Distributed Object-Based Systems

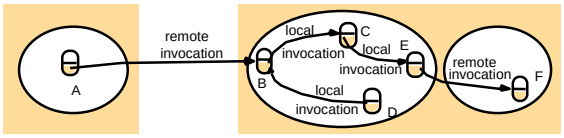
Tanenbaum, Ch. 10

The Distributed Object Paradigm - 1

- The distributed object paradigm is a paradigm that provides abstractions beyond those of the message-passing model.
- In object-oriented programming, objects are used to represent an entity significant to an application. Each object encapsulates:
  - the **state** or data of the entity: in Java, such data is contained in the instance variables of each object;
  - the **operations** of the entity, through which the state of the entity can be accessed or updated.

Local Objects vs. Distributed Objects

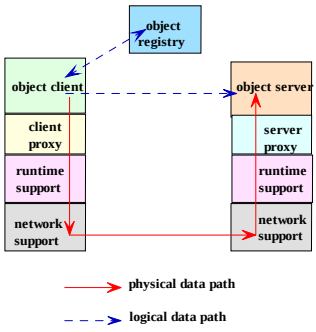
- Local objects are those whose methods can only be invoked by a **local process**, a process that runs on the same computer on which the object exists.
- A distributed object is one whose methods can be invoked by a **remote process**, a process running on a computer connected via a network to the computer on which the object exists.



The Distributed Object Paradigm - 2

- A process running in host A makes a method call to a distributed object residing on host B, passing with the call data for the parameters, if any.
- The method call invokes an action performed by the method on host A, and a return value, if any, is passed from host A to host B.
- A process which makes use of a distributed object is said to be a client process of that object, and the methods of the object are called remote methods (as opposed to local methods, or methods belonging to a local object) to the client process.

An Archetypal Distributed Objects System



5

Distributed Object System

- Logically, the object client makes a call directly to a remote method.
- In reality, the call is handled by a software component, called a ***client proxy***, which interacts which the software on the client host that provides the runtime support for the distributed object system.
- The runtime support is responsible for the interprocess communication needed to transmit the call to the remote host, including the marshalling of the argument data that needs to be transmitted to the remote object.

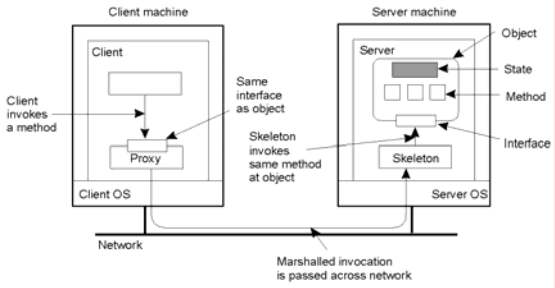
6

Distributed Object System - 2

- A similar architecture is required on the server side, where the runtime support for the distributed object system handles the receiving of messages and the unmarshalling of data, and forwards the call to a software component called the **server proxy**.
- The server proxy interfaces with the distributed object to invoke the method call locally, passing in the unmarshalled data for the arguments.
- The method call results in the performance of some tasks on the server host. The outcome of the execution of the method, including the marshalled data for the return value, is forwarded by the server proxy to the client proxy, via the **runtime support** and **network support** on both sides.

7

Distributed Object System - 3



8

Distributed Object Systems/Protocols

The distributed object paradigm has been widely adopted in distributed applications, for which a large number of mechanisms based on the paradigm are available. Among the most well known of such mechanisms are:

- ~ Java Remote Method Invocation (RMI),
- ~ the Common Object Request Broker Architecture (CORBA) systems,
- ~ the Distributed Component Object Model (DCOM),
- ~ mechanisms that support the Simple Object Access Protocol (SOAP).

Of these, the most straightforward is the Java RMI

Persistent vs transient objects

- Persistent objects continue to exist even if they are not contained in the address space of a server process
- The "state" of a persistent object has to be stored on a persistent store, i.e., secondary storage
- Invocation requests result in an instance of the object being created in the address space of a running process
  - many policies possible for object instantiation and (de)instantiation
- Transient objects only exist as long as their container server processes are running

Static vs dynamic remote method invocations

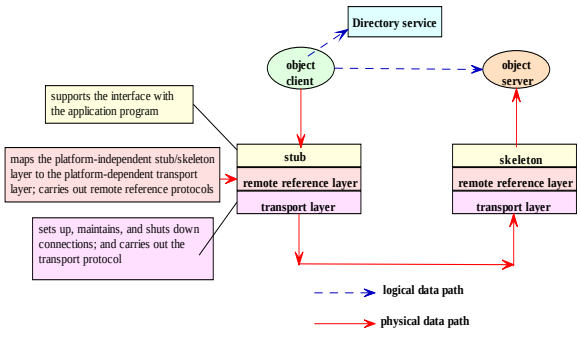
- Typical way for writing code that uses RMI is similar to the process for writing RPC
  - declare the interface in IDL, compile the IDL file to generate client and server stubs, link them with client and server side code to generate the client and the server executables
  - referred to as static invocation
  - requires the object interface to be known when the client is being developed
- Dynamic invocation
  - the method invocation is composed at run-time
    invoke(object, method, input_parameters, output_parameters)
  - useful for applications where object interfaces are discovered at run-time, e.g. object browser, batch processing systems for object invocations, "agents"

Java Remote Method Invocation

Remote Method Invocation

- Remote Method Invocation (RMI) is an object-oriented implementation of the Remote Procedure Call model. It is an API for Java programs only.
- Using RMI, an ***object server*** exports a ***remote object*** and registers it with a directory service. The object provides remote methods, which can be invoked in client programs.
- Syntactically:
  - A remote object is declared with a ***remote interface***, an extension of the Java ***interface***.
  - The remote interface is implemented by the object server.
  - An ***object client*** accesses the object by **invoking the remote methods** associated with the objects using syntax provided for remote method invocations.

The Java RMI Architecture

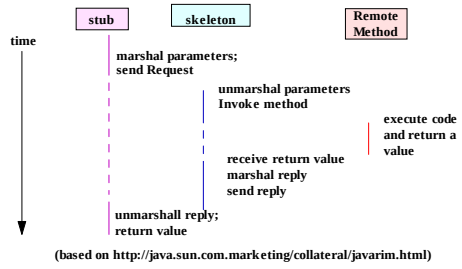


Object Registry

- The RMI API allows a number of directory services to be used for registering a distributed object. One such service is the Java Naming and Directory Interface (JNDI), which is more general than the RMI registry, in the sense that it can be used by applications that do not use the RMI API.
- We will use a simple directory service called the RMI registry, ***rmiregistry***, which is provided with the Java Software Development Kit (SDK). The RMI Registry is a service whose server, when active, runs on the **object server's host machine**, by convention and by default on the TCP port 1099.

