The UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

CS4/MSc

Distributed Systems

Jane Hillston

jeh@dcs.ed.ac.uk

Room 1604

(Lecture 6, CORBA and Java, January 24th 2002)

CORBA Example: Hello World

- Using a `Hello World` program allows us to focus solely on the CORBA aspects of the application.
- We will consider both the client and the server implementation.

The IDL interface

```
module HelloApp
{
     interface Hello
     {
        string sayHello();
     };
};
```

The development process

Developing the client

- Create an ORB object;
- Get an initial object reference, using the naming service;
- Make the invocation on the CORBA object.

Developing the servant

- Write method bodies for each of the operations in the interface.

Developing the server

- Initialise the environment, e.g. create an ORB object for the server.
- Create objects: in this case instantiate the HelloServant class.
- Make objects accessible to the outside world, e.g. register the servant with the naming service.
- Execute a dispatch loop to wait for invocations.

Compiling the IDL

Compiling the IDL file with the `idlj -fall` command results in a subdirectory `HelloApp` containing the following files, corresponding to the description given in the previous lecture:

- `_HelloImplBase.java`,
- `_HelloStub.java`,
- `HelloOperations.java`,
- `Hello.java`,
- `HelloHelper.java`,
- `HelloHolder.java`.

The client application: `HelloClient.java`

```
import HelloApp.*;          // The package containing our stubs.
import org.omg.CosNaming.*; // We will use the naming service.
import org.omg.CORBA.*;     // All CORBA applications need these.
public class HelloClient {
  public static void main(String args[]) {
    try {
        ORB orb = ORB.init(args, null);
        org.omg.CORBA.Object objRef =
                 orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);
        NameComponent nc = new NameComponent("Hello","");
        NameComponent path[] = {nc};

        Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

        String Hello = helloRef.sayHello();  // remote invocation

        System.out.println(Hello);
    } catch(Exception e){
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
    } } }
```

The servant class: HelloServant.java

```
import HelloApp.*;
import org.omg.CORBA.*;

class HelloServant extends _HelloImplBase
{
    public String sayHello()
    {
        return "\nHello world!!\n";
    }
}
```

The server: HelloServer.java

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloServer {
  public static void main(String args[]) {
    try {
      ORB orb = ORB.init(args, null);
      HelloServant helloRef = new HelloServant();
      orb.connect(helloRef);
      org.omg.CORBA.Object objRef=orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);
      NameComponent nc = new NameComponent("Hello", "");
      NameComponent path[] = {nc};
      ncRef.rebind(path, helloRef);
      java.lang.Object sync = new java.lang.Object();
      synchronized(sync) {
         sync.wait();
      }
    } catch(Exception e) { System.err.println("ERROR : " + e);
        e.printStackTrace(System.out);
    } } }
```

Running the CORBA Application (one machine)

1. Start the Java IDL name server:

   ```
   bannocks> tnameserv -ORBInitialPort 1050 &
   ```

2. Start the Hello server:

   ```
   bannocks> java HelloServer -ORBInitialPort 1050 &
   ```

3. Run the Hello application client from another window:

   ```
   bannocks> java HelloClient -ORBInitialPort 1050

   Hello World!!

   bannocks>
   ```

Running the CORBA Application (two machines)

1. Start the Java IDL name server:

```
bannocks> tnameserv -ORBInitialPort 1050 &
```

2. Start the Hello server:

```
bannocks> java HelloServer -ORBInitialPort 1050 &
```

3. Run the Hello application client from another host:

```
clootie> java HelloClient -ORBInitialHost bannocks.dcs.ed.ac.uk
        -ORBInitialPort 1050

Hello World!!

clootie>
```

Object Reference (OR)

Object references

- provide handles to objects but are opaque to their users;
- denote a single object;
- can be passed as strings which can be turned into live object references;
- contain information about the location and type of the object in such a way that the ORB can redirect a request or activate the object if necessary.

Obtaining an Object Reference

There are three ways in which a client can obtain an object reference:

Using a factory object: only servers can create new CORBA objects, so it is common practice to define and implement a *factory object* interface for each object type (cf. the stock example in Lecture 5). This will provide a create method which returns an object reference of the appropriate type. Of course, in order to invoke this method the client needs a reference to the factory object.

Using the naming service: Once an object has been registered by its server, by name, the client can use the naming service to obtain an object reference (cf. Hello example). The reference returned will be of type `org.omg.CORBA.Object`.

From a string: The ORB provides a method to convert any object reference into a string. This can then be stored in a file, sent by email and otherwise communicated to potential clients.

Stringified Object References

- The `object_to_string()` operation (defined on the ORB interface) takes an object and produces a string.
- This string may be passed to the `string_to_object()` operation which will generate a new object reference.
- This reference must be passed to the `narrow()` method of the appropriate interface helper to cast the object reference into a reference to a more specific interface than `Object`.
- The new object reference can be invoked and will send its requests to the same object as was passed to `object_to_string()`.

