WebSphere software

IBM

e business software

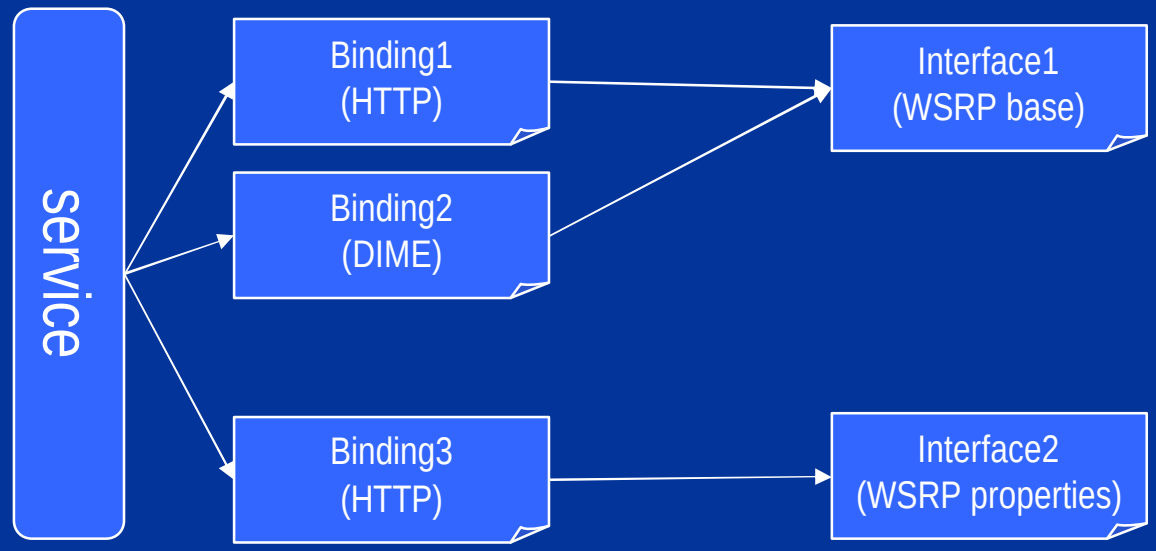
How to create proxies for WSRP (08/27/2002)

Carsten Leue

IBM Software Group

Producer

- One WSRP service may implement one or more different interfaces
- Each interface is defined by an interface WSDL
- Each binding is defined by a binding WSDL that imports the interface WSDL

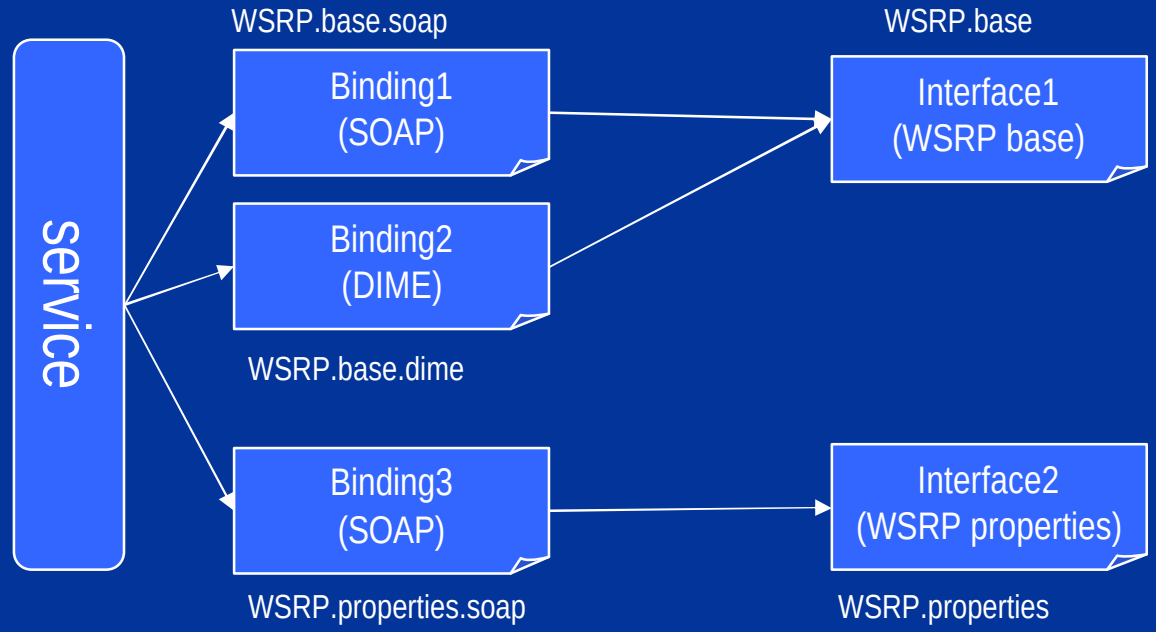


Discovery

- For each service a consumer can discover all exposed bindings and all exposed interfaces
  - Via UDDI as tModelInstanceInfos
  - Via self-description of the service
- The consumer needs to filter all compatible interfaces and bindings to connect to the producer
- How can the consumer find out if a binding or interface is the expected one?
  - By analyzing the WSDL -> tedious and error prone
  - The WSDL files can be named uniquely so interface and binding can be defered from the name (next slide)

