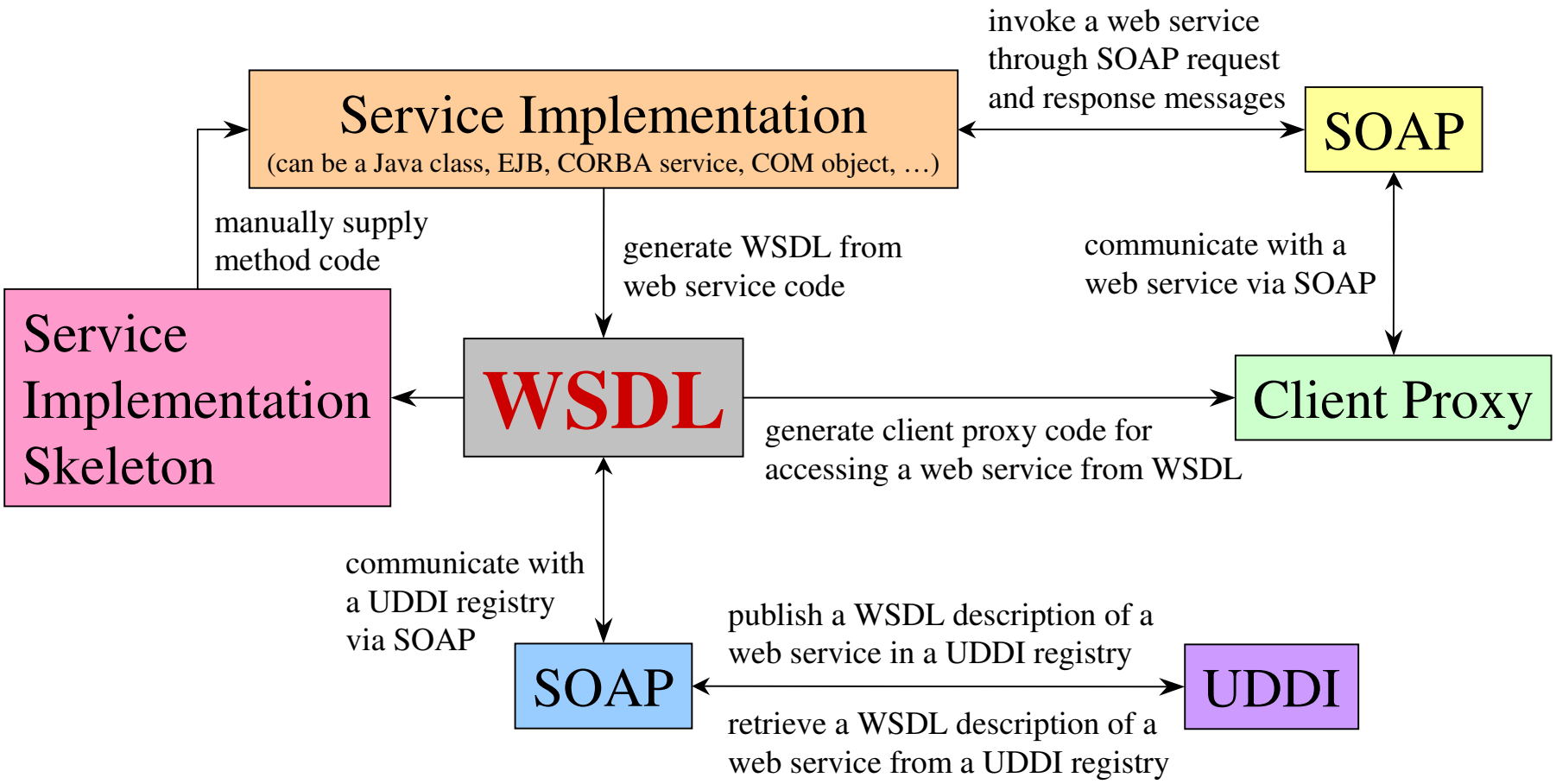
an open source web service toolkit for Java

Mark Volkmann

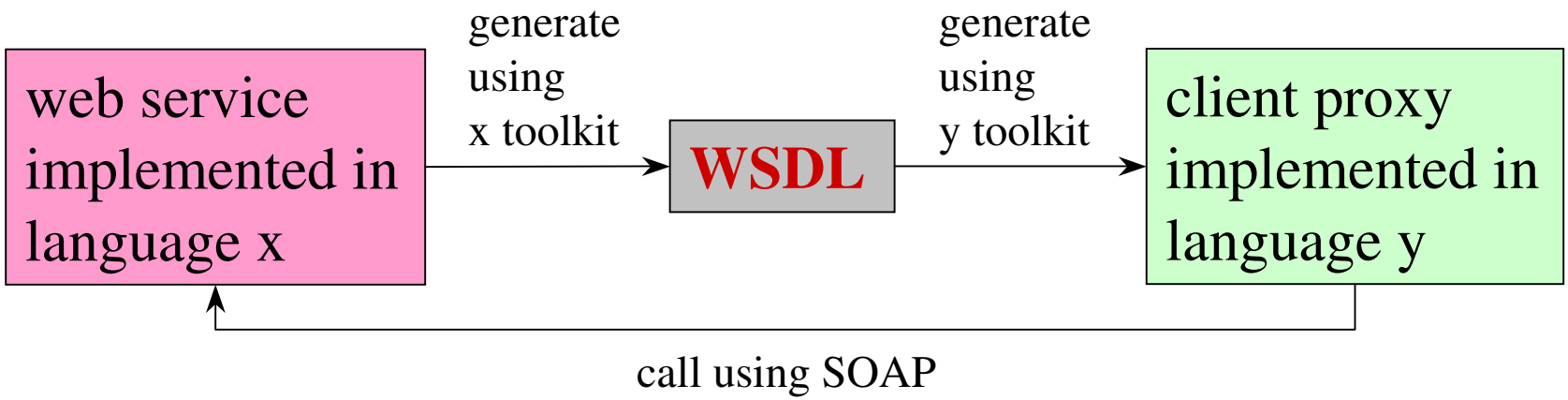
Object Computing, Inc.

