Middleware Programming

Robin Sharp
Informatics and Mathematical Modelling
Technical University of Denmark
Phone: (+45) 4525 3749
e-mail: robin@imm.dtu.dk
More complex C/S Systems

- Internet mail and browser applications are simple examples. In more complex cases:
  - The client may need to search for a server which offers the required functionality. *Need a lookup service in the network.*
  - Client and server may need to exchange arbitrary data in a more efficient way than by using messages in ASCII text. *Need an efficient platform-independent manner of representing arbitrary data.*
  - The client may need to authenticate itself to the server. *Need suitable authentication and key distribution mechanisms.*

- Such common facilities are often introduced by using a middleware layer between T- and A-layers.
Middleware

- Offers **common facilities** for supporting applications.
- Allows the applications to be implemented in a manner which is **independent** of the underlying platform.
- Examples of types of middleware:
  - **Remote Procedure Call (RPC).** Replaces explicit exchange of messages by procedure-call-like construction (e.g. SunRPC).
  - **Remote Object Invocation (ROI).** Activates methods on objects located on remote systems, as in RMI, DCE.
  - **Message-oriented Middleware (MOM).** Exchange of synchronous messages, as in MPI, MQSeries.
  - **Stream-oriented Communication.** Intended to support exchange of continuous media (audio, video, ...).
RPC, ROI and CORBA
Remote Procedure Call (RPC)

- A simple program interface for client/server interaction.
- Looks to the client application like a procedure call.

**Stubs** on client and server sides deal transparently with exchange of messages when procedure is called.
RPC stubs

- Stubs offer both calling and called procedure the same view of the interface as for a local procedure.
- Stubs are compiled from description of the interface in a suitable **Interface Definition Language (IDL)**.
- Description specifies **types** and **directions of flow** for the parameters of the procedure.
RPC stubs (2)

- Compiled stub code in the **client** or **server**:
  - **Marshals** data for transfer to other party: Values are put into a serial (linear) representation in a buffer.
    Not all types of parameter can be used:
    - The **type** must be **serialisable**.
    - **Pointer types** may need special care (hand coding?)
  - **Unmarshals** data received from other party: Data structures are built up again from linear transfer representation.
    (Possibly) exchanges security info. **authenticating** client to server and vice versa.

- Usually one stub procedure for each procedure in the interface. A **dispatcher** on server side parses incoming messages and passes them to appropriate stub procedure.
RPC Semantics

- Not quite true that RPC offers same behaviour as a local procedure call: *Messages can get lost in transfer or at either end.*

- RPC systems offer different guarantees for execution:
  - **Exactly-once:** As for local procedure call. Called procedure is executed exactly once for each call.
    
    *Difficult to guarantee in a distributed system!*
  
  - **At-most-once:** Called proc. executed once or not at all.
  
  - **At-least-once:** Called proc. executed one or more times.
  
  - **Maybe:** No guarantees.

- Note that:
  - **Maybe, At-least-once** may give repeated execution.
  
  - **Maybe, At-most-once** may mean proc. is *not executed.*
Binding

- In an RPC/ROI system, **server** must be associated with an **identifier** used for lookup by **client**.
- Mapping \( \text{id} \rightarrow \text{server} \) is stored in a **registry**.
- Typical operations/methods needed:
  - **bind**: Register service interface and associate it with network name or URI.
  - **rebind**: Associate new service with an already registered name or URI.
  - **unbind**: Remove info. about a service interface with a given name or URI.
  - **lookup**: Obtain reference to service interface with a given name or URI. (Often includes **client binding**: import client stub code, authenticate client and server to one another)
Remote Object Invocation (ROI)

- The OO analogue of RPC, but more tricky because of need to pass references to remote objects.
- On binding to a remote object, the client imports an object proxy from the server.
- Remote references specify server name and path to the object (or to the proxy code).
- Proxy offers same interface to client as the remote object would, and contains code for marshalling, checking security, etc.
- As with RPC, not all object types can be marshalled; they must be serialisable.
Typical ROI system architecture:
Java RMI

- Example of an ROI middleware system.
- User program (as client) can invoke methods of a remote object (as server).
- Remote object implements a Java remote interface (which extends the Remote interface).
- Remote object is identified by a URL specifying server host name and path to object code.
- Bytecode for object proxy is imported to client when client binding takes place.
Example:

```java
public interface RemoteTarget extends Remote {
    public void start(int n)
        throws java.rmi.RemoteException;

    public int add(int i)
        throws java.rmi.RemoteException;

    public void stop()
        throws java.rmi.RemoteException;
}
```