The interaction between the stub and the skeleton

A time-event diagram describing the interaction between the stub and the skeleton:



(based on http://java.sun.com.marketing/collateral/javarim.html)

The API for the Java RMI

- The Remote Interface
- The Server-side Software
  - The Remote Interface Implementation
  - Stub and Skeleton Generations
  - The Object Server
- The Client-side Software

The Remote Interface

- A Java interface is a class that serves as a template for other classes: it contains declarations or signatures of methods whose implementations are to be supplied by classes that implements the interface.
- A java remote interface is an interface that inherits from the Java ***Remote*** class, which allows the interface to be implemented using RMI syntax. Other than the Remote extension and the Remote exception that must be specified with each method signature, a remote interface has the same syntax as a regular or local Java interface.

A sample remote interface

```
// file: SomeInterface.java
// to be implemented by a Java RMI server class.
import java.rmi.*
public interface SomeInterface extends Remote {
   // signature of first remote method
    public String someMethod1( )
         throws java.rmi.RemoteException;
  // signature of second remote method
    public int someMethod2( float ) throws
       java.rmi.RemoteException;
  // signature of other remote methods may follow

} // end interface
```


A sample remote interface - 2

- The **java.rmi.Remote** exception must be listed in the *throw* clause of each method signature.
- This exception is raised when errors occur during the processing of a remote method call, and the exception is required to be caught in the method caller's program.
- Causes of such exceptions include exceptions that may occur during interprocess communications, such as access failures and connection failures, as well as problems unique to remote method invocations, including errors resulting from the object, the stub, or the skeleton not being found.

The Server-side Software

An object server is an object that provides the methods of and the interface to a distributed object. Each object server must

- implement each of the remote methods specified in the interface,
- register an object which contains the implementation with a directory service.

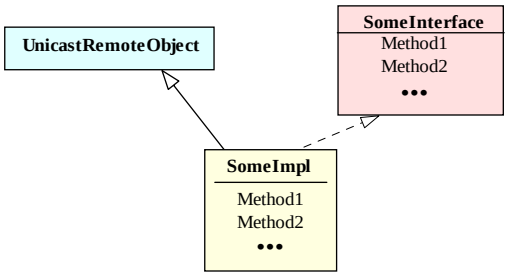
It is recommended that the two parts be provided as separate classes.

The Remote Interface Implementation

A class which implements the remote interface should be provided. The syntax is similar to a class that implements a local interface.

```
import java.rmi.*;
import java.rmi.server.*;
/**
 * This class implements the remote interface SomeInterface.
 */
public class SomeImpl extends UnicastRemoteObject
     implements SomeInterface {
   public SomeImpl() throws RemoteException {
      super( );
   }
   public String someMethod1( ) throws RemoteException {
      // code to be supplied
   }
   public int someMethod2( ) throws RemoteException {
      // code to be supplied
   }
} // end class
```


UML diagram for the SomeImpl class



UMLDiagram for SomeImpl

Stub and Skeleton Generations

In RMI, each distributed object requires a proxy each for the object server and the object client, knowns as the object's skeleton and stub respectively. These proxies are generated from the implementation of a remote interface using a tool provided with the Java SDK: the RMI compiler ***rmic***.

```
rmic <class name of the remote interface implementation>
```

For example:

```
rmic SomeImpl
```

As a result of the compilation, two proxy files will be generated, each prefixed with the implementation class name:

SomeImpl_skel.class
SomeImpl_stub.class.

The stub file for the object

- The stub file for the object, as well as the remote interface file, must be shared with each object client – these file are required for the client program to compile.
- A copy of each file may be provided to the object client by hand. In addition, the Java RMI has a feature called "stub downloading" which allows a stub file to be obtained by a client dynamically.

The Object Server

The object server class is a class whose code instantiates and exports an object of the remote interface implementation. Figure 10 shows a template for the object server class.

```
import java.rmi.*;
……
public class SomeServer  {
   public static void main(String args[]) {
      try{
       // code for port number value to be supplied
          SomeImpl exportedObj = new SomeImpl();
          startRegistry(RMIPortNum);
          // register the object under the name "some"
          registryURL = "rmi://localhost:" + portNum + "/some";
          Naming.rebind(registryURL, exportedObj);
          System.out.println("Some Server ready.");
       }// end try
} // end main
```


The Object Server - 2

```
// This method starts a RMI registry on the local host, if it
// does not already exists at the specified port number.
private static void startRegistry(int RMIPortNum)
      throws RemoteException{
  try {
    Registry registry= LocateRegistry.getRegistry(RMIPortNum);
    registry.list( );
          // The above call will throw an exception
        // if the registry does not already exist
  }
  catch (RemoteException ex) {
    // No valid registry at that port.
    System.out.println(
       "RMI registry cannot be located at port " + RMIPortNum);
    Registry registry= LocateRegistry.createRegistry(RMIPortNum);
    System.out.println(
       "RMI registry created at port " + RMIPortNum);
  }
} // end startRegistry
```


The Object Server - 3

- In our object server template, the code for exporting an object is as follows:

```
// register the object under the name "some"
registryURL = "rmi://localhost:" + portNum +
                    "/some";
Naming.rebind(registryURL, exportedObj);
```

- The ***Naming*** class provides methods for storing and obtaining references from the registry. In particular, the rebind method allow an object reference to be stored in the registry with a URL in the form of

```
rmi://<host name>:<port number>/<reference name>
```

- The ***rebind*** method will overwrite any reference in the registry bound with the given reference name. If the overwriting is not desirable, there is also a ***bind*** method.
- The host name should be the name of the server, or simply "localhost". The reference name is a name of your choice, and should be unique in the registry.

