LECTURE 7: DISTRIBUTED OBJECT- & WEB-BASED SYSTEMS

SECTION 7.1: DISTRIBUTED OBJECT-BASED SYSTEMS
Distributed Objects

• **Introduction**
  – In *distributed object-based* systems, notion of an object plays a key role in establishing 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 because it is relatively 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/manipulate object’s state by invoking methods made available to it 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**
  – This separation between interfaces and the objects implementing them is crucial for distributed systems.
  – Separation allows for placing interface at one machine, with object itself on another machine.
  – This organization is commonly referred to as a *distributed object definition*.
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
  - *Server skeleton* handles (un)marshaling and object invocation

- **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 from a server: if a server exits, the object’s state and 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.
      Example: execute SQL query, return result to caller.
    - *Stateful session bean*: Transient object, but maintains client-related state until session end.
      Example: shopping cart.
    - *Entity bean*: Persistent, stateful object, can be invoked during different sessions.
      Example: 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
    ```java
    User_object *obj_ref;  // Declare a systemwide object reference
    obj_ref = ...;         // Initialize this reference to a distrib. obj
    obj_ref->do_something(); // Implicitly bind and invoke a method
    ```
  - 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 invoke methods on an object running in another Java VM.
  - RMI gives applications transparent, lightweight access to 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 `java.rmi` & `java.rmi.server`
  - An application that uses RMI has 3 components:
    - an interface that declares headers for remote methods;
    - a server class that implements the interface; and
    - one or more clients that call the remote methods.

Distributed Objects (/8)

- A Java RMI application needs to do the following:
  - Locate remote objects: Applications can use one of two mechanisms to obtain references to remote objects:
    - An application can register its remote objects with RMI’s simple naming facility, the rmiregistry, or
    - The application can pass and return remote object references as part of its normal operation.
  - Communicate with remote objects:
    - Details of communication between remote objects are handled by RMI;
    - To coder, remote communication looks like standard Java method invocation.
  - Load class bytecodes for objects that are passed around:
    - As RMI allows caller to pass objects to remote objects, RMI provides necessary mechanisms to load an object’s code and transmit its data.
Distributed Objects (/9)

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

![RMI Architecture Diagram](image)

---

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* unmarshals 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
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

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 (nb. have now created copy of original object)
    – Object-by-value passing tends to introduce nasty problems
Distributed Objects (/11)

- **RMI Parameter Passing**
  
  - **Note:** Systemwide 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

```
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();

            // 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:
   ```
   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 7.1: 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 designed for human-to-machine comms, is used for M2M chatter, e.g. in XML and JSON.**
- **In practice, WS typically provides OO web-based interface to a DB server, used by another web server, or a mobile application, providing user interface to end user.**
- **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 this was extended to include**
  - REST-compliant Web services, in which main purpose of the service is to manipulate representations of Web resources using a uniform set of stateless operations
  - Arbitrary Web services, in which service may expose an arbitrary set of operations

*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.
  - The server is internally organised more or less according to the 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 and availability, WWW servers are often clustered in a way that is transparent to clients.
  - Below a server cluster is used in combination with a front end to implement a Web service.

![Server Cluster Diagram](image1.png)

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 one of the servers, taking some performance metric into account (e.g., load balancing).
    2. **Content-aware distribution:**
       - Front end reads the content of HTTP request and selects best server.

![TCP Handoff Diagram](image2.png)
Background to Web Services (/4)

- **Naming: The Naming Service**
  - Names play a very important role in all computer systems.
  - Used to share resources, uniquely identify entities, refer to locations...
  - An important issue with naming is that a name can be resolved to the entity it refers to, aka **Name resolution**
  - To resolve names, need to implement a **Naming System**.
  - Difference between naming in distributed systems & non-distributed systems lies in their implementation.
  - As we have seen with Chord, for instance, in a distributed system, naming system impln is itself often distributed.
  - How this distribution is done plays a key role in the efficiency and scalability of the naming system.

Background to Web Services (/5)

- **Naming: Names in General**
  - **Name in distributed systems**: string of bits/characters used to refer to it.
  - An entity in a distributed system can be practically anything (e.g. resources such as hosts, printers, disks, and files).
  - Other examples of explicitly entities named are processes, users, mailboxes, Web pages, messages, network connections.
  - Entities can be operated on (e.g., a printer resource offers an interface with operations for printing a document, and others).
  - An entity such as a network connection may provide data send/ receive operations, setting quality-of-service parameters etc.
  - To operate on an entity need an **Access Point**, another kind of entity in a distributed system.
  - 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 an access point of a person, with phone number as address.
  — Indeed, many people nowadays have many phone numbers, corresponding to a point where they can be reached.
  — In a distributed system, a typical example of an access point is a host running a specific server.
  — Here its address is formed by e.g., an IP address+port number (i.e., the server’s transport-level address).
  — Entities may change access points over course time.
  — If a laptop moves to another location, its often assigned a different IP address than previously.
  — In a similar fashion, changing jobs or Internet Service Providers, 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

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:
  - **Directory Service**: Stores service descriptions.
    - 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 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**
Web Services: SOAP-Based (/5)