General Web Service Toolkit Functionality



Future Made Possible By WSDL

- Will be able to easily make calls between any programming languages that can
  - generate WSDL from web services implemented in the language
  - generate client proxies in the language from WSDL



Apache eXtensible Interaction System (Axis)

- An open source web service toolkit for Java
  - supercedes Apache SOAP
- Incredibly flexible
  - as will be seen when its architecture is reviewed later
- JAX-RPC support
  - moving toward full implementation
  - first open source JAX-RPC implementation
  - only free alternative to Sun's JAX-RPC reference implementation
- Download from http://xml.apache.org/axis
  - current version is release candidate 1
  - can also download nightly builds to get the latest features and bug fixes

Most Active Contributors

- IBM
  - Russell Butek, Doug Davis, Glyn Normington, Sam Ruby, Richard Scheuerle, James Snell
- Macromedia
  - Glen Daniels, Tom Jordahl
- Unrealities
  - Rob Jelinghaus
- Computer Associates
  - Davanum Srinivas

Axis Features

- SOAP support
  - full support of SOAP 1.1
  - some support of SOAP 1.2
    - will eventually support all of it
  - some support for
    - SOAP with Attachments
    - Direct Internet Message Encapsulation (DIME) for binary XML
- UDDI support
  - none
  - can use UDDI4J from IBM developerWorks
- EJB support
  - can access session bean methods as web services
    - see org.apache.axis.providers.java.EJBProvider class

Axis Features (Cont'd)

1. SOAP message monitoring
   - TCP Monitor tool (tcpmon) monitors SOAP request/response messages
   - can use with other SOAP toolkits as well
2. Dynamic invocation
   - doesn't use WSDL
   - JAX-RPC Call class
     - invokes a web service operation
     - need to supply SOAP router URL, service namespace, operation name and parameters
     - no compile-time parameter type checking (pass array of Objects)

Axis Features (Cont'd)

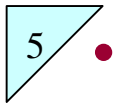
- Web service deployment
  - instant deployment (JWS)
    - simply copy a .java file to the "axis" web app. directory and change the extension to .jws (for Java Web Service)
  - custom deployment using a deployment descriptor
    - uses Web Service Deployment Descriptor (WSDD) XML syntax
      - specific to Axis
    - allows more control for custom type mappings, deployment without source code and more
  - standalone HTTP server
    - a weak alternative to servlet-based servers such as Tomcat
  - self-contained web app.
    - can add axis.jar and its helper jars to any WAR file to add web services to a web app. that will run in any servlet engine

Axis Features (Cont'd)

4

- WSDL support
  - Java2WSDL tool
    - generates WSDL from Java service implementation classes
  - WSDL2Java tool
    - generates client stubs for type-safe invocations
    - generates service skeletons for implementing services described by WSDL
    - generates other necessary server-side files (more on these later)
  - automatically generates WSDL for deployed web services
    - clients can access by appending "**`?wsdl`**" to the web service URL which is typically **`http://host:port/axis/service-name`** (**JWS**) or **`http://host:port/axis/services/service-name`** (**non-JWS**)

Axis Features (Cont'd)



- Type mapping
  - refers to serializing Java objects to and from XML in SOAP messages
  - Java primitive types and registered Java Beans are handled automatically
    - Java Beans are Java classes that follow certain method naming conventions
  - can customize for specified Java classes
    - by writing and registering custom serializers and deserializers
  - maintained in a type mapping registry
- Proxy server support
  - through these system properites
    - `http.proxyHost` and `http.proxyPort`
    - `https.proxyHost` and `https.proxyPort`

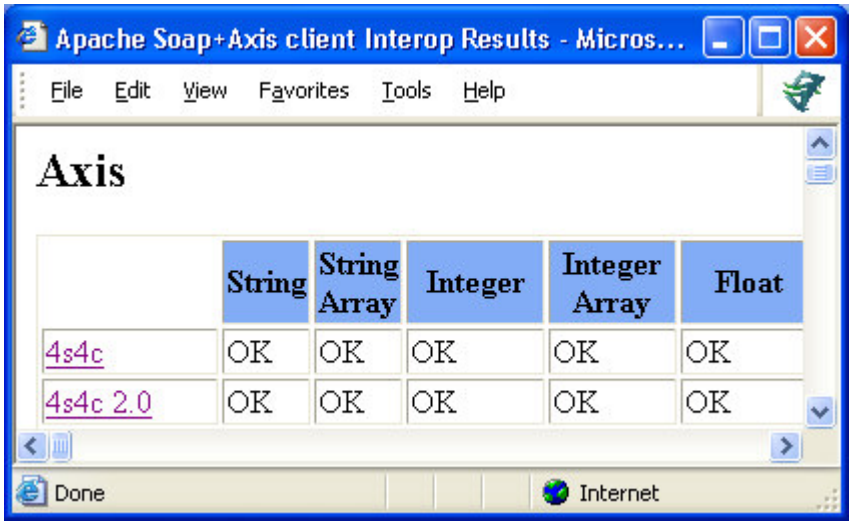
Axis Features (Cont'd)

6

- Pluggable architecture
  - transport-specific, service-specific and global "handlers" executed in a defined order

Interoperability

- Tests are run daily
- SOAPBuilders group defines the tests
  - a group of SOAP implementation developers interested in promoting SOAP interoperability
  - have a Yahoo group at http://groups.yahoo.com/group/soapbuilders/
  - test definitions are at http://www.whitemesa.com/interop.htm
    - contains links to test results for many different SOAP implementations
- Apache SOAP and Axis interop. test results
  - managed by Sam Ruby from IBM
  - posted at http://www.apache.org/~rubys/ApacheClientInterop.html

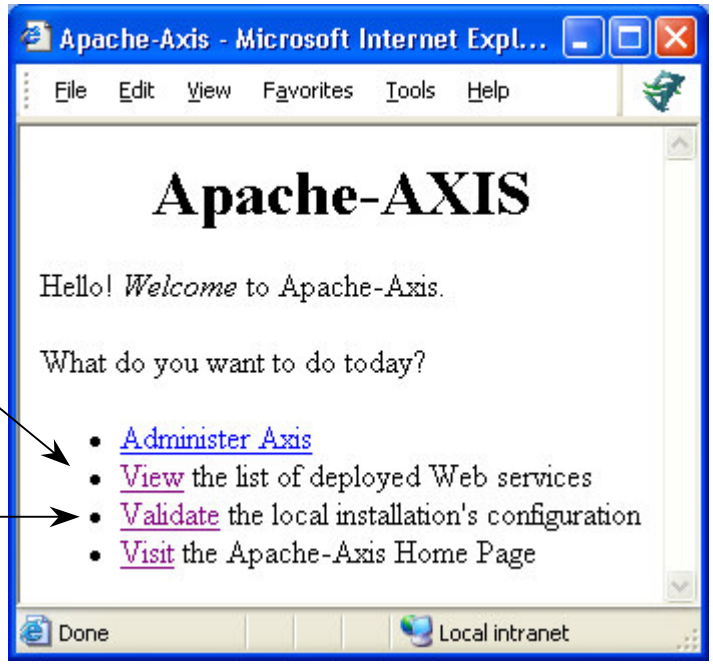


OBJECT COMPUTING, INC.

axis

Testing Axis Installation

- Steps
  - start servlet engine
    - for TOMCAT, this can be done from Windows Explorer by double-clicking startup.bat in %TOMCAT_HOME%\bin
  - from a web browser, visit http://localhost:8080/axis
  - verify that a page like the following is displayed



can view the WSDL of any deployed service

runs happyaxis.jsp (see next page)