The RMI Registry

- A server exports an object by registering it by a symbolic name with a server known as the RMI registry.

```
// Create an object of the Interface
SomeInterfacel obj = new SomeInterface("Server1");
// Register the object; rebind will overwrite existing
//  registration by same name – bind( ) will not.
        Naming.rebind("Server1", obj);
```

- A server, called the RMI Registry, is required to run on the host of the server which exports remote objects.
- The RMIRegistry is a server located at port 1099 by default
- It can be invoked dynamically in the server class:

```
import java.rmi.registry.LocateRegistry;
    …
LocateRegistry.createRegistry ( 1099 );
     …
```


The RMI Registry - 2

- Alternatively, an RMI registry can be activated by hand using the *rmiregistry* utility which comes with the Java Software Development Kit (SDK), as follows:

  rmiregistry <port number>

  where the port number is a TCP port number. If no port number is specified, port number 1099 is assumed.
- The registry will run continuously until it is shut down (via CTRL-C, for example)

The Object Server - 5

- When an object server is executed, the exporting of the distributed object causes the server process to begin to listen and wait for clients to connect and request the service of the object.
- An RMI object server is a concurrent server: each request from an object client is serviced using a separate thread of the server. Note that if a client process invokes multiple remote method calls, these calls will be executed concurrently unless provisions are made in the client process to synchronize the calls.

The Client-side Software

The program for the client class is like any other Java class.

The syntax needed for RMI involves

- locating the RMI Registry in the server host,

  and
- looking up the remote reference for the server object; the reference can then be cast to the remote interface class and the remote methods invoked.

The Client-side Software - 2

```
import java.rmi.*;
….
public class SomeClient {
   public static void main(String args[]) {
     try {
         String registryURL =
          "rmi://localhost:" + portNum + "/some";
        SomeInterface h =
         (SomeInterface)Naming.lookup(registryURL);
        // invoke the remote method(s)
        String message = h.method1();
        System.out.println(message);
        // method2 can be invoked similarly
     } // end try
     catch (Exception e) {
         System.out.println("Exception in SomeClient: " + e);
     }
   } //end main
   // Definition for other methods of the class, if any.
}//end class
```


Looking up the remote object

The ***lookup*** method of the ***Naming*** class is used to retrieve the object reference, if any, previously stored in the registry by the object server. Note that the retrieved reference must be cast to the **remote interface** (**not** its implementation) class.

```
String registryURL =
  "rmi://localhost:" + portNum + "/some";
SomeInterface h =
(SomeInterface)Naming.lookup(registryURL);
```


Invoking the Remote Method

The remote interface reference can be used to invoke any of the methods in the remote interface, as in the example:

```
String message = h.method1();
System.out.println(message);
```

- Note that the syntax for the invocation of the remote methods is the same as for local methods.
- It is a common mistake to cast the object retrieved from the registry to the interface implementation class or the server object class . Instead it should be cast as the interface class.

Steps for building an RMI application

Developing the server-side software

1. Open a directory for all the files to be generated for this application.
2. Specify and compile the remote-server interface in *SomeInterface.java*
3. Implement and compile the interface in *SomeImpl.java*
4. Use the RMI compiler ***rmic*** to process the implementation class and generate the stub file and skelton file for the remote object:

   ```
   rmic  SomeImpl
   ```

   The files generated can be found in the directory as *SomeImpl_Skel.class* and *SomeImpl_Stub.class.*

   Steps 3 and 4 must be repeated each time that a change is made to the interface implementation.
5. Create and compile the object server program *SomeServer.java*.
6. Activate the object server

   ```
   java   SomeServer
   ```

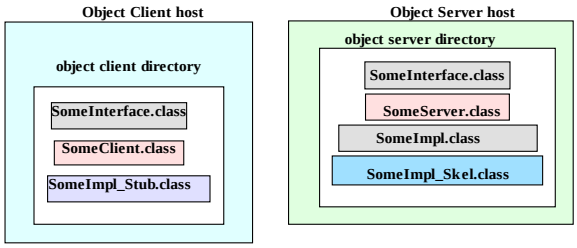

Developing the client-side software

1. Open a directory for all the files to be generated for this application.
2. Obtain a copy of the remote interface class file. Alternatively, obtain a copy of the source file for the remote interface, and compile it using ***javac*** to generate the interface class file.
3. Obtain a copy of the stub file for the implementation of the interface:

   SomeImpl_Stub.class.
4. Develop the client program ***SomeClient.java***, and compile it to generate the client class.
5. Activate the client.

```
java SomeClient
```


Placement of files for a RMI application



Testing and Debugging an RMI Application

1. Build a template for a minimal RMI program. Start with a remote interface with a single signature, its implementation using a stub, a server program which exports the object, and a client program which invokes the remote method. Test the template programs on one host until the remote method can be made successfully.
2. Add one signature at a time to the interface. With each addition, modify the client program to invoke the added method.
3. Fill in the definition of each remote method, one at a time. Test and thoroughly debug each newly added method before proceeding with the next one.
4. After all remote methods have been thoroughly tested, develop the client application using an incremental approach. With each increment, test and debug the programs.

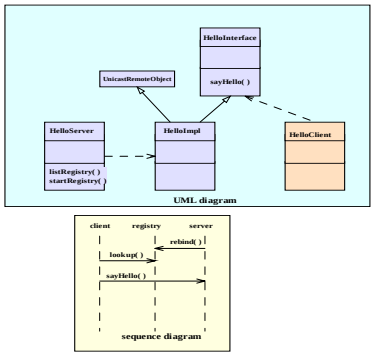
Comparison of the RMI and the socket APIs

The remote method invocation API is an efficient tool for building network applications. It can be used in lieu of the socket API in a network application. Some of the tradeoffs between the RMI API and the socket API are as follows:

- The socket API is closely related to the operating system, and hence has less execution overhead. For applications which require high performance, this may be a consideration.
- The RMI API provides the abstraction which eases the task of software development. Programs developed with a higher level of abstraction are more comprehensible and hence easier to debug.

The HelloWorld Sample

Diagrams for the Hello application



HelloInterface.java

```
import java.rmi.*;

public interface HelloInterface
             extends Remote {
   public String sayHello(String name)
      throws java.rmi.RemoteException;
}
```

HelloImpl.java

```
import java.rmi.*;
import java.rmi.server.*;

public class HelloImpl extends UnicastRemoteObject
     implements HelloInterface {

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

   public String sayHello(String name)
         throws RemoteException {
      return "Hello, World! " + name;
   }
}
```


HelloClient.java

```
import java.io.*;
import java.rmi.*;

public class HelloClient {
   public static void main(String args[ ]) {
      try {
         int RMIPort;
         String hostName;
         InputStreamReader is = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(is);
         System.out.println("Enter the RMIRegistry host number: ");
         hostName = br.readLine();
         System.out.println("Enter the RMIRegistry port number: ");
         String portNum = br.readLine();
         RMIPort = Integer.parseInt(portNum);
         String registryURL = "rmi://"+hostName+":"+portNum+"/hello";
         HelloInterface h = (HelloInterface)Naming.lookup(registryURL);
         System.out.println("Lookup completed ");
         String message = h.sayHello("Daffy Duck");
         System.out.println("HelloClient: " + message); }
      catch (Exception e) {
         System.out.println("Exception in HelloClient: " + e);
      }
   }
}
```


