Last time, we studied how to build a service using SOAP:
The dream of service architecture

You create a subroutine in Java. You ask politely that it be presented as a service. The service is written for you. Your service's nature is advertised on the web. Your service becomes discoverable in searches. Clients are automatically written to call the service -- in any language.
This is not science fiction, but I am glossing over some really important details:

You must comply with a service pattern in order for your subroutine to be wrappable.

This pattern includes requirements on how you store and retrieve local persistent data.

The paradigm has limits: notably, that the service input and output schemas only represent the limits of the language's type system.

You must learn a new programming paradigm: generate and modify.
Details of a cloud-based service

Part I: Code generation strategies. What happens "outside the code"?
Part II: Persistence strategies. What has to happen "inside the code"?
Part I: code generation

How can we avoid the tedium of XML/XSchemas in writing web services?
What will we give up as a result?
Developing SOAP the old way: "top-down"

Define input schema.
Define output schema.
Define request handler for the schema, that generates appropriate output for each input.
Define client that calls the request handler and encodes data according to the schema.

Zzzz...
The new world order: **bottom-up service development**

Start by defining Plain Old Java Objects (POJOs) that you want to turn into web services, e.g., that read NoSQL data. 

**Generate** XML Schemas, service stubs, etc using a **service deployment framework**, e.g., Apache Axis 2. 

**Don't even learn** XML or Schemas! The framework handles all of that!
Service deployment frameworks

Manage "deployment" of a service.
Require only minimal input from you.
Java Code to deploy.
Some configuration options.
Result: a running service.

Two main ones:
Apache Axis (LAMP)
Microsoft Visual Studio Services (IIS)
Apache Axis 2

- Built on top of the Apache Tomcat Java Servlet Engine.
- Supported in native Eclipse J2EE edition (without plugins, but you do have to download local copies of Apache Tomcat and Axis and get them running and bound into Eclipse).
- Takes plain old java objects (POJOs) as input.
- Generates service and client code that treats POJOs as services!
Requirements

The POJO must be "serializable": it must be possible to make a portable copy of the object.
The service calls must be methods in the object.
Using Axis

Define a simple service as a java class.
Tell Axis it should become a service.
"Package" the service.
"Deploy" the service.
Reference: http://wso2.org/library/95
/**
 * The service implementation class
 */

public class SimpleService {
    public String echo(String value) {
        return value;
    }
}

Pasted from <http://wso2.org/library/95>
Informing Axis of the service: services.xml

```
<service>
  <parameter name="ServiceClass" locked="false">SimpleService</parameter>
  <operation name="echo">
    <messageReceiver class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
  </operation>
</service>
```

Pasted from `<http://wso2.org/library/95>`
Packaging the service

Axis expects service files to be found in a specific directory hierarchy.

Creating the hierarchy:

```bash
mkdir temp
javac SimpleService.java -d temp/
mkdir temp/META-INF
cp services.xml temp/META-INF/
cd temp
jar -cvf SimpleService.aar *
```
Deploying the service

Choose a web server platform (e.g., Apache Tomcat+Axis2 or SimpleHTTPserver + Axis2) (Instructions differ.)

Simplest deployment: SimpleHTTPserver included with Axis2.
Using SimpleHTTPServer

Create a "repository" axis2-repo containing

conf subdirectory containing axis2.xml
empty modules subdirectory (contains jars that your code might need).

services directory containing SimpleService.aar

Run the SimpleHTTPServer:

sh http-server.sh /path/to/axis2-repo
Now things get interesting...

Axis2 Services are **reflective**, i.e., they are self-describing on the web.

If one runs the SimpleHTTPserver and browses to [http://localhost:8080/services/SimpleService](http://localhost:8080/services/SimpleService) one gets a description of the service and even how to use it. Weird: one has to have the service **running** before one can **build a client** to it.
Creating a client

Client code is generated by querying the reflection interface for a running service! (It doesn't matter how you wrote the service)

```
```

This generates code in the Axis2 hierarchy that calls the service.

It can be run on a different host than the host on which the service was developed!

It can be written in a different language.
// SimpleServer.echo corresponds to
// SimpleServiceStub.EchoResponse
// in generated code.

package org.apache.axis2;

import org.apache.axis2.