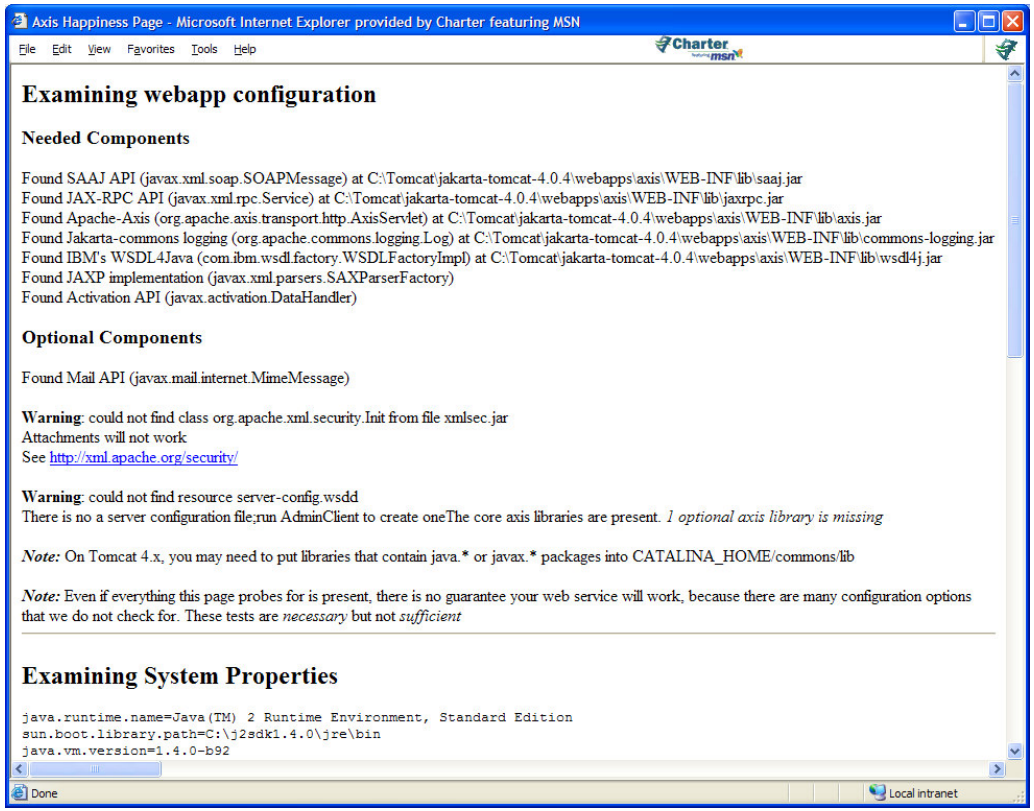
OBJECT COMPUTING, INC.

axis

Testing Axis Installation (Cont'd)

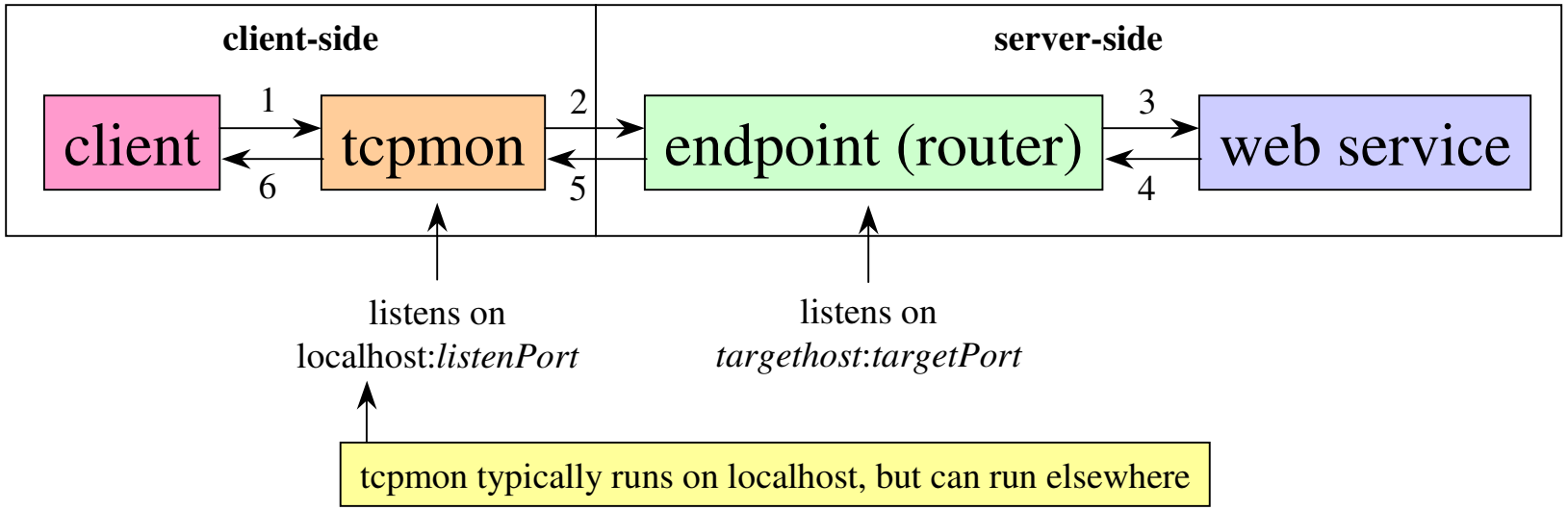
- Happy page
  - a more detailed test of the installation
  - visit http://localhost:8080/axis/happyaxis.jsp

could use in an **HttpUnit** test or in an **Ant get task** to verify that Axis is properly installed during a project build



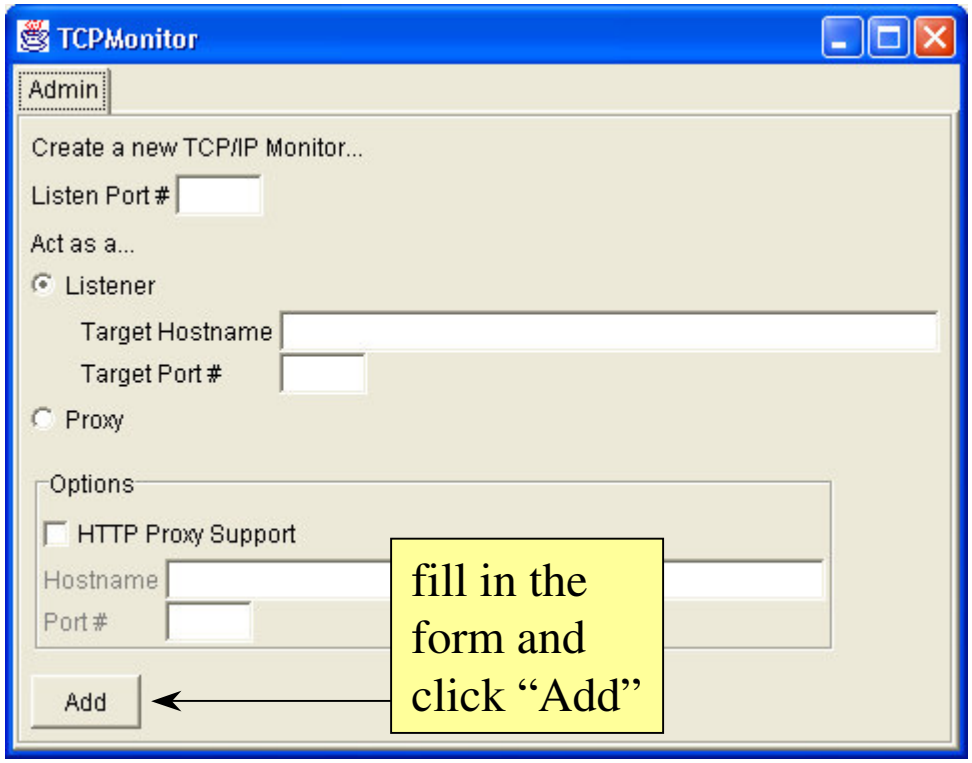
SOAP Message Monitoring

- TCP Monitor tool (tcpmon)
  - intercepts client requests before they are sent to the endpoint and displays them in the GUI
  - forwards requests to endpoint
  - intercepts server responses before they are returned to the client and displays them in the GUI
  - forwards responses to client



