LECTURE 7: DISTRIBUTED OBJECT- & WEB-BASED SYSTEMS

SECTION 7.1: DISTRIBUTED OBJECT-BASED SYSTEMS
Distributed Objects

- **Introduction**
  - In *distributed object-based* systems, an object plays a key role in getting *distribution transparency*.
  - Everything is treated as an object & clients are offered services/resources as objects that they can invoke.
  - Distributed objects form an important paradigm as it’s ‘easy’ to hide distribution aspects behind an object’s interface.
  - As object can be almost anything, also useful paradigm for building systems.
  - Key feature of objects is they encapsulate data (aka *state*), & operations on those data, (aka *methods*).
  - Methods are made available through an *interface*.
  - Process can only access/change object’s state by invoking methods made available via an object’s interface.
  - An object may implement multiple interfaces and for an interface definition, can be several objects offering an implementation of it.

Distributed Objects (/2)

- **Architecture**
  - The separation between interfaces & objects implementing them is crucial for distributed systems.
  - It allows for placing interface at one machine, with object itself on another machine.
  - This organization is commonly referred to as a *distributed object definition*.

Organization of a Distributed Object with a Client-Side Proxy
Distributed Objects (/3)

• **Architecture**
  - Data & operations *encapsulated* in an object,
  - Operations implemented as methods grouped into interfaces
  - Object offers only its *interface* to clients
  - *Object server* is responsible for a collection of objects
  - *Client stub* (proxy) implements interface, marshals call
  - *Server skeleton* handles (un)marshalling and object invocation (+other stuff)

• **Types of objects I**
  - *Compile-time objects*: Language-level objects, from which proxy and skeletons are automatically generated.
  - *Runtime objects*: Implementable in any language, but need *object adapter* to make implementation appear as an object.

• **Types of objects II**
  - *Transient objects*: live only due to server: if server exits, so will the object.
  - *Persistent objects*: live independently of server: if server exits, object state & code remain (passively) on disk

Distributed Objects (/4)

• **Example: Enterprise Java Beans (EJB)**
  - Def: Java object hosted by special server that allows for different means of calling the object by remote clients.
  - Four Different Types of EJBs
    - *Stateless session bean*: Transient object, called once, does its work and is done.
      E.g.: execute SQL query, return result.
    - *Stateful session bean*: Transient object, but keeps client-related state until session end.
      E.g.: shopping cart.
    - *Entity bean*: Persistent, stateful object, can be invoked over many sessions.
      E.g.: object maintaining client info on last number of sessions.
    - *Message-driven bean*: Reactive objects, often triggered by message types. Used to implement publish/subscribe forms of communication.
Distributed Objects (/5)

- **Processes:** Object servers
  - **Servant:** Object implementation, sometimes only implements methods:
    - Collection of C or COBOL functions, that act on structs, records, DB tables, etc.
    - Java or C++ classes
  - **Skeleton:** Server-side stub handles n/w I/O:
    - Unmarshalls incoming requests, calls relevant servant code
    - Marshalls results and sends reply message
    - Generated from interface specifications
  - **Object adapter:** “Manager” of a set of objects:
    - Inspects (as first) incoming requests
    - Ensures referenced object is ‘activated’ (requires identification of servant)
    - Passes request to appropriate skeleton, following specific ‘activation’ policy
    - Responsible for generating object references

---

Distributed Objects (/6)

- **Client-to-object binding:**
  - Object reference
    - Having an object reference allows a client to **bind** to an object:
    - Reference denotes server, object, and communication protocol
    - Client loads associated stub code
    - Stub is instantiated and initialized for specific object
  - Two ways of binding
    - **Implicit:** Methods are invoked directly on referenced object
      - `this_object obj_ref;` // Declare a systemwide object reference
      - `obj_ref->do_something();` // Implicitly bind and invoke a method
    - **Explicit:** Client must explicitly bind to object first before invoking it
      - `Remote_object obj_ref;` // Declare a systemwide object reference
      - `Local_object* obj_ptr;` // Declare a pointer to local objects
      - `obj_ref = ...;` // Initialize the reference to a distrib. obj
      - `obj_ptr = bind(obj_ref);` // Explicitly bind and get ptr to local proxy
      - `obj_ptr->do_something();` // Invoke a method on the local proxy
  - Remote-object references allow us to pass references as parameters.
  - This was difficult with ordinary RPCs.
Distributed Objects (/7)

- Remote Method Invocation (RMI)
  - Java Remote Method Invocation (RMI) system allows an object running in one JVM to call methods on objects running in another.
  - RMI gives applications \textit{transparent, lightweight} access to \textit{remote objects}.
  - RMI defines a high-level protocol and API.
  - Programming distributed applications in Java RMI is simple:
    - It is a single-language system.
    - Remote object coder must consider behaviour in a concurrent environment.

- Java RMI Applications
  - RMI is supported by two java packages \texttt{java.rmi} \& \texttt{java.rmi.server}
  - An application that uses RMI has 3 components:
    - an \textit{interface} that declares headers for remote methods;
    - a \textit{server} class that implements the interface; and
    - one or more \textit{clients} that call the remote methods.

Distributed Objects (/8)

- A Java RMI application needs to do the following:
  - \textit{Locate remote objects}: An application can use one of two mechanisms to obtain references to remote objects:
    1. An application can register its remote objects with RMI's simple naming facility the \texttt{rmiregistry}, or
    2. The application can pass and return remote object references as part of its normal operation.
  - \textit{Communicate with remote objects}:
    - Details of communication between remote objects are handled by RMI;
    - To coder, remote communication looks like standard Java method call.
  - \textit{Load class bytecodes for objects that are passed around}:
    - RMI provides necessary mechanisms to load object’s code* & send its data.
    - Reason for this is that RMI allows caller to pass objects to remote objects.

