Lessons learned so far...

Wednesday, January 26, 2011
4:16 PM

Last lecture: **syntax**:

A **cloud application** is a java serial program
that interacts with **persistent instances** of **distributed objects**,
manufactured by use of a **persistence factory**.

But what does it all mean?

This lecture: **semantics**:

A distributed object exhibits some kind of

consistency: how you read what you wrote.

and

concurrency: what happens when there are multiple concurrent writes.

A cloud application...

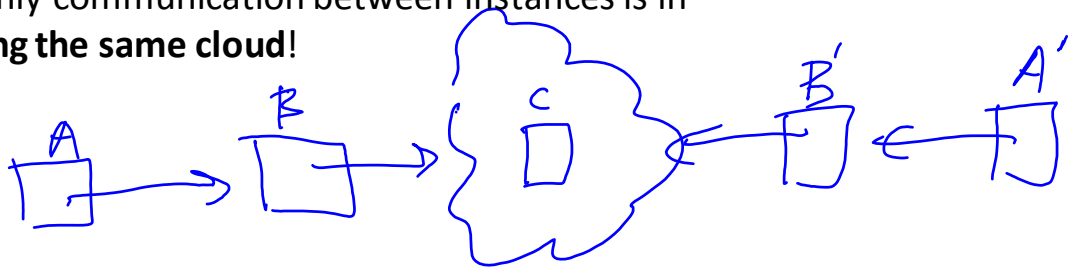
Wednesday, January 27, 2010
9:05 AM

A cloud application…

Is not a parallel or distributed computing program.

Instances of an application **do not communicate with each other.**

The only communication between instances is in **sharing the same cloud**!



Distributable programs

Wednesday, January 27, 2010
9:06 AM

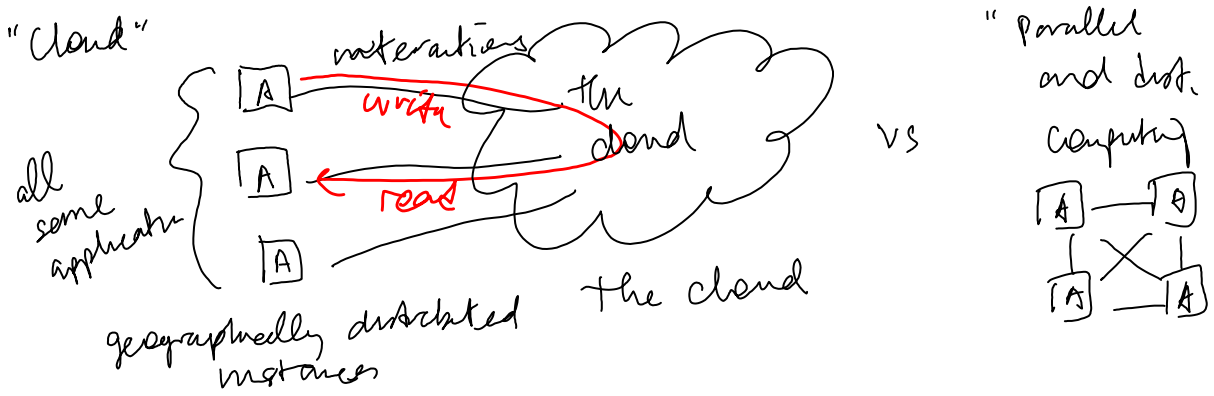
"Distributable" applications

Are regular **serial application programs.**

Can have many concurrent instances running **in different locations.**

All instances **do the same thing.**

All instances have the **same view of the world.**



Distributed objects

Give all instances of an application "**the same view**" of the world.

Are **opaque** to the application.

