This lecture was composed from fragments of two lectures in COMP150CPA: Cloud and Power-Aware Computing.

Alva Couch
couch@cs.tufts.edu
The most important attribute of a distributed system: how \textit{consistency} is handled.

The definition of consistency:

A \textit{consistent distributed object} has the property that if many applications access the object via differing interfaces, all interfaces provide "the same view" of the object.

Example: credit card purchases

You purchase an item.
You ask what you purchased (perhaps through a different interface or mechanism than the one used to purchase the item).
If you always get an accurate depiction of what you purchased -- regardless of interface -- then \textbf{the purchase system is consistent}. 
How inconsistency arises

Tuesday, November 05, 2013
12:18 PM
How inconsistency arises: horizontal scaling

Tuesday, November 05, 2013
12:23 PM
How inconsistency arises: enter the cloud

The requests: same view?
The facts of life for inconsistency

When using a "cloud resource", the server you utilize often changes for every transaction you make. Reason: horizontal scalability. (elasticity) Unless servers somehow communicate, you receive an inconsistent view of the world. "The cloud" is one model of server communication. "The cloud" has its own concepts of consistency.
A cloud application...

Is not a parallel or distributed computing program. Instances of an application do not communicate with each other. Instances have no local persistent memory. The only communication between instances is in sharing the same cloud.
"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.
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)
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...!)

Caveat: when you write a cloud service
All data must be stored in the cloud.
There is no useful concept of local data.
Three different approaches to storing cloud data

ACID: also known as Structured Query Language (SQL)

NoSQL: Not Only SQL: makes a choice of desirable ACID properties, leaves some out.

The CAP Theorem: constraints the choices NoSQL can enable.
Consistent 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.

Why isn't everything ACID? Why do non-ACID systems exist? This is a deep question....
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.
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 database transaction block.
Consistency/Concurrence tradeoffs

Obviously, we want both strong consistency and strong concurrency

But we can't have both at the same time!
Strong consistency requires
Some form of read blocking until a consistent state is achieved,
Which implies a (relatively) slow read time before unblocking.
Which means we can't have strong concurrency!
Strong concurrency requires

Some form of write sequencing.
A (relatively) fast write time, with little blocking.
Which means writes need time to propagate.
Which means we can't have strong consistency!
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 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!
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 propagating "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 propagated.
Relatively slow mechanisms for changing something from "dirty" to "clean".
  Actually propagate the data, then relabel the thing as clean.
Applications get "dirty" info immediately, and then wait until the data is clean before proceeding!
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;
}

Pasted from
<http://code.google.com/appengine/docs/java/datastore/dataclasses.html>
A hard thing to understand

Wednesday, January 27, 2010
1:57 PM
AppEngine persistent object caveats

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

If you are just modifying one object in straightforward ways, one-attribute-at-a-time, you might think there is strong concurrency. 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!
Consider the code:

```java
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

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

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

```java
    tmp = e.getSalary()
    tmp = tmp + 100
    e.setSalary(tmp)
```
So, we can execute this twice according to the following schedule:

<table>
<thead>
<tr>
<th>Instance 1</th>
<th>Instance 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.getSalary</td>
<td>e.getSalary</td>
</tr>
<tr>
<td>e.setSalary</td>
<td>e.setSalary</td>
</tr>
</tbody>
</table>

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

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 whole transaction is cancelled, and the application has to recover somehow (if it can).
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!
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.
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!
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!
The NoSQL controversy

**NoSQL** = "**Not only** Structured Query Language"

Often misunderstood: some more radical authors interpret NoSQL as "Prevent Structured Query Language" :)

In actuality, the NoSQL movement:

Assumes that the majority of queries are simple key fetches that don't require SQL structures.
Optimizes simple key fetches for quick response.
Often contains a fallback to interpreting full SQL, with performance disadvantages.
Or doesn't provide all ACID properties.
A typical NoSQL service

Operations are:

**ValueType get(KeyType key)**

- Gets the last value associated with a key (most of the time).

**int put(KeyType key, ValueType value)**

- Assert the binding of a key to a value.
- Subsequent puts update data.
The most common use-case is retrieval by key. Values are just bags of bits; the user can given them structure as needed.

The CAP theorem: one cannot build a distributed database system that exhibits all of strong Consistency, high Availability, and robustness in the presence of Partitioning (loss of messages).
The CAP Theorem

The CAP conjecture (Eric Brewer, 2000): any distributed datastore can only exhibit two of the following three properties

- **Consistency**: all nodes have the same view of data.
- **Availability**: data is instantly available from the system.
- **Partitionability**: the system continues to respond if messages are lost (due to system or network failure).

Proved to be true by Gilbert and Lynch (2002).

In the cloud context:

Very similar to my initial claim about the tension between consistency and concurrency.

Main contribution: cloud datastores can be categorized in terms of the two (of three) properties C,A,P that they exhibit.

AppEngine is in **class CP = Consistency + Partitionability**

Only other reasonable cloud class is **class AP = Availability + Partitionability**

**We don't want a cloud datastore that loses data (e.g., ¬P)!**
In the cloud context:

CAP theorem claim is related to my initial claim about the tension between consistency and concurrency.

Main contribution: cloud datastores can be categorized in terms of the two (of three) properties C,A,P that they exhibit. AppEngine JDO is in class \( \textbf{CP} = \text{Consistency} + \text{Partitionability} \)

Only other reasonable class is \( \textbf{AP} = \text{Availability} + \text{Partitionability} \)

\textbf{We don't want a cloud datastore that loses data! (e.g., } \neg P)\)
The class AP contains datastores like:
   Amazon Dynamo
   Facebook's Cassandra
   LinkedIn's Project Voldemort
Why so many?
   Everybody needed one.
   Google didn't publish theirs.
   And it was CP, not AP.
How LinkedIn's colleague search actually works

Periodically, a Hadoop job is run to identify potential friends.
Output is stored to a Voldemort (NoSQL) datastore. A web service accesses the datastore in read-only mode.

Some notes:
Data changes slowly, so the Hadoop job only has to be done once/day or less.
The Voldemort datastore is read-only once the job is done.
So we don't need the C in CAP, because there are no consistency issues!
So Voldemort, in class AP, suffices.
What is dynamo?

The back-end cloud datastore for amazon.com itself. Serves requests for shopping carts, purchases. On an **eventually consistent** datastore...!

How?
Unique features of Dynamo

Vector clocks for conflict detection.
Business logic for conflict resolution.
Vector clocks

Every change transaction contains a timestamp and a pointer to the previous version of the object. As transactions flow through the system, they are accumulated in a local history on each node. Local history is

- Pruned if there are no conflicts.
- Resolved (in an application-specific manner) if conflicts occur.
Example: shopping cart

Diagram of a shopping cart with multiple sub-carts and a merge operation.
Business-based recovery

What happens when a conflict occurs is based upon business rules.
If object is a shopping cart, contents are merged.
If object is a purchase record, apparent duplicate purchases are eliminated.
Why Amazon "gets away with" (eventually consistent)
Dynamo
    I told you previously that one would not want to store business data in an eventually consistent store. Amazon "gets away with" doing that, by embedding business logic.
          using vector clock updates.
          for each kind of object, separately.
This is a complex game, which is why Dynamo isn't available to their customers.