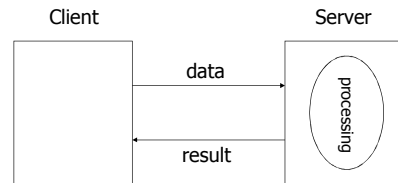


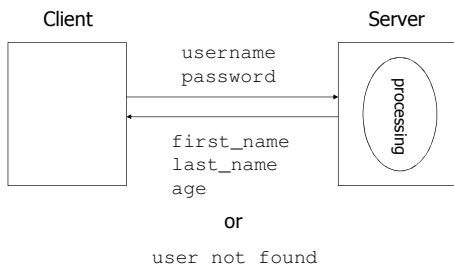
CS520 Web Programming
Introduction to Web Services

Chengyu Sun
California State University, Los Angeles

Client-Server Architecture



Client-Server Example

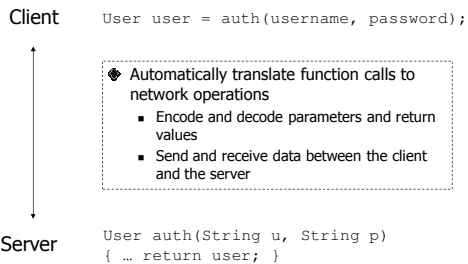


Socket Programming – Client

```
create socket
write string to socket
write string to socket
read string from socket
if( "user not found" ) return null;
else
    return new User (
        read string from socket
        read string from socket
        read integer from socket
    )
close socket
```

- ◆ Tedious networking code
- ◆ Application specific data exchange protocols

Client-Server Interaction as Function Calls



RPC and RMI

- ◆ Remote Procedure Call (RPC)
 - C
- ◆ Remote Method Invocation (RMI)
 - Java

RMI – Server

- ◆ Create a service interface
 - Remote interface
 - Declares the methods to be remotely invoked
- ◆ Create a service implementation
 - Remote object
 - Implements the methods to be remotely invoked
- ◆ Register the service with a RMI registry so a client can find and use this service

RMI – Client

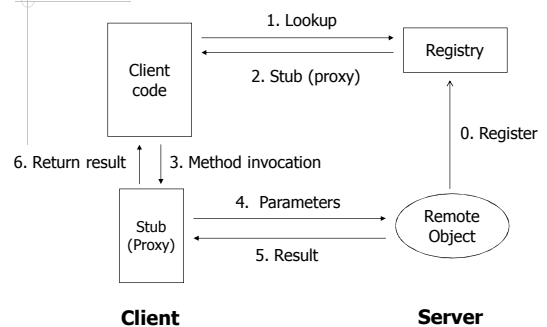
- ◆ Connect to the RMI registry
- ◆ Look up the service by name
- ◆ Invoke the service

RMI Example: AuthService

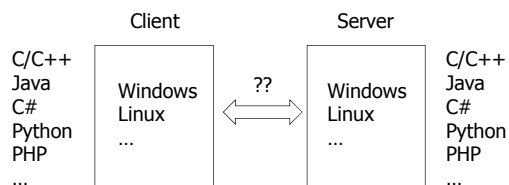
- ◆ Shared by both server and client
 - AuthService
 - User
- ◆ Server
 - AuthServiceImpl
 - AuthServiceStartup
- ◆ Client
 - AuthServiceClient

*Why does User have to implement the Serializable interface?
What exactly does registry.lookup() return?*

How RMI Works



Cross Platform RPC

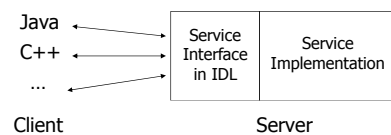


- ◆ The client and the server use different languages and/or platforms

How do we define service interface??

CORBA

- ◆ Common Object Request Broker Architecture
- ◆ Use Interface Definition Language (IDL) to describe service interface
- ◆ Provide mappings from IDL to other languages such as Java, C++, and so on.



IDL Example

```
module bank {  
  interface BankAccount {  
    exception ACCOUNT_ERROR { long errcode; string message;};  
  
    long querybalance(in long acnum) raises (ACCOUNT_ERROR);  
    string queryname(in long acnum) raises (ACCOUNT_ERROR);  
    string queryaddress(in long acnum) raises (ACCOUNT_ERROR);  
  
    void setbalance(in long acnum, in long balance) raises (ACCOUNT_ERROR);  
    void setaddress(in long acnum, in string address) raises (ACCOUNT_ERROR);  
  };  
};
```

(Traditional) Web Services

- ◆ RPC over HTTP
 - Client and server communicate using HTTP requests and responses
- ◆ Many different web service protocols
 - Language support: single language vs. language independent
 - Message encoding: binary vs. text
- ◆ Most widely used: *SOAP*

Web Service Example: HashService

- ◆ HashService
 - @WebService
 - @WebMethod
- ◆ web.xml
- ◆ sun-jaxws.xml
 - <endpoint>