Starting tcpmon

- To start tcpmon
  - insure that axis.jar is in CLASSPATH
  - `java org.apache.axis.utils.tcpmon [listenPort targetHost targetPort]`
  - if optional parameters are omitted, the following screen is displayed



can configure multiple *listenPorts* to support multiple *targetHost*/*targetPort* pairs

To start tcpmon and a client from Ant, use the **parallel task**. See "monitor" target in build.xml at end of this section for an example.

Using tcpmon

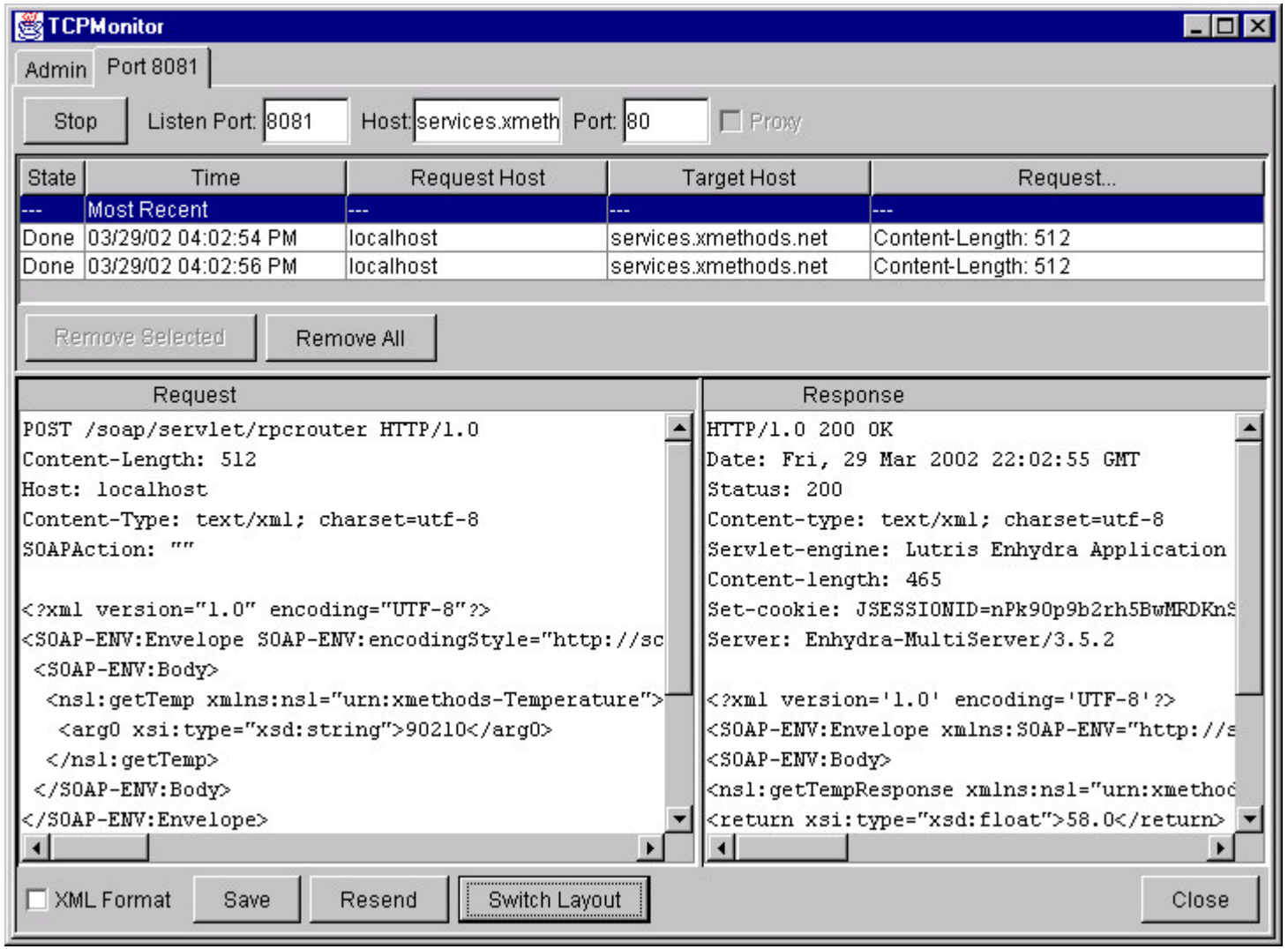
- Steps to use tcpmon
  - modify client to send requests to localhost on *listenPort* instead of *targetHost* on *targetPort*
  - start tcpmon with
    - *listenPort* set to port where clients send requests
    - *targetHost* set to the endpoint host of web service
    - *targetPort* set to the endpoint port of the web service
  - start client

tcpmon Example Screen

select a row to **view** its request and response

can **save** request and response message to a file, but can't reload them

can edit and **resend** request messages;
if length changes, update value of Content-Length HTTP header



OBJECT COMPUTING, INC.

axis

2

Dynamic Invocation

- Uses Axis implementation of JAX-RPC
  - org.apache.axis.client.**Call** implements javax.xml.rpc.**Call**
  - org.apache.axis.client.**Service** implements javax.xml.rpc.**Service**
- Only need to know these details
  - SOAP router URL
    - also called the target endpoint address
  - service namespace
    - required so it can be specified in Axis-generated SOAP requests
  - operation name
  - operation parameter types
  - operation return type

can get this information from a WSDL description of the service

depending on the service, knowing these may not be necessary

2

Dynamic Invocation

This is an example of invoking a web service without using WSDL.
It's much easier to use generated client stubs!
The target URL, operation name and input body namespace can be obtained from a WSDL service description.
The **target URL** for this service is http://services.xmethods.net:80/soap/servlet/rpcrouter.

```
import javax.xml.rpc.namespace.QName;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

public class Client {

  // The values of the following constants were obtained from the WSDL
  // at http://www.xmethods.net/sd/TemperatureService.wsdl.
  private static final String NAMESPACE = "urn:xmethods-Temperature";
  private static final String OPERATION = "getTemp";

  public static void main(String[] args) throws Exception {
    if (args.length < 2) {
      System.err.println("usage: java Client {target-url} {zip-list}");
      System.err.println("where the zipcodes are separated by spaces");
      System.exit(1);
    }
```

urn stands for uniform resource name

Dynamic Invocation (Cont'd)

JAX-RPC objects

```
Service service = new Service();
Call call = (Call) service.createCall();
call.setTargetEndpointAddress(new java.net.URL(args[0]));
call.setOperationName(new QName(NAMESPACE, OPERATION));

call.addParameter("zipcode",
                  org.apache.axis.encoding.XMLType.XSD_STRING,
                  javax.xml.rpc.ParameterMode.IN);
call.setReturnType(org.apache.axis.encoding.XMLType.XSD_FLOAT);

// For each zipcode specified on the command line ...
for (int i = 1; i < args.length; i++) {
  String zipcode = args[i];
  Object temperature = call.invoke(new Object[] {zipcode});
  System.out.println("current temperature in " + zipcode +
                     " is " + temperature);
}
  }
}
```

passing this as a command-line argument instead of hard-coding it allows easy use of **tcpmon**

optional steps depending on the service

Some services don't automatically encode the **return type**. This service does. If not, this is how the client can set it.

