CS 5523 Operating Systems: Remote Objects and RMI

Instructor: Dr. Tongping Liu

Thank Dr. Dakai Zhu and Dr. Palden Lama for providing their slides.
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;
Object-Oriented Programming

- Fundamental idea: encapsulation
  - both data and methods
- Interfaces – define the signature of a set of methods
- Object reference – access object’s data/method
- Method invocation – may affect target objects; may need chain invocation
- Data/variable – could be directly access
- Exceptions – thrown when an error occurs
Distributed Object

Client process:
- disObj1.m1(...);
- disObj1.m2(...);
- disObj2.m3(...);
- disObj2.m4(...);

Client machine

Machine a
- m1

Machine b
- m2

Machine c
- Obj: m3; m4;

Distributed Object

Distributed Object
Remote Objects vs. Message Passing

Host A

Client:

Sum = math.add(n1, n2);

Create a socket
Connect it to server
Put n1, n2 in a msg
Send msg to server
Read/wait reply msg
Extract result from the msg

Host B

Server:
int state;
object math;
int add(int n1, int n2)
{
    ……
}

Create a socket
Bind it to a port
Accept a connection
Read/wait for a msg
Extract n1, n2 from the msg
Compute result
Put it in a reply msg
Send reply msg to client
An Simple Remote Objects

Client:

Sum = math.add(n1, n2);

Server:

object math

int add(int n1, int n2){
    .......
}

proxy

add(n1, n2);

Skeleton

add(n1, n2);
Remote Object Model

- **Remote objects** -- can receive remote invocations; having state information.

- **Remote object reference** – identify remote objects in distributed environments

- **Remote interface** – specifies methods to be invoked remotely

- Process contains objects (local/remote)
  - Local objects: accept only local invocations
  - Remote object: accept both local/remote invocations
  - Remote invocation – different processes (same or different hosts)

- Exceptions – application level
An Example: Remote Object

- Object may implement both remote & local interface
  - Other processes: invoke only methods in remote interface
  - Same process: local object reference and invoke methods in local interface
Objects receiving remote invocations (service objects) are remote objects, e.g., B and F.

Object references are required for invocation, e.g., C must have E’s reference or B must have D’s reference.

What are the local object references?
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;
Remote Object Reference (ROR)

- **Uniquely** identify remote objects in distributed systems
- Needed to invoke remote method of a remote object
- Remote object references may be passed as input arguments or returned as output arguments.

**Compared with ordinary object reference, what additional information is needed for remote object reference?**

- Internet address: 32 bits
- Port number: 32 bits
- Time: 32 bits
- Object number: 32 bits
- Interface of remote object
Interfaces

- Interface for local objects
  - Specify methods and/or data that can be accessed
  - Do not specify an implementation

- Interface for remote objects
  - Specifies methods for remote invocation
  - Input and output parameters are also specified and parameters may be objects
Parameters for Remote Methods

- **Primitive types**
  - → pass by value

- **Ordinary objects**
  - → passed by copy (e.g. using Java serialization; the object must implement the `java.io.Serializable` Interface).

- **Remote objects**
  - → pass the remote object reference (ROR)

**Why not pass a copy of the remote object?**
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;
**RPC Review**

- **1. Client call to procedure**
  - Client machine
  - Client process
  - Client stub
  - k = add(i,j)
  - proc: "add"
  - int: val(i)
  - int: val(j)

- **2. Stub builds message**
  - Client OS
  - proc: "add"
  - int: val(i)
  - int: val(j)

- **3. Message is sent across the network**

- **4. Server OS hands message to server stub**
  - Server process
  - Implementation of add
  - k = add(i,j)
  - proc: "add"
  - int: val(i)
  - int: val(j)

- **5. Stub unpacks message**

- **6. Stub makes local call to "add"**

---

**Middleware generates stubs on both sides**
RMI Overview

How do clients know where the remote objects are?

- **Binding…**
  - **RMI register:** the string name of the object, the remote object itself
  - The **registry returns to the caller a reference (called stub)** to the remote object.
  - Invoke methods on the object (through the stub).