Naming

- Define a name that contains the protocol, binding and interface
- WSRP.<interface>.<binding>
- The publisher of the WSDL MUST ensure that the content of the WSDL matches the name
- Naming should include a version number



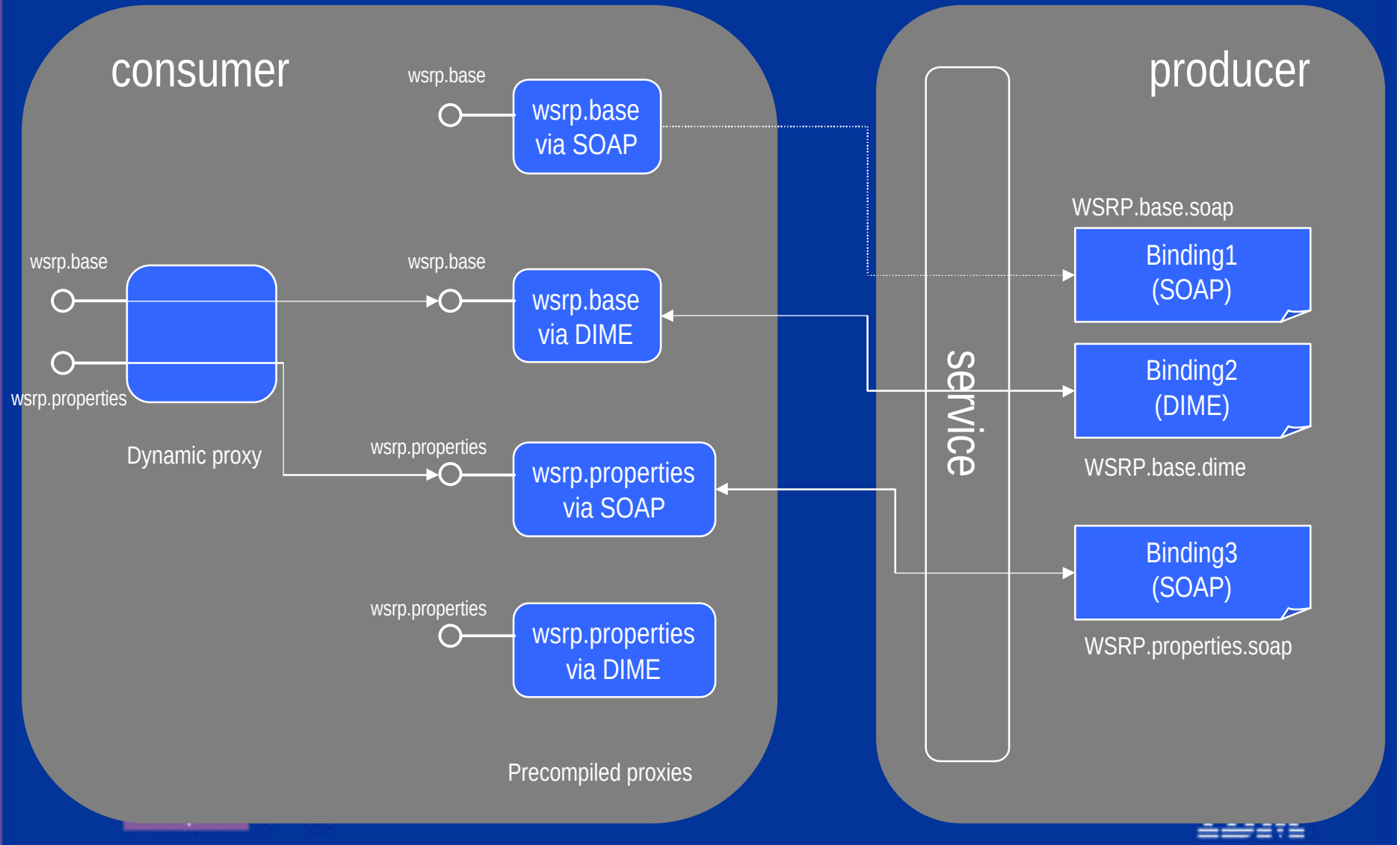
How a consumer can attach using precompiled proxies

- The consumer finds out about all bindings and interfaces the service exposes
- The consumer filters this set by supported interfaces. Per definition the producer must at least expose the WSRP.base interface
- The consumer filters the resulting set by supported bindings. Per definitions all producers must expose SOAP binding, so the resulting list will contain at least one binding for each interface
- The consumer sorts the list of bindings per interface by prefered bindings (e.g. DIME > SOAP-Attachments > SOAP)
- For each interface the consumer selects the best binding