Parameters are named arg0, arg1, … by default.
This allows the client to give them **names** and **data types** the service expects.

Generated HTTP SOAP Request

```
POST /soap/servlet/rpcrouter HTTP/1.0
Content-Length: 518
Host: services.xmethods.net
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ns1:getTemp xmlns:ns1="urn:xmethods-Temperature">
      <zipcode xsi:type="xsd:string">90210</zipcode>
    </ns1:getTemp>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

It would sure reduce message sizes if these namespace definitions were known by default, but that wouldn't be standard XML practice.

operation

By default, Axis specifies the service to be invoked by the namespace of the first element in the SOAP Body.

Generated HTTP SOAP Response

```
HTTP/1.0 200 OK
Date: Sat, 23 Mar 2002 13:25:59 GMT
Content-Length: 465
Content-Type: text/xml; charset=utf-8
Status: 200
Servlet-Engine: Lutris Enhydra Application Server/3.5.2
(JSP 1.1; Servlet 2.2; Java 1.3.0; Linux 2.4.7-10smp x86;
 java.vendor=IBM Corporation)
Set-Cookie: JSESSIONID=cdDkeXGxW5F4cbgFYEjYHnOl;Path=/soap
Server: Enhydra-MultiServer/3.5.2
Via: 1.0 C6100-2 (NetCache NetApp/5.1R2D20)
```

HTTP headers

2

Generated HTTP SOAP Response (Cont'd)

HTTP content

```
<?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:getTempResponse
     xmlns:ns1="urn:xmethods-Temperature"
     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:float">55.0</return>
    </ns1:getTempResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Deploying Web Services Using JWS

- Steps
  - if not there already,
    copy "axis" directory in "webapps" directory of the Axis distribution
    to the deployment directory of a server that supports Java servlets
    - for Tomcat, this is the "webapps" directory
  - copy any Java source file that implements a web service
    into this "axis" directory
    - no special code is required
    - all public, non-static methods are exposed
    - if the class is in a package, copy it to the appropriate subdirectory
  - change the file extension from "`.java`" to "`.jws`"
  - to view the WSDL of a JWS web service,
    enter the following URL in a web browser

    `http://`*`host`*`:`*`port`*`/axis/`*`pkg-subdirs/file-name`*`.jws?wsdl`

    optional part

Example JWS

- Source file BasicMath.java

```
package com.ociweb.math;
public class BasicMath {
    public int add(int n1, int n2) {
        return n1 + n2;
    }
}
```

- Steps to deploy this
  - create `com/ociweb/math` directories under `axis` directory in Tomcat's `webapps` directory
  - copy BasicMath.java to that directory
  - rename it to BasicMath.jws

all the **magic** is provided by JWSHandler, JWSProcessor and RPCProvider

- Access the WSDL with
  - http://localhost:8080/axis/com/ociweb/math/BasicMath.jws?wsdl
  - can generate client stubs using WSDL2Java and this URL (covered soon)

Invoking JWS Services

- Two options
  - use Dynamic Invocation covered earlier
  - use generated client stubs covered later
    - an example of using one follows

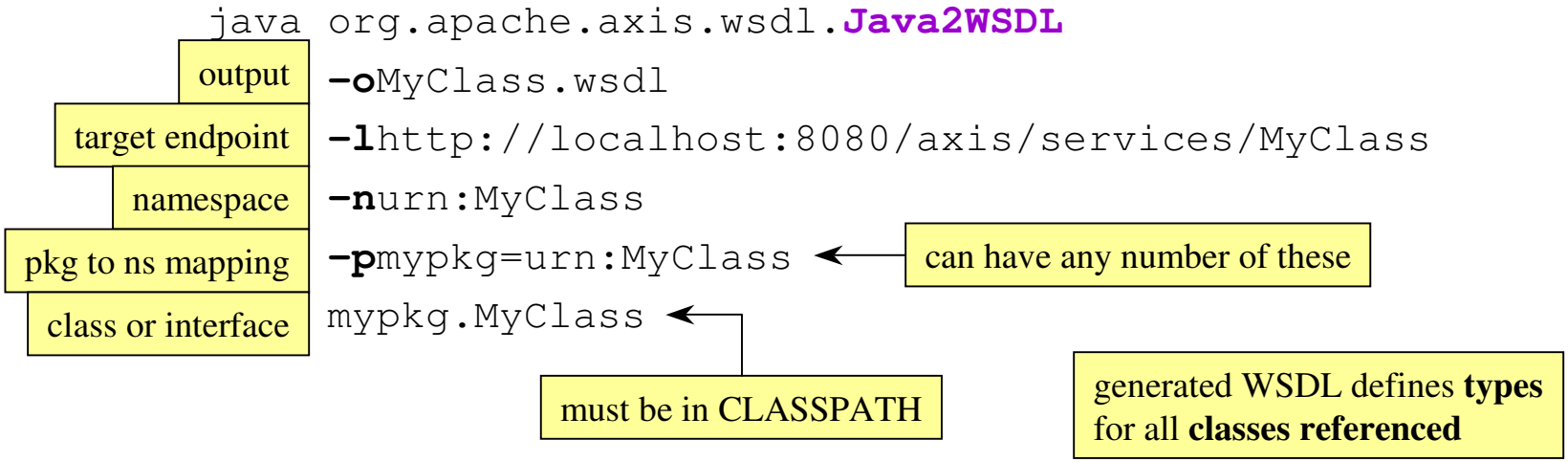
```
import com.ociweb.math.*;

public class BasicMathClient {

    public static void main(String[] args)
    throws java.rmi.RemoteException, javax.xml.rpc.ServiceException {
        BasicMathService service = new BasicMathServiceLocator();
        BasicMath stub = service.getBasicMath();
        System.out.println(stub.add(19, 3));
    }
}
```

Generating WSDL From Java Interfaces and Classes

- Two ways
  - using JWS (covered earlier)
  - using Java2WSDL
- Java2WSDL
  - to generate WSDL for the class `MyClass` (can also use an interface) in the package `mypkg` using the namespace `urn:MyClass`



Client Stubs

- Represent the service on the client-side
  - a.k.a. proxies
- Generated from WSDL using WSDL2Java
  - WSDL can be accessed locally or from a remote URL
  - if Java service implementation already exists,
    WSDL can be generated using Java2WSDL
- Provide type-safe invocation of web services
  - rather than passing parameters as an array of Objects,
    as is done with dynamic invocation,
    calls are made using an interface that specifies parameter types
  - rather that receiving an Object result,
    the interface specifies the actual result type
  - types are checked at compile-time
  - example usage was shown on page 27

WSDL2Java Generates

- For each `<type>`
  - class representation called ***type-name*.java**
  - holder class called ***type-name*Holder.java**, if it is used as an inout or out parameter
- For each `<porttype>`
  - interface called ***port-name.java***
    - describes porttype operations
    - called the Service Definition Interface (SDI)

- **holder classes** support passing parameters by references, something that Java doesn't directly support
- **pre-built holder classes** for primitive types are in the javax.xml.rpc.holders package

WARNING: Watch out for WSDL2Java overwriting existing source files with ***type-name*.java** and ***port-name.java*** files if you generated the WSDL using Java2WSDL!