Running the Client

```
% java HelloClient
Enter the RMIRegistry host number:
localhost
Enter the RMIRegistry port number:
3232
Lookup completed
HelloClient: Hello, World! Daffy Duck
%
```


HelloServer.java (overall structure)

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.net.*;
import java.io.*;

public class HelloServer {
   public static void main(String args[ ]) {
      …
   }
   private static void startRegistry(int RMIPortNum)
  throws RemoteException {
     …
   }
   private static void listRegistry(String registryURL)
           throws RemoteException, MalformedURLException
  {
   …
   }
}
```

```
public static void main(String args[ ]) {
     InputStreamReader is = new InputStreamReader(System.in);
     BufferedReader br = new BufferedReader(is);
     String portNum, registryURL;
     try{
         System.out.println("Enter the RMIregistry port
  number:");
         portNum = (br.readLine()).trim();
         int RMIPortNum = Integer.parseInt(portNum);
         startRegistry(RMIPortNum);
         HelloImpl exportedObj = new HelloImpl();
         registryURL = "rmi://localhost:" + portNum + "/hello";
         Naming.rebind(registryURL,exportedObj);
         System.out.println("Server registered. Registry
  currently contains:");
         listRegistry(registryURL);
         System.out.println("Hello Server ready.");
     } //end try
     catch (Exception re) {
         System.out.println("Exception in HelloServer.main: " +
  re);
     }
  }
```

HelloServer.java (main)


```
// This method starts an RMI registry at port RMIPortNum if
// it does not already exist.
  private static void startRegistry(int RMIPortNum) throws
   RemoteException {
  try {
     Registry registry = LocateRegistry.getRegistry(RMIPortNum);
     registry.list();
  }
  catch (RemoteException e) {
     System.out.println("RMI registry cannot be located at port
   " + RMIPortNum);
     Registry registry =
   LocateRegistry.createRegistry(RMIPortNum);
     System.out.println("RMI registry created at port " +
   RMIPortNum);
```

HelloServer.java (startRegistry)


```
private static void listRegistry(String registryURL)
          throws RemoteException, MalformedURLException
  {
   System.out.println("Registry " + registryURL + "
  contains:");
   String[] names = Naming.list(registryURL);
   for (int i =0;i< names.length;i++)
    System.out.println(names[i]);
  }
```

HelloServer.java (listRegistry)

Running the Server

```
% java HelloServer
Enter the RMIregistry port number:
3232
RMI registry cannot be located at port 3232
RMI registry created at port 3232
Server registered. Registry currently contains:
Registry rmi://localhost:3232/hello contains:
//localhost:3232/hello
Hello Server ready.
```

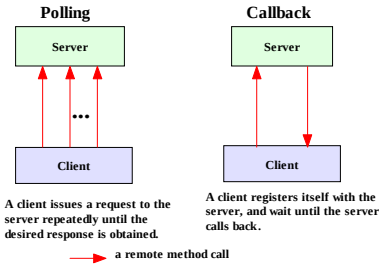
RMI Callbacks

Introduction

- In the client server model, the server is passive: the IPC is initiated by the client; the server waits for the arrival of requests and provides responses.
- Some applications require the server to initiate communication upon certain events. Examples applications are:
  - monitoring
  - games
  - auctioning
  - voting/polling
  - chat-toom
  - message/bulletin board
  - groupware

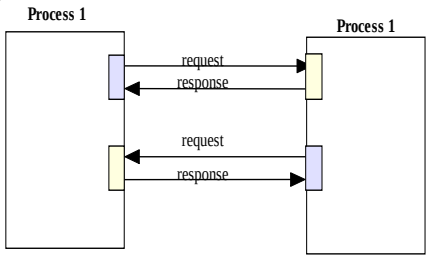
Polling vs. Callback

In the absence of callback, a client will have to poll a passive server repeatedly if it needs to be notified that an event has occurred at the server end.



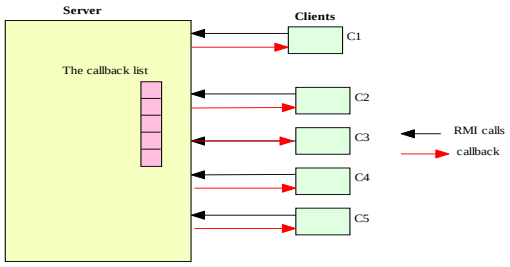
Two-way communications

- Some applications require that both sides may initiate IPC.
- Using sockets, duplex communication can be achieved by using two sockets on either side.
- With connection-oriented sockets, each side acts as both a client and
  a server.

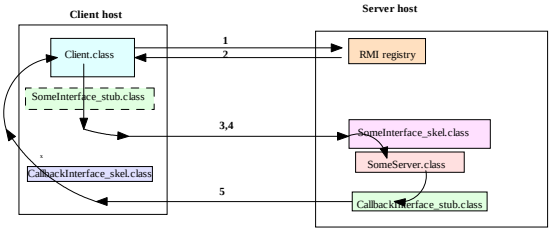


RMI Callbacks

- A callback client registers itself with an RMI server.
- The server makes a callback to each registered client upon the occurrence of a certain event.

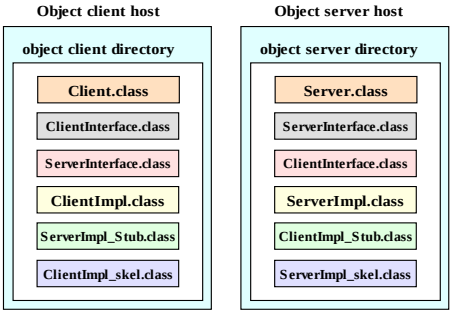


Callback Client-Server Interactions

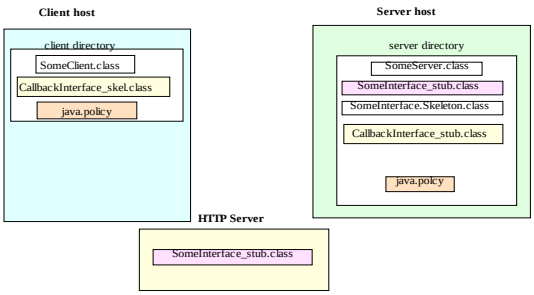


1. Client looks up the interface object in the RMIregistry on the server host.
2. The RMIRegistry returns a remote reference to the interface object.
3. Via the server stub, the client process invokes a remote method to register itself for callback, passing a remote reference to itself to the server. The server saves the reference in its callback list.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the interface object.
5. When the anticipated event takes place, the server makes a callback to each registered client via the callback interface stub on the server side and the callback interface skeleton on the client side.

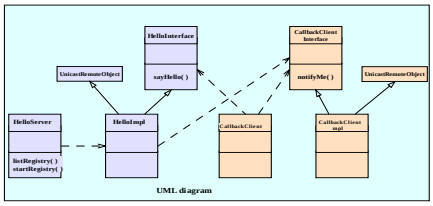
Callback application files

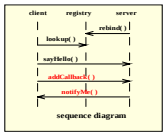


RMI Callback file placements



The Hello Application with Callback





61

RMI Callback Interface

- The server provides a remote method which allows a client to register itself for callbacks.
- A Remote interface for the callback is needed, in addition to the server-side interface.
- The interface specifies a method for accepting a callback from the server.
- The client program is a subclass of RemoteObject and implements the callback interface, including the callback method.
- The client registers itself for callback in its main method.
- The server invokes the client's remote method upon the occurrence of the anticipated event.

62

Remote Interface for Server