* i.e. object translated/’serialized’/’marshalled’ into bytecode
Distributed Objects (/9)

- **RMI Architecture**
  - **Stub**: lives client-side; pretends to be the remote object
  - **Skeleton**: lives on server; talks with true remote object
  - **Reference Layer**: determines if referenced object is local or remote
  - **Transport Layer**: packages remote invocations;
    - dispatches messages between stub & skeleton

```
Client
  ↓
Stub

Server
  ↑
Skeleton

RMI Reference Layer
RMI Transport Layer
TCP/IP
```

Distributed Objects (/10)

- **Java RMI Basics**: (Assumes client stub, server skeleton in place)
  - Client invokes method at **stub**
  - **Stub** marshals request and sends it to server
  - Server ensures referenced object is active:
    - Create separate process to hold object
    - Load the object into server process
    - ...
  - Object **skeleton** unmarshalls request & referenced method is invoked
  - If request contains object reference, invocation is applied recursively
    (i.e., server acts as client)
  - Result is marshalled and passed back to client
  - Client **stub** unmarshalls reply & passes result to client application
Distributed Objects (/10)

• RMI: Parameter passing

  **Object reference**: Much easier than in the case of RPC:
  – Server can simply bind to referenced object, and invoke methods
  – Unbind when referenced object is no longer needed

• **Object-by-value**: Client may also pass a complete object as parameter value:
  – An object has to be marshalled:
    • Marshall its state
    • Marshall its methods, or give ref to where an implementation can be found
  – Server unmarshalls object (n.b. now have copy of original object)
  – Object-by-value passing tends to introduce nasty problems

---

Distributed Objects (/11)

• RMI Parameter Passing
  – **Note**: System-wide object reference usually contains:
    • Server address
    • Port to which adapter listens, and
    • Local object ID.
  – **Extra**: Info on protocol between client & server (TCP, UDP, SOAP, etc.)
Distributed Objects (/12)

• RMI Registry
  – A simple server-side bootstrap naming facility allowing remote clients to get a reference to a remote object
    • Servers name & register their objects to be accessed remotely with the RMI Registry.
    • Clients use the name to find server objects and obtain a remote reference to those objects from the RMI Registry.
  – Registry service is background program with a list of registered server names on a host and invoked by: `rmiregistry port` &
  – Registry service is provided by a Naming object providing two key methods:
    • Bind: to register a name and server
    • Lookup: to retrieve the server bound to a name

RMI Inheritance

```
Java.rmi.RemoteObject
  └── Java.rmi.Remote
       └── Java.rmi.UnicastRemoteObject
            └── MyServerInterface
                └── MyServer
```

implement
extends
Security Manager

• RMI programs must install a *security manager*
  – Otherwise RMI will not download classes

```java
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
}
```

– Security policies specify actions that are *unsafe*
  • For every unsafe action there is a corresponding `checkXXX()` method
  • Actions not allowed throw a `SecurityException`

– Only one security manager can be installed
  • By default, an application has no security manager installed

– Policies are specified using `*.policy` files
  • Server and client application must specify their policy file
    – Default file: `java.home/lib/security/java.policy`
  • Use `-Djava.security.policy` property specify a file

---

RMI Example: Database Interface

```java
import java.rmi.*;
import java.rmi.server.*;
public class Database extends UnicastRemoteObject
    implements DatabaseInterface {
    private int data = 0; // the database

    public Database(int value) throws RemoteException {
        data = value;
    }

    public int read () throws RemoteException {
        return data;
    }

    public void write (int value) throws RemoteException {
        data = value;
        System.out.println("New value is: " + data);
    }
}
```
RMI Example (/2): Database Server

```java
import java.rmi.*;
import java.rmi.server.*;
public class DatabaseServer {
    public static void main(String[] args) {
        try {
            // create Database Server Object
            Database db = new Database(0);

            // register name and start serving
            String name = "rmi://fuji:9999/DB";
            Naming.bind(name, db);
            System.out.println(name + " is running");
        } catch (Exception ex) {
            System.err.println(ex);
        }
    }
}
```

RMI Example (/3): Database Client

```java
import java.rmi.*;
public class DatabaseClient {
    public static void main(String[] args) {
        try {
            // set RMI Security Manager
            System.setSecurityManager(new RMISecurityManager() {
                public void checkConnect(String host, int port) {}
                public void checkConnect(String host, int port, Object Context) {}
            });

            // get database object
            String name = "rmi://fuji:9999/DB";
            DatabaseInterface db = (DatabaseInterface)Naming.lookup(name);
            int value, rounds = Integer.parseInt(args[0]);
            for (int i = 0; i < rounds; i++) {
                value = db.read();
                System.out.println("read: "+value);
                db.write(value+1);
            }
        } catch (Exception ex) {
            System.err.println(ex);
        }
    }
```
RMI Example (/4): Building the Application

Steps involved in Building the Application:

1. Compile the code:
   ```
   javac Database.java DatabaseClient.java
   DatabaseInterface.java DatabaseServer.java
   ```