Object server is responsible for a collection of objects

- **Encapsulate** data and operations
- Object offers only its **interface** (group of methods) to clients,
  - With RMI support, clients can be at a different host
RPC vs. RMI

- **Similarity:**
  - Marshaling and parameter passing

- **Difference:**
  - **RPC:** C based, structure based semantics. **RMI:** java and object-oriented
  - **RPC:** call remote functions, passed everything. **RMI:** remote/system-wide object reference and invoke methods. We can also pass and return objects that can be distributed among many JVM instances, much more powerful.
  - **RMI** can support dynamic invocations.

```plaintext
fobject.append(int)    Invoke(fobject, id(append), int)
```
Architecture to Support Remote Objects

Diagram showing the communication between a client and a server for remote objects. The client side includes object A, proxy for B, and remote reference module. The server side includes the skeleton & dispatcher for B’s class, remote object B, and remote reference module. The diagram illustrates the request and reply communication between the client and server.
Remote Reference Module (RRM)

Server side:
- Create remote object reference (ROR)
- Maintain remote object reference table
- Entry for remote objects
- Map between remote reference and local reference

Client side:
- Create proxy object when first get ROR
- Maintain remote object reference table
- Entry for local proxy (client side)
- Map between remote reference and local reference
Communication Module

- Carry out request-reply protocol

- Provide certain invocation semantics
  - Retry request
  - Duplication message filtering
  - Reply message history cache

- Interact with remote reference module
  - Get remote object’s local reference
  - Pass message and local reference to appropriate dispatcher on server side
Proxy and Skeleton

Proxy - makes RMI transparent to client. Class implements remote interface. Marshals requests and unmarshals results. Forwards request.

Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.
Dispatcher - gets request from communication module and invokes method in skeleton (using *methodID* in message).
Middleware for Remote Objects

- Layer between application and communication/remote reference modules
- Automatically create proxy, skeleton and dispatcher from remote interface definition
- Client side: one proxy for each remote object
  - Implement the methods in remote objects
- Server side: one dispatcher and one skeleton
  - Dispatcher accepts message and select appropriate method in the skeleton: methodID
  - Skeleton: Marshall / unmarshall messages and invokes corresponding method in the remote object
Server/Client Programs and Binder

- **Server program**
  - Dispatcher, skeleton
  - Servant class: implement methods for remote objects

- **Client program**
  - Proxy
  - Use binder to get remote object reference

- **Binder**: kind of name service
  - Mapping between text name and remote object reference
  - System wide register/look up service
Steps in RMI

1. **Naming Service**
2. **Remote object B**
3. **ROR**
4. **Remote reference module**
5. **ROR**
6. **Communication module**
7. **Request**
8. **Reply**
9. **Proxy for B**
10. **Skeleton & dispatcher for B's class**
11. **Remote reference module**
12. **Communication module**
13. **Remote object B**
14. **ROR**
15. **ROR**
16. **Remote reference module**
17. **Communication module**
18. **ROR**

CS5253: Operating Systems @ UTSA
How to Use Remote Objects: Server Side

- Step 0: start binder
- Step 1: server start communication and remote reference module
- Step 2: server create remote object, add it to remote object table, and obtain remote object reference from remote reference module
- Step 3: server publish the remote object to name service → bind the remote object reference with a name; wait for invocation requests
Steps in RMI

1. Naming Service
2. Remote reference module for B
3. ROR
4. Request
5. ROR
6. Remote reference module
7. Communication module
8. Reply
9. Request
10. Reply
11. ROR
12. Skeleton & dispatcher for B’s class
13. Remote object B
14. Remote reference module
15. Communication module
16. Remote reference module
How to Use Remote Objects: Client Side

- Step 4: client start communication and remote reference module
- Step 5: client contact name service for desired remote object reference
- Step 6: client remote reference module create proxy
- Step 7: client call methods in proxy → mashall parameters
- Step 8: client locate remote object through remote reference module
- Step 9: Send method invocation request (contain remote object reference) through communication module
Steps in RMI

1. Naming Service
2. Object B
3. ROR
4. Remote reference module
5. ROR
6. Communication module
7. Request
8. Reply
9. Request
10. Reply
11. Request
12. Reply
13. Request
14. Reply
15. Request
16. Reply
17. Request
18. Reply

Object A
Proxy for B
Skeleton
Dispatcher for B’s class
How to Use Remote Objects: Server Side

