Griglie e Sistemi di Elaborazione Ubiqui

Corso di Laurea Specialistica
in Ingegneria informatica

Lucidi delle Esercitazioni

Anno Accademico 2005/2006

Ing. Antonio Congiusta

Summary

- ✓ 5 steps to writing a service
- ✓ Compilation and GAR generation
- ✓ Hello World example
- ✓ MathService example
- ✓ WSDL details
- ✓ Java implementation details

Developing GT4 Web Services: 5 'easy' steps

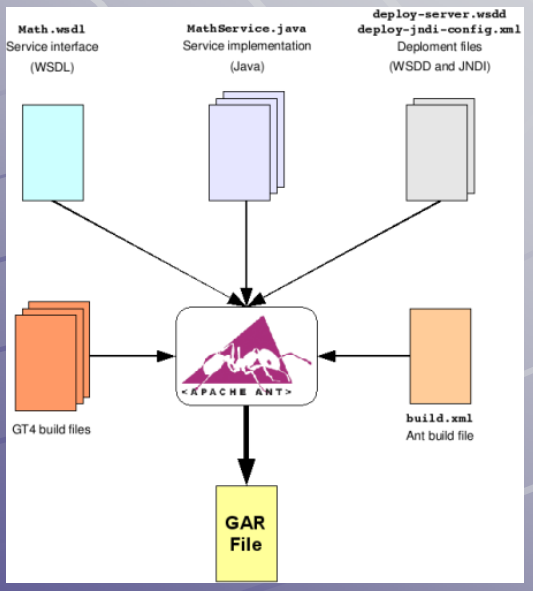
- ✓ Step 1. Define the service's interface.
  - Must write a Web Service Description Language (WSDL) file describing the service's abstract interface (must be done by hand).
- ✓ Step 2. Implement the service.
  - Develop Java routines for the service implementation
  - If WSRF mechanisms are used, also for associated resource properties.
- ✓ Step 3. Define deployment parameters:
  - Write a Web Services Deployment Descriptor (WSDD) and a JNDI file that describes various aspects of the service's configuration.
- ✓ Step 4. Compile everything and generate a GAR file.
  - Compilation generates application specific interface routines that handle the demarshalling/marshalling of the Web service's arguments from/to SOAP messages.
- ✓ Step 5. Deploy the service.

Create a GAR file with Ant

However, creating a GAR file is a pretty complex task which involves the following:

- Processing the WSDL file to add missing pieces (such as bindings)
- Creating the stub classes from the WSDL
- Compiling the stubs classes
- Compiling the service implementation
- Organize all the files into a very specific directory structure

Create a GAR file with Ant



The `globus-build-service` script and buildfile

- ✓ `./globus-build-service.sh` -d <service base directory> -s <service's WSDL file>
- ✓ The "service base directory" is the directory where we placed the deploy-server.wsdd file, and where the Java files can be found (inside a `src` directory).
- ✓ If everything works fine, the GAR file generated will be something like:
  $EXAMPLES_DIR/org_globus_examples_HelloWorld.gar

Deploy the service into a GT4 container

- ✓ Deployment is done with a GT4 tool that, using Ant, unpacks the GAR file and copies the files (WSDL, compiled stubs, compiled implementation, WSDD) into appropriate locations in the GT4 directory tree.
- ✓ This deployment command must be run with a user that has write permission in $GLOBUS_LOCATION.

  globus-deploy-gar $EXAMPLES_DIR/org_globus_examples_HelloService.gar
- ✓ There is also a command to undeploy a service:

  globus-undeploy-gar org_globus_examples_HelloService

A Simple Client

```
public static void main(String[] args) {
   HelloServiceAddressingLocator locator = new
   HelloServiceAddressingLocator();
   try {
        String serviceURI=args[0];
        EndpointReferenceType endpoint = new
                                   EndpointReferenceType();
        endpoint.setAddress(new Address(serviceURI));
        HelloWorldPortType myPort =

        locator.getHelloWorldPortTypePort(endpoint);
        // Perform some operations
        GetResourcePropertyResponse response =
            port.getResourceProperty(HelloQNames.RP_NAME);
        String result = myPort.echo("First invocation");
        response =
            myPort.getResourceProperty(HelloQNames.RP_NAME);
            System.out.println(AnyHelper.toSingleString(
                     response));
            ...      } catch (Exception e)
            {e.printStackTrace(); } }
```