2. Generate stub and skeleton class files:
   ```
   rmic Database (note: not needed for Java 5 or later)
   ```

3. Start the RMI registry (if don’t specify port, 1099 is the default):
   ```
   rmiregistry 9999 &
   ```

4. Start the Server:
   ```
   java -Djava.security.policy=java.policy DatabaseServer
   ```

5. Start the Client:
   ```
   java -Djava.security.policy=java.policy DatabaseClient 10
   ```

**SECTION 7.2:** DISTRIBUTED WEB-BASED SYSTEMS
Introduction to Web Services

- WS offered by one electronic device to another, communicating via web
- Here, web technology (e.g. HTTP), originally to be used for human-to-machine comms, is used for M2M chatter, e.g. in XML and JSON.
- HTTP defines message format, how sent and what Web servers & browsers do in turn
- WS typically provides OO web-based interface to a DB server, used by another web server, or mobile apps showing UI to end users
- In 2002, W3C defined a WS Architecture,
  - Req'd standardized “Web service” impln with interface described in WSDL.
- Other systems interact with the WS using SOAP* messages, typically using HTTP with XML serialization with other Web-related standards.
- Later extended to include
  - REST-compliant WS, where service changes forms of Web resources (URIs) using a uniform set of stateless operations (aka ‘CRUD’)
  - Arbitrary WS where service exposes arbitrary operations (little used)

*Simple Object Access Protocol, now largely falling out of use, though with some specialist applications

Background to Web Services

- Apache Web servers
  - Observation: More than 37% of all 1 billion* Websites are Apache.
  - Server is internally organised roughly according to steps needed to process an HTTP request.
  - The anatomy of an Apache Web Server is shown below:

Background to Web Services (/2)

- **Server Clusters**
  
  - **Essence:** To improve performance & availability, WWW servers are often clustered in a way that is transparent to clients.
  
  - Below a server cluster is used with a front end to implement a WS.

---

Background to Web Services (/3)

- **Problem with Server Clusters:**
  
  - Front end gets easily overloaded, thus need for special measures.

  1. **Transport-layer switching:**
     
     - Front end simply passes TCP request to a server, according to some performance metric (e.g. load balancing).

  2. **Content-aware distribution:**
     
     - Front end reads the content of HTTP request and selects best server.
Background to Web Services (/4)

• Naming: The Naming Service
  – Names play a very important role in all computer systems.
  – For sharing resources, uniquely identifying entities, referring to locations...
  – Important issue for naming:
    • a name must be resolvable to its entity it refers to,
    • for Name resolution need to implement a Naming System.
  – Naming in distributed systems & non-distributed systems differs in the implementation.
  – In Chord, DS naming system implementation is itself often distributed.
  – How this distribution is done dictates efficiency & scalability of the naming system.

Background to Web Services (/5)

• Naming: Names in General
  – Name in DS: string of bits/characters used to refer to it.
  – Entities
    • In DS can be anything (e.g. resources such as hosts, printers, disks & files).
    • Other examples of explicitly named entities are processes, users, mailboxes, Web pages, messages, network connections.
  – Entities can be operated on
    • e.g., a printer offers an interface with operations for printing docs & others
    • e.g. network connection offers data send/receive, set QoS parameters etc.
  – Operating on entities need an Access Point, another DS entity:
    • The name of an access point is called an address.
    • Address of entity’s access point entity is called an address of that entity.
  – Note: A location-independent name for an entity $E$, is independent from the addresses of the access points offered by $E$. 
Background to Web Services (/6)

- **Naming: Names in General** (cont’d)
  - Entities can offer more than one access point
    - e.g. phone is person’s access point, with phone number as address
    - people have many phone numbers, for their many addresses.
  - In DS, a typical access point is a host running a specific server.
    - address is e.g. IP address+port (i.e. server’s transport-level address).
  - Entities may change access points over course time.
    - laptop moves location, it’s often assigned a different IP address
    - similarly, changing jobs or ISPs, means changing e-mail addresses.

Background to Web Services (/7)

- **Naming: Identifiers**
  - Pure name
    - A name that has no meaning at all; it is just a random string.
    - Pure names can be used for comparison only.
  - **Identifier**: A name having the following properties:
    - P1: Each identifier refers to at most one entity
    - P2: Each entity is referred to by at most one identifier
    - P3: An identifier always refers to the same entity (prohibits reusing an identifier)
  - Observation
    - Identifier needn’t necessarily be a pure name i.e. can have content
Background to Web Services (/8)

• Naming: Uniform Resource Locator (URL)
• Often contain information on how/where to access a document.
• Some URLs
  - Using only a DNS Name
  - Combining a DNS name with a port number
  - Combining a DNS name with a port number

section 7.2.1: soap-based web services
Web Services: SOAP-Based

• The Principle of a Web Service
  – Standardization dictates how those services are described such that they can be looked up by a client application.
  – Also, need to ensure that service call proceeds according to server application rules.
  – This is no different from what is needed to realize a remote procedure call.

Web Services: SOAP-Based (/2)

• Standardization needed so client can look up/access services.
  – Three Components:
      • Adheres to Universal Description, Discovery & Integration standard (UDDI).
      • As its name suggests, this prescribes DB layout with service descriptions.
      • Allows Web service clients to browse for relevant services.
    – Interface: Services described in Web Services Definition Lang (WSDL).
      • Formal language akin to IDLs used to support RPC-based communication.
      • Description contains precise definitions of interfaces provided by a service.
        – e.g. procedure specification, data types, (logical) location of services, etc.
      • A WSDL description is one that can be automatically translated to client-side and server-side stubs, akin to in ordinary RPC-based systems.
    – Communication: Simple Object Access Protocol (SOAP) is used
      • Specification of how communication takes place.
      • SOAP is used, which is essentially a framework for standardizing communication between two processes.
Web Services: SOAP-Based (/3)

