CS 683 Emerging Technologies
Spring Semester, 2003
Doc 14 IDL, RMI, CORBA, WSDL

Contents

References

Web Services Description Language (WSDL) 1.1 **http://www.w3.org/TR/wsdl**

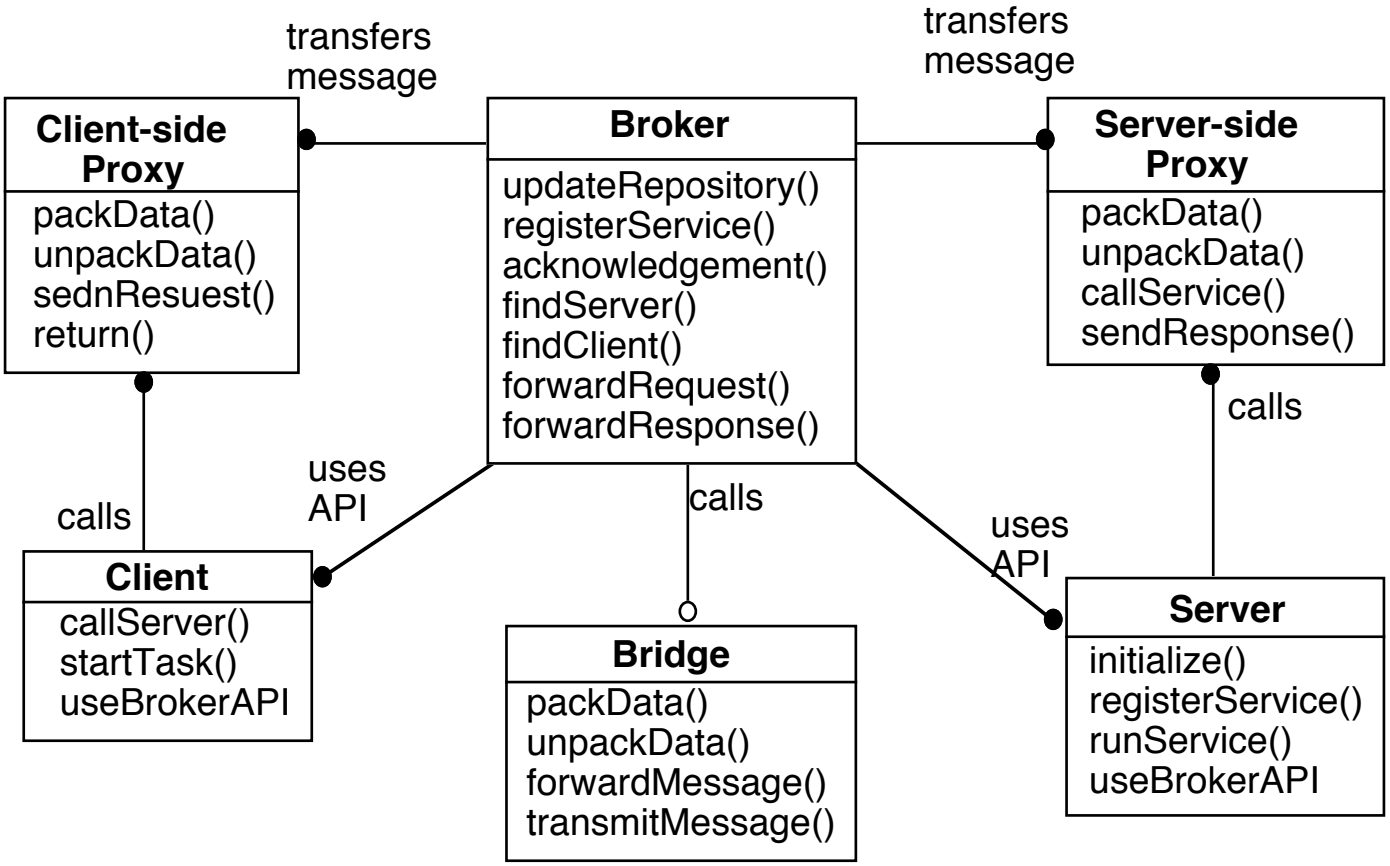
Building Web Services with Java, Graham et all, Chapter 6

CS 696 Emerging Technologies: Distributed Objects Spring Semester, 1998, Lecture notes Doc 4 and Doc 19.