A Simple Client

- ✓ First, we create an **EndpointReferenceType**
- ✓ Next, we obtain a reference to the service's portType
- ✓ Once we have that reference, we can work with the web service *as if it were a local object*. For example, to invoke the *remote* **echo** operation
- ✓ Finally, notice how all the code must be placed inside a **try/catch** block. We must always do this, since all the remote operations can throw **RemoteException**s (for example, if there is a network failure and we can't contact the service).

A Simple Client

- ✓ We are now going to compile the client. Before running the compiler, make sure you run the following:

```
source $GLOBUS_LOCATION/etc/globus-devel-env.sh
```

  The **globus-devel-env.sh** script takes care of putting all the Globus libraries into your CLASSPATH. When compiling the service, Ant took care of this but, since we're not using Ant to compile the client, we need to run the script.

- ✓ To compile the client, do the following:

```
javac \ -classpath ./build/stubs/classes/:$CLASSPATH \
org/globus/examples/clients/HelloService_instance/Cli
ent.java
```

- ✓ **./build/classes** is a directory generated by Ant were all the compiled stub classes are placed. We need to include this directory in the CLASSPATH so our client can access generated stub classes such as **HelloServiceAddressingLocator**.

Running the Container

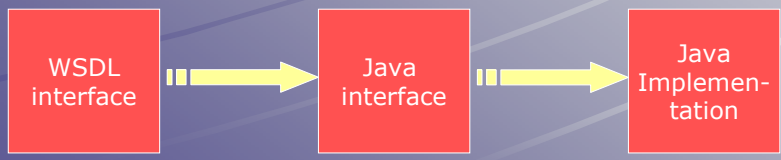
- Now, before running the client, we need to to start up the standalone container. Otherwise, our web service won't be available, and the client will crash.
  globus-start-container -nosec
- When the container starts up, you'll see a list with the URIs of all the deployed services.
  http://127.0.0.1:8080/wsrf/services/examples/HelloService
- This is the service as it would appear in a default GT4 installation, with the standalone container located in **http://localhost:8080/wsrf/services**.
- If the service is correctly deployed, we can now run the client:
- **java \ -classpath ./build/stubs/classes/:$CLASSPATH \ org.globus.examples.clients.HelloService_instance.Client \ http://127.0.0.1:8080/wsrf/services/examples/HelloService**

5 Steps to Writing a Service

- Step 1: Define the interface in WSDL
  - what operations will be available to clients
  - the service interface is called the port type (portType).
- Step 2: Implement the service in Java
  - Service
  - Service resource
  - Service home
- Step 3: Configure the deployment
  - WSDD (and JNDI)
- Step 4: Create a GAR file with Ant
- Step 5: Deploy the service into a Web services container

Top-down approach

- ✓ *We use the top-down approach. The WSDL contains the abstract definition of the service including types, messages and portTypes.*
- ✓ *Starting with a document/literal WSDL and then generating the Java artifacts from that leads to the most interoperability.*
- ✓ *We use some of the tools that come with the GT4 toolkit to generate the binding and stubs.*



Tutorial Example: MathService

- ✓ Supports the operations:
  - Add
  - Subtract
- ✓ Has the following *properties*
  - Value (integer)
  - Last operations performed (string)
- ✓ On initialization:
  - Value set to zero
  - Last op set to NONE