- Step 10: server get invocation requests (contains remote object reference) through communication module
- Step 11: server consult with remote reference module and get local reference for the remote object
- Step 12: server hand the request to the dispatcher/skeleton of the remote object’s class \(\rightarrow\) which method
- Step 13: server call method in skeleton \(\rightarrow\) unmarshall parameters in request and invoke real-method in remote object
- Step 14: perform operation in remote object and return results to skeleton
- Step 15: server marshall results in skeleton and send out message (remote object reference) through communication module
How to Use Remote Objects: Client Side

- Step 16: get result message (contain remote object reference) in communication module
- Step 17: obtain proxy reference from remote object reference and hand the result message to the proxy
- Step 18: unmarshal the results in proxy and return results to the caller

*Complete a Remote Method Invocation 😊*

Why do people say RMI is simpler than sockets?

RMI = Sockets + Object Serialization + Some Utilities
User don’t need to worry about communication
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;
Case Study: Java RMI

- Define a remote interface
  
  HelloInterface.java

- Server side:
  
  - Implement the interface
    Hello.java
  
  - Develop the server
    HelloServer.java

- Client side:
  
  - Develop a client
    HelloClient.java

- Run the RMI registry, the server, and the client

```java
//HelloInterface.java
import java.rmi.*;

public interface HelloInterface extends Remote {

    public String add(String s) throws RemoteException;

    public String say() throws RemoteException;

}
```
import java.rmi.*; //Hello.java
import java.rmi.server.*;

public class Hello extends UnicastRemoteObject implements HelloInterface {
    private String message;

    public Hello (String msg) throws RemoteException {message = msg; }

    public String add(String more) throws RemoteException{
        message = new String (message + more); return message;  }

    public String say( ) throws RemoteException {return message; }
Server: Create Servant Object and Bind

//HelloServer.java
import java.rmi.*;
public class HelloServer{

    public static void main(String args[]){

        try {
            Naming.rebind("Hello", new Hello("Hello, world!"));
            System.out.println("Hello Server is ready.");
        } catch (Exception e) {
            System.out.println("Hello Server failed: " + e);
        }
    }
}
The binder: RMI Registry

For server

- Void **rebind** (String name, Remote obj)
  - Register an object by name
  - Override previous registration

- Void **bind** (String name, Remote obj)
  - Register an object by name
  - If existent throw exception

- Void **unbind** (String name, remote obj)

For Client

- Remote **lookup** (String name)
  - ROR is returned if found

- String[ ] **list**()
  - Show all names bound in this registry
Run the Server

- Compile the interface and remote class
  - javac HelloInterface.java Hello.java

- Compile server
  - javac HelloServer.java

- Generate skeletons & stubs (old Java compiler)
  - rmic Hello --> Hello_Skel.class & Hello_Stub.class

- Start RMI registry
  - rmiregistry (default at port 1099)

- Start Hello server
  - java HelloServer
Implementations of Client

- First, get remote object reference
  - It can retrieve a remote object reference from **RMIregistry** on the machine where the remote object resides
  - It received a remote object reference from a previous call

- Invoke methods specified in remote interface

- Compile the client code
  - Needs the interface file in order to compile
import java.rmi.*;
import java.rmi.server.*;
public class HelloClient{
public static void main (String[] argv) {
    try {
        HelloInterface hello =
            (HelloInterface) Naming.lookup ("//localhost/Hello");
        System.out.println (hello.say( ));
        System.out.println (hello.add("Here is added information!!!"));
    } catch (Exception e) {
        System.out.println ("HelloClient exception: " + e);
    }
}
Java RMI: Compile and Run the Client

- Compile the interface class
  javac HelloInterface.java

- Compile client
  javac HelloClient.java

- Start Hello client
  java HelloClient [X]

> java HelloClient
Hello, world!
Hello, world!Here is added information!!!
> java HelloClient
Hello, world!Here is added information!!!
Hello, world!Here is added information!!!Here is added information!!!
> java HelloClient
Hello, world!Here is added information!!!Here is added information!!!
Hello, world!Here is added information!!!Here is added information!!!Here is added information!!!
Summary of Java RMI Steps