Some Basic Questions

How do I find a service?
Where is the service?
What does the service do?
How does one interact with the service?

The Broker Pattern



A broker

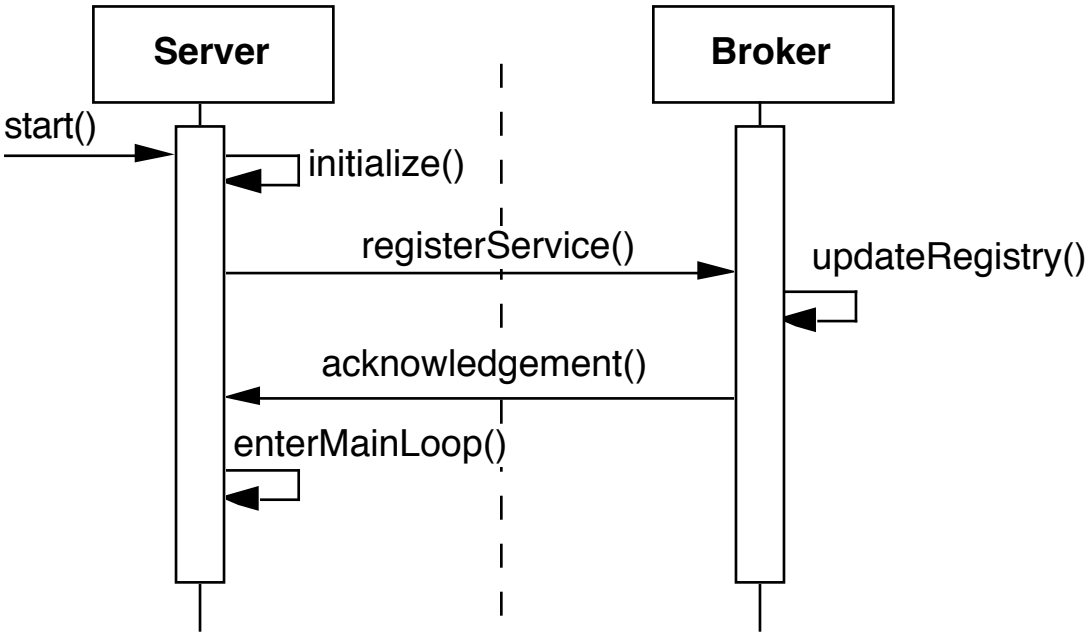
- Handles the transmission of requests from clients to servers
- Handles the transmission of responses from servers to clients
- Must have some means to identify and locate server
- If server is hosted by different broker, forwards the request to other broker
- If server is inactive, active the server
- Provides APIs for registering servers and invoking server methods