Are distributed in ways that the **application cannot detect.**

Classes and instances

Monday, January 24, 2011
7:58 PM

Best to think about cloud clients and services in terms of classes and instances:

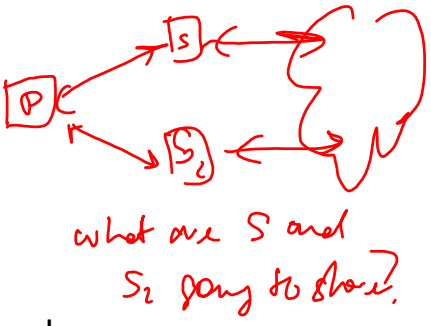
- For a client:
  - The class determines the kind of application.
  - An instance is one copy of a program that satisfies class requirements
  - There can be many concurrent instances of a client (e.g., 1000 cellphone users)
- For a service:
  - The class is the kind of service
  - An instance is one copy of a program that provides the service.
  - There can be many instances of the same service (e.g., geographically distributed)

Binding

Monday, January 24, 2011
8:01 PM

The concept of binding

- You run an app on your cellphone.
- It connects to a service.
- It does something (e.g., recording your Lat/Long location)
- It logs out.
- What can you assume about this transaction?



You cannot assume that

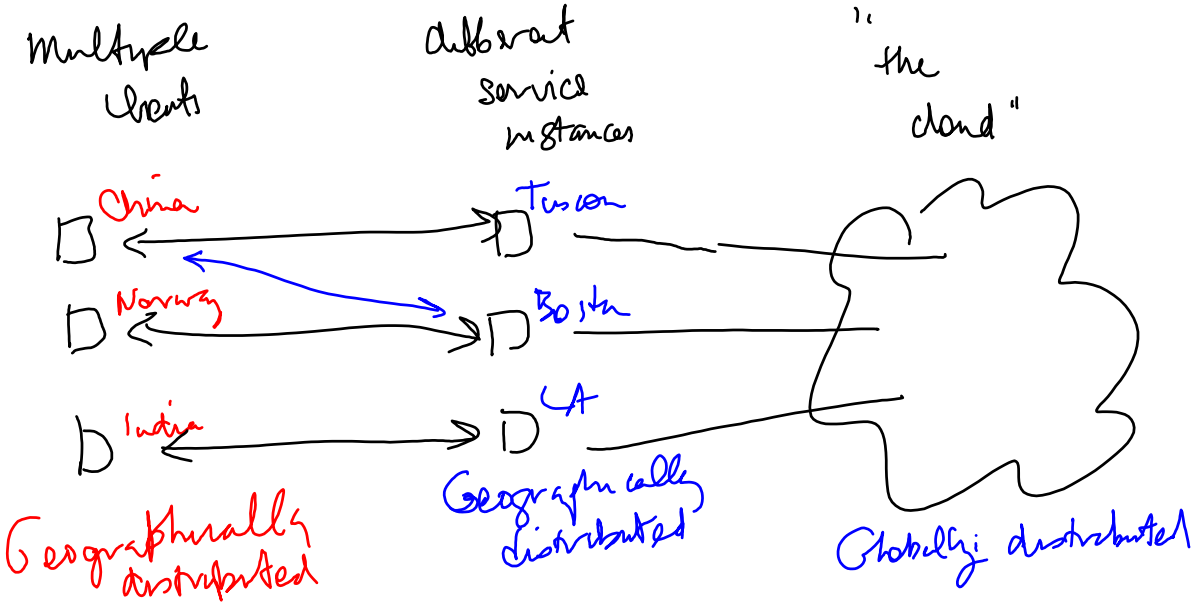
- you'll ever get the same server again. or that
- the server you get will have the same view of the cloud.
- unless you know something more about the cloud…!

When you write a service

- All data must be stored in the cloud.
- There is no useful concept of local data.

A model of cloud execution

Monday, January 24, 2011
7:54 PM



ACID

Wednesday, January 27, 2010
11:09 AM

Datastores should exhibit what is commonly called ACID:

Atomicity: requested operations either occur or not, and there is nothing in between "occurring" and "not occurring".

Consistency: what you wrote is what you read.

Isolation: no other factors other than your actions affect data.

Durability: what you wrote remains what you read, even after system failures.

Consistency and concurrency

Two visible properties of a distributed object: consistency and concurrency

Consistency: the extent to which "what you write" is "what you read" back, afterward.