• **Service-Oriented Architectures**
  • So far, a Web service is offered in terms of a single invocation.
    • In practice, more complex invocation structures needed before a service can be considered as completed.
      e.g. book order requires selecting a book, paying, and ensuring its delivery.
    • So must model actual service as a transaction with multiple ordered steps.
    • Means dealing with a complex service built from number of basic services.
  • **SOA principles for organising s/w not restricted to Web services use**
    • Loose Coupling (independent & self-contained)
    • Discoverability
    • Abstract service description (independent of implementation)
    • Encapsulation (autonomy and abstraction)
    • Compositionality (can be composed of other services)
    • Additional for web services: based on open standards & vendor neutral

Web Services: SOAP-Based (/4)

• **Java Web Services**: Java supports web services thro JAX-WS
  – **JAX-WS** = Java API for XML-Web Services.
  – Java Web Services can be deployed in the following ways:
    • Core Java only
    • Core Java with the current Metro release (helps when building a client)
    • Stand-alone web container (e.g. Tomcat)
    • Java application server (e.g. Glassfish – useful for implementing EJB)
  – Can implement SOAP-based web service as a single Java class
  – But usually consists of the following:
    • SEI (Service Endpoint Interface): Declares methods (web service operations)
    • SIB (Service Implementation Bean)
      – Defines the methods declared in the Interface
      – Can be either POJO (Plain Old Java Object) or EJB (Enterprise Java Bean)
Web Services: SOAP-Based (/4)

• Writing a Web Service Client
  – Web service client is a program using Web service, e.g. Java application
  – How to access the Web services:
    • Send a HTTP POST request with request as SOAP message to server
    • Better: use wsimport to generate Java stubs to do this for you
  – However, wsimport needs a description of Web services offered by
    the Web server:
    • Use WSDL document generated by the Web server
    • URL of this document can be obtained by looking at Web services
      section at http://localhost:4848

package ch01.ts;   // time server
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

/**
* The annotation @WebService signals that this is the
* SEI (Service Endpoint Interface). @WebMethod signals
* that each method is a service operation.
* The @SOAPBinding annotation impacts the under-the-hood
* construction of the service contract, the WSDL
* (Web Services Definition Language) document. Style.RPC
* simplifies the contract and makes deployment easier.
*/
@WebService
@SOAPBinding(style = Style.RPC) // more on this later
public interface TimeServer {
    @WebMethod String getTimeAsString();
    @WebMethod long getTimeAsElapsed();
    // These methods can be called akin to an RMI interface
    // But no remote exceptions thrown.
}
package ch01.ts;

import java.util.Date;
import javax.jws.WebService;

/**
 * The @WebService property endpointInterface links the
 * SIB (this class) to the SEI (ch01.ts.TimeServer).
 * Note that the method implementations are not annotated
 * as @WebMethods.
 */

@WebService(endpointInterface = "ch01.ts.TimeServer")
// Links the service to the interface
public class TimeServerImpl implements TimeServer {
    public String getTimeAsString() { return new Date().toString(); }
    public long getTimeAsElapsed() { return new Date().getTime(); }
}

package ch01.ts;

import javax.xml.ws.Endpoint;

/**
 * This application publishes the Web service whose SIB is ch01.ts.TimeServerImpl.
 * For now, the service is published at network address 127.0.0.1., which is localhost,
 * and at port number 9876, as this port is likely available on any desktop machine.
 * The publication path is /ts, an arbitrary name.
 * The Endpoint class has an overloaded publish method. In this two-argument version,
 * the first argument is the publication URL as a string and the second argument is
 * an instance of the service SIB, in this case ch01.ts.TimeServerImpl.
 * The application runs indefinitely, awaiting service requests. It needs to be
 * terminated at the command prompt with control-C or the equivalent.
 * Once the application is started, open a browser to the URL
 * http://127.0.0.1:9876/ts?wsdl
 * to view the service contract, the WSDL document. This is an easy test to
 * determine whether the service has deployed successfully. If the test succeeds,
 * a client then can be executed against the service.
 */

public class TimeServerPublisher {
    public static void main(String[] args) {
        // 1st argument is the publication URL
        // 2nd argument is an SIB instance, implementor obj to create interface implns dynamically
        Endpoint.publish("http://127.0.0.1:9876/ts", new TimeServerImpl());
        // After publish has been called, endpoints starts accepting incoming requests
    }
}
**TimeServer (/4)**

- **TimeServer: Compiling and Running**
  - Compiling the SEI, SIB and publisher `javac ch01/ts/*.java`
  - Running the publisher `java ch01.ts.TimeServerPublisher`
  - Testing the web service with the browser:
    - Access the URL: `http://127.0.0.1:9876/ts?wsdl`
    - Accessing WSDL using `curl`:
      ```bash
curl http://127.0.0.1:9876/ts?wsdl
```
- **TimeServer will Return the current time:**
  - Either as a string or
  - Elapsed milliseconds from Unix epoch, midnight January 1, 1970 GMT.

---

**TimeServer (/5): Ruby Client**

```ruby
#!/usr/bin/ruby

# one Ruby package for SOAP-based services
require 'soap/wsdlDriver'

wsdl_url = 'http://127.0.0.1:9876/ts?wsdl'

# Get a service object from the WSDL_url
service = SOAP::WSDLDriverFactory.new(wsdl_url).create_rpc_driver

# Save request/response messages in files named '...soapmsgs...'
# since want to inspect them
service.wiredump_file_base = 'soapmsgs'