Metro

- ◆ <http://metro.java.net/>
- ◆ A Java web service library backed by SUN/Oracle
- ◆ Implementation of the latest Java web service specifications
- ◆ Guaranteed interoperability with .NET Windows Communication Foundation (WCF) web services
- ◆ Easy to use

Other Java Web Service Libraries

- ◆ Apache Axis2
 - <http://axis.apache.org/axis2/java/core/>
- ◆ Apache CXF
 - <http://cxf.apache.org/>

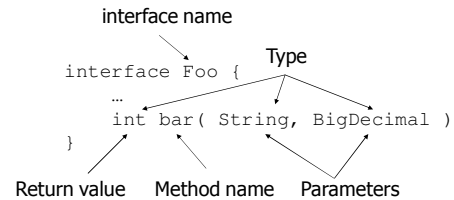
WSDL

- ◆ A language for describing web services
 - Where the service is
 - What the service does
 - How to invoke the operations of the service
- ◆ Plays a role similar to IDL in CORBA

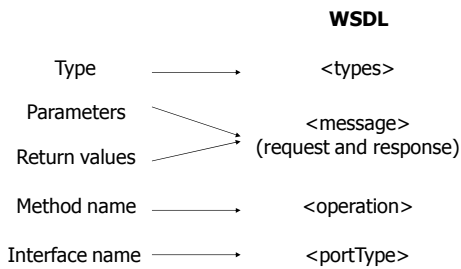
Sample WSDL Documents

- ◆ HashService - <http://localhost:8080/ws/hash?wsdl>
- ◆ Amazon ECS - <http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl>

How Do We Describe an API



How Do We Describe an Web Service API

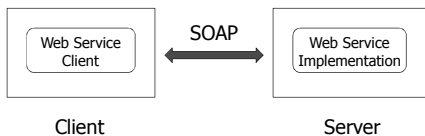


Web Service Example: Consume HashService

- ◆ Generate client side interface and stub from WSDL using Metro's `wsimport`
- ◆ Write client code

SOAP

- ◆ <http://www.w3.org/TR/soap/>
- ◆ Simple Object Access Protocol



A Sample SOAP Message

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:doSpellingSuggestion xmlns:ns1="urn:GoogleSearch"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <key xsi:type="xsd:string">0000000000000000000000000000000000</key>
      <phrase xsi:type="xsd:string">britney spears</phrase>
    </ns1:doSpellingSuggestion>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A Sample SOAP RPC Response

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:doSpellingSuggestionResponse xmlns:ns1="urn:GoogleSearch"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:string">britney spears</return>
    </ns1:doSpellingSuggestionResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A Sample Fault Response

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Client</faultcode>
      <faultstring>Client Error</faultstring>
      <detail>
        <m:dowJonesfaultdetails xmlns:m="DowJones">
          <message>Invalid Currency</message>
          <errorcode>1234</errorcode>
        </m:dowJonesfaultdetails>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Encoding

- ◆ <http://schemas.xmlsoap.org/encoding>
- ◆ Include all built-in data types of *XML Schema Part 2: Datatypes*
 - `xsi` and `xsd` name spaces

SOAP Encoding Examples

```
int a = 10;      <a xsi:type="xsd:int">10</a>
float x = 3.14159; <x xsi:type="xsd:float">3.14159</x>
String s = "SOAP"; <s xsi:type="xsd:string">SOAP</s>
```

Compound Values and Other Rules

```
<iArray xsi:type="SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[3]">
  <val>10</val>
  <val>20</val>
  <val>30</val>
</iArray>
```

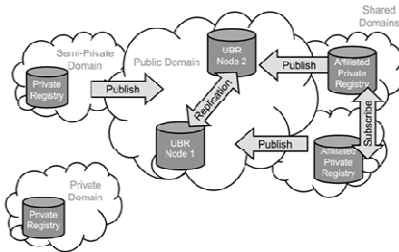
```
<Sample>
  <iVal xsi:type="xsd:int">10</iVal>
  <sVal xsi:type="xsd:string">Ten</sVal>
</Sample>
```

- ◆ References, default values, custom types, complex types, custom serialization ...

UDDI

- ◆ Universal Description Discovery and Integration
- ◆ A registry for web services
- ◆ A web API for publishing, retrieving, and managing information in the registry

UDDI Registries



Problems with SOAP Web Service

- ◆ Very complex
 - Based on some very complex specifications
 - Very difficult to create supporting libraries
 - Virtually impossible to use without supporting libraries
- ◆ Not very efficient



RESTful Web Services

A RESTful Web Service

Request

Get user with id=1: /service/user/1



XML Response

or

JSON Response

```
<user>
  <id>1</id>
  <firstName>John</firstName>
  <lastName>Doe</lastName>
  <email>jdoe1@localhost</email>
</user>
```

```
{
  "id": 1,
  "firstName": "John",
  "lastName": "Doe",
  "email": "jdoe1@localhost"
}
```

Is This Really A Web Service?

- ◆ Where is the method call?
- ◆ Why does it look like a web *application*?
- ◆ Why is it called *RESTful*?

Where Is The Method Call?

