Enabling Grids for E-science in Europe

www.eu-egee.org

17th October 2004

WSDL

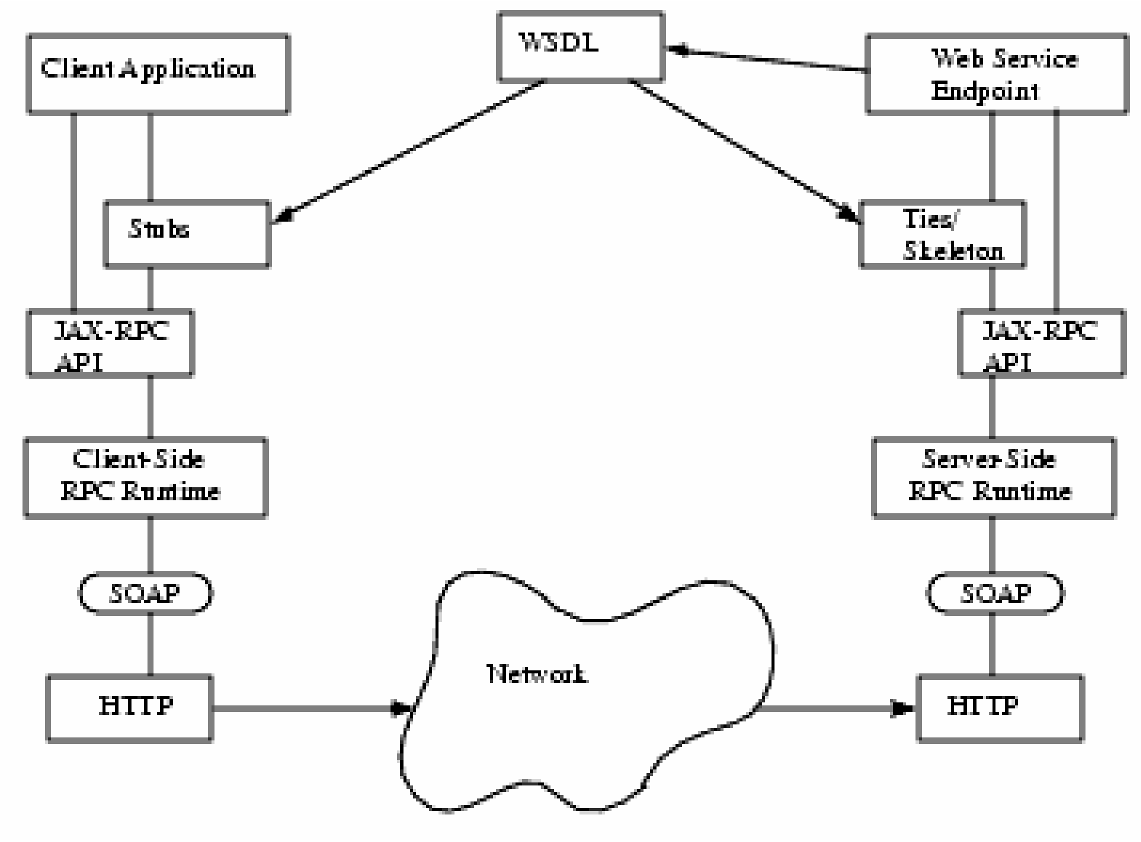
JAX-RPC

EGEE is a project funded by the European Union under contract IST-2003-508833

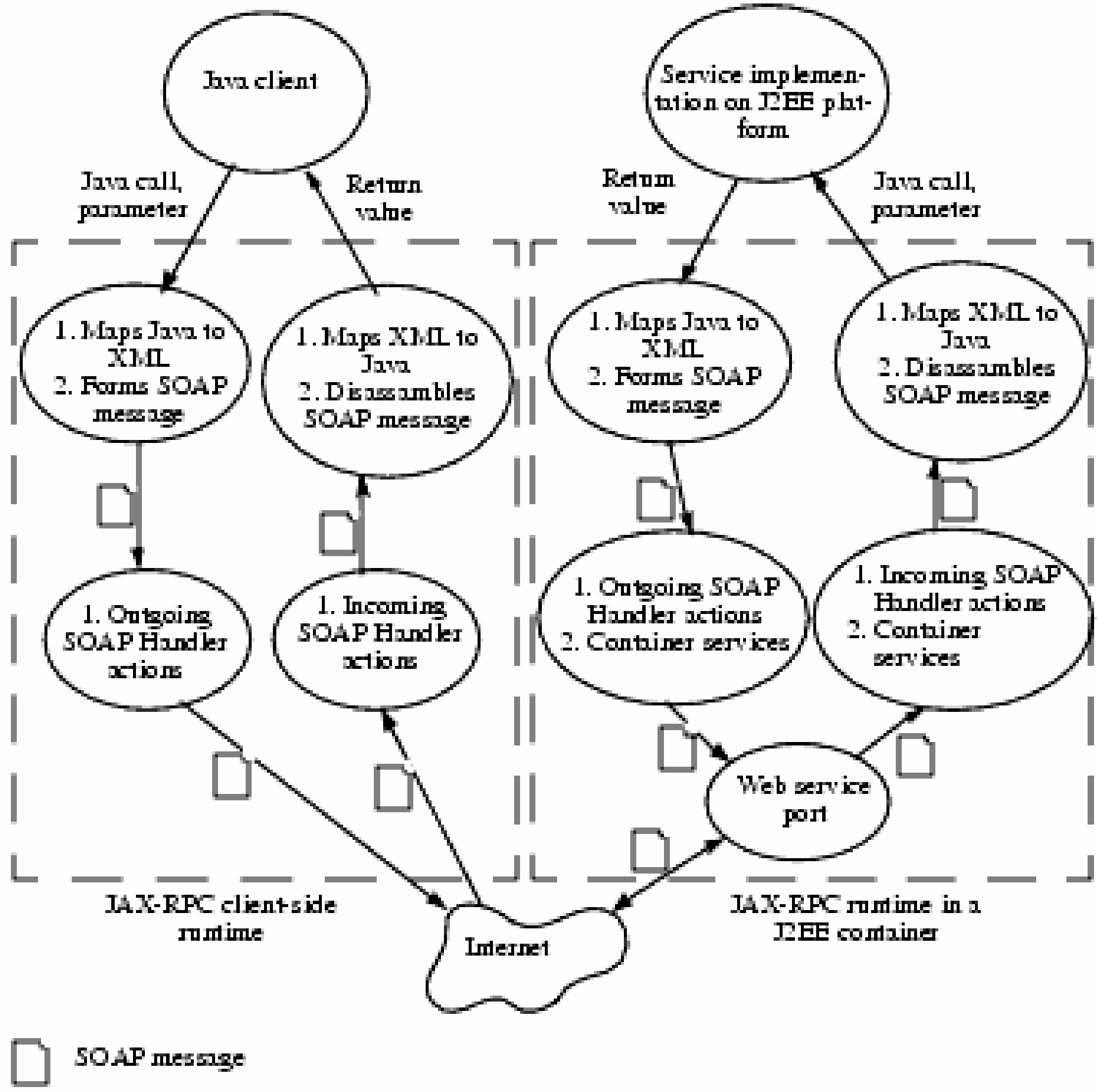
JAX-RPC API packages

javax.xml.rpc	Core classes for the client side programming model
javax.xml.rpc.encoding	Java primatives <-> XML SOAP messages
javax.xml.rpc.handler	processing XML messages
javax.xml.rpc.handler.soap	
javax.xml.rpc.holders	support the use of IO parameters
javax.xml.rpc.server	minimal API for web service inplementation
Javax.xml.rpc.soap	specific SOAP bindings

JAX-RPC Architecture



Java web service flow


Java client
Java call, parameter
Return value
1. Maps Java to XML
2. Forms SOAP message