```
public interface HelloInterface extends Remote {
  // remote method
  public String sayHello() throws java.rmi.RemoteException;
  // method to be invoked by a client to add itself to the
   callback list
  public void addCallback(
     HelloCallbackInterface CallbackObject)
     throws java.rmi.RemoteException;
}
```

63

Remote Interface for Callback Client

```
// an interface specifying a callback method
public interface HelloCallbackInterface extends java.rmi.Remote
{
   // method to be called by the server on callback
   public void callMe (
         String message
     ) throws java.rmi.RemoteException;
}
```

64

HelloServer, with callback

```
public class HelloServer extends UnicastRemoteObject implements
    HelloInterface {
    static int RMIPort;
    // vector for store list of callback objects
    private static Vector callbackObjects;

    public HelloServer() throws RemoteException {
      super();
      // instantiate a Vector object for storing callback objects
      callbackObjects = new Vector();
    }
    // method for client to call to add itself to its callback
    public void addCallback(  HelloCallbackInterface CallbackObject) {
       // store the callback object into the vector
        System.out.println("Server got an 'addCallback' call.");
       callbackObjects.addElement (CallbackObject);
    }
```


HelloServer, with callback - 2

```
public static void main(String args[]) {
      …
    registry = LocateRegistry.createRegistry(RMIPort);
      …
    callback( );
      …
  } // end main
  private static void callback( ) {
      …
    for (int i = 0; i < callbackObjects.size(); i++)  {
      System.out.println("Now performing the "+ i +"th callback\n");
      // convert the vector object to a callback object
      HelloCallbackInterface client =
          (HelloCallbackInterface) callbackObjects.elementAt(i);
        …
      client.callMe ( "Server calling back to client " + i);
        …
```


Alogorithm for building an RMI Callback Application

Server side:

1. Open a directory for all the files to be generated for this application.
2. Specify the remote-server interface, and compile it to generate the interface class file.
3. Build the remote server class by implementing the interface, and compile it using javac.
4. Use rmic to process the server class to generate a stub.class file and a skelton.class file: *rmic SomeServer*
5. If stub downloading is desired, copy the stub file to an appropriate directory on the HTTP host.
6. Activate the RMIRegistry, if it has not already been activated.
7. Set up a java.policy file.
8. Activate the server, specifying (i) the codebase if stub downloading is desired, (ii) the server host name, and (iii) the security policy file.
9. Obtain the CallbackInterface. Compile it with *javac*, then use *rmic* to generate the stub file for the callback.

Alogorithm for building an RMI Callback Application

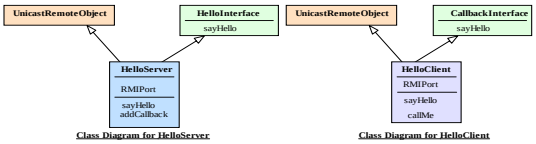
Client side:

1. **Open a directory for all the files to be generated for this application.**
2. **Implement the client program or applet, and compile it to generate the client class.**
3. **If stub downloading is not in effect, copy the server interface stub class file by hand.**
4. **Implement the callback interface. Compile it using *javac*, then using *rmic* to generate a stub class and a skeleton class for it.**
5. **Set up a java.policy file.**
6. **Activate the client, specifying (i) the server host name, and (ii) the security policy file.**

HelloClient, with callback

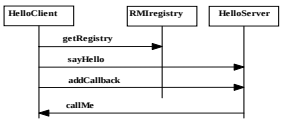
```
    HelloClient() { // constructor
        System.setSecurityManager(new RMISecurityManager());
        // export this object as a remote object
          UnicastRemoteObject.exportObject(this);
             //  …
          Registry registry = LocateRegistry.getRegistry("localhost",  RMIPort);
          h = (HelloInterface) registry.lookup("helloLiu");
          h.addCallback(this);  //  …
    } // end constructor
    // call back method - this displays the message sent by the server
    public void callMe (String message) {
       System.out.println( "Call back received: " + message );
    }
    public static void main(String args[]) {  // …
          HelloClient client   = new HelloClient();  // …
          while (true){
            ; } // end while
    } // end main
} // end HelloClient class
```


HelloServer, HelloClient



Class Diagram for HelloServer

Class Diagram for HelloClient



Event diagram for the Hello application

The Common Object Request Broker Architecture (CORBA)

CORBA

- The Common Object Request Broker Architecture (CORBA) is a standard architecture for a distributed objects system.
- CORBA is designed to allow distributed objects to interoperate in a heterogenous environment, where objects can be implemented in different programming language and/or deployed on different platforms

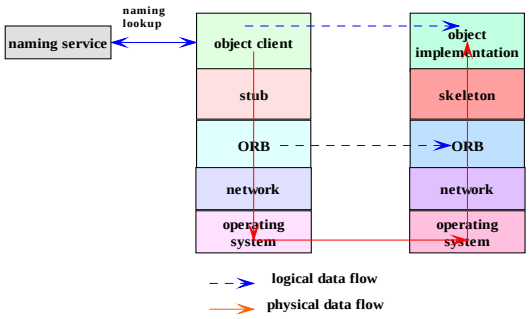
CORBA vs. Java RMI

- CORBA differs from the architecture of Java RMI in one significant aspect:
  - RMI is a proprietary facility developed by Sun MicroSystems, Inc., and supports objects written in the Java programming langugage only.
  - CORBA is an architecture that was developed by the Object Management Group (OMG), an industrial consortium.

CORBA

- CORBA is not in inself a distributed objects facility; instead, it is a set of protocols.
- A distributed object facility which adhere to these protocols is said to be CORBA-compliant, and the distributed objects that the facility support can interoperate with objects supported by other CORBA-compliant facilities.
- CORBA is a very rich set of protocols. We will instead focus on the key concepts of CORBA related to the distributed objects paradigm. We will also study a facility based on CORBA: the Java IDL.

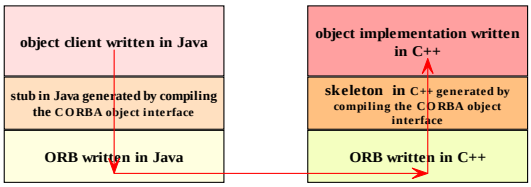
The Basic Architecture



CORBA Object Interface

- A distributed object is defined using a software file similar to the remote interface file in Java RMI.
- Since CORBA is language independent, the interface is defined using a universal language with a distinct syntax, known as the ***CORBA Interface Definition Language*** (***IDL***).
- The syntax of CORBA IDL is similar to Java and C++. However, object defined in a CORBA IDL file can be implemented in a large number of diverse programming languages, including C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLScript.