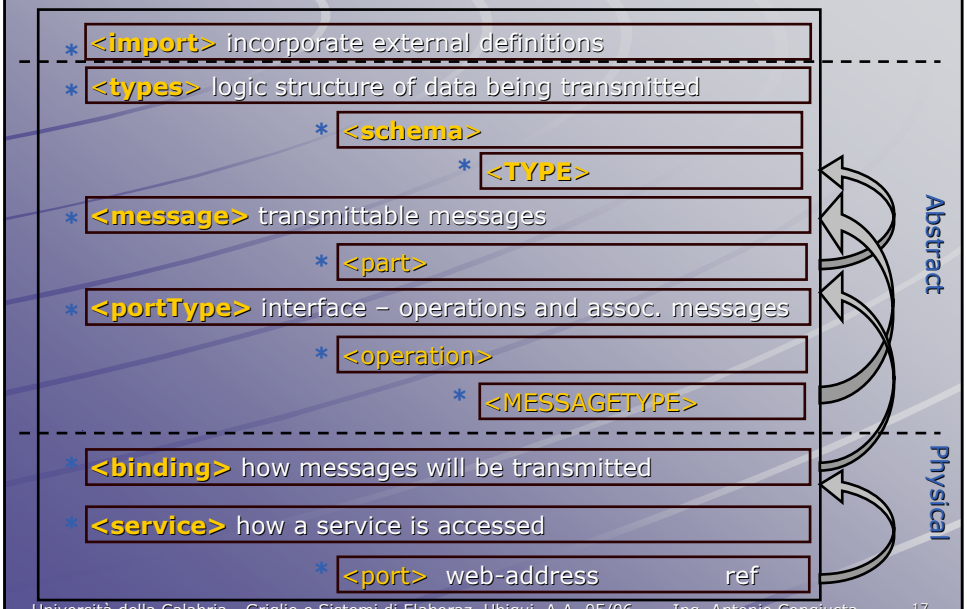
Step 1: Define the interface in WSDL: Two options

- ✓ **Writing the WSDL directly.**
  - This is the most versatile option.
  - Total control over the description of our portType.
- ✓ **Generating WSDL from a Java interface.**
  - The easiest option, but very complicated interfaces are not always converted correctly to WSDL

The *interface* in Java

```
public interface Math {
  public void add(int a);
  public void subtract(int a);
  public int getValueRP();
}
```

WSDL structure



Steps to WSDL: key TAGS

- ✓ Write the root element **<definitions>**
- ✓ Write the **<PortType>**
  - It has 2 **resource properties**
- ✓ Write an input and output **<message>** for each operation in the PortType.
- ✓ Write the **<types>**

Root Element -- <definitions>

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathService"
   targetNamespace="http://www.globus.org/namespaces/2004/02/prog
   tutorial/MathService"
   xmlns:tns="http://www.globus.org/namespaces/2004/02/progtutori
   al/MathService"
   xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
   xmlns:gwsdl="http://www.gridforum.org/namespaces/2003/03/gridW
   SDLExtensions"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns=http://schemas.xmlsoap.org/wsdl/
   ...
/definitions>
```

- ✓ <definitions> TAG as 2 important properties:
  - name: the '**name**' of the **WSDL file**. Not related with the name of the PortType
  - targetNamespace: The target namespace of the GWSDL file.

<import …>

- ✓ TAG used to import WSDL files
- ✓ You don't have to import all WSDL files, just the ones you plan to use

```
<import location="../../wsrf/properties/WS-
   ResourceProperties.wsdl"
   namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-
   WS-ResourceProperties-1.2-draft-01.wsdl"
/>
```

Write the PortType

