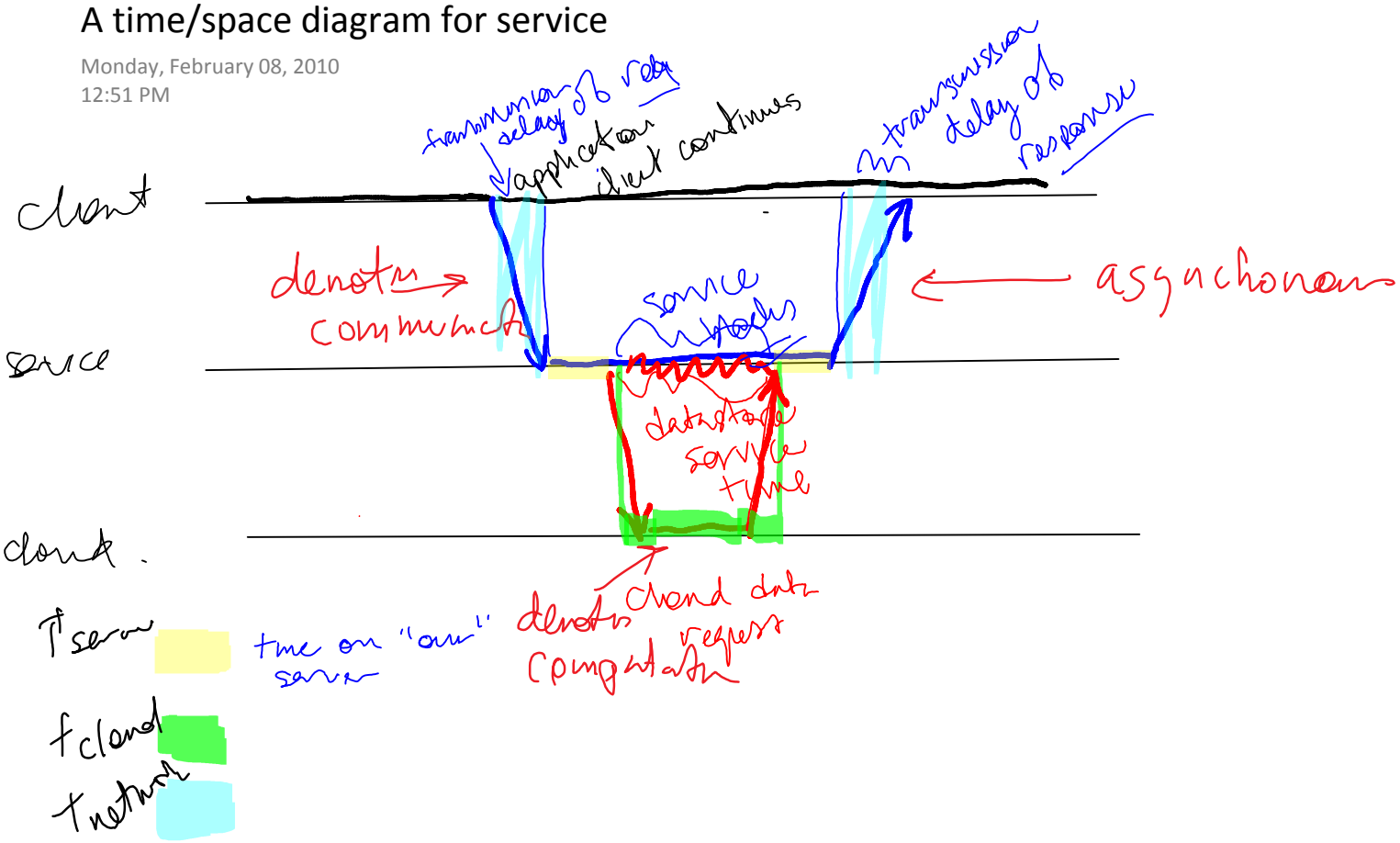


What does horizontal scaling do?

