CS 5523 Lecture 8: Introduction to Remote invocation

- *Questions on Laboratory 1*
- *Models of programming in distributed systems*
- *Objects and remote objects*
- *Remote invocation and remote object references*
- *A simple "Hello World" application in CORBA*
- *Marshalling*

Programming models for distributed applications:

- *Remote method invocation (RMI) – an object in one process can invoke methods of objects in another process*
- *Remote procedure call (RPC) – client calls the procedure in a server program that is running in a different process*
- *Event notification – objects receive notification of events at other objects for which they have registered*

These mechanism must be location-transparent. The first two are traditional client-server (pull), while event notification is a push strategy

Basic steps for client-server (pull strategies):

- *Client or its proxy marshalls the information that would be used for local access (do operation, call, or invocation) into a message and sends to the remote server.*
- *The server or its proxy unmarshalls the message and performs the request as though it were made locally.*
- *The server or its proxy then marshalls the result into a message and sends it to the remote client.*
- *The client or its proxy unmarshalls the message and treats the result as though it were obtained locally.*

What is a proxy and why might it be useful?

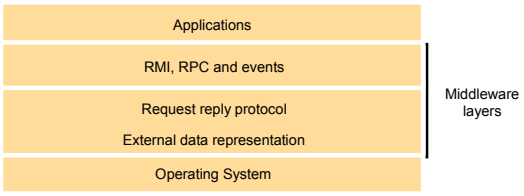
Marshalling:

- *marshalling – process of transforming a collection of data items into a form suitable for transmission as a message*
- *unmarshalling – process of disassembling a message into its pre-marshalled equivalent.*

The process requires a predefined format. Examples:

- *XDR standardized external data representation (RPC)*
- *CORBA common data representation (CDR)*
- *Java object serialization (Java RMI)*
- *Convert to ASCII (HTTP)*
- *Microsoft's format*

Figure 5.1
Middleware layers



Systems that support RMI:

- *CORBA – Common Object Request Broker Architecture*
- *Java RMI*
- *Microsoft's Distributed Common Object Model (DCOM, now COM)*
- *SOAP/.NET*

Review of objects:

- *An object encapsulates both data and methods*
- *Objects are accessed via object references*
- *Interfaces – provide definitions of signature of a set of methods*
- *Actions are performed in OO by having objects invoke methods of other objects, the invoker is called a "client" of the object*
- *Invocation can cause:*
  - *the state of the receiver to be changed (modifier methods)*
  - *additional invocations of methods on other objects*
- *Exceptions are thrown when an error occurs. If object doesn't "catch" the exception, the exception is delivered to the caller (similar to signals, but at the programming language level)*

Figure 5.3
Remote and local method invocations

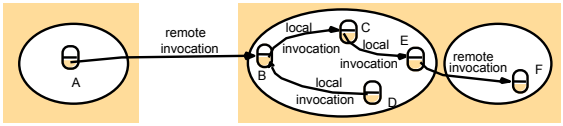
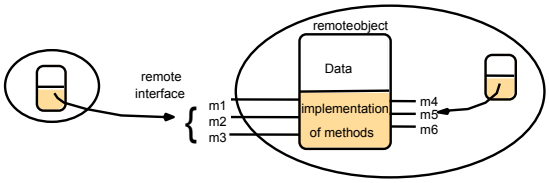


Figure 5.4
A remote object and its remote interface



Remote object reference:

- *An object must have the remote object reference of an object in order to do remote invocation of an object*
- *Remote object references may be passed as input arguments or returned as output arguments.*
- *Parameters of a method in Java are input parameters*
- *Returned result of a method in Java is the output parameter*
- *Objects are serialized to be passed as parameters*
- *When a remote object reference is returned, it can be used to invoke remote methods*
- *Non-remote serializable objects are copied by value*

Remote object reference:

- *An object must have the remote object reference of an object in order to do remote invocation of an object*
- *Remote object references may be passed as input arguments or returned as output arguments.*

Figure 4.10
Representation of a remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Remote interface (RMI):