# Invoke service operations.
result1 = service.getTimeAsString
result2 = service.getTimeAsElapsed

# Output results.
puts "Current time is: #{result1}"
puts "Elapsed milliseconds from the epoch: #{result2}"```

---
TimeServer (/6): Perl Client

```perl
#!/usr/bin/perl -w

use SOAP::Lite;
# provides under-the-hood functionality allowing client to issue
# appropriate SOAP request & process the ensuing SOAP response

my $url = 'http://127.0.0.1:9876/ts?wsdl';
# request url ends with a query string asking for WSDL doc

my $service = SOAP::Lite->service($url);
# PERL client gets WSDL and SOAP::Lite library then generates
# appropriate service object. In consuming WSDL doc, SOAP::Lite gets
# info needed (e.g. WS operations & their data types)

print "Current time is: ",
    $service->getTimeAsString();
print "Elapsed milliseconds from the epoch: ",
    $service->getTimeAsElapsed(), "\n";
```

TimeServer (/7): HTTP Request

POST http://127.0.0.1:9876/ts HTTP/ 1.1
Accept: text/html
Accept: multipart/*
Accept: application/soap
User-Agent: SOAP::Lite/Perl/0.69
Content-Length: 434
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tns="http://ts.ch01/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:Body>
<tns:getTimeAsString xsi:nil="true"/>
</Body>
</soap:Envelope>
```

- HTTP Startline specifies it’s a POST method
- `<soap:Body>` contains a single method whose localname is `getTimeAsString`
TimeServer (/8): HTTP Response

HTTP/1.1 200 OK
Content-Length: 323
Content-Type: text/html; charset=utf-8
Client-Date: Mon, 28 Apr 2008 02:12:54 GMT
Client-Peer: 127.0.0.1:9876
Client-Response-Num: 1

<?xml version="1.0"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:ans='http://ts.ch01'/>
  <soapenv:Body>
    <ans:getTimeAsStringResponse xmlns:ans='http://ts.ch01'/>
      <return>Thu Mar 21 14:45:17 GMT 2013</return>
    </ans:getTimeAsStringResponse>
  </soapenv:Body>
</soapenv:Envelope>

• HTTP/1.1 200 OK signals all processed normally

TimeServer (/9): WSDL Document Structure

• A WSDL document has two parts:
  – Interface (abstract)
    • Available services: operations grouped in portTypes
    • Which messages are needed by operations: A message can have parts
    • Used data types and XML-elements
  – Implementation (concrete)
    • binding to message layer (e.g. SOAP):
      How message parts mapped to body/header elements of SOAP messages
    • bindings to transport layer (e.g. HTTP):
      Where do I find the service?
    • A service may offer several ports,
      i.e. ways to call it
TimeServer (/10): WSDL Document Structure

```xml
<message name="getTimeAsString"></message>
<message name="getTimeAsStringResponse">
  <part name="return" type="xsd:string"></part>
</message>
<message name="getTimeAsElapsed"></message>
<message name="getTimeAsElapsedResponse">
  <part name="return" type="xsd:long"></part>
</message>
```

- For the `TimeServer` service, four messages

```xml
<portType name="TimeServer">
  <operation name="getTimeAsString" parameterOrder="">
    <input message="tns:getTimeAsString"></input>
    <output message="tns:getTimeAsStringResponse"></output>
  </operation>
  <operation name="getTimeAsElapsed" parameterOrder="">
    <input message="tns:getTimeAsElapsed"></input>
    <output message="tns:getTimeAsElapsedResponse"></output>
  </operation>
</portType>
```

- `portType` for `TimeService` has two operations, each with one input message & one output message

TimeServer (/11): Generating Client Support Code From WSDL

- After `TimeServerPublisher` generated WSDL, execute:

  ```bash
  ```

  - The `-keep` option specifies that the source files should be kept
  - The `-p client` option specifies Java package in which generated files are to be placed
  - Above command generates two source & two compiled files in the subdirectory `client`

- Approaches to Web Services 1: The Contract-First Approach

  - Above approach, where WSDL contract is used to generate all required artifacts for WS development, deployment, & invocation is known as the `Contract-First Approach`. 
TimeServer (/12): Generating WS Artifacts From Java Code

• Approaches to Web Services 2: The Code-First Approach
  - A second approach, where Java classes are available and used to generate all required artifacts for WS development, deployment, & invocation is known as Code-First Approach.
  - Command wsgen -cp [Compiled Java Code] achieves this.
  - Run the publisher to deploy the web service.
• This contrasts with the Contract-First seen earlier which was a top-down approach to generate JAX-WS Artifacts
• In general, for a number of reasons Contract-First approach is preferred to Code-First

Following lists the process to create a WS starting from Java sources, classes, or a WSDL file (server side):

• Starting from Java classes use Code-First:
  - Use wsgen to generate portable artifacts (e.g. SE Interface & Implementation classes etc).
  - Deploy the Web Service
• Starting from a WSDL file use Contract-First:
  - Use wsimport to generate portable artifacts.
  - Implement the service endpoint.
  - Deploy the Web Service
• Following lists the process to invoke a web service (client side):
  - Starting from deployed web service's WSDL
  - Use wsimport to generate the client-side artifacts.
  - Implement the client to invoke the web service.
TimeServer (/13): A Compromise Approach

• A third Approach: Code First, Contract Aware
  – Updating Code-First service, might find that WSDL changes too.
  – To get around this, there is a style called Code First, Contract Aware.
  – Write code first but annotate to tightly constrain generated WSDL.

• Some annotations:
  – @WebMethod, indicates a method exposed as Web Service operation,
  – @SOAPBinding specifies WS mapping onto SOAP message protocol
  – @WebParam maps a parameter to a WS msg part & XML element,
  – @WebResult specifies that operation result in generated WSDL is something other than default return e.g. IntegerOutput.