- If the servers are not too busy, service request time stays roughly the same as for one server, because
  - Switching time is negligible.
  - Datastore behavior dominates server behavior.
- As servers become busy, their computation time becomes a **bottleneck** resource.
  - Switching time is still negligible.
  - Response time = datastore response time + service response time.

A time/space diagram for service

Monday, February 08, 2010
12:51 PM



Time analysis for switched service

Monday, February 08, 2010
1:02 PM

Time between request and response ($t_{service}$)

= service time for service on server instance (t_{server})

+ time spent waiting for datastore (t_{cloud})

+ transmission time for request

+ transmission time for response } ($t_{network}$)

= $t_{server} + t_{cloud} + t_{network}$.

Service switching does not i**nfluence:**

transmission time $t_{network}$

datastore time t_{cloud}

Service switching does control:

server time (other than datastore) t_{server}.

Some performance facts

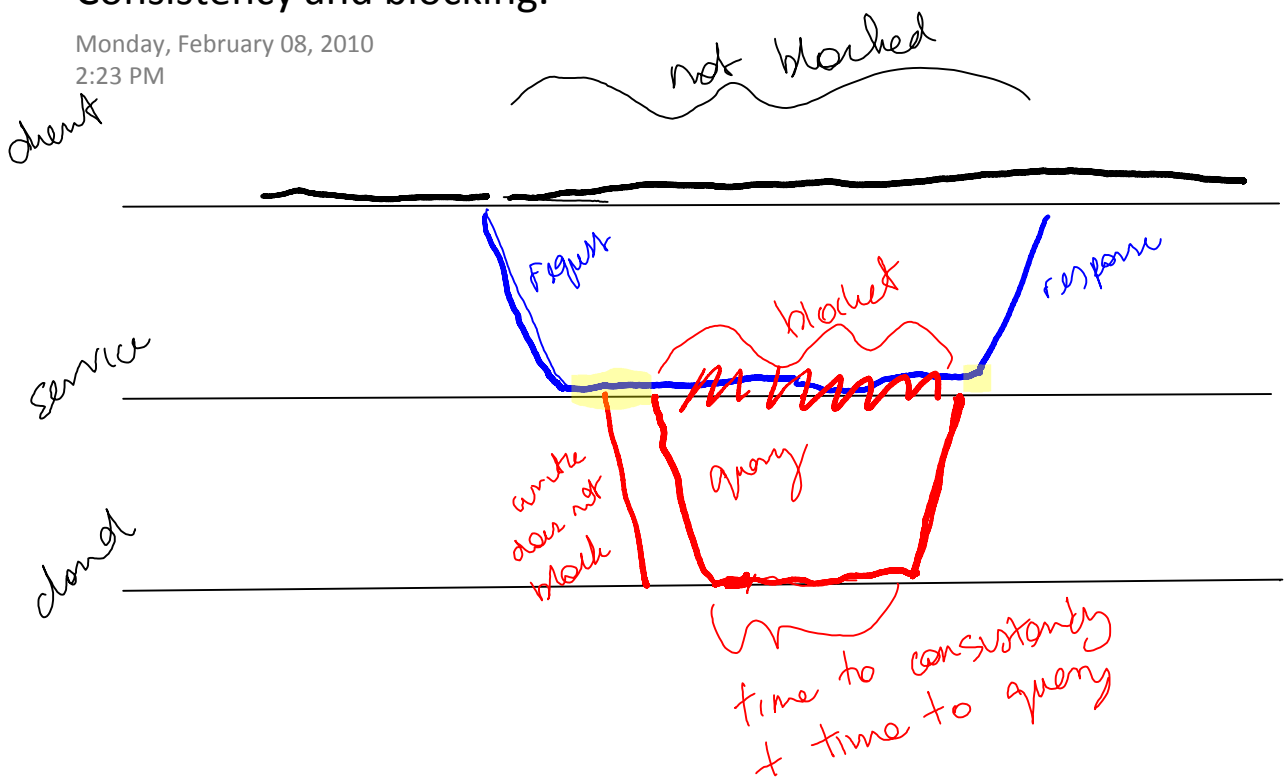
Monday, February 08, 2010
2:20 PM

Some performance facts about AppEngine

- Reads and writes to the datastore are handled differently
  - Writes never block; the datastore does the write while the application is running.
  - Reads block until data is consistent.
- Thus "writing one datum" takes **negligible time**, while "writing and reading one datum" in sequence takes (potentially) longer than "reading one datum" later, after data is known to be consistent!

Consistency and blocking.

Monday, February 08, 2010
2:23 PM



So what are we scaling with switching?

Monday, February 08, 2010
1:07 PM

So what are we scaling with switching?

We control how much time our application spends with the service.

The cloud itself controls response time for the datastore, via similar scaling mechanisms.

So the service switch solves **only 1/3 of the problem**.

Network time and cloud time are managed through **their own (unique) provisioning mechanisms**!