WSDL2Java Generates (Cont'd)

- For each `<binding>`
  - stub class called ***port-name*BindingStub.java**
    - implements *port-name*.java
    - used by clients to invoke web service methods
    - uses JAX-RPC Service and Call interfaces
  - implementation class shell called ***port-name*SoapBindingImpl.java**
    - implements *port-name*.java
    - web service method implementations to be completed by developer
    - won't be overwritten if it already exists
    - don't need this if a class implementing the operations already exists
  - optionally, a skeleton class called ***port-name*SoapBindingSkeleton.java**
    - implements *port-name*.java
    - server-side counterpart to *port-name*ServiceBindingStub
    - forwards all calls to service methods to implementation methods
    - benefit is questionable; calls can go directly to your implementation instead

only has "SOAP" in the name when the transport is SOAP

- neither of these is needed
- see p. 35 for steps to bypass them

WSDL2Java Generates (Cont'd)

- For each `<service>`
  - interface called ***port-name*Service.java**
    - describes methods for obtaining porttype interface implementations
      - one get method for each port defined in the service
  - locator class called ***port-name*ServiceLocator.java**
    - implements *port-name*Service.java (above)
    - locates an implementation of the service using the URI specified in the WSDL or a specified URI
    - creates an instance of the binding stub class and returns it as the porttype interface type
  - optionally, a JUnit test class called ***port-name*ServiceTestCase.java**
    - for testing methods in the porttype interface
    - test methods must be completed manually

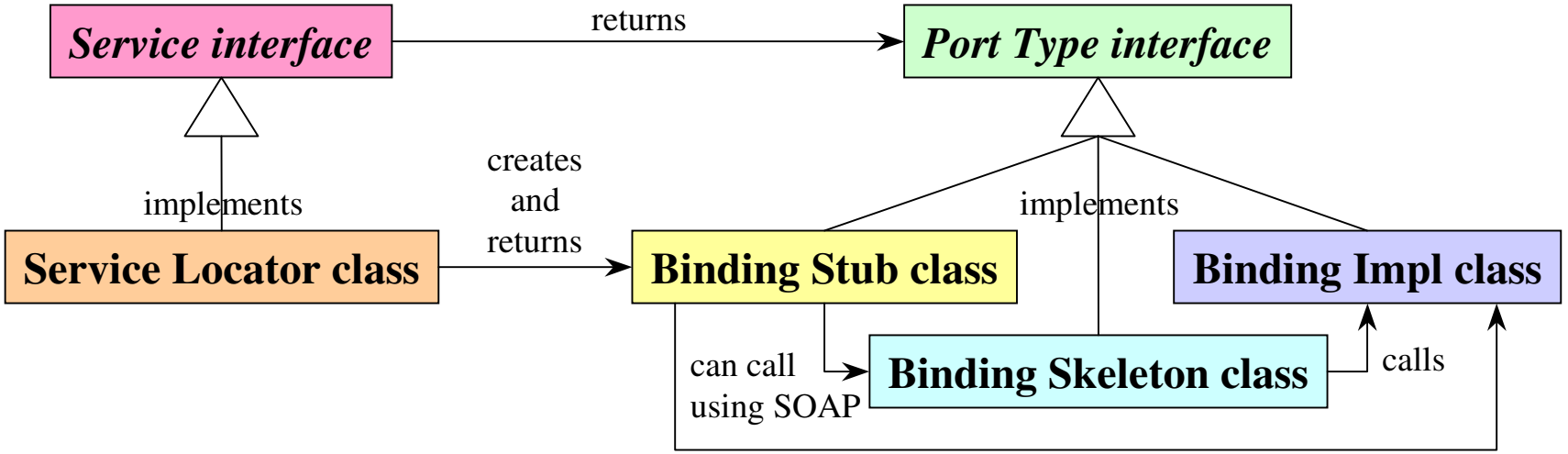
WSDL2Java Generates (Cont'd)

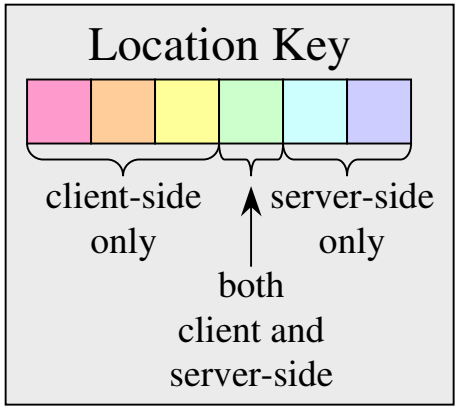
- For each WSDL file
  - deployment descriptor called **deploy.wsdd**
    - used by AdminClient to deploy the service
    - implementation class referred to will either be the skeleton class (if skeleton generated) or the impl. class (if no skeleton generated)
    - can change to refer to your own class
  - undeployment descriptor called **undeploy.wsdd**
    - used by AdminClient to undeploy the service

`-Strue` to generate skeleton class
`-Sfalse` to suppress it (default)

more on WSDD later

Relationships Between Generated Files





Steps to use in client code

1) create instance of Service Locator
   (hold in variable of Service interface type)
2) use it to obtain a Binding Stub
   (hold in variable of Port Type interface type)
3) invoke web service methods on it

← this was done in BasicMathClient

Deploying Without Using Generated Skeleton and Impl Classes

- Can deploy a standard Java class without using generated skeleton and impl classes
  - matches what is done with JWS
- Still run WSDL2Java
  - to generate service interface, service locator class, binding stub class and deployment descriptors
- Modify generated deploy.wsdd file
  - change value of the className parameter to the service to refer to your class instead of the generated skeleton class

```
<parameter name="className" value="HERE"/>
```

Using WSDL2Java

- Command

```
java org.apache.axis.wsdl.WSDL2Java wsdl-uri
```

- Option highlights

`-N`*`namespace=package`*
maps a **namespace** to a Java package

`-o` *`dir`* specifies **output** directory where generated files should be written

`-p` *`pkg`* overrides default **package** name for generated classes

`-s` generates **service-side interfaces and classes** required by a service implementation, including a **skeleton** class

`-t` generates JUnit **test** case for the web service

`-v` prints a message about each file that is generated

`-S` option controls this (see page 33)

package of generated files defaults to reverse of server name
for example, server www.xmethods.net uses package net.xmethods.www

Generated test code doesn't sufficiently exercise the web service. Add code to it.
CAUTION: Save modified test code somewhere else
because WSDL2Java will overwrite it if it is run again!