Bridge

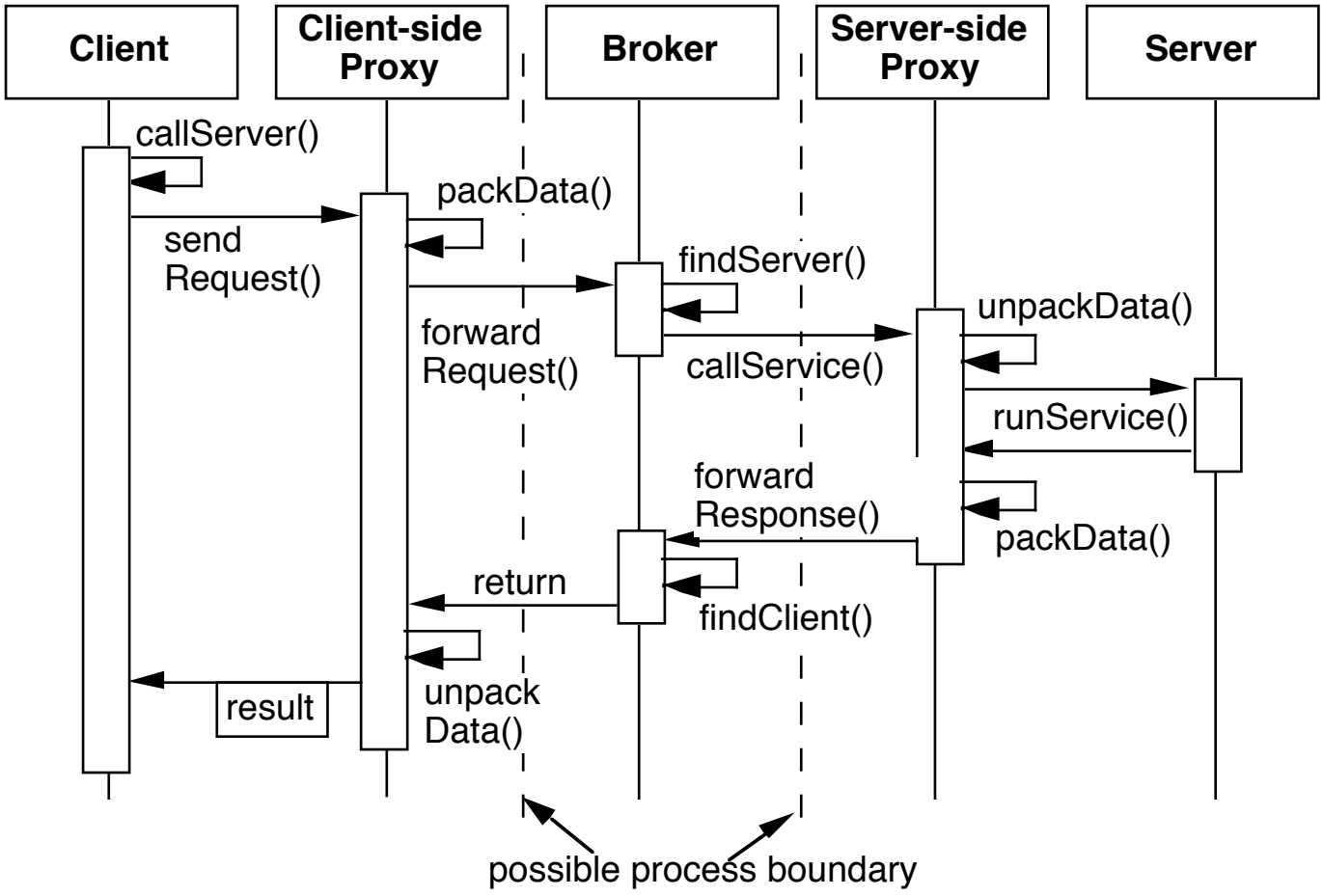
- Optional components used for hiding implementation details when two brokers interoperate

Dynamics

Registering Server



Client Server Interaction



Variants

Direct Communication Broker System

- Broker gives the client a communication channel to the server
- Client and server interact directly
- Many CORBA implementation use this variant

Message Passing Broker System

- Clients and servers pass messages rather than services (methods)