All management mechanisms run independently.

Two choices for switching semantics

Thursday, February 04, 2010
1:43 PM

Two choices for the design of the service switch

One session per server: the client-server binding persists over one "session".

Distributed sessions: requests go to whatever server is least busy, regardless of which server first responded to the client.

"One session per server" is called "stickiness".

Hulu is sticky: it sometimes runs out of servers.

One session per server

Thursday, February 04, 2010
1:45 PM

One session per server:

- Allows server to **store session state locally.**
- **Faster response** to individual queries than using a distributed datastore due to **local session state.**
- Service switch must track **sessions**, called **flows** in networking parlance.
- **Fragile:** if session server goes down, session **dies.**
- Example: first generation Microsoft .net® programs (but not current generation).
- Some services can only be provided this way, e.g., streaming media.

Thursday, February 04, 2010
1:46 PM

Distributed sessions

Session information is stored **in the cloud**.

No mechanism for using "the same server" for two sequential requests. Service switch can be "**flowless**".

Each application instance must **fetch session context** from the cloud (slower). Thus there is **at least one datastore transaction for every request**.

Very robust: virtually no impact from a server failure.

This is the architecture of Google Appengine.

Aside: robustness and fragility

Monday, February 08, 2010
1:16 PM

An application is

- Fragile (or "brittle") if minor changes in the environment can break it.
- Robust if minor changes in the environment will not affect it.

Reality: failures happen.

- In 2002, google estimated that one server a day in their infrastructure fails due to **wire rot.**
- Need a **robust approach** to avoid service failure.
- This implies **flowless switching**.