- For each of these languages, OMG has a standardized mapping from CORBA IDL to the programming language, so that a compiler can be used to process a CORBA interface to generate the proxy files needed to interface with an object implementation or an object client written in any of the CORBA-compatible languages.

Cross-language CORBA application



Inter-ORB Protocols

- To allow ORBs to be interoperable, the OMG specified a protocol known as the ***General Inter-ORB Protocol*** (***GIOP***), a specification which "provides a general framework for protocols to be built on top of specific transport layers."
- A special case of the protocol is the ***Inter-ORB Protocol*** (***IIOP***), which is the GIOP applied to the TCP/IP transport layer.

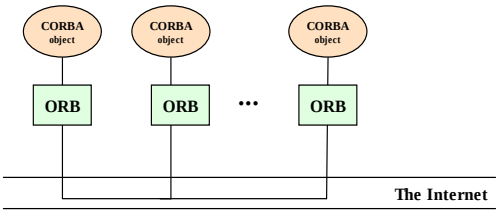
Inter-ORB Protocols

The IIOP specification includes the following elements:

1. **Transport management requirements**: specifies the connection and disconnection requirements, and the roles for the object client and object server in making and unmaking connections.
2. **Definition of common data representation**: a coding scheme for marshalling and unmarshalling data of each IDL data type.
3. **Message formats**: different types of message format are defined. The messages allow clients to send requests to object servers and receive replies. A client uses a Request message to invoke a method declared in a CORBA interface for an object and receives a reply message from the server.

Object Bus

An ORB which adheres to the specifications of the IIOP may interoperate with any other IIOP-compliant ORBs over the Internet. This gives rise to the term "***object bus***", where the Internet is seen as a bus that interconnects CORBA objects



ORB products

There are a large number of proprietary as well as experimental ORBs available:
(See CORBA Product Profiles, http://www.puder.org/corba/matrix/)

- Orbix IONA
- Borland Visibroker
- PrismTech's OpenFusion
- Web Logic Enterprise from BEA
- Ada Broker from ENST
- Free ORBs

81

Object Servers and Object Clients

- As in Java RMI, a CORBA distributed object is exported by an ***object server***, similar to the object server in RMI.
- An ***object client*** retrieves a reference to a distributed object from a naming or directory service, to be described, and invokes the methods of the distributed object.

82

CORBA Object References

- As in Java RMI, a CORBA distributed object is located using an ***object reference***. Since CORBA is language-independent, a CORBA object reference is an abstract entity mapped to a language-specific object reference by an ORB, in a representation chosen by the developer of the ORB.
- For interoperability, OMG specifies a protocol for the abstract CORBA object reference object, known as the ***Interoperable Object Reference*** (***IOR***) protocol.

83

Interoperable Object Reference (IOR)

- For interoperability, OMG specifies a protocol for the abstract CORBA object reference object, known as the ***Interoperable Object Reference*** (***IOR***) protocol.
- An ORB compatible with the IOR protocol will allow an object reference to be registered with and retrieved from any IOR-compliant directory service. CORBA object references represented in this protocol are called ***Interoperable Object References*** (***IOR***s).

84

Interoperable Object Reference (IOR)

An IOR is a string that contains encoding for the following information:

- The type of the object.
- The host where the object can be found.
- The port number of the server for that object.
- An object key, a string of bytes identifying the object.

  The object key is used by an object server to locate the object.

Interoperable Object Reference (IOR)

The following is an example of the string representation of an IOR [5]:

```
IOR:000000000000000d49444c3a677269643a312e3000000
00000000001000000000000004c000100000000015756c74
72612e6475626c696e2e696f6e612e696500009630000002
83a5c756c7472612e6475626c696e2e696f6e612e69653a67
7269643a303a3a49523a67726964003a
```

The representation consists of the character prefix "`IOR:`" followed by a series of hexadecimal numeric characters, each character representing 4 bits of binary data in the IOR.

CORBA Naming Service

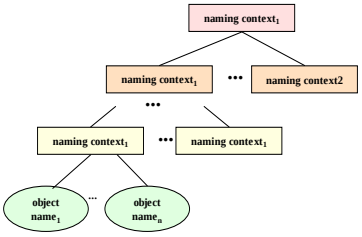
- CORBA specifies a generic directory service. The ***Naming Service*** serves as a directory for CORBA objects, and, as such, is platform independent and programming language independent.
- The Naming Service permits ORB-based clients to obtain references to objects they wish to use. It allows names to be associated with object references. Clients may query a naming service using a predetermined name to obtain the associated object reference.

CORBA Naming Service

- To export a distributed object, a CORBA object server contacts a Naming Service to ***bind*** a symbolic name to the object The Naming Service maintains a database of names and the objects associated with them.
- To obtain a reference to the object, an object client requests the Naming Service to look up the object associated with the name (This is known as ***resolving*** the object name.)
- The API for the Naming Service is specified in interfaces defined in IDL, and includes methods that allow servers to bind names to objects and clients to resolve those names.

CORBA Naming Service

To be as general as possible, the CORBA object naming scheme is necessary complex. Since the name space is universal, a standard naming hierarchy is defined in a manner similar to the naming hierarchy in a file directory



A Naming Context

- A naming context correspond to a folder or directory in a file hierarchy, while object names corresponds to a file.
- The full name of an object, including all the associated naming contexts, is known as a *compound name*. The first component of a compound name gives the name of a naming context, in which the second component is accessed. This process continues until the last component of the compound name has been reached.
- Naming contexts and name bindings are created using methods provided in the Naming Service interface.

A CORBA object name

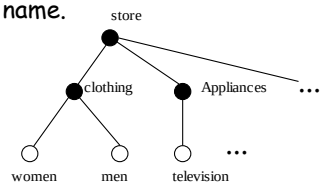
The syntax for an object name is as follows:

```
<naming context > …<naming context><object name>
```

where the sequence of naming contexts leads to the object name.

Example of a naming hierarchy

As shown, an object representing the men's clothing department is named store.clothing.men, where store and clothing are naming contexts, and men is an object name.



Interoperable Naming Service

The ***Interoperable Naming Service*** (***INS***) is a URL-based naming system based on the CORBA Naming Service, it allows applications to share a common initial naming context and provide a URL to access a CORBA object.

CORBA Object Services

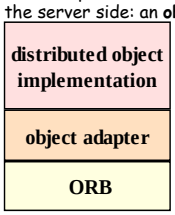
CORBA specify services commonly needed in distributed applications, some of which are:

- ***Naming Service***:
- ***Concurrency Service***:
- ***Event Service***: for event synchronization;
- ***Logging Service***: for event logging;
- ***Scheduling Service***: for event scheduling;
- ***Security Service***: for security management;
- ***Trading Service***: for locating a service by the type (instead of by name);
- ***Time Service***: a service for time-related events;
- ***Notification Service***: for events notification;
- ***Object Transaction Service***: for transactional processing.

Each service is defined in a standard IDL that can be implemented by a developer of the service object, and whose methods can be invoked by a CORBA client.

Object Adapters

In the basic architecture of CORBA, the implementation of a distributed object interfaces with the skeleton to interact with the stub on the object client side. As the architecture evolved, a software component in addition to the skeleton was needed on the server side: an **object adapter**.



Object Adapter

- An object adapter simplifies the responsibilities of an ORB by assisting an ORB in delivering a client request to an object implementation.
- When an ORB receives a client's request, it locates the object adapter associated with the object and forwards the request to the adapter.
- The adapter interacts with the object implementation's skeleton, which performs data marshalling and invoke the appropriate method in the object.

The *Portable Object Adapter*

- There are different types of CORBA object adapters.
- The ***Portable Object Adapter***, or ***POA***, is a particular type of object adapter that is defined by the CORBA specification. An object adapter that is a POA allows an object implementation to function with different ORBs, hence the word portable.

A Java IDL application example

The CORBA Interface file **Hello.idl**

```
01. module HelloApp
02. {
03.   interface Hello
04.   {
05.   string sayHello();
06.   oneway void shutdown();
07.   };
08. };
```


Compiling the IDL file (using Java 1.4)

The IDL file should be placed in a directory dedicated to the application. The file is compiled using the compiler ***idlj*** using a command as follows:

```
idlj -fall Hello.idl
```

The *–**fall*** command option is necessary for the compiler to generate all the files needed.

In general, the files can be found in a subdirectory named <some name>App when an interface file named <some name>.idl is compiled.

If the compilation is successful, the following files can be found in a ***HelloApp*** subdirectory:

HelloOperations.java	*Hello.java*
HelloHelper.java	*HelloHolder.java*
_HelloStub.java	*HelloPOA.java*

These files require no modifications.

The *Operations.java file

- There is a file HelloOperations.java
- found in HelloApp/ after you compiled using idlj
- It is known as a ***Java* operations *interface*** in general
- It is a Java interface file that is equivalent to the CORBA IDL interface file (***Hello.idl***)
- You should look at this file to make sure that the method signatures correspond to what you expect.

HelloApp/HelloOperations.java

The file contains the methods specified in the original IDL file: in this case the methods ***sayHello( )*** and ***shutdown***().

```
package HelloApp;
01. package HelloApp;
04. /**
05. * HelloApp/HelloOperations.java
06. * Generated by the IDL-to-Java compiler (portable),
07. *  version "3.1" from Hello.idl
08. */
09.
10. public interface HelloOperations
11. {
12.   String sayHello ();
13.   void shutdown ();
14. } // interface HelloOperations
```


HelloApp/Hello.java

The signature interface file combines the characteristics of the Java *operations* interface (*HelloOperations.java*) with the characteristics of the CORBA classes that it extends.

```
01. package HelloApp;
03. /**
04. * HelloApp/Hello.java
05. * Generated by the IDL-to-Java compiler (portable),
06. * version "3.1" from Hello.idl
07. */
09. public interface Hello extends HelloOperations,
10.   org.omg.CORBA.Object,
11.   org.omg.CORBA.portable.IDLEntity
12. { ...
13. } // interface Hello
```


HelloHelper.java, the Helper class

- The Java class `HelloHelper (Figure 7d)` provides auxiliary functionality needed to support a CORBA object in the context of the Java language.
- In particular, a method, ***narrow***, `allows` a CORBA object reference to be cast to its corresponding type in Java, so that a CORBA object may be operated on using syntax for Java object.

HelloHolder.java, the Holder class

- The Java class called `HelloHolder` (`Figure 7e`) holds (contains) a reference to an object that implements the `Hello interface`.
- The class is used to handle an `out` or an `inout` parameter in IDL in Java syntax (In IDL, a parameter may be declared to be ***out*** if it is an output argument, and ***inout*** if the parameter contains an input value as well as carries an output value.)

_HelloStub.java

- The Java class ***HelloStub*** (`Figure 7e`) is the stub file, the client-side proxy, which interfaces with the client object.
- It extends `org.omg.CORBA.portable.ObjectImpl` and implements the ***Hello.java*** interface.

HelloPOA.java, the server skeleton

- The Java class ***HelloImplPOA*** (`Figure 7f`) is the skeleton, the server-side proxy, combined with the portable object adapter.
- It extends *org.omg.PortableServer.Servant*, and implements the ***InvokeHandler*** interface and the ***HelloOperations*** interface.

The application

Server-side Classes

- On the server side, two classes need to be provided: the servant and the server.
- The servant, ***HelloImpl***, is the implementation of the ***Hello*** IDL interface; each ***Hello*** object is an instantiation of this class.

The Servant - HelloApp/HelloImpl.java