```
<definitions ... >
<portType name="MathPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty"
    wsrp:ResourceProperties="tns:MathResourceProperties">
   <operation name="add">
        <input message="tns:AddInputMessage"/>
        <output message="tns:AddOutputMessage"/>
   </operation>
   <operation name="subtract">
        <input message="tns:SubtractInputMessage"/>
        <output message="tns:SubtractOutputMessage"/>
   </operation>
   <operation name="getValueRP">
        <input message="tns:GetValueRPInputMessage"/>
        <output message="tns:GetValueRPOutputMessage"/>
   </operation>
</portType>
</definitions>
```


<portType> tag important attributes

- ✓ name: name of the PortType
- ✓ wsdlpp:extends:
  - One of the *main differences* with plain WSDL.
  - Part of preprocessor provided by *Globus*
  - Allow definition of PortType as an extension of an existing PortType.
  - In this example we tell preprocessor to include GetResourceProperty and ImmediateResourceTermination

<portType> tag important attributes

- ✓ wsrp:ResourceProperites:
  - Specifies service resource properties
- ✓ An <operation> tag for each method in the PortType
  - Operation tag has an **input** tag, an **output** tag, and a **fault** tag
  - Input/output tags have a message attribute, which specifies what message should be passed along when the operation is invoked and returns

Write input and output messages for each port type

```
<message name="AddInputMessage">
   <part name="parameters" element="tns:add"/>
</message>
<message name="AddOutputMessage">
   <part name="parameters" element="tns:addResponse"/>
</message>
```

- ✓ Messages are composed of **parts** – these messages have one **part**, in which a *single XML element* is passed along

Define the XML elements inside the <types> tag

The <types> tag contains an <xsd:schema> tag.

```
<types>
<xsd:schema
   targetNamespace="http://www.globus.org/namespaces/2004/02/progt
   utorial/MathService"
   attributeFormDefault="qualified"
   elementFormDefault="qualified"
   xmlns="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="add">
        <xsd:complexType>
              <xsd:sequence>
                   <xsd:element name="value" type="xsd:int"/>
              </xsd:sequence>
        </xsd:complexType>
   </xsd:element>
   <xsd:element name="addResponse">
         <xsd:complexType/>
   </xsd:element>
</xsd:schema>
 </types>
```


Multiple XML elements inside <types> tag

```
<types>
<xsd:schema
   targetNamespace="http://www.globus.org/namespaces/2004/02/p
   rogtutorial/MathService"
   attributeFormDefault="qualified"
   elementFormDefault="qualified"
   xmlns="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="add">
        <xsd:complexType>
              <xsd:sequence>
                   <xsd:element name="value1" type="xsd:int"/>
                   <xsd:element name="value2" type="xsd:int"/>
                        </xsd:sequence>
        </xsd:complexType>
   </xsd:element>
   <xsd:element name="addResponse">
         <xsd:complexType/>
   </xsd:element>
</xsd:schema>
 </types>
```

Declaring the resource properties

- ✓ Example slide:
  - declaration specifies that **MathResourceProperties** contains two resource properties, Value and LastOp, each of which appear only once (we could specify array resource properties by changing the values of **maxOccurs**).

- ✓ **It is important to notice → No bindings**: Bindings are an essential part of a normal WSDL file. However, we don't have to add them manually, since they are generated automatically by a GT4 tool that is called when we build the service.


```
<types>
<xsd:schema
targetNamespace="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:tns="http://www.globus.org/namespaces/examples/core/MathService_instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


        <!- Resource Properties-->

        <xsd:element name="Value" type="xsd:int"/>
        <xsd:element name="LastOp" type="xsd:string"/>

        <xsd:element name="MathResourceProperties">
        <xsd:complexType>
                <xsd:sequence>
                        <xsd:element ref="tns:Value" minOccurs="1"
maxOccurs="1"/>
                        <xsd:element ref="tns:LastOp" minOccurs="1"
maxOccurs="1"/>
                </xsd:sequence>
        </xsd:complexType>
        </xsd:element>


</xsd:schema>
</types>
```

Step 2: Implement the Service

```
package org.globus.examples.services.core.first.impl;
import java.rmi.RemoteException;
import org.globus.wsrf.Resource;
import org.globus.wsrf.ResourceProperties;
import org.globus.wsrf.ResourceProperty;
import org.globus.wsrf.ResourcePropertySet;
import org.globus.wsrf.impl.ReflectionResourceProperty;
import org.globus.wsrf.impl.SimpleResourcePropertySet;
import
   org.globus.examples.stubs.MathService_instance.AddResponse;
import
   org.globus.examples.stubs.MathService_instance.SubtractRespo
   nse;
import
   org.globus.examples.stubs.MathService_instance.GetValueRP;
public class MathService implements Resource ,
   ResourceProperties  {