Client Stub Invocation

(uses same service as earlier dynamic invocation example)

```
import net.xmethods.www.*;
```

package of generated files

```
public class Client {

  public static void main(String[] args) throws Exception {
    if (args.length == 0) {
      System.err.println("usage: java Client {target-url} {zip-list}");
      System.err.println("where the zipcodes are separated by spaces");
      System.exit(1);
    }
```

Client Stub Invocation (Cont'd)

```
    TemperatureService service = new TemperatureServiceLocator();
    TemperaturePortType stub =
      service.getTemperaturePort(new java.net.URL(args[0]));

    for (int i = 1; i < args.length; i++) {
      String zipcode = args[i];
      float temperature = stub.getTemp(zipcode);
      System.out.println("current temperature in " + zipcode +
                         " is " + temperature);
    }
  }
}
```

Don't have to pass target URL, but it's useful for using tcpmon.

Notice that parameters are not passed to the service in an array of Objects and the proper return type is returned instead of an Object.

Customized Deployment

- Benefits over JWS deployment
  - deploy/undeploy multiple services
  - deploy/undeploy handlers
  - deploy classes with no source
  - custom type mapping
- Server-side classes
  - these are web service implementation classes and classes they use
  - when using Axis webapp
    - copy **.class** files to **webapps/axis/WEB-INF/classes**
    - copy **.jar** files to **webapps/axis/WEB-INF/lib**
  - a problem if different web services want to use different versions of a class or JAR
    - solution is to deploy services in different web apps. that each contain Axis

Customized Deployment (Cont'd)

- Web Service Deployment Descriptor (WSDD)
  - specifies components to be deployed and undeployed
    - services, handlers and more
  - specifies type mappings
  - download Axis source to get DTD and XML Schema for WSDD
    - in java/wsdd directory
- AdminClient uses WSDD
  - to deploy or undeploy components, pass it a WSDD file

    typically called `deploy.wsdd` or `undeploy.wsdd`

    ```
    java org.apache.axis.client.AdminClient filename.wsdd
    ```
    - modifies **server-config.wsdd** in the axis webapp WEB-INF directory
  - to list deployed components

    ```
    java org.apache.axis.client.AdminClient list
    ```
    - outputs **server-config.wsdd** which is used when the server is restarted to redeploy all previously deployed services

WSDD To Deploy And Undeploy

- Generated by WSDL2Java
- Example to deploy a service

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="CarQuote" provider="java:RPC">
    <parameter name="className"
               value="com.ociweb.auto.CarQuoteImpl"/>
    <parameter name="allowedMethods" value="*"/>
  </service>
</deployment>
```

identifies the pivot handler to be used; Axis provides three: RPC, MSG and EJB; MSG is for message or document-centric calls as opposed to RPC-style calls

makes all public, non-static methods of the class available; can be a whitespace-delimited list of method names

don't need to have source for this class

- Example to undeploy a service

```
<undeployment xmlns="http://xml.apache.org/axis/wsdd/">
  <service name="CarQuote"/>
</undeployment>
```

WSDD To Register a Java Bean For Automatic Serialization

- Example for the Car class

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <service name="CarQuote" provider="java:RPC">
    <parameter name="className"
               value="com.ociweb.cardealer.CarQuoteImpl"/>
    <parameter name="allowedMethods" value="getQuote"/>
  </service>

  <beanMapping xmlns:ns="urn:CarQuote" qname="ns:Car"
    languageSpecificType="java:com.ociweb.cardealer.Car"/>
</deployment>
```

appropriate when the class follows Java Bean conventions and all of its properties should be serialized

beanMapping is needed if a Car object is passed to or returned from a web service operation

Custom Type Mapping

- Steps
  - write **serializer class**
    - implements org.apache.axis.encoding.Serializer
  - write **serializer factory class** that returns serializer instances
    - implements org.apache.axis.encoding.SerializerFactory
  - write **deserializer class**
    - extends org.apache.axis.encoding.DeserializerImpl
  - write **deserializer factory class** that returns deserializer instances
    - implements org.apache.axis.encoding.DeserializerFactory
  - **register them** using WSDD and the Admin client

add as child of `deployment` element

```
<typeMapping
  xmlns:ns="urn:CarQuote"
  qname="ns:Car" type="java:com.ociweb.cardealer.Car"
  serializer="com.ociweb.cardealer.CarSerializerFactory"
  deserializer="com.ociweb.cardealer.CarDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
```

WSDD DTD Highlights

(attributes omitted)

root element

no specified order for these

```
<!ELEMENT deployment (documentation?, globalConfiguration?,
    (beanMapping|typeMapping|chain|handler|transport|service)*)>
<!ELEMENT chain (documentation?, parameter*, handler*)>
<!ELEMENT documentation (#PCDATA)>
<!ELEMENT faultFlow (documentation?, parameter*, (chain|handler)*)>
<!ELEMENT globalConfiguration (documentation?, parameter*, transport*,
    requestFlow?, provider, responseFlow?, faultFlow*)>
<!ELEMENT handler (documentation?, parameter*)>
<!ELEMENT operation EMPTY>
<!ELEMENT parameter EMPTY>
<!ELEMENT provider (documentation?, parameter*, operation*)>
<!ELEMENT requestFlow (documentation?, parameter*, (chain|handler)*)>
<!ELEMENT responseFlow (documentation?, parameter*,
    (chain | handler)*)>
<!ELEMENT service (documentation?, parameter*, typeMapping*,
    requestFlow?, provider, responseFlow?, faultFlow*)>
<!ELEMENT transport (documentation?, parameter*,
    requestFlow?, responseFlow?, faultFlow*)>
<!ELEMENT beanMapping (documentation?)>
<!ELEMENT typeMapping (documentation?)>
```

can go inside or outside service elements depending on whether they are specific to a service

Axis Components

- Axis engines
  - coordinate message processing by invoking a series of handlers
  - run on server and optionally on client

see diagrams coming up

- Handlers
  - operate on request and response messages
    - **can** examine a message and **modify** it before passing it on
  - can invoke software outside Axis
  - example uses
    - authentication, compression, encryption, logging, message transformation

somewhat similar to Java servlet Filters

- Chains
  - collections of handlers that are executed in a specified order
  - a chain is itself handler, allowing chains to contain other chains