```
// The servant -- object implementation -- for the Hello
// example.  Note that this is a subclass of HelloPOA,
// whose source file is generated from the
// compilation of Hello.idl using j2idl.
06.  import HelloApp.*;
07. import org.omg.CosNaming.*;
08. import java.util.Properties; …
15. class HelloImpl extends HelloPOA {
16.   private ORB orb;
18.   public void setORB(ORB orb_val) {
19.     orb = orb_val;
20.   }
22.   // implement sayHello() method
23.   public String sayHello() {
24.     return "\nHello world !!\n";
25.   }
27.   // implement shutdown() method
28.   public void shutdown() {
29.     orb.shutdown(false);
30.   }
31. } //end class
```


The server - HelloApp/HelloServer.java

```
public class HelloServer {
  public static void main(String args[]) {
   try{
   // create and initialize the ORB
   ORB orb = ORB.init(args, null);
   // get reference to rootpoa & activate the POAManager
   POA rootpoa =
   (POA)orb.resolve_initial_references("RootPOA");
   rootpoa.the_POAManager().activate();
   // create servant and register it with the ORB
   HelloImpl helloImpl = new HelloImpl();
   helloImpl.setORB(orb);
   // get object reference from the servant
   org.omg.CORBA.Object ref =
      rootpoa.servant_to_reference(helloImpl);
   // and cast the reference to a CORBA reference
   Hello href = HelloHelper.narrow(ref);
```


HelloApp/HelloServer.java - continued

```
// get the root naming context
// NameService invokes the transient name service
org.omg.CORBA.Object objRef =
  orb.resolve_initial_references("NameService");
// Use NamingContextExt, which is part of the
// Interoperable Naming Service (INS) specification.
NamingContextExt ncRef =
NamingContextExtHelper.narrow(objRef);
// bind the Object Reference in Naming
String name = "Hello";
NameComponent path[] = ncRef.to_name( name );
ncRef.rebind(path, href);
System.out.println
("HelloServer ready and waiting ...");
// wait for invocations from clients
  orb.run();
```


The object client application

- A client program can be a Java application, an applet, or a servlet.
- The client code is responsible for creating and initializing the ORB, looking up the object using the Interoperable Naming Service, invoking the narrow method of the ***Helper*** object to cast the object reference to a reference to a ***Hello*** object implementation, and invoking remote methods using the reference. The object's ***sayHello*** method is invoked to receive a string, and the object's shutdown method is invoked to deactivate the service.

```
// A sample object client application.
import HelloApp.*;

import org.omg.CosNaming.*; …
public class HelloClient{
  static Hello helloImpl;
  public static void main(String args[]){
     try{
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
   NamingContextExt ncRef =
      NamingContextExtHelper.narrow(objRef);
   helloImpl =
   HelloHelper.narrow(ncRef.resolve_str("Hello"));
   System.out.println(helloImpl.sayHello());
   helloImpl.shutdown();
```


Compiling and Running a Java IDL application

1. Create and compile the Hello.idl file on the server machine:

   idlj -fall Hello.idl

2. Copy the directory containing Hello.idl (including the subdirectory generated by ***idlj***) to the client machine.
3. In the ***HelloApp*** directory on the client machine: create ***HelloClient.java***. Compile the *.java files, including the stubs and skeletons (which are in the directory ***HelloApp***):

   javac *.java HelloApp/*.java

Compiling and Running a Java IDL application

4. In the ***HelloApp*** directory on the server machine:
   - Create ***HelloServer.java***. Compile the `.java` files:
     javac *.java HelloApp/*.java
   - On the server machine: Start the Java Object Request Broker Daemon, ***orbd***, which includes a Naming Service.

   To do this on Unix:

   ```
   orbd -ORBInitialPort 1050 -ORBInitialHost
      servermachinename&
   ```

   To do this on Windows:

   ```
   start orbd -ORBInitialPort 1050 -ORBInitialHost
      servermachinename
   ```


Compiling and Running a Java IDL application

5. On the server machine, start the Hello server, as follows:

   java HelloServer –ORBInitialHost <nameserver host name> -ORBInitialPort 1050

6. On the client machine, run the ***Hello*** application client. From a DOS prompt or shell, type:

   java HelloClient -ORBInitialHost *nameserverhost* -ORBInitialPort 1050

   all on one line.

   Note that *nameserverhost* is the host on which the IDL name server is running. In this case, it is the server machine.

Compiling and Running a Java IDL application

7. Kill or stop ***orbd*** when finished. The name server will continue to wait for invocations until it is explicitly stopped.
8. Stop the object server.

Summary - 1

- The distributed object paradigm is at a higher level of abstraction than the message-passing paradigm.
- Using the paradigm, a process invokes methods of a remote object, passing in data as arguments and receiving a return value with each call, using syntax similar to local method calls.
- In a distributed object system, an object server provides a distributed object whose methods can be invoked by an object client.

Summary - 2

- Each side requires a proxy which interacts with the system's runtime support to perform the necessary IPC.
- an object registry must be available which allow distributed objects to be registered and looked up.
- Among the best-known distributed object system protocols are the Java Remote Method Invocation (RMI), the Distributed Component Object, Model (DCOM), the Common Object Request Broker Architecture (CORBA) , and the Simple Object Access Protocol (SOAP).

Summary - 3

- Java RMI is representative of distributed object systems.
  - The architecture of the Java Remote Method Invocation API includes three abstract layers on both the client side and the server side.
  - The software for a RMI application includes a remote interface, server-side software, and client-side software.
  - What are the tradeoffs between the socket API and the Java RMI API?

Summary-4

- **Client callback:**
  - Client callback is useful for an application where the clients desire to be notified by the server of the occurrence of some event.
  - Client callback allows an object server to make remote method call to a client via a reference to a client remote interface.

Summary-5

- **Client callback:**
  - To provide client callback, the client-side software
    - supplies a remote interface,
    - instantiate an object which implements the interface,
    - passes a reference to the object to the server via a remote method call to the server.
  - The object server:
    - collects these client references in a data structure.
    - when the awaited event occurs, the object server invokes the callback method (defined in the client remote interface) to pass data to the client. Two sets of stub-skeletons are needed: one for the server remote interface, the other one for the client remote interface.
  - Two sets of stub-skeletons are needed: one for the server remote interface, the other one for the client remote interface.

Summary-6

- The key topics introduced with CORBA are:
  - The basic CORBA architecture and its emphasis on object interoperability and platform independence
  - Object Request Broker (ORB) and its functionalities
  - The Inter-ORB Protocol (IIOP) and its significance
  - CORBA object reference and the Interoperable Object Reference (IOR) protocol
  - **CORBA Naming Service** and the **Interoperable Naming Service (INS)**
  - Standard CORBA **object services** and how they are provided.
  - **Object adapters**, portable **object Adapters** (**POA**) and their significance.