1. Maps XML to Java
2. Disassambles SOAP message
1. Outgoing SOAP Handler actions
1. Incoming SOAP Handler actions
JAX-RPC client-side runtime
Internet
Service implementation on J2EE platform
Return value
Java call, parameter
1. Maps Java to XML
2. Forms SOAP message
1. Maps XML to Java
2. Disassambles SOAP message
1. Outgoing SOAP Handler actions
2. Container services
1. Incoming SOAP Handler actions
2. Container services
Web service port
JAX-RPC runtime in a J2EE container
SOAP message

Client operation modes

- JAX-RPC allows two modes of operation
  - Synchronous request – response
  - One-way RPC

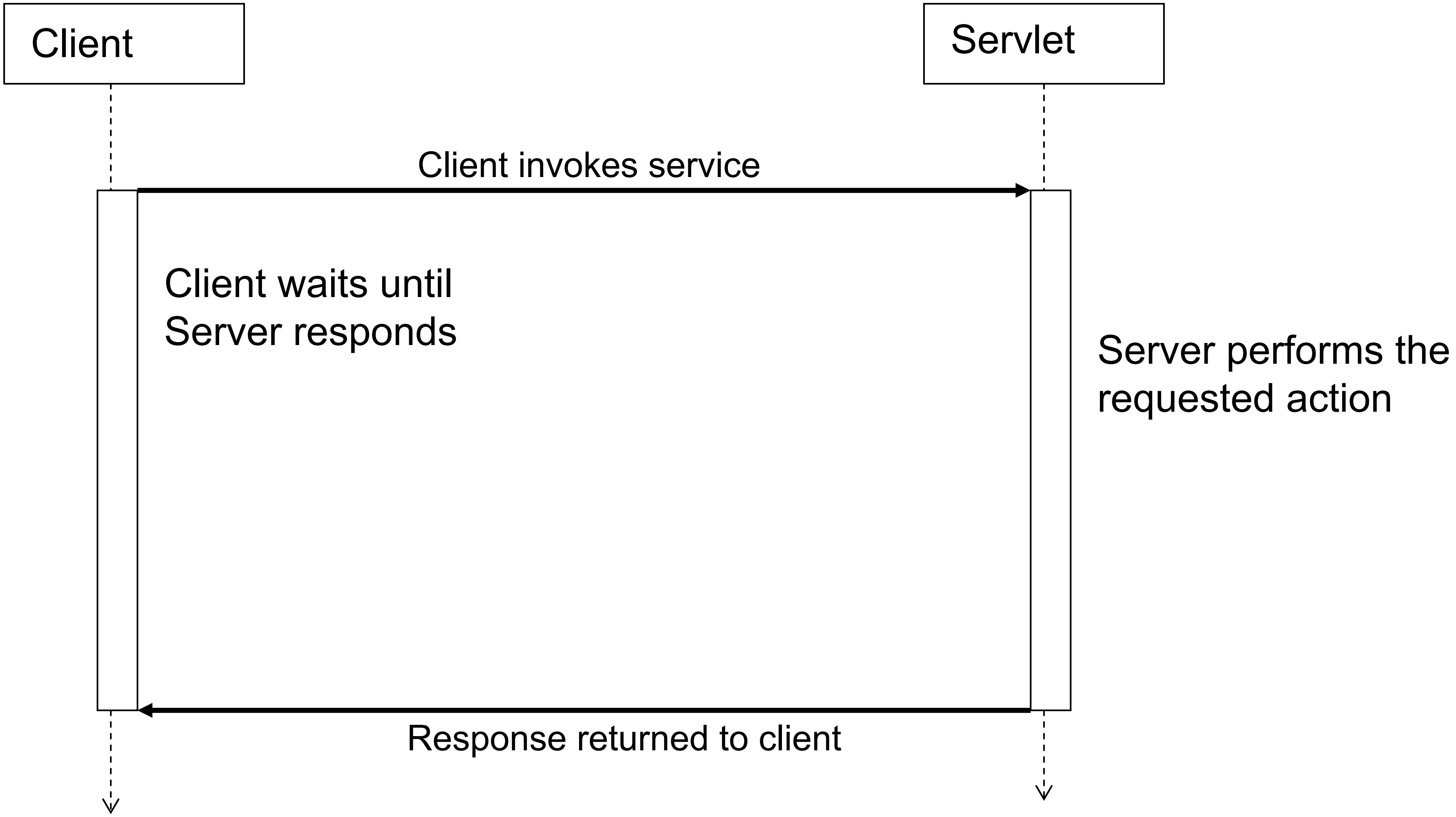
- Synchronous
  - This involves blocking the client until it receives a response
  - Is similar to a traditional java method call