```
module CORBA { //pseudo IDL
    interface ORB {
        string object_to_string(in Object obj);
        Object string_to_object(in string obj);
    ......... };
};
```

Using Stringified Object References: Server

```
try{
  ORB orb = ORB.init(args, null);

  HelloServant helloRef = new HelloServant();
  orb.connect(helloRef);

  // print stringified object reference
  System.out.println( orb.object_to_string(helloRef));

  java.lang.Object sync = new java.lang.Object();
  synchronized(sync) {
    sync.wait();
  }
}
```

Using Stringified Object References: Client

```
canna> java HelloStringClient IOR:000000000000001749444c3a48656
c6c6f4170702f48656c6c6f3a312e30000000000001000000000000003800010
000000000011 63616d2e6463732e65642e61632e756b000030680000000018afa
bcafe00000002222fb433000000080000000000000000
```

```
try{
  ORB orb = ORB.init(args, null);
  org.omg.CORBA.Object obj = orb.string_to_object( args[0] );

  Hello helloRef = HelloHelper.narrow(obj);
  if (helloRef == null) {
     System.err.println("error in stringified object reference");
     System.exit(-1);
  }
  System.out.println(helloRef.sayHello());
}
```

`resolve_initial_references()`

There is an additional way of obtaining an object reference provided solely for the purpose of bootstrapping object references in a newly started application. The operation takes a string argument that names one of a few recognised objects. It returns a CORBA Object which must be narrowed appropriately. Two string values are presently defined:

"NameService" returns a reference to a root naming context.

"InterfaceRepository" returns a reference to an *interface repository* which contains interface definitions.

We saw the use of this in the Hello World example:

```
org.omg.CORBA.Object objRef
           orb.resolve_initial_references("NameService")
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

The Naming Service

- In the CORBA specification the Common Object Services (COS) Naming Service provides a tree-like directory for object references much like a files system provides a directory structure for files.
- Object references are stored in the namespace by name and each (name, object reference) pair is called a **binding**.
- Name bindings are organised into **naming contexts** (analogous to a subdirectory in a file system); each naming context is itself a name binding.
- The root of the tree structure is the **initial naming context** and this is the only persistent binding in the namespace; the rest of the namespace is lost if the name server process is stopped.
- To use the naming service, an ORB must know the name and port of a host running the service, or have access to a stringified object reference to the initial naming context of the name server.

Adding Objects to the Namespace

```
try{
  Properties props = new Properties();
  props.put("org.omg.CORBA.ORBInitialPort", "1050");
  ORB orb = ORB.init(args, props);
  NamingContext ctx = NamingContextHelper.narrow(
                              orb.resolve_initial_references("NameService"));
  NamingContext objref = ctx;
  NameComponent nc1 = new NameComponent("plans", "text");
  NameComponent[] name1 = {nc1};
  ctx.rebind(name1, objref);

  NameComponent nc2 = new NameComponent("Personal", "directory");
  NameComponent[] name2 = {nc2};
  NamingContext ctx2 = ctx.bind_new_context(name2);

  NameComponent nc3 = new NameComponent("schedule", "text");
  NameComponent[] name3 = {nc3};
  ctx2.rebind(name3, objref);

  NameComponent nc4 = new NameComponent("calender", "text");
  NameComponent[] name4 = {nc4};
  ctx2.rebind(name4, objref);
}
```

Peer-to-peer interactions

CORBA is not limited to strict client-server interactions, i.e. a “client” may also be a “server” within the same application. One common scenario in which this occurs is when the “client” wishes to retrieve information from a “server”, possibly at a some time in the future. This is known as *callback*. The client passes a callback object to the server. The server notifies the client by issuing a method request on the callback object.

```
module CiaoApp
{
   interface CiaoCallback {
      void callback(in string message);
   };
   interface Ciao {
      string sayCiao(in CiaoCallback objref, in string message);
   };
};
```