```java
package ch01.team;
import java.util.List;
import javax.jws.WebService;
import javax.jws.WebMethod;
package ch01.team;
import java.util.List;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Teams {
    private TeamsUtility utils;
    public Teams() {
        utils = new TeamsUtility();
        utils.make_test_teams();
    }

    @WebMethod
    public Team getTeam(String name) {
        return utils.getTeam(name);
    }

    @WebMethod
    public List<Team> getTeams() {
        return utils.getTeams();
    }
}
```

A Harder SOAP Example: The Teams Web Service
A Harder SOAP Example (/2)

```java
package ch01.team;
import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;

public class TeamUtility {
    private Map<String, Team> team_map;
    public TeamUtility() {
        team_map = new HashMap<String, Team> ();
        make_test_teams();
    }
    public Team getTeam(String name) {
        return team_map.get(name);
    }
    public List<Team> getTeams() {
        List<Team> list = new ArrayList<Team> ();
        for (String key : team_map.keySet()) {
            list.add(team_map.get(key));
        }
        return list;
    }
    private void make_test_teams() {
        List<Team> teams = new ArrayList<Team> ();
        Player burns = new Player(“George Burns”, “George”);
        Player allen = new Player(“Gracie Allen”, “Gracie”);
        List<Player> ba = new ArrayList<Player> ();
        ba.add(burns);
        ba.add(allen);
        Team burns_and_allen = new Team(“BurnsAllen”, ba);
        team.add(burns_and_allen);
        Player abbott = new Player(“Milliam Abbott”, “Bud”);
        Player costello = new Player(“Lou Cristillo”, “Lou”);
        List<Player> ac = new ArrayList<Player> ();
        ac.add(abbott);
        ac.add(costello);
        Team abbott_and_costello = new Team(“Abbott and Costello”, ac);
        teams.add(abbott_and_costello);
        Player chico = new Player(“Leonard Marx”, “Chico”);
        Player harpo = new Player(“Alphip Marx”, “Groucho”);
        List<Player> mb = new ArrayList<Player> ();
        mb.add(chico);
        mb.add(harpo);
        Team marx_brothers = new Team(“Marx Brothers”, mb);
        teams.add(marx_brothers);
        store_teams(teams);
    }
    private void store_teams(List<Team> teams) {
        for (Team team : teams) {
            team_map.put(team.getName(), team);
        }
    }
}
```

A Harder SOAP Example (/3)

```java
package ch01.team;
import java.util.List;
public class Team {
    private List<Player> players;
    private String name;
    public Team() {
        players = new ArrayList<Player> ();
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setPlayers(List<Player> players) {
        this.players = players;
    }
    public int getPlayers() {
        return players.size();
    }
    public class TeamPublisher {
        public static void main(String[] args) {
            int port = 8888;
            String url = “http://localhost:” + port + “/team”;
            ServerSocket s = new ServerSocket(port);
            Socket s1 = s.accept();
            BufferedReader in = new BufferedReader(new InputStreamReader(s1.getInputStream()));
            PrintWriter out = new PrintWriter(new OutputStreamWriter(s1.getOutputStream()));
            String line = in.readLine();
            while (line != “exit”) {
                String[] parts = line.split(“,”);
                String name = parts[0];
                int count = Integer.parseInt(parts[1]);
                Player player = new Player(name, “”);
                Team team = new Team();
                team.setName(name);
SECTION 7.2.2: REST-BASED WEB SERVICES

Introduction to REST

- **REST**, or **REpresentational State Transfer**, is a distributed communication architecture
  - Overall SOAP WS architecture has many layers with protocols & standards for security & reliability=>tedious for WS developers.
  - REST is fast becoming the lingua franca for Cloud Computing
  - Central REST abstraction is the **Resource** i.e. anything with a URI.
  - In practice, resource is an info item that has hyperlinks to it.
Contrast Between SOAP & REST

- REST & SOAP are quite different

<table>
<thead>
<tr>
<th>SOAP &amp; REST: Protocol Layering</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP Technology Stack</td>
</tr>
</tbody>
</table>

Contrast Between SOAP & REST (/2)

- REST & SOAP are quite different

<table>
<thead>
<tr>
<th>No.</th>
<th>SOAP</th>
<th>REST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>SOAP is a protocol.</td>
<td>REST is an architectural style.</td>
</tr>
<tr>
<td>2)</td>
<td>SOAP stands for Simple Object Access Protocol.</td>
<td>REST stands for Representational State Transfer.</td>
</tr>
<tr>
<td>3)</td>
<td>SOAP can’t use REST because it is a protocol.</td>
<td>REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.</td>
</tr>
<tr>
<td>4)</td>
<td>SOAP uses services interfaces to expose the business logic.</td>
<td>REST uses URI to expose business logic.</td>
</tr>
<tr>
<td>5)</td>
<td>JAX-WS is the Java API for SOAP web services.</td>
<td>JAX-RS is the Java API for RESTful web services.</td>
</tr>
<tr>
<td>6)</td>
<td>SOAP defines standards to be strictly followed.</td>
<td>REST does not define too much standards like SOAP.</td>
</tr>
<tr>
<td>7)</td>
<td>SOAP requires more bandwidth and resource than REST.</td>
<td>REST requires less bandwidth and resource than SOAP.</td>
</tr>
<tr>
<td>8)</td>
<td>SOAP defines its own security.</td>
<td>RESTful web services inherits security measures from the underlying transport.</td>
</tr>
<tr>
<td>9)</td>
<td>SOAP permits XML data format only.</td>
<td>REST permits different data format such as Plain text, HTML, XML, JSON etc.</td>
</tr>
<tr>
<td>10)</td>
<td>SOAP is less preferred than REST.</td>
<td>REST more preferred than SOAP.</td>
</tr>
</tbody>
</table>