- One – way
  - No client blocking
  - Service performs a operation without replying.
  - Not analogous to traditional method calls

Comparing One-way and traditional methods

- A traditional java method call like
  - Public void request (int arg1, int arg2);
  - Does not return a value to the caller
  - However if it appeared in a web service interface definition it would be mapped to a synchronous request – response RPC
  - This is because it indicates that an exception may still need to be thrown to the client.
  - A one – way RPC cannot throw an exception.

Synchronous method invocation



Client

Servlet

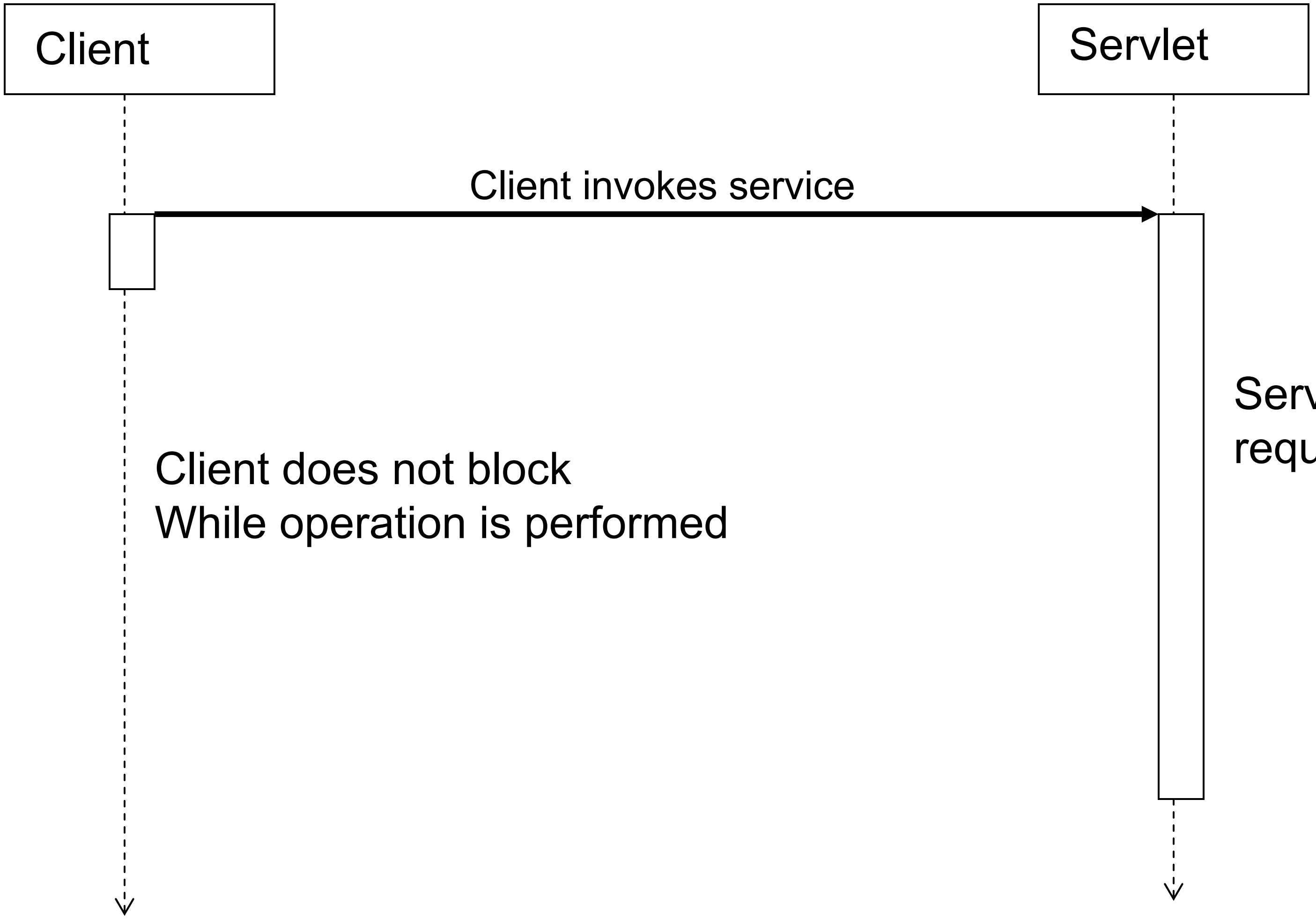
Client invokes service

Client waits until Server responds

Server performs the requested action

Response returned to client

One – way RPC invocation



Client

Servlet

Client invokes service

Client does not block
While operation is performed

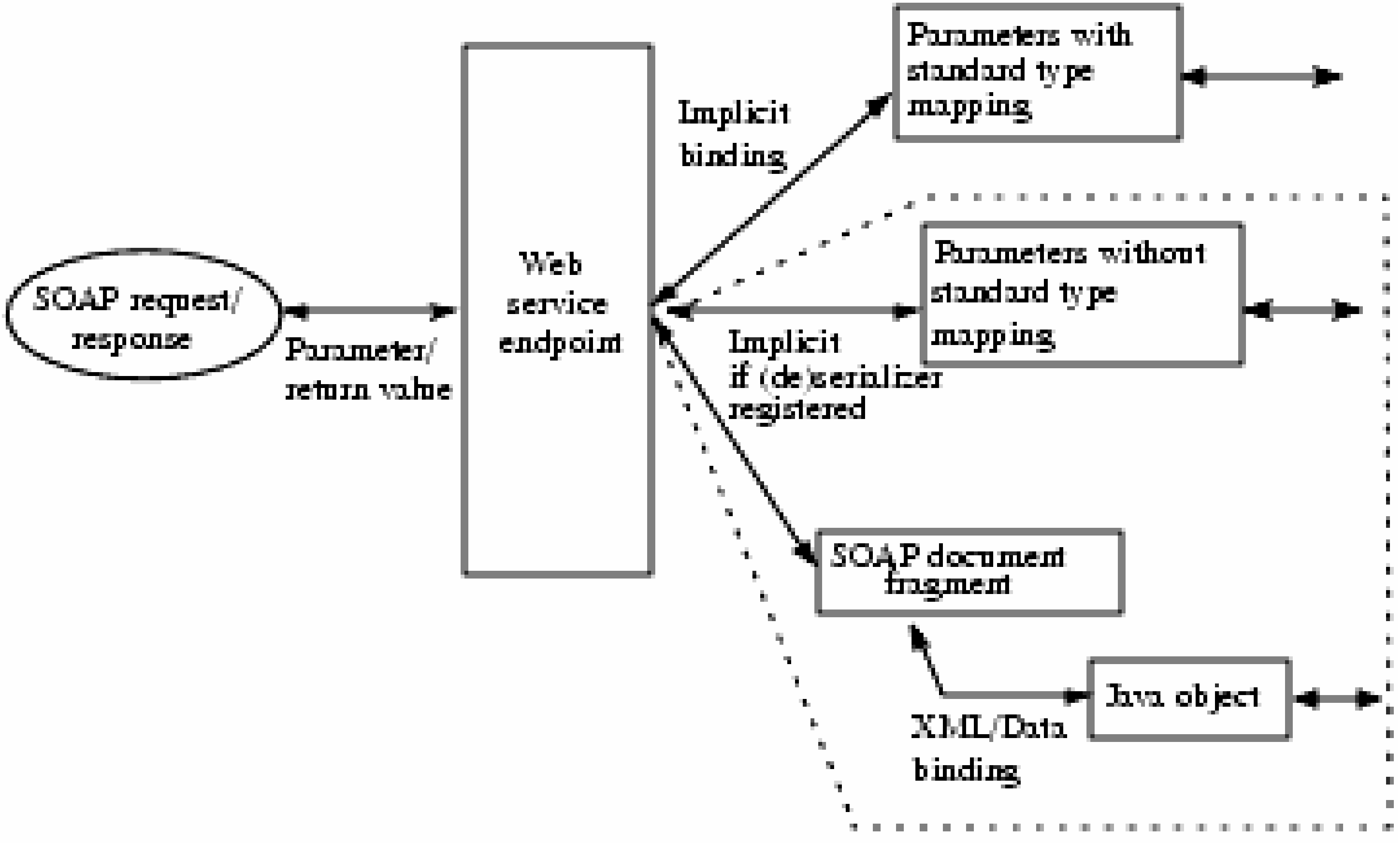
Server performs the requested action

Defining a service

- A service can be defined starting with:
  - A java interface
  - A WSDL document
- Which to use?
  - If the service end point interface is defined in java it may not be interoperable with services/clients defined in other languages
  - If the service is initially defined in WSDL it will be open

Using JAX-RPC to create a service from a Java interface

Binding Parameters and Return Values with JAX-RPC



Interface method definitions

A java web service end point interface must obey the following rules:

- The interface must extend `java.rmi.remote`
- Interface methods must declare that it throws `java.rmi.RemoteException`
- Service dependent exceptions can be thrown if they are checked exceptions derived from `java.lang.Exception`
- Method name-overloading is permitted
- Service endpoint interfaces may be extensions of other interfaces

Supported data types

- Java primitives (eg. `bool, int, float`, etc)
- Primitive wrappers (`Boolean, Interger, Float`, etc)
- Standard java classes (required - `java.lang.String, java.util.Calendar, java.util.Date, java.math.BigDecimal, java.math.BigInterger`)
- Value types
- Holder classes
- Arrays (where all elements are supported types)

Object by reference is not supported

Value Types

- Class has a public no-argument constructor
- May be extended from any other class, may have static and instance methods, may implement any interface (except `java.rmi.Remote` and any derived)
- May have static fields, instance fields that are public, protected, package private or private but these must be supported types.

Warning about comparing classes

- The values returned by service methods are in fact local classes created by JAX-RPC from the XML serialisation
- This means that comparisons using `==` should be avoided
- `equals ()` should be used instead
- (inner static classes will not compare correctly)

Serializer

- If you want to pass an un-supported java class you have to create your own serializer/deserializer to translate to and from XML.

- This not a trivial task as there is no JAX-RPC framework.