- 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** program 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`

```java
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.
 * @WebService
 * @SOAPBinding(style = Style.RPC)
 * // more on this later
 * @WebService
 * @SOAPBinding(style = Style.RPC) // more on this later
 * public interface TimeServer {
 *     @WebMethod String getTimeAsString();
 *     @WebMethod long getTimeAsElapsed();
 *     // These methods can be call 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());
    }
}
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`
      `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

```
#!/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.
p = "Current time is: \{result1\}"
p.puts p
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 "\verb+\n+Current time is: ",
$service->getTimeAsString();
print "\verb+\n+Elapsed milliseconds from the epoch: ",
$service->getTimeAsElapsed(), "\verb+\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
<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"
  <soap:Body>
    <tns:getTimeAsString xsi:nil="true" />
  </soap: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"
><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

• HTTP/1.1 200 OK signals all processed normally

Lecture 7: Distributed Obj & Web-based Systems  CA4006 Lecture Notes (Martin Crane 2015)  45

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 the transport layer (e.g. HTTP) Where do I find the service?
    • A service may offer several ports, i.e. ways to call it

Lecture 7: Distributed Obj & Web-based Systems  CA4006 Lecture Notes (Martin Crane 2015)  46
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

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

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

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

• After TimeServerPublisher generated WSDL, execute:
  – 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 `wagen -cp [Compiled Java Code]` achieves this.
- Run the publisher 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**

---

TimeServer (/12): How to pick a tool?

- The following lists the process to create a web service starting from Java sources, classes, or a WSDL file (server side):
  - Starting from Java classes use **Code-First**:
    - Use `wagen` 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 a Web Service operation,
  - `@SOAPBinding` specifies WS mapping onto SOAP message protocol
  - `@WebParam` maps a parameter to a WS message part & XML element,
  - `@WebResult` specifies that operation result in generated WSDL is something other than default return e.g. `IntegerOutput`.

Introduction to REST

- REST, or Representational State Transfer, is a distributed communication architecture
- It is fast becoming the lingua franca for Cloud Computing
- The central abstraction in REST is the *Resource*.
- A resource in the RESTful sense is anything that has an URI.
- In practice, a resource is an informational item that has hyperlinks to it.
Contrast Between SOAP & REST

• REST & SOAP are quite different
  – SOAP is a messaging protocol, REST is a architectural style.
  – SOAP uses WSDL for comms, REST exchanges data in XML/JSON
  – SOAP calls services by RPCs, REST invokes them via URL path.
  – SOAP can be over other protocols, REST is only over HTTP.
  – HTTP lighter, human–parsable data–exchange format than XML

Contrast Between SOAP & REST (/2)

• 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 a successful request to read a resource, it’s typed representation (e.g. text/html) transfers from resource’s server to the requester.

HTTP (REST) Operations

- **Essence**: Communication in Web generally based on HTTP.
- **Some HTTP Operations and their effects is shown below:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Idempotent?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>Yes</td>
<td>Request to return the header of a document</td>
</tr>
<tr>
<td>Get</td>
<td>Yes</td>
<td>Request to return a document to the client</td>
</tr>
<tr>
<td>Put</td>
<td>Yes</td>
<td>Request to store a document</td>
</tr>
<tr>
<td>Post</td>
<td>No</td>
<td>Provide data that are to be added to a document (collection)</td>
</tr>
<tr>
<td>Delete</td>
<td>Yes</td>
<td>Request to delete a document</td>
</tr>
</tbody>
</table>
Example in REST

• Shown below is client communications with a resource & typed responses it gets:
  – GET request could return my biography (html or video)
  – Typical html resource representations have other resource links
  – Could then be target of HTTP requests with appropriate CRUD verbs

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 Ruby Client on a Web Service

- Using Ruby’s Rest **`Client.gem`** with CrunchBase REST interface
  - Here use Rest **`Client.gem`** (e.g. with interactive Ruby shell **`irb`**).
  - URI is a CrunchBase request given to Rest Client’s **`Get`** method.
  - If you want, emit the resp.body, (& see all JSON data returned).
  - Can use **`JSON.parse`** method to parse response into a Ruby object structure.
  - Lastly, extract desired value (‘number of employees=50’ here).

```ruby
#!/usr/bin/env ruby
require 'rest_client' # don't need as rest_client.gem has all these
#require 'json' # don't need as rest_client.gem has all these
class Crunchery
  @record=nil
  def initialize (name="IBM")
    @name=name
  end
  def print_data (company="IBM")
    # default value is "IBM"
    base_url="http://api.crunchbase.com"
    key="6ytgczfnsrbeganbj2fvm5hw" # my key; you have to get your own
    company="whatsapp" # the company
    url=#{base_url}/v/1/company/#{company}.js?api_key=#{key}
    puts url # echo url variable to screen
    resp=RestClient.get url
    @record=JSON.parse(resp) # output whatapp's roll no.
    puts @record['number_of_employees']
  end
end
if __FILE__ == $0
  mg = Crunchery.new("whatsapp")
  mg.print_data
  puts "end reached"
end
```

C:\Ruby193>ruby Crunchery.rb
http://api.crunchbase.com/v/1/company/whatsapp.js?api_key=6ytgczfnsrbeganbj2fvm5hw
50
end reached
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.
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
  
  Use of hash to alter/delete

---

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.