Concurrency: what happens when two instances of your application try to do conflicting things at the same exact time?

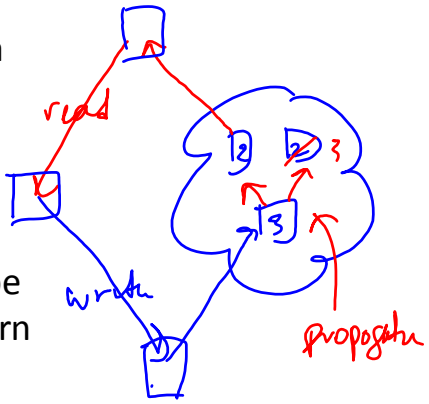
Consistency

The extent to which "what you write" into a distributed object is "what you read" later.

Two kinds:

Strong consistency: if you write something into a distributed object, you always read what you wrote, even immediately after the write.

Weak (eventual) consistency: if you write something into a distributed object, then eventually -- after some time passes -- you will be able to read it back. Immediate queries may return stale data.



A file analogy

Strong consistency is like writing to a regular file or database: **what you write is always what you get back.**

Eventual consistency is like writing something on pieces of paper and mailing them to many other people. What you get back depends upon **which person you talk to** and **when you ask. Eventually,** they'll all know about the change.

Concurrency

How conflicting concurrent operations are handled.

Two kinds:

Strong concurrency: if two or more conflicting operations are requested at the same time, they are serialized and done in arrival order, and both are treated as succeeding. Thus **the last request determines the outcome.**

Weak ("opportunistic") concurrency: if two conflicting operations are requested at the same time, **the first succeeds** and **the second fails**. Thus **the first request determines the outcome**.

Wednesday, January 27, 2010
9:41 AM

A file analogy

On linux, file writes exhibit strong concurrency, in the sense that conflicting writes all occur and the last one wins.

Likewise, in a database, a stream of conflicting operations are serialized and all occur -- the last one determines the outcome.

Opportunistic concurrency only occurs when there is some form of data locking, e.g., in a transaction block.

Strong consistency and concurrency

Wednesday, January 27, 2010
9:54 AM

Consistency/Concurrency tradeoffs

Obviously, we want both strong consistency and strong concurrency

But we can't have both at the same time!

Strong consistency requirements

Wednesday, January 27, 2010
10:01 AM

Strong consistency requires

- Some form of write blocking until a consistent state,
- Which implies a (relatively) slow write time before unblocking.
- Which means we can't afford to "wait" for the write to end in order to sequence writes.
- Which means **we can't have strong concurrency!**

Strong concurrency requirements

Wednesday, January 27, 2010
10:01 AM

Strong concurrency requires

- Some form of write sequencing.
- A (relatively) fast write time, with little blocking.
- Which means writes need time to propogate.
- Which means **we can't have strong consistency!**

Two approaches to PAAS

Wednesday, January 27, 2010
10:03 AM

Google's "appEngine"
- Provides strong consistency
- At the expense of opportunistic concurrency.

Amazon's "dynamo"
- Provides strong concurrency.
- At the expense of exhibiting eventual consistency.

AppEngine

Wednesday, January 27, 2010
10:18 AM

AppEngine properties

Strong consistency: what you write is always what you read, even if you read at a (geographically) different place!

Opportunistic concurrency: updates can fail; application is responsible for repeating failed update operations. Updates should be contained in "try" blocks!

Wednesday, January 27, 2010
11:23 AM

Modifying distributed objects in AppEngine

A distributed object retrieved from the persistence manager remains "attached" to the cloud.

If you "set" something in a persistent object, **this implicitly modifies the cloud version** and **every copy in other concurrently running application instances! This is what strong consistency means!**

So if some concurrent application instance sets something else in an object instance you fetched, **your object will reflect that change** (via strong consistency).