Trader System

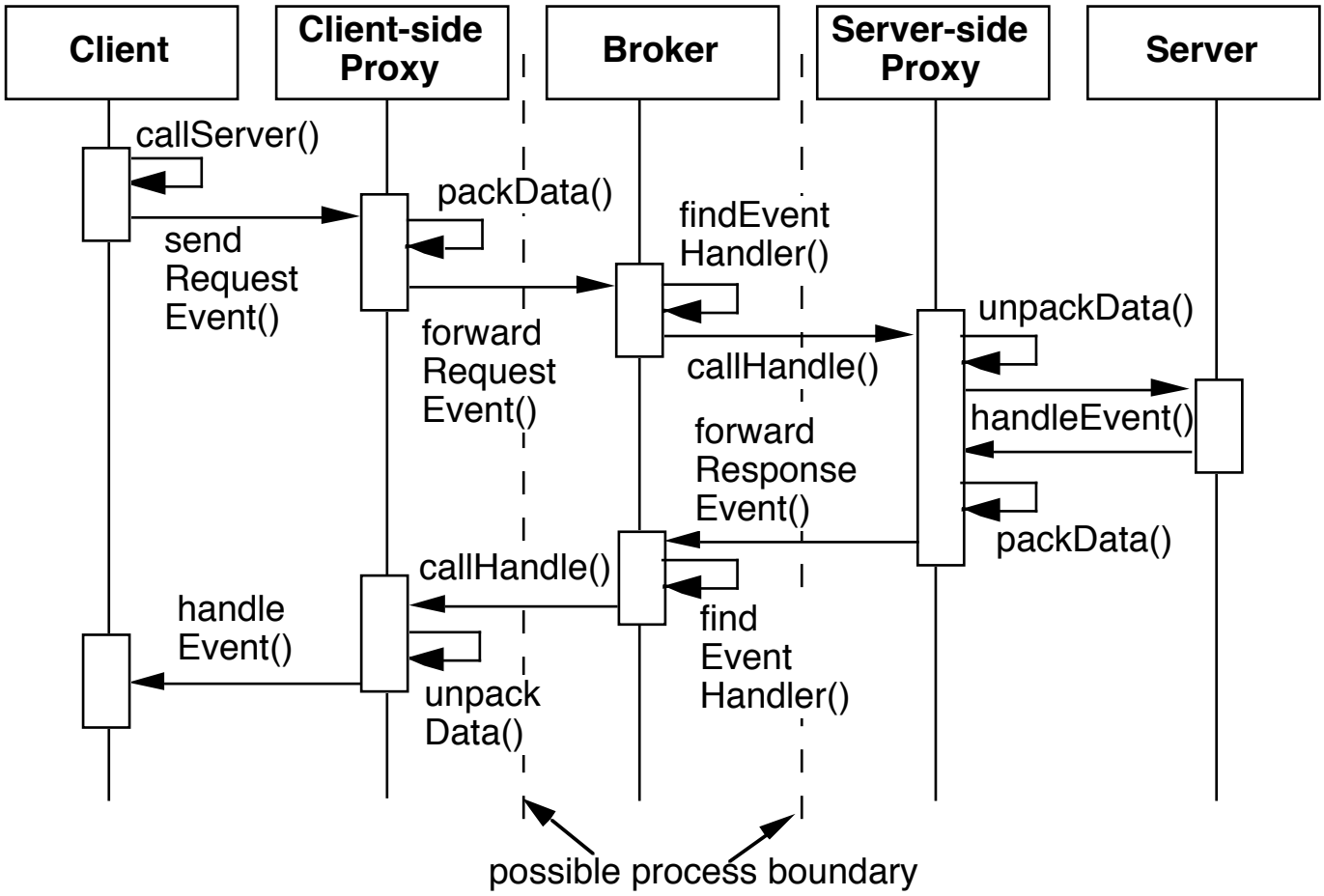
- Clients request a service, not a server
- Broker forwards the request to a server that provides the service

Adapter Broker System

- Hide the interface of the broker component to the servers using an additional layer
- The adapter layer is responsible for registering servers and interacting with servers
- For example if all server objects are on the same machine as application a special adapter could link the objects to the application directly

Callback Broker System

- Eliminate the difference between clients and servers
- When an event is registered with a broker, it calls the component that is registered to handle the event