- Must extend `Remote` interface.
- All methods must be declared as raising `java.rmi.RemoteException`.
Java remote object

- Must implement the given `remote interface`.
- Must extend a suitable `RemoteObject` subclass.
- Constructor must throw `RemoteException`.

```java
public class Target extends UnicastRemoteObject
    implements RemoteTarget

private int bcount;
private boolean active;

public Target() throws RemoteException
{ super(); }

public void start(int n) ...
public int add (int i) ...
public void stop () ...  
```

Implementations of methods of the remote interface
Main method of remote object class must:
- Create an object of the class.
- Register object with RMI registry, so clients can find it.

```java
public static void main(String args[]){
  try
  {
    Target t = new Target();
    String url = "//localhost/Target";
    Naming.rebind(url, t);
    System.out.println( "Bound server at "
                      + url + " to registry" );
  }
  catch(Exception e){ e.printStackTrace(); }
}
```
Java client for remote object

- Looks up object in registry to get reference to remote interface.
- Calls remote methods using reference.
- May need suitable security manager + permissions.

```java
public static void main(String args[]) {
    String server = "sss";
    try {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager( ... );
        }
        String url = "/" + server + "/Target";
        RemoteTarget rTarget = (RemoteTarget) Naming.lookup( url );
        rTarget.start(20); ...
    }
    catch (SecurityException e) {
        ... } catch (Exception e) {
        ... }
}
```
An RMI application: Summary

- Compile remote interface def.: `javac RemoteTarget.java`
- Compile remote object impl.: `javac Target.java`
- Compile client implementation: `javac Blipper.java`
- Compile code for stubs: `rmic Target`  
  (If remote object is a package, use full qualified package name)
  This produces `Target_Stub.class` for client stub and `Target_Skel.class` for server stub, both on server system.
- Start RMI registry: `rmiregistry &`
- Start remote object: `java Target &`
  May need to specify codebase and/or security policy file.
- Start client(s): `java Blipper &`
  If remote object is correctly registered and permissions given, client will be able to activate methods of the remote object.
CORBA

A software architecture and environment for developing and implementing distributed applications.

Developed by **Object Management Group (OMG)**.

**CORBA = Common ORB Architecture**

**ORB = Object Request Broker**: “software bus” which can connect different types of software component, possibly developed in different languages (C, Java,…)

ORB defined in terms of its **interface** -- many different implementations, sometimes giving rise to differences visible at application level.

We focus on **Java ORB**, using conventions to make implementations **portable** to other ORBs.
CORBA (2)

- System architecture:

- Familiar features: Client stub, server stub (skeleton).
- New features:
  - Object adaptor
  - Interface and implementation repositories
  - Dynamic invocation interface.
Common Object Services

A characteristic feature of CORBA, including:

- **Naming Service**: for registration of bindings between names and object references.
- **Event Service**: allows components on the ORB to register and de-register their interest in receiving particular asynchronous events.
- **Security Service**: provides security facilities, such as authentication, non-repudiation and audit trails.
- **Concurrent Service**: provides a lock manager.
- **Time Service**: provides clock synchronisation over multiple computers.
CORBA IDL

- Used to describe client/server interface, for example:

```cpp
module BlipTarget
{
  interface Blip
  {
    void start(in long n);
    long add (in long i);
    void stop ();
    oneway void shutdown();
  };
};
```

- "C++" types (IDL long <-> Java int, etc.)
- Direction of information flow indicated by "in", "out", "in out".
- Compiled to desired target language by appropriate IDL compiler.
A CORBA application: Summary

- Compile remote interface def.: `idlj Blip.idl`
- Compile server implem.: `javac BlipTarget.java`
- Compile client implementation: `javac BlipClient.java`
- Compile code derived from IDL: `javac BlipApp/*.java`

This produces class files for client & server stubs and helper classes.