So mostly, **you observe** what appears to be **strong concurrency.**

- The illusion of strong consistency
  - How is this actually done?
  - It's actually smoke and mirrors!
- Creating strong consistency
  - Every object is "dirty" if changed, and "clean" if not.
  - Very fast mechanisms for propogating "dirty" information (e.g., a bit array).
    - A class of objects is dirty if any instance is.
    - An instance is dirty if its data isn't completely propogated.
  - Relatively slow mechanisms for changing something from "dirty" to "clean".
    - Actually propogate the data, then relabel the thing as clean.
  - Applications get "dirty" info immediately, and then **wait** until the data is clean before proceeding!

An example object

Wednesday, January 27, 2010
11:28 AM

```
import com.google.appengine.api.datastore.Key;

import java.util.Date;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.IdentityType;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;


@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Employee {
   @PrimaryKey
   @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
   private Key key;

   @Persistent
   private String firstName;

   @Persistent
   private String lastName;

   @Persistent
   private Date hireDate;

   public Employee(String firstName, String lastName, Date hireDate) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.hireDate = hireDate;
   }

   // Accessors for the fields.  JDO doesn't use these, but your
application does.

   public Key getKey() {
      return key;
```

```
    }

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Date getHireDate() {
        return hireDate;
    }
    public void setHireDate(Date hireDate) {
        this.hireDate = hireDate;
    }
}
```

wait until consistent!

propagate!

Pasted from <http://code.google.com/appengine/docs/java/datastore/dataclasses.html>

Entity identity and keys

- An object of type Key helps one locate persistent objects in the cloud
- By default, the system assigns it.
- You can also make one that is easier to remember.

Creating an object with a settable key

Wednesday, January 27, 2010
1:45 PM

```
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;
import com.google.appengine.api.datastore.Key;

// ...
   @PrimaryKey
   @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
   private Key key;

   public void setKey(Key key) {
      this.key = key;
   }
```

Pasted from
<http://code.google.com/appengine/docs/java/datastore/creatinggettinganddeletingdata.html>

Making an object with a known key

Wednesday, January 27, 2010
1:46 PM

```
import com.google.appengine.api.datastore.Key;
import com.google.appengine.api.datastore.KeyFactory;

// ...
    Key key = KeyFactory.createKey(Employee.class.getSimpleName(),
                                   "Alfred.Smith@example.com");
    Employee e = new Employee();
    e.setKey(key);
    pm.makePersistent(e);
```

Pasted from <http://code.google.com/appengine/docs/java/datastore/creatinggettinganddeletingdata.html>

Retrieving an object via a known key

Wednesday, January 27, 2010
1:47 PM

```
Key k = KeyFactory.createKey(Employee.class.getSimpleName(),
                                        "Alfred.Smith@example.com");
Employee e = pm.getObjectById(Employee.class, k);
```

Pasted from
<http://code.google.com/appengine/docs/java/datastore/creatinggettinganddeletingdata.html>

Key caveats

- One can use any integer or string field of an object as a basis for a datastore key.
- If you don't, one is selected for you automatically.
- One can retrieve the basis value from the key.
- Thus you do not need to store the basis value explicitly in the record; it is implicitly present.
- In the examples, the basis value is an email address.