Aside: consistency versus flow-based switching

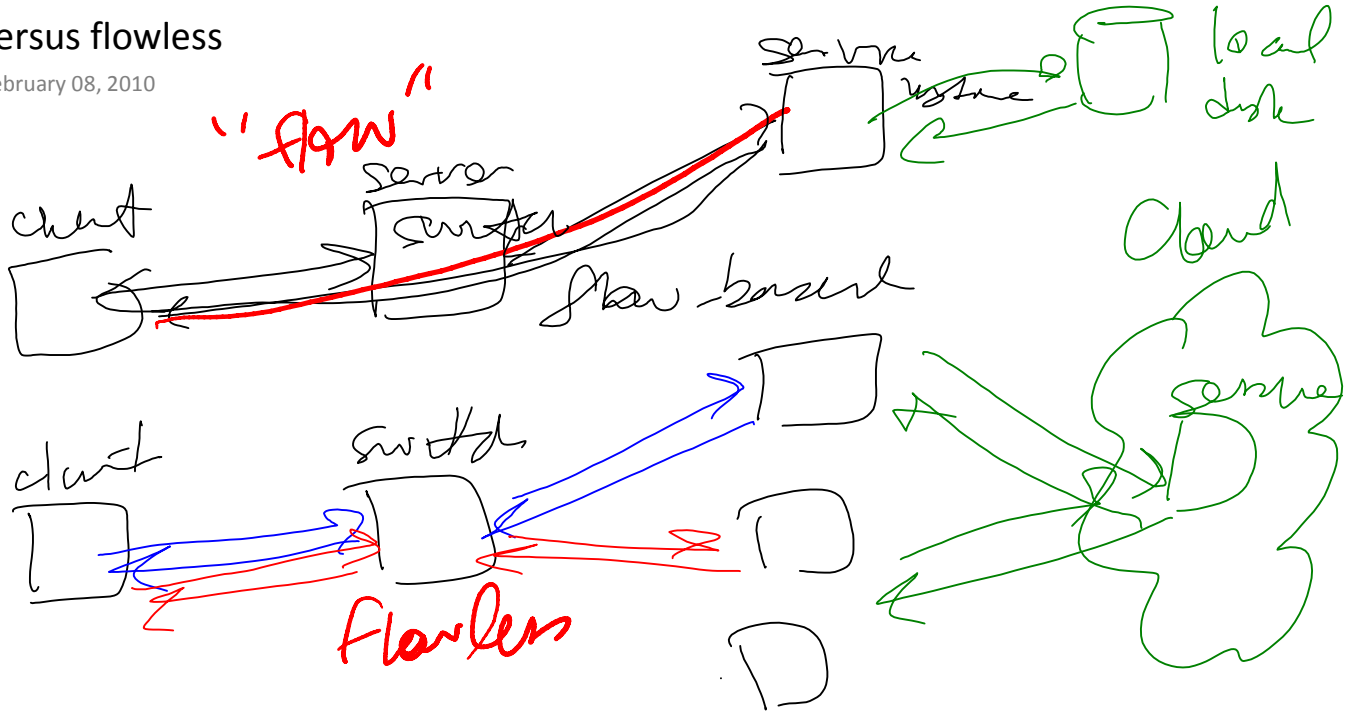
Thursday, February 04, 2010
4:42 PM

Consistency versus flow-based switching

- If you have **strong consistency**, you don't need **flow-based switching,** because the distributed object is "always right" about the data about the object, no matter which application instance is contacted next.
- What can go wrong with combining **weak consistency** and **flowless switching**? Suppose you purchase a product and a different application instance on a different server instance prints your receipt. **What if that instance doesn't see your transaction yet?**

Flow versus flowless

Monday, February 08, 2010
6:36 PM



Why ask why?

Thursday, February 04, 2010
3:15 PM

We already have enough information to justify the commandments of cloud programming:

AppEngine has **flowless switching** for **high robustness.**

We can't store data other than in the cloud, because in a sequence of requests, we might get a **different server instance** and **different application instance** for a **second request**!

We need **strong consistency** in the cloud, because otherwise, server instances and application instances will see **different versions** of data.

Monday, February 08, 2010
1:13 PM

Distributed sessions

From now on, we assume we have **flowless switching** (and thus **distributed sessions)**.

When would we want to throw more application instances at the problem?

How does performance change with number of application instances?

Provisioning

Thursday, February 04, 2010
3:03 PM

Provisioning and resources

The number of servers (machines) available is **finite.**

Service provisioning refers to the decision to have N application instances (on N distinct server instances).

This is always a tradeoff between the **cost** of running the servers and the **value** of providing the service.

Monday, February 08, 2010
1:20 PM

Scalability

An application is **scalable** (in some load attribute) if provisioning N servers allows it to handle N times as much load of that kind.

Watch out: not everything is scalable.

From the time-space diagram:

We can keep service computing time down by scaling, but

We cannot reduce the base processing time.

So

Appengine applications are

not scalable in processing time per request, but they **are scalable in request volume.**

Request volume scalability

Monday, February 08, 2010
1:24 PM

If there are few requests, each request takes some average time $t = t_{server} + t_{cloud} + t_{network}$

t_{server} = processing time spent on your server instance in your application instance

t_{cloud} = processing time spent waiting for cloud interactions.