SimpleServiceStub.EchoResponse;

public class Client {
    public static void main(String[] args) throws Exception {
        SimpleServiceStub stub = new SimpleServiceStub();

        SimpleServiceStub.Echo request = new SimpleServiceStub.Echo();
        request.setParam0("Hello world");

        EchoResponse response = stub.echo(request);
        System.out.println("Response : " + response.get_return());
    }
}

Pasted from <http://wso2.org/library/95>
What did we give up?

XSchemas are generated by the language type system.
They do not fully enforce preconditions.
In Axis, it isn't possible to do so.
    Reason: just-in-time compilation.
XML is slow to encode/decode. This is a performance issue. So we don't use this for non-public interfaces.
## A tale of two service generators

<table>
<thead>
<tr>
<th>Apache Axis</th>
<th>Microsoft Visual Studio Web Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Just-in-time compilation</td>
<td>Generate and modify</td>
</tr>
<tr>
<td>XSchema preconditions are based upon</td>
<td>Can modify XSchemas to assert more</td>
</tr>
<tr>
<td>language type system</td>
<td>than language-based preconditions</td>
</tr>
</tbody>
</table>

Monday, November 26, 2012
2:46 PM
Your java subroutine must be written to be:

**horizontally scalable**: it is possible to deploy more service instances without problems.

**stateless**: any instance of the service can handle any request.

**consistency aware**: it is not possible for multiple instances to get confused about persistent service state.
Relationship between Parts I and II

Part I: how to generate a service from a java class.
Part II: how to make that class intelligently use persistent data.
A simple example of persistent state

An "availability" service: remembers whether you're busy now or available to chat.
Key: your user identifier (e.g., Windows LiveID or Google ID)
Value: boolean: TRUE if you're available.

Two service calls:
put(ID, avail): set the availability for a user ID to a specific state
avail = get(ID): get the availability for a user ID.
A cloud model of a service

Monday, November 26, 2012
11:25 AM

"Does output of `get` reflect the put?"
We want horizontal scalability of the front end: we should be able to add front-ends without modifying behavior. We want vertical scalability of the datastore: we should be able to add storage resources without changing behavior.
"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...

When you write a service
All data must be stored in the cloud.
There is no useful concept of local data.
There is no concept of server-to-server communication
except through the datastore.
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.
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.
Consistency/Concurrency 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 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 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!
Two approaches to Consistency and Concurrency

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!
Modifying distributed objects in AppEngine

A distributed object is controlled by a persistence manager.
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 String getLastName() {
        return lastName;
    }

    public Date getHireDate() {
        return hireDate;
    }
}
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>
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.
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

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);

Past from
<http://code.google.com/appengine/docs/java/datastore/creatinggettinganddeletingdata.html>
Retrieving an object via a known key

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

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

Changing persistent objects
Wednesday, January 27, 2010
1:52 PM
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?

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!

If -- on average -- there is one app looking at the object, then -- on average -- no one waits!
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
\begin{verbatim}
e.setSalary(e.getSalary()+100)
\end{verbatim}
is the same thing as (and is implemented as!)
\begin{verbatim}
tmp = e.getSalary()
tmp = tmp + 100
e.setSalary(tmp)
\end{verbatim}
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 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.:

```java
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!
}"
```
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

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.
Employee e = pm.getObjectById(Employee.class, k);
e.setSalary(e.getSalary()+100); // Give Alfred a raise!

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;
        }
    }
}
End of Lecture on 11/26/2012
Monday, November 26, 2012
5:33 PM
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** = "**N**ot only **S**tructured **Q**uery **L**anguage"

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 syntax
Optimizes simple key fetches for quick response.
Often contains a fallback to interpreting full SQL, with performance disadvantages.
A typical NoSQL service

Operations are:

`ValueType get(KeyType key)`
- Assert the binding of a key to a value.
- Subsequent puts update data.

`int put(KeyType key, ValueType value)`
- Gets the last value associated with a key (most of the time).
Roots of NoSQL

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 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. 
(High) **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).
Impact of the CAP theorem

In the context of cloud services:

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

Only other reasonable class is $\mathbf{AP} = \text{Availability} + \text{Partitionability}$

We don't usually 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 (AppEngine/BigTable) in open source.
Inside Dynamo

An arbitrary choice...
Because it's very well documented...
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 Dynambo

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 (queue) on each (redundant) storage node.

Local history is
- Pruned if there are no conflicts.
- Resolved (in an application-specific manner) if conflicts occur.
Example: shopping cart
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" Dynamo
one would usually not want to store business data in an eventually consistent store.
customer dissatisfaction: "buy this" responds with "you haven't bought it (yet)"
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 directly to their customers.