Known Uses

CORBA

IBM's SOM/DSOM

Mircosoft OLE 2.x

RMI

Consequences
Benefits

Location Transparency

Clients (servers) do not care where servers (clients)are located

Changeability and extensibility of components

Changes to server implementations are transparent to clients if they don't change interfaces

Changes to internal broker implementation does not affect clients and servers

One can change communication mechanisms without changing client and server code

Portability of Broker System

Porting client & servers to a new system usually just requires recompiling the code

Benefits - Continued

Interoperability between different Broker System

Different broker systems may interoperate if they have a common protocol for the exchange of messages

DCOM and CORBA interoperate

DCOM and RMI interoperate

RMI and CORBA interoperate

Reusability

In building new clients you can reuse existing services

Liabilities

Restricted Efficiency

Lower fault tolerance compared to non-distributed software

Benefits and Liabilities

Testing and Debugging

A client application using tested services is easier to test than creating the software from scratch

Debugging a Broker system can be difficult

Some RMI

A First Program - Hello World

Modified from "Getting Started Using RMI"

The Remote Interface

```
public interface Hello extends java.rmi.Remote
   {
   String sayHello() throws java.rmi.RemoteException;
   }
```

The Server Implementation

```
// Required for Remote Implementation
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

// Used in method getUnixHostName
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class HelloServer
      extends UnicastRemoteObject
      implements Hello
   {

   public HelloServer() throws RemoteException
      {
      }

   // The actual remote sayHello
   public String sayHello() throws RemoteException
      {
      return  "Hello World from " + getUnixHostName();
      }
```

// Works only on UNIX machines

```
protected String getUnixHostName()
   {
   try
      {
      Process  hostName;
      BufferedReader  answer;

      hostName = Runtime.getRuntime().exec( "hostname" );
      answer = new BufferedReader(
                  new InputStreamReader(
                     hostName.getInputStream()) );

      hostName.waitFor();
      return answer.readLine().trim();
      }
   catch (Exception noName)
      {
      return "Nameless";
      }
   }
```

// Main that registers with Server with Registry

```
public static void main(String args[])
   {
   // Create and install a security manager
   System.setSecurityManager(new RMISecurityManager());

   try
      {
      HelloServer serverObject = new HelloServer ();

      Naming.rebind("//roswell.sdsu.edu/HelloServer",
                    serverObject );

      System.out.println("HelloServer bound in registry");

      }
   catch (Exception error)
      {
      System.out.println("HelloServer err: ");
      error.printStackTrace();
      }
   }
}
```

The Client Code

```
import java.rmi.*;
import java.net.MalformedURLException;

public class HelloClient
  {
  public static void main(String args[])
    {
    try {
      Hello remote = (Hello) Naming.lookup(
                        "//roswell.sdsu.edu/HelloServer");

      String message = remote.sayHello();
      System.out.println( message );
      }
    catch ( Exception error)
      {
      error.printStackTrace();
      }
    }
  }
```

Note the multiple catches are to illustrate which exceptions are thrown

Running The Example
Server Side

Step 1. Compile the source code

Server side needs interface Hello and class HelloServer

javac Hello.java HelloServer.java

Step 2. Generate Stubs and Skeletons (to be explained later)

The rmi compiler generates the stubs and skeletons

rmic HelloServer

This produces the files:

HelloServer_Skel.class
HelloServer_Stub.class

The Stub is used by the client and server
The Skel is used by the server

The normal command is:

rmic fullClassname

Step 3. Insure that the RMI Registry is running

For the default port number

```
rmiregistry &
```

For a specific port number

```
rmiregistry portNumber &
```

On a UNIX machine the rmiregistry will run in the background and will continue to run after you log out

This means you manually kill the rmiregistry

Step 4. Register the server object with the rmiregistry by running HelloServer.main()

```
java HelloServer &
```