Client side Implementation

wscompile

- Generates
  - Compiled class files + optionally source files for stubs to interface with client side JAX-RPC
  - WSDL file
  - Model file

Example commandline

```
wscompile -gen:client -d output/client -classpath classpath config-file
```

(add `-keep -s` to retain java source files)

config.xml

```
<?xml version=“1.0” encoding=“UTF-8” ?>
<configuration xmlns=“http://java.sun.com/xml/ns/jax-rpc/ri/config”>
    <service name=“……..”
    targetNamespace=“………………………”
    typeNamespace=“……………………………..”
    packageName=“……………………………….”>

        <interface name=“……………………………”/>
    </service>
</configuration>
```

name = name of service

targetNamespace = namespace of WSDL for names associated with the service eg. port type

typeNamespace = namespace of WSDL for data types

packageName = name of java package

Generated files

Some of the client side generated files:

Service	Service.java
	Service_Impl.java
	Service_SerializerRegistry.java
Exception	ServiceException_SOAPSerializer.java
	ServiceException_SOAPBuilder.java
Value type	Info_SOAPSerializer.java
	Info_SOAPBuilder.java
Interface	Interface_Stub.java
	method.java

Service.java file

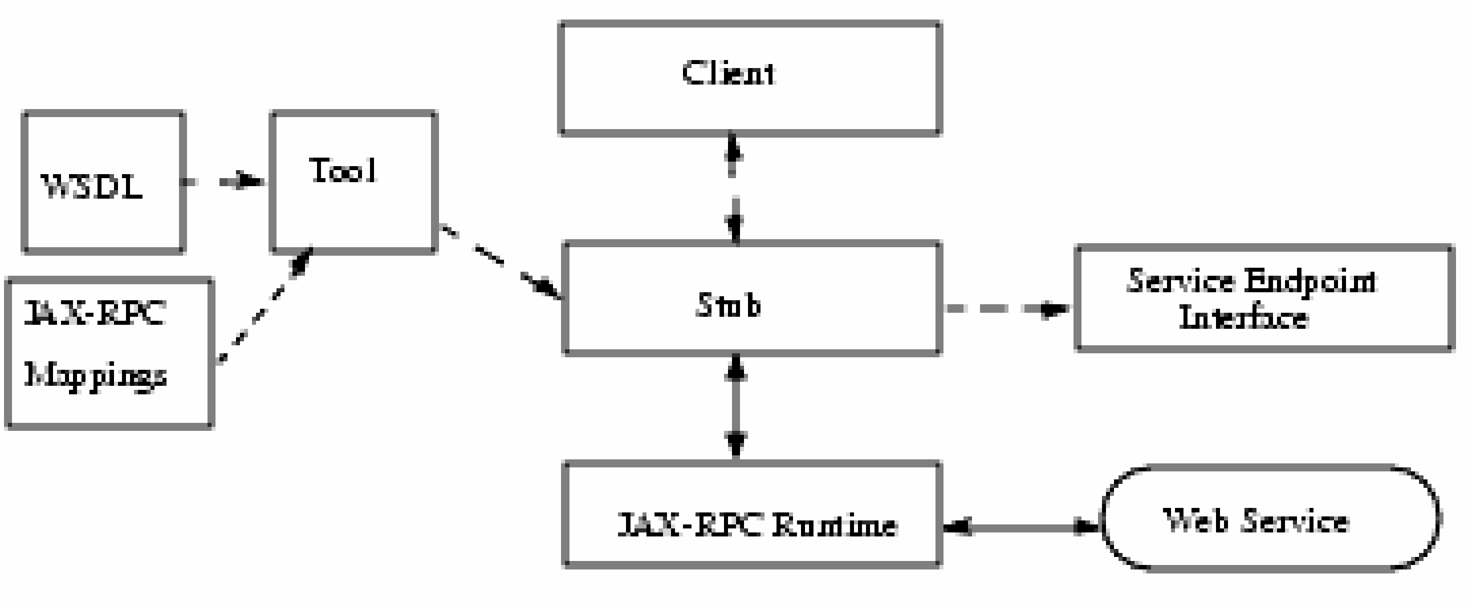
- The `Service.java` file corresponds to the definition of the interface for the web service, ie it contains the same info as the `<service>` element in the config file.

```
package servicePackage;

import javax.xml.rpc.*;

Public interface Service extends javax.aml.rpc.Service
{
      public servicePackage getServicePort();
}
```

Stub Communication Model



Referencing the stub

- In order to get an object to reference the stub you have to instantiate Service_Impl.
  - (Unfortunately this name is only recommended)

-
  ```
  Service_Impl service = new Service_Impl ();
  ```

-
  ```
  value* name = (value)service.getServicePort ();
  ```

- With this reference you can call the methods of the service.

Stub Interface (javax.xml.rpc.Stub)

```
Public interface Stub
{
   public abstract Object _getProperty (String name) throws
   JAXRPCException;
   public abstract Iterator  _getPropertyNames ();
   public abstract void _setProperty(String name, Object
   value) throws    JAXRPCException;
}
```

These methods allow the stub to be configured by setting various properties.

Stub configuration

Property name	type	description
ENDPOINT_ADDRESS_PROPERTY	String	Address of the service to connect
SESSION_MAINTAIN_PROPERTY	Bool	Whether to enter and maintain session – default false
USERNAME_PROPERTY PASSWORD_PROPERTY	String	Authentication required for HTTP

Server side Implementation

Deploying to a web container

- Create a WAR file
  - Java class file for service endpoint interface
  - Java class files for service implementation and resources
  - `web.xml` file containing deployment information
  - Class files for JAX-RPC tie classes

- JAX-RPC tie classes are implementation specific.

Deploying with JWSDP - Tomcat

Additional WAR files required for JWSDP

`WEB-INF/web.xml`	Web application deployment descriptor
`WEB-INF/jaxrpc-ri.xml`	JWSDP-specific deployment information
`WEB-INF/model`	Model file generated by `wscompile`

web.xml file

```
<?xml version=“1.0” encoding=“UTF-8” ?>

<!DOCTYPE web-app
   PUBLIC “-//Sun Microsystems, Inc.//DTD Web Application
   2.3//EN”
   “http://java.sun.com/j2ee/dtds/web-app_2_3.dtd”>


<web-app>
   <display-name>Service Name</display-name>
   <description>A web service application</description>
</web-app>
```

Creating a deployable WAR file

```
wsdeploy -o targetFileName portableWarFileName
```

The process is informed by the content of the `jaxrpc-ri.xml` file.