$t_{network}$ = time spent waiting for request to be transmitted and response received.

t_{cloud} and $t_{network}$ can be considered **constant** (they have their own management mechanisms!)

What are we scaling?

Monday, February 08, 2010
1:30 PM

What are we scaling?

An idle service has an average service time $t_{minimum}$, while a busy service might require increased service time $t_{server} > t_{minimum}$.

So total service response time

$= t_{server}+t_{cloud}+t_{network}$

$\geq t_{minimum}+t_{cloud}+t_{network}$.

Scaling

Thursday, February 04, 2010
2:01 PM

Scaling has two kinds

Demand scaling refers to changes in demand (requests per second) for enterprise reasons, e.g., Christmas rush, breaking news, slashdot, etc.

Service scaling refers to changes in **service provisioning**, e.g., number of servers allocated, for the purpose of **meeting demand.**

The reason for the cloud

Thursday, February 04, 2010
1:57 PM

The main technical reason for the cloud:

Respond to demand scaling with application scaling.
I.e., if there are more requests, **deploy more application instances** on **server instances.**
Likewise, if there are less requests, **deploy less application instances on server instances.**

SLOs and SLAs

Thursday, February 04, 2010
4:57 PM

A measure of scaling: SLOs and SLAs

Service-level objectives (SLOs) express response time goals for a service, i.e., the time between making a request and receiving a response.

Service-level agreements (SLAs) are legal documents describing the scope and consequences of expectations for a service

An SLO describes what is desirable; an SLA describes penalties for not meeting expectations!

Thursday, February 04, 2010
5:40 PM

A Typical SLA

- Several levels of service: "gold", "silver", "bronze".
- Response time expectations for each service, e.g.,
  - "gold service: elapsed time between request and response < 0.1 second."
  - "silver service: elapsed time between request and response < 1.0 second."
- Economic penalty for each response time violation ("SLA violation").

Thursday, February 04, 2010
5:44 PM

The goal of an SLA

- Define the legal obligations of a service provider.
- Establish monetary rewards for good service, and penalties for poor service.
- "Motivate" the service provider to provide good service.

Monday, February 08, 2010
1:39 PM

SLAs and provisioning

SLAs determine bounds on total response time = $t_{server} + t_{cloud} + t_{network} \leq t_{threshold}$ so that

$t_{server} \leq t_{threshold} - t_{cloud} - t_{network} = \mathbf{t_{maximum}}$.

Thus, the goal of service switching is to insure that

$\mathbf{t_{minimum} \leq t_{server} \leq t_{maximum}}$.

$t_{minimum} \leq t_{server}$ for theoretical reasons.

$t_{server} \leq t_{maximum}$ only if **scale meets demand.**

Too much of a good thing...

Monday, February 08, 2010
1:47 PM

Too much of a good thing…

$t_{server} \geq t_{minimum}$ no matter how many application instances you throw at it!

Once $t_{server} \leq t_{maximum}$, you have provisioned "enough" servers.

Any more servers **cost you,** but don't **benefit the customer.**

Robbing Peter to pay Paul

In a cloud, the number of server instances is fixed and the number of application instances per server instance is (as a first approximation) fixed.

If we provision a server instance for one application, its resources are **not available** for others.

Thus we must often make tradeoffs between provisioning one application instance or another on a server instance.

Case history: CNN on 9/11

Monday, February 08, 2010
1:49 PM

Bill Lefebre: head of operations at cnn.com on 9/11/2001

- Ridiculous, unforeseen request volume.
- Server instances provisioned manually crashed **before they could come up.**
- Final solution: shut down cnn for 1/2 hour, reboot everything, **bring all servers up at once.**
- Moral: in incredible circumstances, **provisioning is not enough!**