}
```


Service implementation

- ✓ Since our Java class will implement both the service and the resource, we need to implement the Resource interface. However, this interface doesn't require any methods. It is simply a way of tagging a class as being a resource.
- ✓ By implementing the ResourceProperties interface we are indicating that our class has a set of resource properties which we want to make available. This interface requires that we add the following to our class:

```
private ResourcePropertySet propSet;
public ResourcePropertySet getResourcePropertySet() {
  return this.propSet;
}
```

Public class MathService

```
public class MathService implements Resource, ResourceProperties {
    /* Resource Property set */
    private ResourcePropertySet propSet;

    /* Resource properties */
    private int value;
    private String lastOp;

    /* Get/Setters for the RPs */
    public int getValue() {
          return value;
    }

    public void setValue(int value) {
          this.value = value;
    }

    public String getLastOp() {
          return lastOp;
    }

    public void setLastOp(String lastOp) {
          this.lastOp = lastOp;
    }
    /* Required by interface ResourceProperties */
    public ResourcePropertySet getResourcePropertySet() {
          return this.propSet;
    }
}
```


The QNames interface

When we have to refer to just about anything related to a service, we will need to do so using its *qualified name*, or QName for short.

This is a name which includes a *namespace* and a *local name*. For example, the QName of the Value RP is:

{http://www.globus.org/namespaces/examples/core/MathService_instance}Value

A qualified name is represented in Java using the QName class

```
package org.globus.examples.services.core.first.impl; import
javax.xml.namespace.QName;
public interface MathQNames {
public static final String NS =
"http://www.globus.org/namespaces/examples/core/MathService_instan
ce";
public static final QName RP_VALUE = new QName(NS, "Value");
public static final QName RP_LASTOP = new QName(NS, "LastOp");
public static final QName RESOURCE_PROPERTIES = new QName(NS,
"MathResourceProperties");
}
```

Public class MathService

Next, we have to implement the constructor. Here we will initialize the resource properties.

```
/* Constructor. Initializes RPs */
public MathService() throws RemoteException {
     this.propSet = new SimpleResourcePropertySet(
                          MathQNames.RESOURCE_PROPERTIES);

        /* Initialize the RP's */
        try {

                ResourceProperty valueRP = new
ReflectionResourceProperty(
                                    MathQNames.RP_VALUE, "Value", this);
                this.propSet.add(valueRP);
                setValue(0);

                ResourceProperty lastOpRP = new
ReflectionResourceProperty(
                                    MathQNames.RP_LASTOP, "LastOp", this);
                this.propSet.add(lastOpRP);
                setLastOp("NONE");
        } catch (Exception e) {
                throw new RuntimeException(e.getMessage());
        }
```


Operation implementation

```
public AddResponse add(int a) throws RemoteException {
  value += a;
  lastOp = "ADDITION";
return new AddResponse();
}
```

You'll notice that, even though we defined the add operation as having no return type in the WSDL file, now the return type is AddResponse. A similar thing happens in the getValueRP method:

```
public int getValueRP(GetValueRP params) throws RemoteException
{
   return value;
}
```

Boxed parameters

The parameters and the return values will *in some cases* be 'boxed' inside stub classes (which are generated automatically from the WSDL file). This is more evident when we have several parameters. For example, if we declared the following operation in our WSDL file:

```
void multiply(int a1, int a2);
```

The actual Java code would look like this:

```
public MultiplyResponse multiply(Multiply params) throws
RemoteException {
int a1 = params.getA1()
int a2 = params.getA2()
// Do something
return new MultiplyResponse(); }
```

Multiply and MultiplyResponse are stub classes. Notice how the two parameters (a1 and a2) are 'boxed' inside a single Multiply parameter, and how we return a MultiplyResponse object, even though we don't really want to return anything.

Question Time

? ? ? ? ? ?

The more you ask…

…the less I question you!