The archive contains:

class files and resources

compiled class files for the ties

compiled class files for serializers

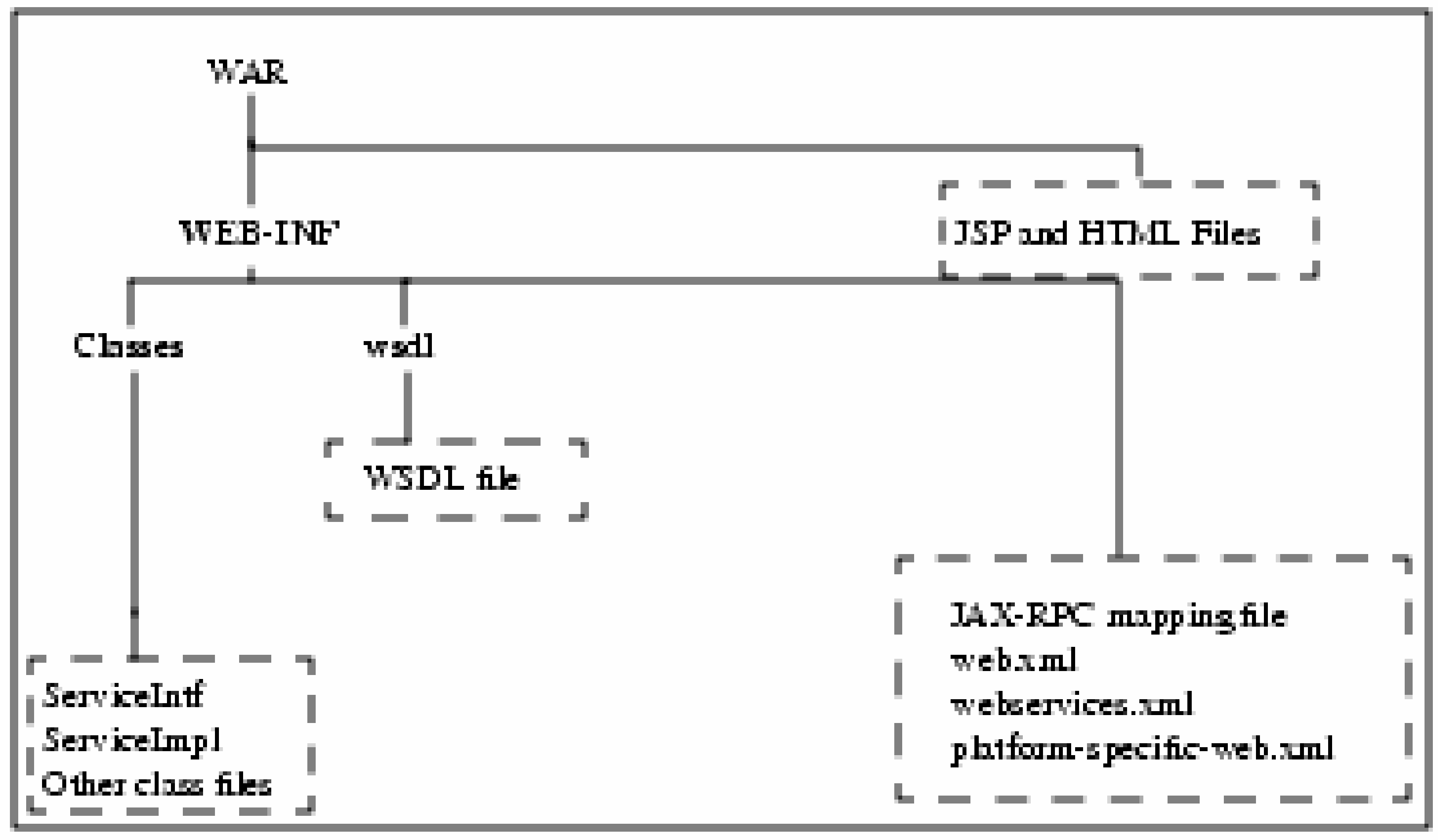
WSDL (in WEB-INF directory)

model file for the service (in WEB-INF)

modified `web.xml` file

`jaxrpc-ri-runtime.xml` (based on `jaxrpc-ri.xml`)

Package Structure for JAX-RPC Service Endpoint



Modified web.xml

```
<?xml version=“1.0” encoding=“UTF-8”?>
<!DOCTYPE web-app PUBLIC “-//Sun Microsystems, Inc. //DTD Web Application 2.3//EN”
    “http://java.sun.com/j2ee/dtds/web-app_2_3.dtd”>
    <web-app>
          <display-name> Service name </display-name>
          <description>……………………….</description>
    <listener>
          <listener-class>com.sun.xml.rpc.server.http.JAXRPCContextListener
          </listener-class>
    <listener>
    <servlet>
          <servlet-name>Servlet</servlet-name>
          <display-name>Servlet.</display-name>
          <description>………………....</description>
          <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet</servlet-class>
          <load-on-startup>1</load_on_startup>
    </servlet>
    <servlet-mapping>
          <servlet-name>Servlet</servlet-name>
          <url-pattern>/Servlet</url-pattern>
    </servlet-mapping>
</web-app>
```

jaxrpc-ri.xml file

```
<?xml version=“1.0” encoding=“UTF-8”?>
<webServices xmlns=“http://java.sun.com/xml/ns/jax-rpc/ri/dd”
   version=“1.0”
   targetNamespaceBase=“ {WSDL file location} ”
   typeNamespaceBase=“ {types} “>

   <endpoint name =“Servicename”
        displayname=“Servicename Port”
        description=“………………..”
        model=“/WEB-INF/model”
        interface=“ classpath “
        implementation=“ classpath “/>
   <endpointMapping>
        endpointName=“Service”
        urlPattern=“ /Service “/>
</webServices
```

May contain any number of `endpoint` elements and any number of `endpointMapping`