How a consumer can attach using precompiled proxies

- The consumer instantiates a precompiled WSRP-proxy for each binding/interface and points it to the correct access point
- In the JAVA case the consumer now instantiates a java.lang.ref.proxy object and makes it expose all discovered and supported interfaces
- The consumer delegates each method invokation to the appropriate WSRP-proxy (e.g. by managing a map between method object and WSRP-proxy)

Object Model



Implementation Hints

- How to implement support for multiple WSDL factors using AXIS?
  - No hacking
  - Keep session across factors
- Example
  - Assume that the service exposes the two factors Additive and Substractive as defined below

```
interface Additive {
    public int add(int a,int b);
}

interface Substractive {
    public int sub(int a,int b);
}
```

Step1

- Compile the JAVA interfaces
- Generate WSDL from JAVA interface
  - `java org.apache.axis.wsdl.Java2WSDL -w Interface -o Additive.wsdl com.wsrp.Additive`
  - `java org.apache.axis.wsdl.Java2WSDL -w Interface -o Substractive.wsdl com.wsrp.Substractive`
- Result
  - two interface WSDLs (`Additive.wsdl` and `Substractive.wsdl`)

Step2

- Generate Proxies (Javaspeak = Stubs) for each WSDL factor
  - `java org.apache.axis.wsdl.WSDL2Java -v -o .\ Additive.wsdl`
  - `java org.apache.axis.wsdl.WSDL2Java -v -o .\ Substractive.wsdl`

- Result
  - two helper files for each interface WSDL
    - `Additive.java` that defines the JAVA interface
    - `AdditiveSoapBindingStub.java` that defines the code to call the methods of the interface via SOAP

Interim Result

- Results
  - Interfaces for each WSDL
  - Separate proxies for each interface
- What we want
  - One single java object that exposed all implemented interfaces via java typecasts
  - Make sure that the different factors use one single session

Step3.1

- Handcraft your dynamic proxy
  - Extend `org.apache.axis.client.Service` to manage a shared call object
  - Implement `java.lang.reflect.InvocationHandler` to serve java's dynamic proxy

```
public class TestProxy extends Service
   implements InvocationHandler {
```

Step3.2

- Handle a common call object for all factors (so there will only be one session)

```
// reuse the call object for all proxies
private Call call;

// override from Service
public javax.xml.rpc.Call createCall() throws ServiceException
{
 // lazy initialization
 if (call==null)
   call = (Call)super.createCall();
 // reuse the call object
 call.removeAllParameters();
 call.setMaintainSession(true);
 return call;
}
```

Step3.3

- Provide some logic to dispatch incoming calls to the proxies

```
// map the interface methods against implementation classes
Map methodMap = new HashMap();

/* find out about the supported methods of an interface and
   map against the implementing proxy
*/
private void fillMethods(Class cls,Object handler)
{
  Method[] m = cls.getMethods();
  for (int i=m.length-1; i>=0; --i)
    methodMap.put(m[i],handler);
}

// simply dispatch an incoming method call
public Object invoke(Object arg0, Method arg1, Object[] arg2)
throws Throwable
{
  return arg1.invoke(methodMap.get(arg1),arg2);
}
```

Step3.4

- Initialize our handler with all supported factors and the respective proxies. In the real world this information would be discovered at runtime from the metadata or UDDI

```
Class[] ifaces = new Class[2];

private TestProxy(URL url) throws ServiceException, AxisFault
{
  // some reflection
  fillMethods(Additive.class,
    new AdditiveSoapBindingStub(url,this));
  fillMethods(Substractive.class,
    new SubstractiveSoapBindingStub(url,this));

  // supported interface
  ifaces[0] = Additive.class;
  ifaces[1] = Substractive.class;
}
private Class[] getInterfaces() {return ifaces;}
```

Step3.5

- Instantiate the proxy

```
// create the proxy object
static public Object createProxy(URL url) throws
ServiceException,
  AxisFault
{
  // the handler
  TestProxy proxy = new TestProxy(url);
  // create the java proxy
  return Proxy.newProxyInstance(
    proxy.getClass().getClassLoader(),
    proxy.getInterfaces(),
    proxy
  );
}
```

Step4

- Use the proxy
  - Once initialized the resulting meta-proxy can be typecast to all implemented java interfaces
  - Calls to the meta-proxy will automatically be dispatched to the implementation proxy
  - There will only be one single session

```
Object o = TestProxy.createProxy(new URL(„http:...“));

Additive a = (Additive)o;
int plus = a.add(1,2);

Substractive s = (Substractive)o; // or (Substractive)a
int minus = s.sub(1,2);
```