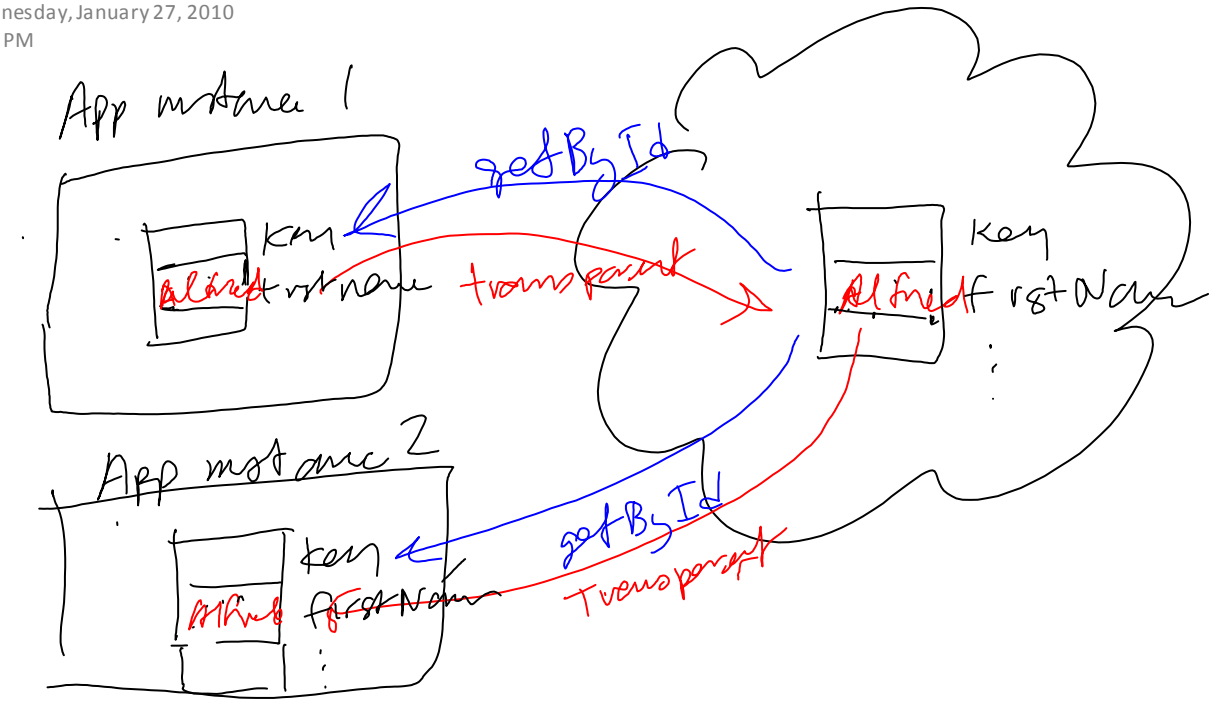
Changing persistent objects

Wednesday, January 27, 2010
1:52 PM

```
Key k = KeyFactory.createKey(Employee.class.getSimpleName(),
                                        "Alfred.Smith@example.com");
Employee e = pm.getObjectById(Employee.class, k);
e.setFirstName("Alfred");  // happens in distributed space, immediately!
```

A hard thing to understand

Wednesday, January 27, 2010
1:57 PM



Persistent object caveats

- Changes to a persistent object occur when requested. Access functions **must** be used; persistent data **must** be private; the PM factory adds management code to make this happen!
- Changes are reflected **everywhere the object is being referenced!**

What does strong consistency mean?

Wednesday, January 27, 2010
2:02 PM

What does strong consistency mean?

- When you change something in an instance of a persistent object, it is changed in **every other image** of that instance, including **inside other instances** of your application!

But this is a **polite illusion**; in reality, other instances of your application **wait** for data they need to arrive!

Is concurrency actually weak?

Wednesday, January 27, 2010
2:07 PM

Is concurrency actually weak?

If you are just modifying one object in straightforward ways, one-attribute-at-a-time, you might think there is strong consistency.

But **problems arise** when you're trying to update an object, e.g., from itself.

The **solution** to these problems -- and not the problems themselves -- makes concurrency weak!

Updating an entity from its own values

Wednesday, January 27, 2010
2:08 PM

Consider the code:

```
Key k = KeyFactory.createKey(Employee.class.getSimpleName(),
                                        "Alfred.Smith@example.com");
// assume existence of persistent getSalary and setSalary methods
Employee e = pm.getObjectById(Employee.class, k);
e.setSalary(e.getSalary()+100);  // Give Alfred a raise!
```

Consider what happens when **two application instances** invoke this code at **nearly the same time.**

The problem of concurrent updates

Wednesday, January 27, 2010
2:11 PM

The code

```
e.setSalary(e.getSalary()+100)
```

is the same thing as (and is implemented as!)

```
tmp = e.getSalary()
tmp = tmp + 100
e.setSalary(tmp)
```

So, we can execute this as:

Wednesday, January 27, 2010
2:13 PM

So, we can execute this twice according to the following schedule:

Instance 1	Instance 2
e.getSalary	
	e.getSalary
e.setSalary	
	e.setSalary

And Alfred gets a $100 raise rather than a $200 raise :(

Transactions

Wednesday, January 27, 2010
2:16 PM

Transactions allow us to avoid that problem:

Identify operations that should be done without interruption.