- Design the remote interfaces being implemented
  
- Server create and to register the remote objects.
    - Implement remote methods in the remote interfaces
    - Create and install a security manager
      - Regulates whether the server itself might download classes when accessing other remote servers
    - Create one or more instances of a remote object
    - Register at least one of the remote objects with the RMI remote object registry (or another naming service)

- Client program to use the methods of remote object
  - Looks up server in remote object registry and gets ROR
  - Use normal method call syntax for calling remote method
Other Issues for Remote Objects

- When/who create remote objects?
  - Remote Object Factory method

- Can server contact client?
  - Call-back
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Remote Object Factory
  - RMI callbacks
  - CORBA case study
  - Transient vs. Permanent objects;
Remote Object Model

- **Remote objects** -- can receive remote invocations; having state information.

- **Remote object reference** – identify remote objects in distributed environments

- **Remote interface** – specifies methods to be invoked remotely

- Process contains objects (local/remote)
  - Local objects: accept only local invocations
  - Remote object: accept both local/remote invocations
  - Remote invocation – different processes (same or different hosts)

- Exceptions – application level
Remote Object Reference (ROR)

- **Uniquely** identify remote objects in distributed systems
- Needed to invoke remote method of a remote object
- Remote object references may be passed as input arguments or returned as output arguments.

Compared with ordinary object reference, what additional information is needed for remote object reference?

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
</tr>
</tbody>
</table>
**RPC Review**

**Middleware** generates stubs on both sides

1. Client call to procedure
2. Stub builds message
3. Message is sent across the network
4. Server OS hands message to server stub
5. Stub unpacks message
6. Stub makes local call to "add"
How do clients know where the remote objects are?  

**Binding…**  

- **RMI register**: the string name of the object, the remote object itself  
- The **registry returns to the caller a reference (called stub)** to the remote object.  
- Invoke methods on the object (through the stub).
RPC vs. RMI

**Similarity:**
- Marshaling and parameter passing

**Difference:**
- RPC: C based, structure based semantics. RMI: java and object-oriented
- RPC: call remote functions, passed everything. RMI: remote/system-wide object reference and invoke methods. We can also pass and return objects that can be distributed among many JVM instances, much more powerful.
- RMI can support dynamic invocations.

```java
fobject.append(int) Invoke(fobject, id(append), int)
```
Middleware for Remote Objects

- Layer between application and communication/remote reference modules
- Automatically create proxy, skeleton and dispatcher from remote interface definition
- Client side: one proxy for each remote object
  - Implement the methods in remote objects
- Server side: one dispatcher and one skeleton
  - Dispatcher accepts message and select appropriate method in the skeleton: methodID
  - Skeleton: Marshall / unmarshall messages and invokes corresponding method in the remote object
Proxy and Skeleton

Proxy - makes RMI transparent to client. Class implements remote interface. Marshals requests and unmarshals results. Forwards request.

Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.
Steps in RMI

1: ROR
2: Remote object B
3: ROR
4: Remote reference module
5: ROR
6: Communication module
7: Request
8: Reply
9: proxy for B
10: Request
11: Reply
12: skeleton & dispatcher for B’s class
13: remote object B
14: Remote reference module
15: Communication module

Naming Service
Case Study: Java RMI

- Define a remote interface
  HelloInterface.java

- Server side:
  - Implement the interface
    Hello.java
  - Develop the server
    HelloServer.java

- Client side:
  - Develop a client
    HelloClient.java

- Run the RMI registry, the server, and the client

```java
//HelloInterface.java
import java.rmi.*;

public interface HelloInterface
  extends Remote {

  public String add(String s)
      throws RemoteException;

  public String say()
      throws RemoteException;
}
```
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Remote Object Factory
  - RMI callbacks
  - CORBA case study
  - Transient vs. Permanent objects;
REMOTE OBJECT FACTORY

The **constructor** method of a remote object is **NOT** included in the interface. So, it cannot be called by the client.

Where the remote objects come from?

- **First remote object**: initiated by the server at startup and registered
- **Factory method (remote method) →** create remote objects, and return remote object reference (ROR) based on client’s requests
Factory Classes

- When a remote object reference (ROR) to a remote object is obtained through the RMI registry and then used to request additional RORs, *the registered remote object is referred to as a factory class.*

- Using RORs obtained through method calls on factory objects, client applications can dynamically request the creation of new remote objects, without the objects being registered individually with the server registry.
Factory Class Example

Consider a remote banking system using the Account object. The server provides services to remote clients running on PCs or ATMs etc.