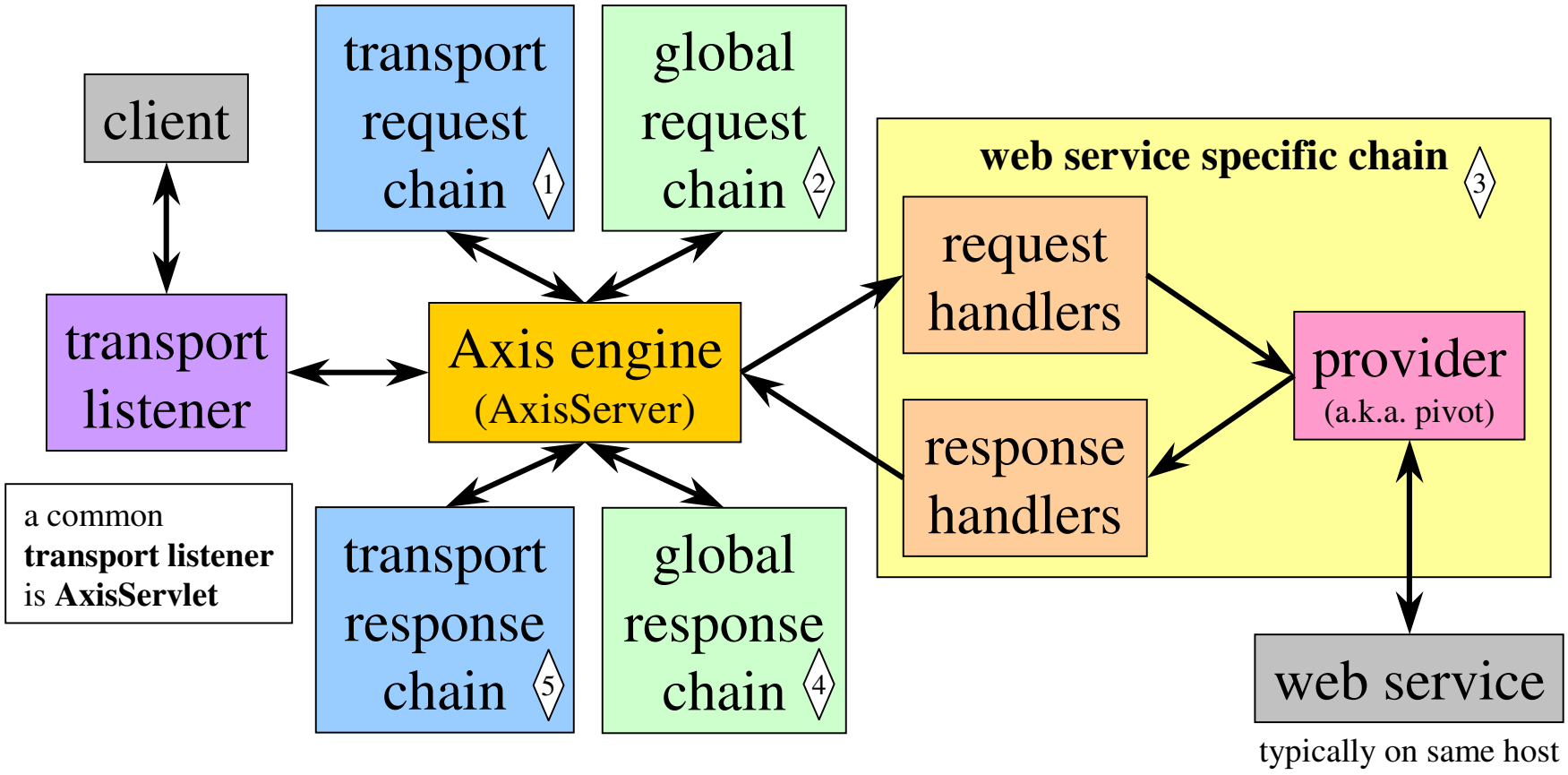
Composite pattern

- Provider (a.k.a. “pivot point” handler)
  - point in the chain where handlers switch from processing the request to processing the response
  - invokes web service operation

Axis Components (Cont'd)

- Transports
  - handle message protocol conversions
    - request from client to Axis Engine (for example, HTTPTransport)
    - response from Axis Engine to client
- Serializers/Deserializers
  - convert Java data (primitives and objects) to and from XML
- Deployment/Configuration
  - deployment refers to component registrations
    - services, handlers, chains, transports,
      bean mappings and type mappings (register serializers and deserializers)
  - configuration refers to setting Axis options
    - such as security controls for remote administration
  - Administration subsystem provides the easiest way to do this
    - pass WSDD file to AdminClient

Axis Server-side Architecture



- configured by **server-config.wsdd** in webapps/axis/WEB-INF which is modified by **AdminClient**
- actually there is a **single transport chain** that is considered to have two sides
- an Axis "**service**" is a "**targeted chain**" (can have request handlers, a pivot handler and response handlers)

Configuring Handlers With WSDD

- Three kinds of handler elements
  - unnamed - `<handler type="java:full-class-name"/>`
  - named - `<handler name="name" type="java:full-class-name"/>`
    - can be used in multiple chain/phases
  - reference to named - `<handler type="name"/>`
- Handlers are only added to chains through deployment
  - not dynamically at run-time
- Three chains
  - transport, global and service
- Two phases
  - request and response

six places for a handler to be invoked

Configuring Handlers With WSDD (Cont'd)

- Global handlers

Handlers in the **request flow** execute before the pivot.
Handlers in the **response flow** execute after the pivot.

```
<globalConfiguration>
  <requestFlow> handler-element* </requestFlow>
  <responseFlow> handler-element* </responseFlow>
</globalConfiguration>
```

- Transport-specific handlers

```
<transport name="http">
  <requestFlow> handler-element* </requestFlow>
  <responseFlow> handler-element* </responseFlow>
</transport>
```

value to use for HTTP transport

- Service-specific handlers

```
<service name="service-name" provider="java:RPC">
  <requestFlow> handler-element* </requestFlow>
  <responseFlow> handler-element* </responseFlow>
</transport>
```

typical value

Exposing EJBs as Web Services

- Currently only works with stateless session beans
- Don't have to write any code!
- Steps
  - deploy EJB in a J2EE application server
  - create a WSDD file for deploying a corresponding Axis service
    - see details on next page
  - run AdminClient on the WSDD to deploy the Axis service
  - client code to invoke the service is identical to code for invoking non-EJB web services

Exposing EJBs as Web Services (Cont'd)

- WSDD details

this is how it knows to use EJBProvider

```
<service name="service-name" provider="java:EJB">
  <parameter name="jndiURL"
             value="naming-service-url"/>
  <parameter name="jndiContextClass"
             value="initial-context-factory-class-name"/>
  <parameter name="beanJndiName"
             value="ejb-jndi-name"/>
  <parameter name="homeInterfaceName"
             value="home-interface-name"/>
  <parameter name="remoteInterfaceName"
             value="remote-interface-name"/>
  <parameter name="className"
             value="remote-interface-name"/>
  <parameter name="allowedMethods"
             value="list-of-methods-to-expose"/>
</service>
```

verify need to specify these

Summary - Axis Pros and Cons

- Pros
  - great architecture
  - free and open source
  - well supported by IBM committers
  - supports **JAX-RPC** API
  - supports SOAP with Attachments API for Java (**SAAJ**)
  - large number of unit tests
  - regularly tested for interoperability
- Cons
  - significant changes are still being made
    - doesn't yet support asynchronous message processing
    - doesn't provide explicit support for SOAP intermediaries
      - can be implemented with custom handlers
  - open source
    - some see this as a positive, while others see it as a negative

SAAJ 1.1 specification is a maintenance release of the **JAXM** 1.0 specification