The file is private to `JAX-RPC` and you don’t need to edit it

Using JAX-RPC to create a service from a WSDL definition

- WSDL is an interface definition

Getting the WSDL

- WSDL can be downloaded from a UDDI registry

- If the service uses `JAXRPCServlet` you can attach `?WSDL` (or `?model`) to the URL request to get the WSDL (or model file).
  - Eg `http://localhost:8080/Service/Servicename?WSDL`

A config.xml file

```
<?xml version=“1.0” encoding=“UTF-8”?>

<configuration xmlns=“http://java.sun.com/xml/ns/jax-
  rpc/ri/config”>

  <wsdl
  location=“http://localhost:8080/Service/Servicename?W
  SDL” packageName=“example.wsdlexample.servicename”/>

</configuration>
```

Format of config file depends on whether `wscompile` is given a WSDL file, model file or Java

Generate client side artifacts

```
wscompile -gen:client -keep -s
  generated/client -d output/client -classpath
  classpath config.xml
```

J2EE 1.4 bug

In some versions of J2EE 1.4 generated WSDL files contain errors in the <soap:address> definitions tag and have to be manually edited.

Eg. http://localhost:8080//Service/Servicename

Which would have to be edited to

http://localhost:8080/Service/Servicename

Some of the client side files generated by wscompile from WSDL

Service	Service.java
	Service_Impl.java
	Service_SerializerRegistry.java
Exception	ServiceException.java
	ServiceException_SOAPSerializer.java
	ServiceException_SOAPBuilder.java
Value type	Info.java
	Info_SOAPSerializer.java
	Info_SOAPBuilder.java
Interface	Interface_Stub.java
	method.java

Stub interface

```
Service_Impl service = new Service_Impl ();


Object name = (Object)service.getServicePort();
Info[] name = Service.getServiceInfo();
```

The web service address is preconfigured using information from the WSDL `<soap:address>` element within the service's `<port>` element for its `portType.`

J2EE client

- J2EE allows container-resident clients to get references to Service objects defined in the JNDI environment.
- So code can be vendor independent
- The client has to be packaged in a JAR file to be deployed.

JAR application client entry

- To create the entry in the JNDI environment you include a `webservicesclient.xml` file in the JAR

- This file resides in the `META-INF` directory

webservicesclient.xml file

```
<?xml version=“1.0” encoding=“UTF-8”?>
<!DOCTYPE webservicesclient PUBLIC
   “-//IBM Corporation, Inc//DTD J2EE Web services client
   1.0//EN”
   “http://www.ibm.com/standards/xml/webservices/j2ee/j2ee_web
   _services_client_1_0.dtd”>
<webservicesclient>
   <service-ref>
       <description>……………</description>
       <service-ref-name>service/Service</service-ref-name>
       <service-interface>classpath</service-interface>
       <wsdl-file>Filename.wsdl</wsdl-file>
       <jaxrpc-mapping-file>META-INF/model</jaxrpc-mapping-
   file>
   </service-ref>
<webservicesclient>
```

Elements

- `<service-ref>` defines the reference to the web service
- `<service-ref-name>` defines where the reference appears in the JNDI relative to java:comp/env
- `<service-interface>` fully qualified path to the generated class
- `<wsdl-file>` location of WSDL file relative to the root of the JAR file.
- `<jaxrpc-mapping-file>` mapping of WSDL definition to java service endpoint interface

Generation

- The information in the webservicesclient.xml file is read by the deployment tools.

- These generate a class which implements the Service interface

- They also generate the client side stubs which the application will call.

Obtaining a Service object

```
InitialContext ctx = new InitialContext ();
Object service = (object)PortableRemoteObject.narrow
   (ctx.lookup (“java:comp/env/service/Service”),
   object.class);

Object name = (object)service.getServicePort();

((Stub)name)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
   args[0]);
```

wscompile

You can use the information in a config.xml file which specifies a WSDL definition to generate the classes required for the service:

```
wscompile -import -f:norpcstructures -d
  output/interface config.xml
```

`-f:norpcstructures` – avoids generating SOAP message creation classes.

Files required in the JAR

File type	Filename
Service end point interface	*Classpath.service.name*
	Classpath.service.Info
	Classpath.service.Exception
Service interface	*Classpath.service.Service*
Application implementation	*Classpath.client.ServiceAppClient*
WSDL file	*Service.wsdl*
Deployment descriptors	*META-INF/application-client.xml*
	META-INF/mapping.xml or META-INF/model
	META-INF/webservicesclient.xml
Manifest file	*META-INF/MANIFEST.MF*