Callback example: Servants

```
class CiaoServant extends _CiaoImplBase
{
    public String sayCiao(CiaoCallback callobj, String msg)
    {
        callobj.callback(msg);
        return "\n Ciao Mondo!! \n";
    }
}

class CiaoCallbackServant extends _CiaoCallbackImplBase
{
    public void callback(String notification)
    {
        System.out.println(notification);
    }
}
```

Callback Client

```
ORB orb = ORB.init(args, null);

org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");
   .................
Ciao ciaoRef = CiaoHelper.narrow(ncRef.resolve(path));

CiaoCallbackServant ciaoCallbackRef = new CiaoCallbackServant();
orb.connect(ciaoCallbackRef);

// call the ciao server object and print results
String ciao = ciaoRef.sayCiao(ciaoCallbackRef, "\n Buongiorno\n");
System.out.println(ciao);
```

The callback server is unaltered: see `CiaoServer.java` in the example programs.

Objects with Persistent State

- The Java IDL ORB only supports transient objects, but using file storage it is possible for an object to store its state and reinitialise itself from that state each time the server is restarted.
- The following example illustrates this with a rather unreliable secret server with the following IDL definition:

```
module SecretApp{
  interface Secret{
    exception cantWriteFile{};
    exception cantReadFile{};

    string saveSecret(in string message) raises (cantWriteFile);

    string lastSecret() raises (cantReadFile);
  };
};
```

Object Implementation with Persistent State (1)

```
class SecretServant extends _SecretImplBase{
  public String saveSecret(String msg)
      throws SecretApp.SecretPackage.cantWriteFile{
    try {
      synchronized(this) {
      File secretFile = new File("secretStorage.txt");
      FileOutputStream fos = new FileOutputStream(secretFile);
      byte[] buf = new byte[msg.length()];
      msg.getBytes(0, msg.length(), buf, 0);
      fos.write(buf);
      fos.close(); }
    } catch(Exception e) {
          throw new SecretApp.SecretPackage.cantWriteFile();
    }
 return "\n Your secret is safe with me... \n";
 }
```

Object Implementation with Persistent State (2)

```
 public String lastSecret()
    throws SecretApp.SecretPackage.cantReadFile{
   try {
     synchronized(this) {
     File secretFile = new File("secretStorage.txt");
     FileInputStream fis = new FileInputStream(secretFile);
     byte[] buf = new byte[1000];
     int n = fis.read(buf);
     String lastmsg = new String(buf);
     fis.close();
     return lastmsg; }
   } catch(Exception e) {
      throw new SecretApp.SecretPackage.cantReadFile();
   }
  }
}
```

Delegated Implementation (1)

- The servant class which implements a CORBA object inherits from the `ImplBase` class provided by the `idlj` compiler.
- This becomes a problem for classes which need to inherit some or all of their functionality from another Java class: Java does not allow multiple inheritance for classes.
- The problem is overcome by using **delegation**, sometimes known as the **tie mechanism**.
- A **pseudo-implementation** or **tie class** is introduced which does not implement the operations but calls methods on another object that actually implements the operations' semantics.
- The necessary ORB classes are generated by compiling the IDL definition of the object with the `idlj` compiler and a special flag,

```
idlj -fallTIE Hello.idl
```

Delegated Implementation (2)

In the `HelloApp` subdirectory this generates a file `Hello_Tie.java` instead of `_HelloImplBase`. This class acts as the skeleton, receiving invocations from the ORB and delegating them to the servant that actually does the work.

```
class HelloServant extends HelloBasic implements HelloOperations
{
}
class HelloBasic {
    public String sayHello() {
        return "\nHello world !!\n";
    }
}
```

Delegated Implementation(3): `Hello_Tie`

```
public class Hello_Tie extends org.omg.CORBA.portable.ObjectImpl
                implements HelloApp.Hello, org.omg.CORBA.portable.InvokeHandler
{
  // Constructors
  public Hello_Tie () {
  }
  public Hello_Tie (HelloApp.HelloOperations impl) {
    super ();
    _impl = impl;
  }
  .
  .
  public String sayHello ()  {
    return _impl.sayHello();
  } // sayHello
  .
  .
  private HelloApp.HelloOperations _impl;
} // class Hello_Tie
}
```

Delegated Implementation (4)

In the server the delegation is made explicit, as the object which is registered with the the ORB is not a reference to the servant, but a reference to the a `Hello_Tie` object:

```
HelloServant servant = new HelloServant();
Hello_Tie helloRef = new Hello_Tie(servant);
orb.connect(helloRef);
```