Keep other concurrent things from interfering between begin() and commit(). E.g.:

```
Key k = KeyFactory.createKey(Employee.class.getSimpleName(),
                                    "Alfred.Smith@example.com");
pm.currentTransaction().begin();
Employee e = pm.getObjectById(Employee.class, k);
e.setSalary(e.getSalary()+100);  // Give Alfred a raise!
try {
    pm.currentTransaction().commit();
} catch (JDOCanRetryException ex) {
    // ouch: something prevented the transaction!
    throw ex; // share the pain!
}
```

How this works

The transaction block (from begin() to commit()) attempts to execute before any other changes can be made to e.

If no other changes have been made to e between begin() and commit(), the **transaction succeeds.**

If some change in e has been made meanwhile, the transaction fails, e gets the changed values, the **whole transaction is cancelled,** and **the application has to recover somehow** (if it can).

How this behaves

Wednesday, January 27, 2010
2:34 PM

How this behaves

- If two applications try to do this, then
  - the object will only change due to a commit.
  - If two commits interleave, an exception is thrown.
- So we know we've goofed!

It is the use of transactions that "creates" weak concurrency, but without them, we have chaos!

Optimistic concurrency

- Transaction blocks delimit things that should be done together.
- If something changes about the object between begin() and commit(), the transaction **throws an exception.**
- So, the application **knows that its request failed.**

Coping with optimistic concurrency via retries

Wednesday, January 27, 2010
2:25 PM

```
for (int i = 0; i < NUM_RETRIES;  i++) {
      pm.currentTransaction().begin();
      Employee e = pm.getObjectById(Employee.class,  k);
      e.setSalary(e.getSalary()+100);  // Give Alfred a raise!
      try {
            pm.currentTransaction().commit();
            break; // from retry loop!
      } catch (JDOCanRetryException  ex) {
            if (i == (NUM_RETRIES  - 1)) {
                  throw ex;
            }
      }
}
```

A subtle point

Wednesday, January 27, 2010
10:27 AM

A subtle point of object consistency

To modify an object, you must read it.

If two object operations are attempted at the same time, **it is possible for both to contain stale data** with respect to each other.

Then the only reasonable choice is for the loser to start over by reading e again! Otherwise data is lost!

Thus **the only reasonable choice that preserves transactional integrity is opportunistic concurrency!**

Avoiding user misconceptions

Transactions are used -- even when concurrency is strong enough -- to avoid user misconceptions.

Remember that **every instance sees every change.**

So, if you are doing multiple things, "embarrassing outcomes" are possible

Embarrassing outcomes

You change a person's name from "John Smith" to "Jon Smythe".

On a printed report, he appears as "Jon Smith".

Oops!

Why?

Avoiding embarrassing moments

- Enclose blocks of changes in transaction blocks.
- Use blocks to freeze data at current (consistent) state.

When is strong concurrency a good choice?

Wednesday, January 27, 2010
10:34 AM

When is strong concurrency a good choice?

So far, we see that the outcome of concurrency is weak (transaction) consistency.

Why does strong concurrency work well in Amazon dynamo? **This is really subtle.**

Dynamo is intended as a **data warehouse**. Thus it is not a database itself, but rather, a **historical record** of a database.

Thus **it is never necessary to invoke a transaction** on it!

Controlling concurrency

- Concurrency control is not a place for programmer creativity.
- Concurrency problems are **extremely difficult to debug.**
- A problem may not repeat for hours or days or months!
- Better to employ some known **concurrency pattern** with properties you want!

Clouds and design patterns

A **design pattern** is a tested and robust solution to a well-understood problem.

Not so much a program, as much as instructions for writing a program.

In dealing with a known problem (concurrency), it is best to utilize a known solution.