One cumbersome situation:

- we run an RMI registry, create an Account object for every account we have on record, and register each one with the RMI registry using the account name.

```java
Registry local = LocateRegistry.getRegistry();
local.bind("Abrams, John", new AccountImpl("John Abrams"));
local.bind("Adams, John", new AccountImpl("John Adams"));
```

- Take long to start a server with thousands of accounts, while many are unnecessary
- Accounts need to be added or removed from the RMI registry
Factory Class Example

Define a factory class for *Account* objects, as in:

```java
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface AccountManager extends Remote {
    public Account getAccount(String name) throws RemoteException;
    public Boolean newAccount(Account s) throws RemoteException;
}
```

- *getAccount()* lets a client ask for an account, returning a remote reference to an *Account*.
- *Account* object is created on the server but is not registered with the registry, preventing the RMI registry in sync with the database.
- Only *AccountManager* object is registered with the registry. It can access the bank’s database directly to find accounts and create corresponding *Account* remote objects.
- *newAccount()* allows clients to add new accounts to the database.
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Remote Object Factory
  - RMI callbacks
  - CORBA case study
  - Transient vs. Permanent objects;
Enable the server to invoke the methods on the client.
RMI Callback

Typically in the client-server model,

- the server is passive:
- the IPC is initiated by the client;
- the server waits for requests and provides responses

Some applications require the server to initiate communication upon certain events such as

- Auctioning: user submits bid, server inform if a higher bid by others.
- chat-room: user type message, server forwards messages from other users. message/bulletin board etc.

With the RMI callback feature, client creates remote objects (callback objects) that implements an interface for server to call.

So we can now develop interactive distributed applications.
Callback (Remote Objects on Clients)

- **Callback**: server’s action in notifying the client
  - Instead of client **polling** the server, the server calls a method in the client when it is updated.

- Client creates a **remote object** that implements an interface for server to call.

- Server let clients to “**register**” their callbacks.

- When an event occurs, the server calls the interested clients.
In the absence of callback feature, how would a client be notified if it needs to know that a certain event has occurred at the server or not?

**Polling**

A client issues a request to the server repeatedly until the desired response is obtained.

**Callback**

A client registers itself with the server, and wait until the server calls back.

---

a remote method call
Good and Bad about Callbacks

Advantages

- More efficient than polling
- More timely than polling
- Provides a way of server inquiring about client status

Disadvantages

- May leave server with inconsistent state if client crashes or exits without notifying the server
- Requires the server to make a series of synchronous RMI’s
1. Client looks up the interface object in the RMI registry on the server host.
2. The RMI registry 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.
RMI Callback example

// Remote Interface that Server exports
public interface HelloInterface extends Remote {
    // remote method
    public String sayHello() throws java.rmi.RemoteException;
    // register itself to the callback list
    public void addCallback(
        HelloCallbackInterface CallbackObject
    ) throws java.rmi.RemoteException;
}

// Remote Interface for Callback Client
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;
}

http://www2.cs.uic.edu/~i441/RMICallback/
http://searchdaily.net/tag/callback-pattern-rmi-example/
http://docs.oracle.com/cd/E13211_01/wle/rmi/callbak.htm
Just google ….
public class HelloServer extends UnicastRemoteObject implements HelloInterface {

    // vector for store list of callback objects
    private static Vector callbackObjects;

    public HelloServer() throws RemoteException {
        callbackObjects = new Vector();
    }

    // method for clients to register their callbacks
    public void addCallback(HelloCallbackInterface CallbackObject) {
        // store the callback object into the vector
        callbackObjects.addElement(CallbackObject);
    }

    public static void main(String args[]) {
        registry = LocateRegistry.createRegistry(RMIPort);
        callback();
    }

    private static void callback() {
        for (int i = 0; i < callbackObjects.size(); i++) {
            // convert the vector object to a callback object
            HelloCallbackInterface client = (HelloCallbackInterface) callbackObjects.elementAt(i);
            client.callMe( "Server calling back to client " + i);
        }
    }
}
RMI Callback example (client)

```java
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("hello");
    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
```
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Remote Object Factory
  - RMI callbacks
  - CORBA case study
  - Transient vs. Permanent objects;
Case Study: CORBA (Common Object Request Broker Architecture)