- *The remote interface specifies the methods of an object that are available for remote invocation*
- *Input and output parameters are specified. The parameters may be objects*
- *Use:*
  - *When the remote method is invoked, the actual arguments corresponding to the input parameters are marshalled into a packet and sent to the server.*
  - *The server demarshals the packet, performs the procedure, remarshals the output arguments, and sends the return packet to the caller.*
  - *Client demarshals the return packet*
  - *Need a common format definition for how to pass objects (e.g., CORBA IDL or Java RMI)*

Interfaces:

- *Specify procedures (methods) and variables that can be accessed in a module*
- *No information other than that specified by the interface can be communicated.*
- *Do not specify an implementation*
- *Types of interfaces:*
  - *Service interface (RPC)*
  - *Remote interface (RMI)*

Remote interface:

- *CORBA – uses IDL to specify remote interfaces*
- *JAVA – uses ordinary interfaces that are extended by the keyword* `remote`.

Example of CORBA IDL

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

Map into Java by running the IDL-to-java: `idlj -fall Hello.idl`

This generates the following files in the HelloApp subdirectory:

Hello.java, HelloHelper.java, HelloHolder.java,

HelloOperations.java, HelloPOA.java and _HelloStub.java.

HelloClient.java

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

public class HelloClient {
  static Hello helloImpl;

  public static void main(String args[]){
    try{
        ORB orb = ORB.init(args, null); // create and initialize the ORB
                                        // get the root naming context
        org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
        // Use NamingContextExt instead of NamingContext. This is
        // part of the Interoperable naming Service.
        NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
        String name = "Hello"; // resolve the Object Reference in Naming
        helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));
        System.out.println("Obtained a handle on server object: " + helloImpl);
        System.out.println(helloImpl.sayHello());
        helloImpl.shutdown();

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

HelloServer.java

```
// HelloServer.java
// Copyright and License
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;

class HelloImpl extends HelloPOA {
  private ORB orb;

  public void setORB(ORB orb_val) {
    orb = orb_val;
  }

  // implement sayHello() method
  public String sayHello() {
    return "\nHello world !!\n";
  }

  // implement shutdown() method
  public void shutdown() {
    orb.shutdown(false);
  }
}
```

HelloServer.java (continued)

```
public class HelloServer {

  public static void main(String args[]) {
    try{
      ORB orb = ORB.init(args, null); // create and initialize the ORB
                                      // get reference to rootpoa & activate the POAManager
      POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();
      HelloImpl helloImpl = new HelloImpl(); // create servant and register it with the ORB
      helloImpl.setORB(orb);
                                      // get object reference from the servant
      org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
      Hello href = HelloHelper.narrow(ref);
                                      // get the root naming context
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
              // Use NamingContextExt which is part of the Interoperable Naming Service (INS) spec
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      String name = "Hello";          // bind the Object Reference in Naming
      NameComponent path[] = ncRef.to_name( name );
      ncRef.rebind(path, href);
      System.out.println("HelloServer ready and waiting ...");
      orb.run(); // wait for invocations from clients
    } catch (Exception e) {
      System.err.println("ERROR: " + e);
      e.printStackTrace(System.out);
    }
    System.out.println("HelloServer Exiting ...");
  }
}
```

Running CORBA (server on pandora, port 20000):

- *Compile the client and server:*

```
javac HelloClient.java HelloApp/*.java
javac HelloServer.java HelloApp/*.java
```

- *Start the Java Object Request Broker Daemon on server host:*

```
orbd -ORBInitialPort 20000 -ORBInitialHost pandora.cs.utsa.edu &
```

- *Start the HelloServer on server host:*

```
java HelloServer -ORBInitialPort 20000
```

- *Start the client on another machine, say ten23:*

```
java HelloClient -ORBInitialHost pandora.cs.utsa.edu -ORBInitialPort 20000
```

- *Be sure to kill your orbd when finished….*

Why is this easier than just doing sockets?

Figure 4.7
CORBA CDR for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Figure 4.8
CORBA CDR message

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Figure 4.9
Indication of Java serialized form

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

For next time:

- *Read CDK 4.3 and Chapter 17*