Lecture 7: Distributed Obj & Web-based Systems  CA4006 Lecture Notes (Martin Crane 2017)
Contrast Between SOAP & REST (/3)

<table>
<thead>
<tr>
<th>SOAP Web Services</th>
<th>RESTful Web Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS Security:</td>
<td>WS Security:</td>
</tr>
<tr>
<td>• Defines own security (WS Security)</td>
<td>• Supports just standard security to set up</td>
</tr>
<tr>
<td>• Has standard impln of data integrity &amp; data privacy</td>
<td>encrypted link between server &amp; client SSL</td>
</tr>
<tr>
<td>Atomic Transaction:</td>
<td>ACID Transactions:</td>
</tr>
<tr>
<td>• Supports ACID transactions.</td>
<td>• Supports transactions, but not ACID compliant.</td>
</tr>
<tr>
<td>• Internet apps mostly don’t need transactional</td>
<td>• Limited by HTTP (can’t provide 2-phase commit</td>
</tr>
<tr>
<td>reliability, enterprise apps sometimes do.</td>
<td>across distributed transactional resources)</td>
</tr>
<tr>
<td>Messaging:</td>
<td>Reliable Messaging:</td>
</tr>
<tr>
<td>• Has successful/retry logic built in</td>
<td>• Has no standard messaging system</td>
</tr>
<tr>
<td>• End-to-end reliable even thro SOAP intermediaries.</td>
<td>• Expects clients to retry if comms failures</td>
</tr>
<tr>
<td>Slow:</td>
<td>Fast:</td>
</tr>
<tr>
<td>• Uses XML format that must be parsed to be read.</td>
<td>• No strict specification like SOAP.</td>
</tr>
<tr>
<td>• Defines many standards to be followed while</td>
<td>• Consumes less bandwidth and resource.</td>
</tr>
<tr>
<td>developing the SOAP applications.</td>
<td></td>
</tr>
<tr>
<td>• = slow &amp; consumes more b/w &amp; resource.</td>
<td></td>
</tr>
<tr>
<td>WSDL dependent:</td>
<td>Permits different data format:</td>
</tr>
<tr>
<td>• Uses WSDL and doesn’t have any other mechanism</td>
<td>• Different data format possible</td>
</tr>
<tr>
<td>to discover the service.</td>
<td>• E.g. Plain Text, HTML, XML and JSON.</td>
</tr>
</tbody>
</table>

Contrast Between SOAP & REST (/4)

• REST tries to isolate complexity at endpoints (Clients & Service):
  — Service:
    • Could need logic/computation to process XML to maintain Resources & generate their representation.
  — Client:
    • May have to process XML to extract info from XML representation.
• But this complexity is kept from the transport level.
• SOAP complicates the transport level as a SOAP message is encapsulated as transport message body.
More on Resources in REST

- Resources have certain properties:
  - *State*: i.e. they are mutable.
- Note:
  - In a RESTful request on it, resource itself stays service-side.
  - If request succeeds, requester gets resource’s *representation* (this transfers from server to requester machine).
  - For successful request to read resource, it’s typed *representation* (e.g. `text/xml`) transfers from resource’s server to the requester.

Roy Fielding’s Principles of REST

1. The web has **addressable resources** each with a URI.
2. The web has a **uniform and constrained interface**.
   - HTTP is synchronous request/response network protocol
   - Has a small number of methods.
   - Use these to manipulate resources.
3. Web is **representation oriented** – providing diverse formats.
4. The web may be used to **communicate statelessly** – providing scalability
5. HATEOAS: Hypermedia is used as the engine of application state.
Principles of REST 1: Addressability

scheme://host:port/path?queryString#fragment

- The **scheme** need not be HTTP. May be FTP or HTTPS.
- The **host** field may be a DNS name or a IP address.
- The **port** may be derived from the scheme. Using HTTP implies port 80.
- The **path** is a set of text segments delimited by the “/”.
- The **queryString** is a list of parameters represented as name=value pairs with each delimited by an “&”.
- The **fragment** is used to point to a particular place in a document.

REST Principles 2: Uniform Constrained Interface

- Small number of HTTP Operations:
  - No need for IDL
  - Interoperability
CA4006 Lecture Notes (Martin Crane 2017) 65

**REST Principles 2: Uniform Constrained Interface (2)**

**REST Response…**

is a representation of a resource. It could have several representations (e.g. XML, JSON, text, etc.)

![Diagram](image)

... contains metadata in the Header:

- Status Code
- Message length
- Date
- Content Type
- Etc.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
</tr>
<tr>
<td>201</td>
<td>Created</td>
</tr>
<tr>
<td>204</td>
<td>No Content</td>
</tr>
<tr>
<td>302</td>
<td>Found</td>
</tr>
<tr>
<td>304</td>
<td>Not Found</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
</tr>
<tr>
<td>415</td>
<td>Unsupported Media Type</td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
</tr>
</tbody>
</table>

**Status Codes**

5 classes of codes:

- 1xx: Informational code
- 2xx: Success code
- 3xx: Redirection code
- 4xx: Client Error code
- 5xx: Server Error code

---

**Principles of REST 3: Representation-Orientated**

- Representations of resources are exchanged.
  - GET returns a representation.
  - PUT & POST sends representations to server so underlying resources may change.
- Representations may be in many formats: XML, JSON, etc.
- HTTP uses CONTENT-TYPE header to specify message format the server is sending.
- The value of the CONTENT-TYPE is a MIME typed string.
- Examples:
  - text/plain
  - text/html
Principles of REST 4: Communicate Statelessly