Provides an interesting way of combining method invocation and message-oriented communication
CORBA History

- Starting from 1991 for distributed communication
- Claimed as “next-generation technology for e-commerce” before
  
  **Bleeding-edge technology** ➔ **Popular Middleware** ➔ **A niche technology in obscurity**

- Good: independent in os and languages, high tunability, comprehensive, inherent object-based approach
- Bad: location transparency, design deficiency, poor implementations
The Common Object Request Broker Architecture (CORBA) is a standard architecture for a distributed object-based system.

CORBA is designed to allow distributed objects to interoperate in a heterogeneous environment, where objects can be implemented in different programming language and/or deployed on different platforms.

Java RMI is platform independent too but it is language dependent.
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 language only.
- CORBA is a suite of specifications developed by the Object Management Group (OMG), http://www.omg.org/
  ✔ Using a facility supporting CORBA, objects can be written in any language
CORBA

- CORBA is not itself 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. But we will focus on the key concepts of CORBA related to the distributed objects paradigm.
The Basic Architecture

naming service

object client

stub

ORB

network

operating system

object implementation

skeleton

ORB

network

operating system

logical data flow

physical data flow

Distributed Computing, M. L. Liu
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, C++

- Object defined in a CORBA IDL file can be implemented in a large number of diverse programming languages, e.g., C/C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLScript.

For each language, OMG has a standardized mapping from CORBA IDL to the language,

- So 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

- Object client written in Java
- Stub in Java generated by compiling the CORBA object interface
- ORB written in Java

- Object implementation written in C++
- Skeleton in C++ generated by compiling the CORBA object interface
- ORB written in C++

Distributed Computing, M. L. Liu
CORBA Interfaces

- Specify a name and a set of methods
- Parameters and results about methods
  - Parameters are specified by keywords: **in**, **out**, or **inout**
  - Primitive type or constructed type → pass by value
  - CORBA object of interface type → remote object reference
- User-defined exceptions in interfaces and thrown by methods in the interface
- At-most-once invocation semantics by default

```c
int add(in int I, in int j, out int k);
```
Asynchronous Method Invocation

Basic rule: Asynchronous method invocation do not affect the implementation of an object

Server handle normal sync invocation
Client’s communication system handles it

```
int add(in int i, in int j, out int k);
```

```
//Downcall by the client
void sendcb_add(in int i, in int j);
```

```
//Upcall to the client
Void replycb_add(in int ret, in int k);
```
Messaging: Async. Method Invocation

Callback model

Polling model
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Remote Object Factory
  - RMI callbacks
  - CORBA case study
  - Transient vs. Permanent objects;
Transient vs. Permanent Remote Objects

- **Transient Remote Objects**: exist within the process that creates the object
  - Good: allocate resources when necessary
  - Bad: may take time for invocations

- **Persistent Objects** can survive when process dies and be later activated by new process. Must be persistent to storage.
  - Long run objects: sleep for efficient resource usage and activated whenever necessary
  - **Persistent object store** provides a simple storage management (like a database system)
States for Persistent Objects

- **Active**: Ready for method invocation
- **Passive**: not active, and cannot accept invocation
  - A passive object consists of two parts
    - Implementation of its methods
    - Internal state in marshalled form

**Activator**
- Registering passive objects that are available for activation
- Activate: start server and active remote object within it
- *Keeping track of locations of servers for remote objects that it has activated.*

**When to store information about object in persistent storage?**
- At the time of passivated
- At end of important operations (e.g. end of transaction)
Object Location

- Remote object reference contains IP address and port # of server process to guarantee uniqueness
  - Advantage: can be used as address
  - Disadvantage: object cannot be migrated to other server

Location service – to support migration

- Database to map remote object reference to their current location
- Location services can be replicated on each machine; information on machines is kept more or less consistent by some update propagation mechanism
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Remote Object Factory
  - RMI callbacks
  - CORBA case study
  - Transient vs. Permanent objects;