Client Side

The client can be run on the same machine or a different machine than the server

Step 1. Compile the source code

Client side needs interface Hello and class HelloClient

javac Hello.java HelloClient.java

Step 2. Make the HelloServer_Stub.class is available

Either copy the file from the server machine

or

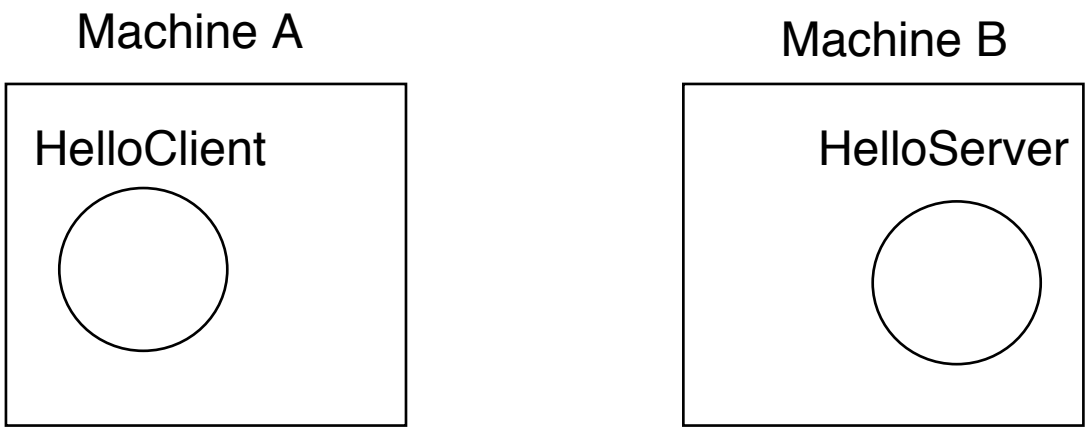
Compile HelloServer.java on client machine and rum rmic

Step 3. Run the client code

java HelloClient

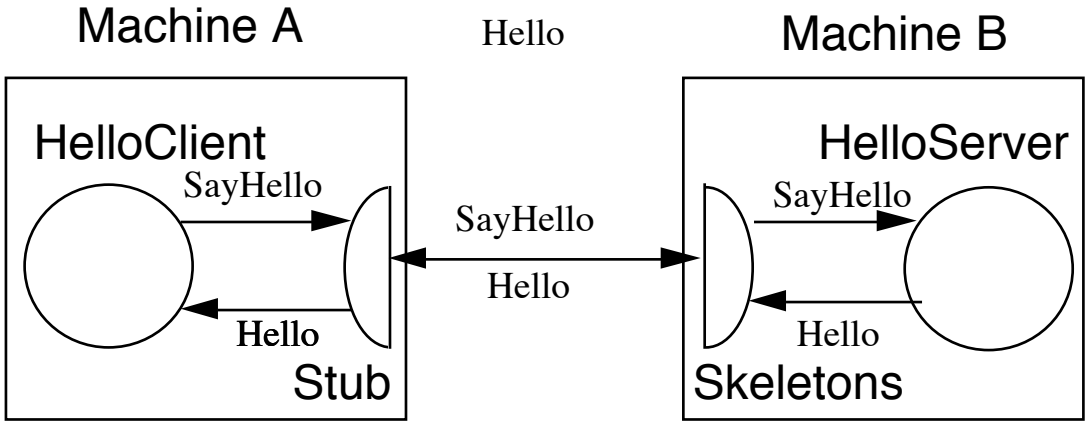
Proxies

How do HelloClient and HelloSever communicate?