JAR file uses

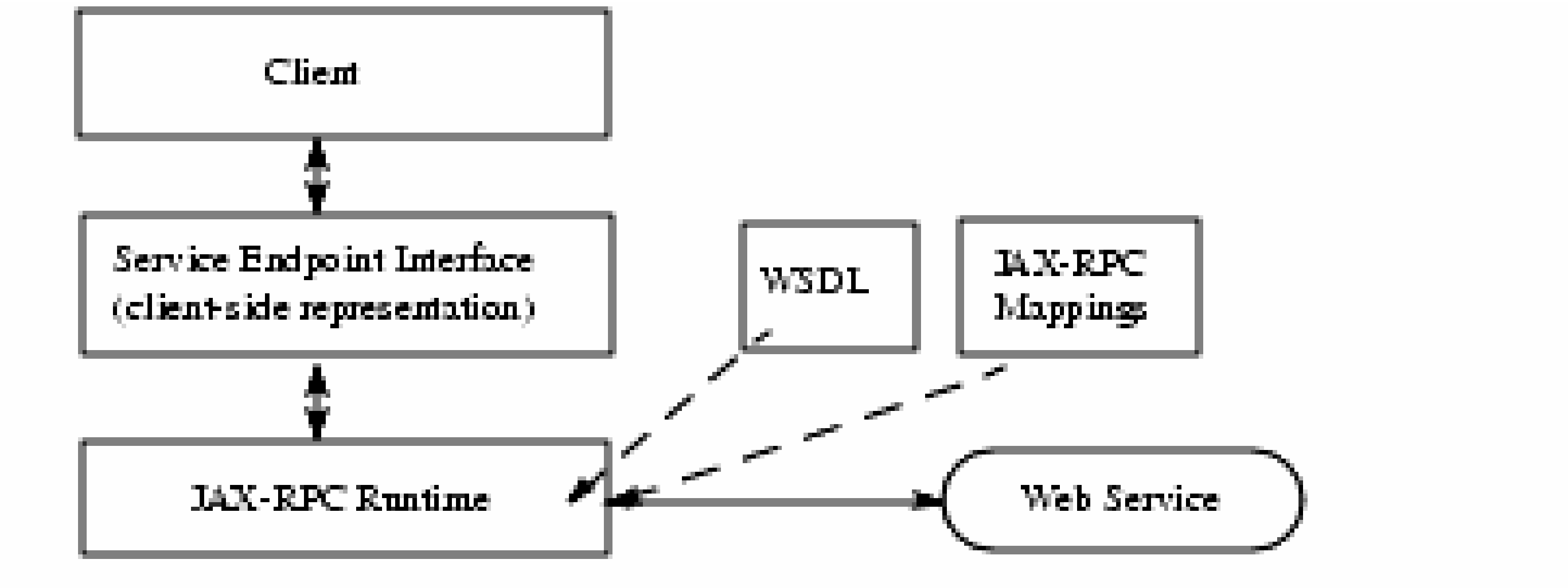
- Deployment to the server to create stubs
- Source for class files for application client

After deployment

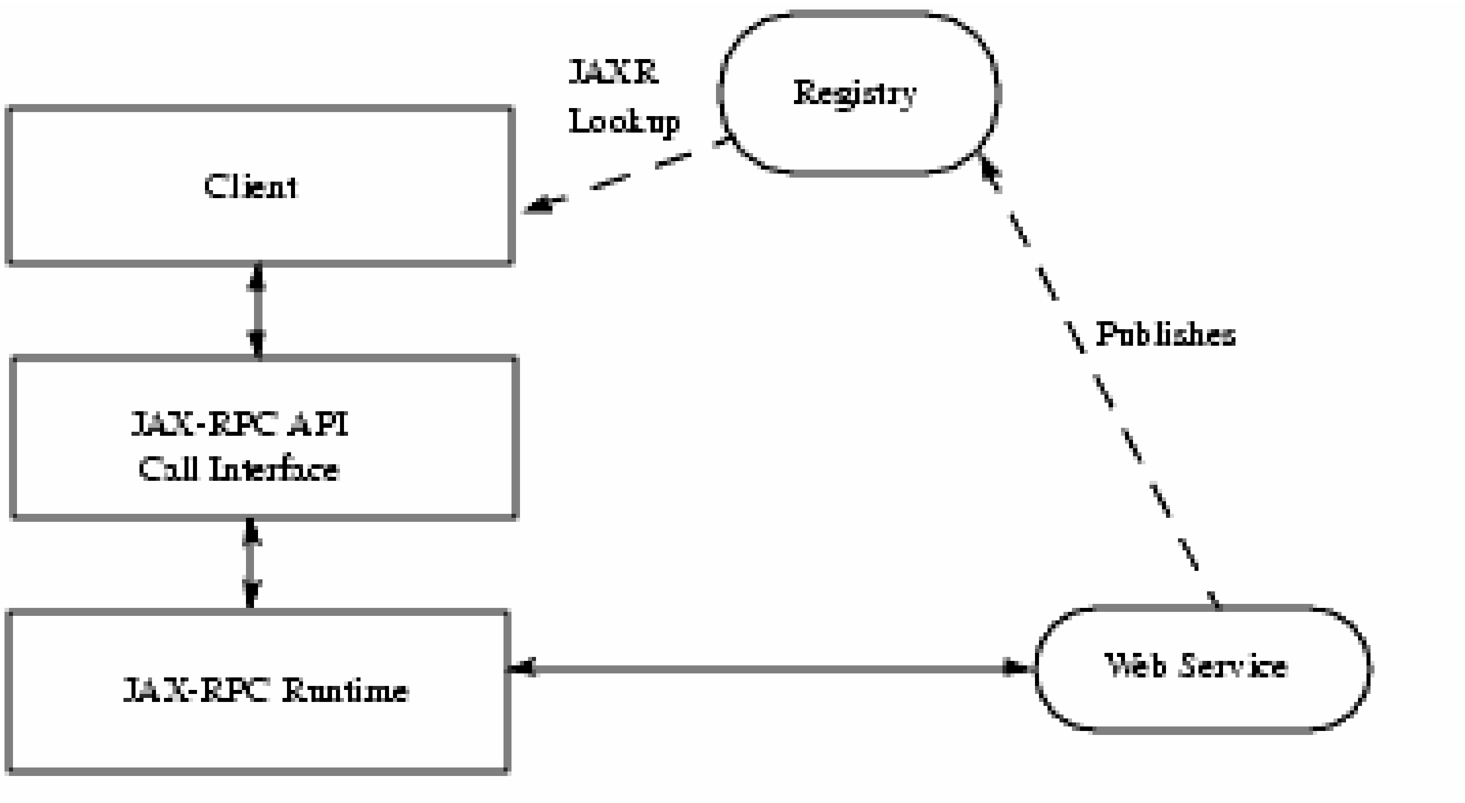
- Generated stubs are written to a file called `stubs.jar`
- The JAR also has a file called sun-j2ee-ri.xml

Accessing a Service Using a Dynamic Proxy

Enabling Grids for E-science in Europe



DII Call Interface


JAXR
Lookup
Registry
Client
Publishes
JAX-RPC API
Call Interface
JAX-RPC Runtime
Web Service