- The application may have state but there is no client session data stored on the server.
- Server only records & manages state of resources it exposes.
- Any session-specific data is held & maintained by the client for sending to server with each request as needed.
- Server is easier to scale. No replication of session data concerns.
  - Client sessions only kept server-side due to browser limitations
  - Around 2008 browsers got powerful enough to maintain their own session state=>fat clients possible

Principles of REST 5: HATEAOSS

- Final REST principle is idea of using Hypermedia As The Engine Of Application State (HATEOAS).
- Hypermedia is document-centric approach with added support to insert links to other services & info in that document format.
- REST client doesn’t need any prior info on interacting with any application or server except understanding of hypermedia.
- REST client enters REST application thro simple fixed URL.
- All future actions client takes discoverable in resource representations returned from the server.
- Provide further guidance in the response!!!
Principles of REST 5: HATEAOS (/2)

A Subtlety: Opacity of URIs

• A URI is meant to be opaque
  – Means that URI: http://bedrock/citizens/fred has no inherent connection to the URI: http://bedrock/citizens/
  – Although Fred happens to be a citizen of Bedrock.
  – Of course, good designers devise URIs akin to what they identify, but URIs have no intrinsic hierarchical structure.

• A Note of caution
  – URI syntax resembles that for file system navigation, but this can mislead:
  – URIs are opaque identifiers, each naming exactly one resource.
A User Interface Client on a Web Service

• Example
  – Note: password is user hash from registration with Bibsonomy.com.

A User Interface Client on a Web Service (/2)

• Example
  – The bookmark results of the previous `Get` operation.
A User Interface Client on a Web Service (/3)

- Example
  - RestClient uses **Post** to add a Bookmark to Bibsonomy.com.
  - Nb: Change content-type to application/xml & charset to **UTF-8**.

A User Interface Client on a Web Service (/4)

- Example: The bookmark results of the previous **Post** operation.
  - Success!
  - Take note of hash – needed later!
A User Interface Client on a Web Service (/5)

- Example: RestClient uses `Put` to change a Bookmark thus
  
  http://www.bibsonomy.org/api/users/martycrane/posts/hash

A User Interface Client on a Web Service (/6)

- Example: The bookmark results of the previous `Put` operation.
A User Interface Client on a Web Service (/7)

- Example: RestClient uses Delete to remove a Bookmark thus
  http://www.bibsonomy.org/api/users/martycrane/posts/hash

A User Interface Client on a Web Service (/8)

- Example: The bookmark results of the previous Delete operation.
A JAX-RS REST Example: Customer Class

```java
class Customer {
    private int id;
    private String firstName;
    private String lastName;
    private String street;
    private String city;
    private String state;
    private String zip;

    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 int getId() { return id; }
    public void setId(int id) {
        this.id = id;
    }
}
```

CustomerResource Class

```java
@Path("/customers")
public class CustomerResource {
    private AtomicInteger idCounter = new AtomicInteger();

    @GET
    public Customer getCustomer(@PathParam("id") int id) {
        final Customer cust = customerDB.get(id);
        if (cust == null) {
            throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
        return cust;
    }

    @PUT
    public void updateCustomer(@PathParam("id") int id, Customer cust) {
        if (cust == null) throw new WebApplicationException(Response.Status.NOT_FOUND);
        Customer update = readCustomer(id);
        Customer curr = customerDB.get(id);
        if (curr == null) throw new WebApplicationException(Response.Status.NOT_FOUND);
        curr.setFirstName(update.getFirstName());
        curr.setLastName(update.getLastName());
        curr.setStreet(update.getStreet());
        curr.setState(update.getState());
        curr.setZip(update.getZip());
        curr.setCountry(update.getCountry());
        update = curr;
        customerDB.put(id, update);
    }
}
```
Writing a Client MyClient Class

package com.restfully.shop.test;

import org.junit.Test;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.Response;

public class MyClient {

    public static void main(String[] args) throws Exception {
        Client client = ClientBuilder.newClient().build();
        String xml = "<customers>
        <first-name>Bill</first-name>
        <last-name>Burke</last-name>
        <street>256 Clarendon Street</street>
        <city>Boston</city>
        <state>MA</state>
        <zip>02115</zip>
        <country>USA</country>
        </customers>";

        Response response = client.target("http://localhost:8080/services/customers")
                .request(javax.ws.rs.core.MediaType.APPLICATION_XML).post(Entity.entity(xml, javax.ws.rs.core.MediaType.APPLICATION_XML));

    }

}
package com.restfully.shop.services;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

/* services can be either singletons or on a per-request model: former is where one and only one
Java object services HTTP requests; latter is Java object is created to process each incoming
request and is thrown away at the end of that request. We use the former */

@ApplicationPath("/services") /* specs relative base URL path for all JAX-RS services */
public class ShoppingApplication extends Application {

    private Set<Object> singletons = new HashSet<Object>();

    public ShoppingApplication() {singletons.add(new CustomerResource());}

    /* ShopAppgetSingletons() returns Set initialized in constructor & CustomerResource instance. */
    @Override
    public Set<Object> getSingletons() {return singletons;}

    /* WAR file distributes JavaServer Pages, Java classes, other resources of web application. */
    static content
    WEB-INF/ /* WEB-INF dir contains a file named Web.xml defining web application structure */
    Web.xml
    classes/
    com/restfully/shop/domain/
    Customer.class
    com/restfully/shop/services/
    CustomerResource.class
    ShoppingApplication.class

Lecture 7: Distributed Obj & Web-based Systems