Client talks to a Stub that relays the request to the server over a network

Server responds via a skeleton that relays the response to the Client



Some Corba

A Simple CORBA Example using OrbixWeb

Step 1 Create an interface for the server using the CORBA interface definition language (IDL)

Place the following code in the file Hello.idl

```
interface Hello
   {
   string sayHello();
   };
```

Step 2 Compile the IDL interface

```
idl Hello.idl
```

This creates a directory called java_output which contains:

Hello.java	HelloPackage/	_HelloSkeleton.java
HelloHelper.java	_HelloImplBase.java	_HelloStub.java
HelloHolder.java	_HelloOperations.java	_tie_Hello.java

IDL output

Hello
Client interface to server

_HelloStub
Client side proxy for server

_HelloSkeleton
Server side proxy

_HelloImplBase
Abstract class to use as parent class to server

HelloPackage
A Java package used to contain any IDL types nested in the Hello interface

HelloHelper
A Java class that allows IDL user-defined types to be manipulated in various ways

HelloHolder
Used for passing Hello objects as parameters

Step 3 Implementing the Server using Inheritance

Classes generated by OrbixWeb IDL compiler
Hello

```
public interface Hello
   extends org.omg.CORBA.Object
{
   public String sayHello() ;
   public java.lang.Object _deref() ;
}
```

_HelloImplBase

```
import IE.Iona.OrbixWeb._OrbixWeb;

public abstract class _HelloImplBase
   extends _HelloSkeleton
   implements Hello
{
   public _HelloImplBase() {
      org.omg.CORBA.ORB.init().connect(this);
   }

   public _HelloImplBase(String marker) {
      _OrbixWeb.ORB(org.omg.CORBA.ORB.init()).connect(this,marker);
   }

   public _HelloImplBase(IE.Iona.OrbixWeb.Features.LoaderClass loader) {
      _OrbixWeb.ORB(org.omg.CORBA.ORB.init()).connect(this,loader);
   }

   public _HelloImplBase(String marker,
                         IE.Iona.OrbixWeb.Features.LoaderClass loader) {
      _OrbixWeb.ORB(org.omg.CORBA.ORB.init()).connect(this,marker,loader);
   }

   public java.lang.Object _deref() {
      return this;
   }

}
```

Programmer Implemented Classes
HelloImplementation

```
public class HelloImplementation extends _HelloImplBase
   {
   public String sayHello()
      {
      return "Hello World";
      }
   }
```

HelloServer

```
import IE.Iona.OrbixWeb._CORBA;
import IE.Iona.OrbixWeb.CORBA.ORB;

public class HelloServer
   {
   public static void main (String args[])
      {
      org.omg.CORBA.ORB ord =
         org.omg.CORBA.ORB.init();

      try
         {
         Hello server = new HelloImplementation();
         _CORBA.Orbix.impl_is_ready( "HelloServer" );
         System.out.println("Server going Down");
         }
      catch ( org.omg.CORBA.SystemException corbaError)
         {
         System.out.println("Exception " + corbaError);
         }
      }
   }
```

Step 4 Compiling the Server

The classpath must include the following:

Java JDK classes
org.omg.CORBA package
IR.Iona.OrbixWeb package

Must compile the following classes:

Hello.java
_HelloSkeleton.java
_HelloImplBase.java
HelloImplementation.java
HelloServer.java

Step 5 Registering and running the Server

Make sure that the OrbixWeb daemon (orbixdj) is running on the server machine

You start the deamon by the command:

orbixdj -textConsole

Now register the server via:

putit HelloServer -java HelloServer

Now run the server via

java HelloServer

Note running the server is not normally required, however, since the server is not in a package it is hard to get the ORB to activate the server. We will address this issue later

Details of the above process will be discussed later

Step 6 Writing the client

HelloClient.java

```
import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.ORB;

public class HelloClient
  {
  public static void main(String args[])
    {
    ORB.init();

    String hostname = "eli.sdsu.edu";
    String serverLabel = ":HelloServer";
    Hello server = HelloHelper.bind( serverLabel, hostname);
    System.out.println( server.sayHello() );
    }
  }
```

Step 7 Compiling and Running the client

Compile the classes:

_HelloStub.java
HelloClient.java

Now run the client with the command:

java HelloClient

Output - Client Window

[New IIOP Connection (eli.sdsu.edu,IT_daemon, null,null,pid=0)]
[New IIOP Connection (eli.sdsu.edu,HelloServer, null,null,pid=0)]
Hello World

Output - Server Window

[HelloServer: New Connection (eli.sdsu.edu:59201)]
[HelloServer: End of IIOP connection (eli.sdsu.edu:59201)]

Output - Daemon Window

[IT_daemon: New Connection (eli.sdsu.edu:59200)]
[IT_daemon: End of IIOP connection (eli.sdsu.edu:59200)]

IDL

Interface Definition language

RMI

```
public interface Hello extends java.rmi.Remote
   {
   String sayHello() throws java.rmi.RemoteException;
   }
```

CORBA

```
interface Hello
   {
   string sayHello();
   };
```

WSDL

WSDL Elements

- Types
  Container for data type definitions

  XML schema defining types used

- Message
  Abstract, typed definition of the data being communicated

  Defines the set of parameters used in method signatures

- Operation
  Abstract description of an action supported by the service

  Set of messages

- Port Type
  Abstract set of operations supported by one or more endpoints

  Each child operation defines an abstract method signature

- Binding
  A concrete protocol and data format specification for a particular port type

  Details of how elements in portType are converted into concrete representation of data formats (XML) and protocols(http)

- Port
  A single endpoint defined as a combination of a binding and a network address

- Service
  A collection of related endpoints

Standard WSDL Namespaces

Namespace	URI	Definition
wsdl	http://schemas.xmlsoap.org/wsdl/	WSDL framework
soap	http://schemas.xmlsoap.org/wsdl/soap/	WSDL SOAP binding
http	http://schemas.xmlsoap.org/wsdl/http/	WSDL HTTP GET & POST binding
mime	http://schemas.xmlsoap.org/wsdl/mime/	WSDL MIME binding
soapenc	http://schemas.xmlsoap.org/soap/encoding/	SOAP 1.1Encoding
soapenv	http://schemas.xmlsoap.org/soap/envelope/	SOAP 1.1 Envelope
xsi	http://www.w3.org/2000/10/XMLSchema-instance	XSD Instance namespace
xsd	http://www.w3.org/2000/10/XMLSchema	Schema namespace
tns	Varies	This namespace

Example

```
<?xml version="1.0"?>
<definitions name="BabelFishService"
   xmlns:tns="http://www.xmethods.net/sd/BabelFishService.wsdl"
   targetNamespace="http://www.xmethods.net/sd/BabelFishService.wsdl"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
   <message name="BabelFishRequest">
      <part name="translationmode" type="xsd:string"/>
      <part name="sourcedata" type="xsd:string"/>
   </message>
   <message name="BabelFishResponse">
      <part name="return" type="xsd:string"/>
   </message>
   <portType name="BabelFishPortType">
      <operation name="BabelFish">
         <input message="tns:BabelFishRequest" name="BabelFish"/>
         <output message="tns:BabelFishResponse"
            name="BabelFishResponse"/>
      </operation>
   </portType>
   <binding name="BabelFishBinding" type="tns:BabelFishPortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="BabelFish">
         <soap:operation soapAction="urn:xmethodsBabelFish#BabelFish"/>
         <input>
            <soap:body use="encoded" namespace="urn:xmethodsBabelFish"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
         </input>
```

```
        <output>
          <soap:body use="encoded" namespace="urn:xmethodsBabelFish"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </output>
      </operation>
    </binding>
    <service name="BabelFish">
      <documentation>
          Translates text of up to 5k in length, between a variety of languages.
      </documentation>
      <port name="BabelFishPort" binding="tns:BabelFishBinding">
        <soap:address
          location="http://services.xmethods.net:80/perl/soaplite.cgi"/>
      </port>
    </service>
</definitions>
```