Pieces of the puzzle so far

Google AppEngine/GWTJ: a platform for cloud computing.

Map/Reduce: a core technology of cloud computing.

Google Map/Reduce
- closed system
- real-time queries
  - Can specify custom map operations (slowly).
  - Only one kind of reduce (collect).

Hadoop Map/Reduce
- Open system
- Specify map/combine/reduce operations.
  - Not real-time.
  - Requires another piece to expose data.
Some missing pieces

From Hadoop to applications.
   How to store massive M/R computations.
   The concept of a service.
The hidden role of business logic.
   Why clouds are important.
A rather important missing piece

Hadoop Map/Reduce is not a real-time technology. Instead, we feed Map/Reduce output into an object store. This is typically a Distributed Hash Table (DHT).

Dynamo
Voldemort

raw

Hadoop Map/Reduce

periodic

1/day

(relatively slow)

DHT

very fast

server

client
You might expect that in thinking about clouds, you would have a cloudy perspective!

Clouds are about **closing boxes** and **keeping them closed**.

Cloud abstractions allow us **not** to know things. Skill of doing this relies upon a new way of approaching programming.
I was all set to bury you in details of SOAP. I quickly decided that everything I wanted to tell you was irrelevant, useless, and peripheral! I had to function at a completely new level of abstraction.
The old-style programmer can't program the cloud. My generation has difficulty. The new-style programmer approaches the cloud differently, by being able to close boxes and keep them closed. The next-generation programmer is not going to be concerned with boxes, but instead, with patterns, best practices, and "technological shamanism".
A pattern contains detail about limits and scope. An example is a "ritual": do this, get that.
Abstraction botches

An **abstraction botch** occurs when the choice of level of abstraction is so wrong that it limits either functionality or further development.
The worst abstraction botch...

... is to write code that only works for one vendor/platform/configuration.
... so you're trapped into using them, forever!

The "wrong" way to implement JDO:
write it to write and read directly from Oracle10g, mysql, etc. Oracle would really like you to do this!

The "right" way to implement JDO:
use an abstraction layer that can run it on top of any grid-enabled datastore, regardless of who wrote it!

There is a business case for appropriate layering of abstraction... because you aren't locked into one vendor.
Don't pay any attention...

Don't pay any attention to that man behind the curtain!
(Wizard of OZ)
Cloud programming is about using strategic abstraction to hide details.
There is danger in even looking behind the curtain:
  It isn't empowering.
  It is hopelessly confusing.
Cloud programming requires a different way of thinking.
Two schools of programming...

The old school of programming: you must know exactly what your program is doing, or "you'll be sorry".
  Avoid abstractions.
  Use operational semantics (what something does is defined through exactly how it does it).

The cloud school of programming: you don't need to know exactly what is happening; only that at some level, your program's behavior is predictable.
  Embrace abstractions.
  Use axiomatic semantics and/or contracts (what something does is defined through external invariants of its behavior).
Implementing a service

Figure out what it is supposed to do.
Implement basic code that does it.
Embed this inside a web server, somehow.

Example: location awareness:
what it should do: take in IP address, output <lat,long>
how it should do it: look up IP address in a datastore.
how to embed this inside a web server?
Service Architecture

Two parts to a service

How it's implemented, e.g., via a datastore backend.

How it's requested and responds.

Key to service programming

I specify how it's implemented.

Interface to the outside world is generated.
Seran

conforms to XSD schema

XML Request

XML Response

conforms to some
specified XSD schema:

describes what XML

should contain

SOAP:

- checks that

  input schema

  is correct

- calls same Seran

- checks that

  output is correct
Slightly more detailed model

A SOAP request is described as a series of verifications, a HUGE amount of plug-and-chug programming.
Three kinds of "services"

Remote procedure call.
SOAP/REST (web) service.
Servlets
Remote Procedure Calls

Client requests data from server.
Server replies to client.
Utilizes a portable data format that is assured to mean the same thing on client and server.
Most common format: XML-RPC.

Other interesting formats
Native RPC
Google protocol buffers.
SOAP: Simple object access protocol
   Client sends XML "request" to server.
   Server responds with XML to client.
   XML Schemas determine legal inputs and outputs.

REST: Representative State Transfer
   Client sends a request to a specific URL with embedded CGI data.
   Server responds with XML or plaintext.
   Service is invoked via a regular webserver URL.

   www.customer.com/invoice/54/paid=true
Choosing the format of a service

Servlets: most suitable for services that output end-user data.
REST: most suitable for services whose input is simple and whose output is used by other parts of an application.
SOAP: best for complex inputs and outputs.
Basic SOA building block: Simple Object Access Protocol (SOAP)

Apache Axis: builds SOAP services from Java code.
You specify a plain-old-java class.
Axis translates this into a web service.

(End of lecture on 3/9/2011)
SOAP is:
   A request/response protocol.
   Request and response are coded as XML, validated through XML schemas.
Components of a SOAP request:
   An XML schema for valid input.
   XML input conforming to the schema.
Components of a SOAP response:
   An XML schema for valid output.
   XML output conforming to the schema.
A SOAP service
   Receives requests conforming to the input schema.
   Generates responses conforming to the output schema.
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 Hadoop HDFS files. **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.

Deployment and code generation
A deployment framework is not a code generator.
The XML schemas for your service aren't even stored.
Instead, they are generated from Java code through inspection when they are needed.
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!
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

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

Creating the hierarchy:

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`

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>
So, what have we done?

On our server:
   We provided a Plain Old Java Object (POJO).
   Server-side Axis implemented a web service based upon it.
Meanwhile, halfway around the world (or on Mars):
   Client-side Axis implemented a client stub class that calls it.
   We wrote a client that calls the stub.
Neither side cares even whether the other is using Axis!
But wait, there's more! (how much would you pay?)
Axis provides for free:
   WSDL: a definition of your service in a standard form.
   WSAPI: a standard for inspecting a service.
But these represent a dream that never happened!
There will be a "service marketplace". Companies will compete to provide services. We'll dynamically bind to services based upon price; ("if it's Tuesday, we'll use google maps, but on Thu, we'll use MapQuest!") Application programs will intelligently choose between services dynamically. Not.
Problems with the dream

Trust: can we trust the vendor to be honest?
Ontology: can we trust the vendor to even use the same language as us?
The problem of trust

Can you trust your service provider to be honest?
Can you trust your service provider to keep providing?
   Case law: no, you can't.
What guarantees do you get from the provider?
   Case law: don't believe them.
The problem of ontology

Vendors describe the same things with different words

Location service 1:
  Input: ADDRESS
  Output: Lat, Long

Location service 2:
  Input: IP
  Output: Latitude, Longitude

How do we know these are the same?

This problem really comes into its own when one is trying to communicate business-critical, legal, or medical concepts!
What is an ontology?

A taxonomy is a mapping from names to things. Each web service has its own taxonomy, as expressed by its API (WSDL).

An **ontology** is a **mapping between taxonomies**. E.g., to solve the problem on the previous page:

"IP" is an address in one taxonomy.

"ADDRESS" is an address in the other.

An ontology must relate "IP" <-> "ADDRESS"
Beyond the service marketplace:
  One buys a "service stack" consisting of a set of interoperating services.
  One vendor, explicit contract.
  No dynamic switching.