- ◆ Answer: does it have to be a method call?

HTTP request: `http://<host>/service/user/1`

`User user = getUser(1);`

HTTP response

The downside is that now it's the client's responsibility to turn an HTTP response into a "return value".

Why Does It Look Like A Web Application?

- ◆ Answer: it does, and it's a good thing.

Now all web technologies/languages can be used to create web services (and you don't have to implement complex specifications like SOAP).

Why Is It Called RESTful?

- ◆ Representational State Transfer
- ◆ Introduced by Roy Fielding in his Ph.D. dissertation on network-based software architecture
- ◆ Describes the common characteristics of *scalable, maintainable, and efficient* distributed software systems

The REST Constraints

- ◆ Client and server
- ◆ Stateless
- ◆ Support caching
- ◆ Uniformly accessible
- ◆ Layered
- ◆ (Optional) support code-on-demand

RESTful Web Services

- ◆ Web applications for *programs*
 - Generate responses in formats to be read by machines (i.e. XML and JSON) rather than by humans (i.e. HTML)
- ◆ Simulate how the static web (the largest REST system) works
 - Use URLs that look like static web pages
 - Utilize HTTP request methods and headers
 - *Stateless*

RESTful Web Service Example

- ◆ User Management
 - List
 - Get
 - Add
 - Update
 - Delete

Create a RESTful Web Service

- ◆ Identify resources and operations
- ◆ Determine resource representation, i.e. data exchange format between the service and the clients
- ◆ Design URL and request mapping
- ◆ Implement the operations

Resource Representation

- ◆ Data format should be easily "understandable" by all programming languages
- ◆ XML
 - Already widely in use as a platform independent data exchange format
 - XML parsers are readily available in many languages
- ◆ JSON
 - Much more concise than XML
 - Can be used directly in JavaScript

URL Design and Request Mapping Conventions (1)

◆ Operation: get a user

◆ URL

- /user/{id} or
- /user/get?id={id}

Path variable based design is usually preferred to request parameter based design.

URL Design and Request Mapping Conventions (2)

◆ Operation: get a user

◆ Choose which data format to use

◆ Solution:

- /user/{id}.{format}
- Check the Accept request header

Checking Accept header is preferred in theory, but the URL based solution is more convenient in practice, e.g. <https://dev.twitter.com/docs/api/1.1>

URL Design and Request Mapping Conventions (3)

◆ Map HTTP Request Methods to CRUD operations

- | | | |
|-----------------|----|------------|
| ■ POST (or PUT) | ←→ | ■ Create |
| ■ GET | ←→ | ■ Retrieve |
| ■ PUT (or POST) | ←→ | ■ Update |
| ■ DELETE | ←→ | ■ Delete |

Request Mapping Example

| Operation | HTTP Request |
|---------------|--|
| Get a user | GET /user/1 HTTP 1.1 |
| Delete a user | DELETE /user/1 HTTP 1.1 |
| Update a user | PUT /user/1 HTTP 1.1 { "id":1, "firstName":"John", "lastName":"Doe", "email":"jdoe@localhost" } |

Service Implementation – Know Your Libraries

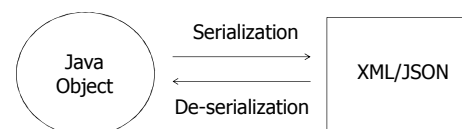
◆ Map HTTP requests to service operations

- Modern webapp framework like Spring
- Jersey - <https://jersey.java.net/>

◆ Convert between objects and XML/JSON

- Simple XML Serialization - <http://simple.sourceforge.net/>
- Jackson - <http://jackson.codehaus.org/>

Serialization and Deserialization



Service Implement Example: Simple XML Serialization

- ◆ Dependency
 - `org.simpleframework:simple-xml`
- ◆ Usage
 - Content type
 - `Serializer` and `Persister`

Service Implementation Example: Jackson

- ◆ Dependency
 - `com.fasterxml.jackson.core:jackson-databind`
- ◆ Additional view resolver
 - `BeanNameViewResolver`
- ◆ Additional view
 - `MappingJackson2JsonView`

Using Multiple View Resolvers in Spring

- ◆ View resolution order
 - Order of the resolver beans, or
 - Based on the `order` property of the beans
- ◆ `InternalResourceViewResolver` should always be the last

Access RESTful Web Service

- ◆ Apache HttpClient
 - <http://hc.apache.org/httpcomponents-client-ga/>
- ◆ HttpURLConnection
 - <http://developer.android.com/reference/java/net/URLConnection.html>
- ◆ Examples:
 - `XmlClient` and `JsonClient`
 - CSNSAA

Summary

- ◆ RPC and RMI
- ◆ CORBA
 - IDL
- ◆ SOAP, WSDL, UDDI
 - Create and consume SOAP web services using Metro
- ◆ RESTful web services

Further Readings

- ◆ *Java Web Services Up and Running* by Martin Kalin
- ◆ *RESTful Java Web Services* by Jose Sandoval
- ◆ *The Rise and Fall of CORBA* by Michi Henning