- Start ORB Name Service daemon:
  `orbd -ORBInitialPort 1050 &`

- Start server:
  `java BlipTarget -ORBInitialPort 1050 -ORBInitialHost localhost &`

- Start client(s):
  `java BlipClient -ORBInitialPort 1050 -ORBInitialHost localhost &`

- Don’t forget to stop ORB name server daemon when finished!
Web services and SOAP
Another approach: SOAP

- SOAP: Simple Object Access Protocol
- Offers a way to pass arguments to an application and return results from the application via an existing Application Layer protocol (typically HTTP).
- System structure (using HTTP):

```
Web Client → Arguments in HTTP POST request → Web Server
            |          | Results in HTTP POST response |
            |          |                              |
            |          | Web Server                  |
            |          |                              |
            |          | Object                      |
```

- Arguments and results are embedded in SOAP messages encoded in XML.
SOAP messages in XML

- SOAP messages are syntactically **XML documents** with a hierarchical structure:
- E.g. for method invocation:

```xml
<?xml version="1.0"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENC:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <m:add xmlns:m="http://www.soapware.org/">
      <n xsi:type="xsd:int">3</n>
    </m:add>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
Results from invoked method are embedded in a similar manner:

```xml
<?xml version="1.0"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENC:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <m:addResponse xmlns:m="http://www.soapware.org/">
   <addresult xsi:type="xsd:int"> 27 </addresult>
  </m:addResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

... at least if the invocation terminated correctly.
If errors occur, a Fault Response message is returned:

```xml
<?xml version="1.0"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENC:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Client</faultcode>
      <faultstring>Client error.
        Too many parameters in call of "add".
      </faultstring>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
**SOAP Types**

- A subset of XML datatypes.
- As in most programming languages, fall into:
  - **Simple types**: scalar values, no internal structure.
  - **Compound types**: structured values.
- Some simple types in SOAP 1.1:

<table>
<thead>
<tr>
<th>Set of values in type</th>
<th>Type name</th>
<th>Value Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal fractions</td>
<td>xsd:decimal</td>
<td>12.345</td>
</tr>
<tr>
<td>Signed floating point numbers</td>
<td>xsd:float</td>
<td>-12.345E3</td>
</tr>
<tr>
<td>Signed double precision numbers</td>
<td>xsd:double</td>
<td>-12.34567890E3</td>
</tr>
<tr>
<td>Boolean values</td>
<td>xsd:boolean</td>
<td>true</td>
</tr>
<tr>
<td>Strings of characters</td>
<td>xsd:string</td>
<td>good morning</td>
</tr>
<tr>
<td>Date/times</td>
<td>xsd:dateTime</td>
<td>2001-10-01T04:05:06</td>
</tr>
<tr>
<td>Base64 encoded binary</td>
<td>SOAP-ENC:base64</td>
<td>GWalP2A=</td>
</tr>
<tr>
<td>32-bit signed integers</td>
<td>xsd:int</td>
<td>-1234567</td>
</tr>
<tr>
<td>16-bit signed integers</td>
<td>xsd:short</td>
<td>-1234</td>
</tr>
<tr>
<td>Negative integers</td>
<td>xsd:negativeInteger</td>
<td>-32768</td>
</tr>
</tbody>
</table>
SOAP Types (2)

- New types can be **derived** from already defined ones by definition of subtypes:
  - **Enumerations**: Selected explicitly from a simple base type.
  - **Subsets** selected by **restriction** rules. E.g. `int`, `short`, `negativeInteger` etc. are all subsets of `decimal`.

- **Compound** SOAP types can be:
  - **Structs**: *sets* of named elements, of any types, whose order is unimportant.
  - **Arrays**: ordered *sequences* of elements, of same or different types. (Names, if any, are unimportant.)

**Note**: SOAP compound types and derived types are (currently) only a subset of those available in XML.
Members of **struct** types are *sets* of named elements, whose order is not significant.

**Example:** A **struct** of type **Bibentry**.

```xml
<e:Bibentry>
  <author>Alfons Aaberg</author>
  <title>My life as a latchkey child</title>
  <pubyear>2015</pubyear>
</e:Bibentry>
```

Three elements: two strings and an integer.

Reference to elements by name (author, title,...).

**Type definition given in schema**, e.g.:

```xml
<element name="Bibentry">
  <complexType>
    <element name="author" type="xsd:string"/>
    <element name="title" type="xsd:string"/>
    <element name="pubyear" type="xsd:int"/>
  </complexType>
</element>
```
Members of array types are ordered sequences of elements, of same or different types. Element names are not significant.

Example: An array of type int[5] (5 int elements)

```
<Primes SOAP-ENC:arrayType="xsd:int[5]">
  <item xsi:type="xsd:int">2</item>
  <item xsi:type="xsd:int">3</item>
  <item xsi:type="xsd:int">5</item>
  <item xsi:type="xsd:int">7</item>
  <item xsi:type="xsd:int">11</item>
</Primes>
```

Or, alternatively, with implicit element types:

```
<Primes SOAP-ENC:arrayType="xsd:int[5]">
  <number>2</number>
  <number>3</number>
  <number>5</number>
  <number>7</number>
  <number>11</number>
</Primes>
```
Web Services with SOAP

- Simple idea: Implement programs which send and receive HTTP POST requests/responses containing SOAP messages.
- More advanced environments hide details (like RMI and CORBA).
- Examples:
  - Apache SOAP (Java-based)
  - SOAP::lite (Perl-based)
  - .NET (C#-based, but other languages also OK?)
Apache SOAP

A toolkit for producing Web services:

- Offers a Java environment for:
  - Defining interface to service (à la RMI).
  - Producing implementation on server.
  - Producing implementation of client.

- Offers facilities for deploying service:
  - Defining service offered to users.
  - Registering in registry.
  - Browsing in registry.
Apache SOAP server definitions

- Interface definition:

```java
public interface IBlip
{
    public void start(int n);
    public int add(int i);
    public void stop();
}
```

- Server-side implementation:

```java
public class WSBlip implements IBlip
{
    private int bcount;
    private boolean active;

    public void start(int n)
    {
        System.out.println(new Date() + " Target activated.");
        bcount = n;
        active = true;
    }

    public int add(int i) { ... }

    public void stop() { ... }
}
```

Compile as usual using `javac`.
Apache SOAP Client code

- Not quite as easy as Server-side code!
- Uses modified versions of methods, referring to remote service by URL.
- Each method needs to:
  - Set up Call object describing call of remote method.
  - Create vector with parameters and insert into Call object.
  - Invoke remote service.
  - Deal with (correct or faulty) SOAP response.
- Apache SOAP environment offers classes whose methods make this (relatively) easy.
Example: add method in client

```java
public static int add(URL url, int n) throws Exception {
    // Create Call object and set SOAP encoding spec.
    Call call = new Call();
    call.setEncodingStyleURI( Constants.NS_URI_SOAP_ENC );
    // Set parameters for service locator
    call.setTargetObjectURI( "urn:xmethods-Blipper" );
    call.setMethodName( "add" );
    // Create vector with parameter(s) and insert into Call object
    Parameter param1 = new Parameter("n", int.class, n, null);
    Vector params = new Vector();
    params.addElement( param1 );
    call.setParams( params );

    // Invoke remote service and deal with response
    Response resp = call.invoke(url, "/Blipper");
    if( resp.generatedFault() )
    {
        Fault f = resp.getFault();
        System.err.println("Fault= "+f.getFaultCode()+", "+f.getFaultString());
        throw new Exception();
    } else
    {
        // The call was successful. Retrieve return value.
        Parameter result = resp.getReturnValue();
        Int addobj = (Int) result.getValue();
        return addobj.intValue();
    }
}
```
Apache SOAP Web Service deployment

- Offers facilities to inform Web server about:
  - **Identity** of Web service being offered, as URN.
  - **Scope**: Activation mode of service.
    - E.g. **Request**: New instance of service for each new request.
  - **Methods** made available.
  - **Provider type**: Java class, Bean script, EJ B,…
  - **Provider Class**: Name of the Java class offered.
  - **Static?**: True if methods are static on Java class.

  … or whatever is appropriate for the implementation type.

- Offers facilities for **browsing** deployed services.

- Offers facilities for **undeploying** services which are no longer needed.
Deployment Descriptors

- XML documents which describe the deployed service.
  E.g. for Apache SOAP:

```xml
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:demo:wsblip"
             type="message"
             checkMustUnderstands="false">
  <isd:provider type="java"
                scope="Request"
                methods="start add stop">
    <isd:java class="WSBlip" static="true"/>
  </isd:provider>
  <isd:faultListener>
    org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
</isd:service>
```
SOAP in context…

- SOAP is really just one protocol layer in the multi-layer Web Services Architecture:

<table>
<thead>
<tr>
<th>Service Flow</th>
<th>WSFL, BPEL, …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Publication</td>
<td>UDDI, WSEL</td>
</tr>
<tr>
<td>and Discovery</td>
<td></td>
</tr>
<tr>
<td>Service description</td>
<td>WSDL</td>
</tr>
<tr>
<td>XML-based messaging</td>
<td>SOAP</td>
</tr>
<tr>
<td>Networking</td>
<td>HTTP, FTP, SMTP, …</td>
</tr>
</tbody>
</table>
Thank you for your attention

Course 02152, DTU, Autumn 2008