Al Sham Private University
Faculty of Informatics Engineering

جامعة الشام الخاصة
كلية الهندسة المعلوماتية

نظم معلومات موزعة
Distributed Information Systems

Lecture 5: Remote method invocation (RMI)

اعداد: أ. غاندي هسام

Introduction

- Remote method invocation (RMI) is closely related to RPC but extended into the world of distributed objects.
- In RMI, a calling object can invoke a method in a potentially remote object and constructed on top of request-reply protocols.
- The programmer is able to use the full expressive power of object-oriented programming in the development of distributed systems software.
- all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.

- RMI allows the programmer to pass parameters not only by value, as input or output parameters, but also by object reference.
- Passing references is particularly attractive if the underlying parameter is large or complex.
- The remote end, on receiving an object reference, can then access this object using remote method invocation, instead of having to transmit the object value across the network.

Design issues for RMI

The object model

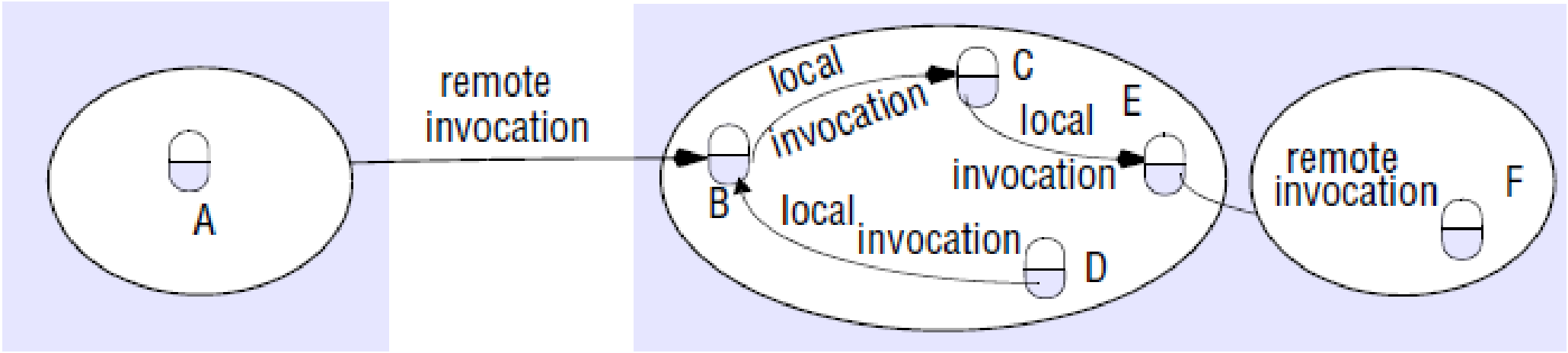
- Object references: Objects can be accessed via object references. For example, in Java, a variable that appears to hold an object actually holds a reference to that object.
- Interfaces: An interface provides a definition of the signatures of a set of methods (that is, the types of their arguments, return values and exceptions) without specifying their implementation.
- Actions : Action in an object-oriented program is initiated by an object invoking a method in another object. An invocation can include additional information (arguments) needed to carry out the method.
- Exceptions: Programs can encounter many sorts of errors and unexpected conditions of varying seriousness.
- Garbage collection: It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed.

Distributed objects

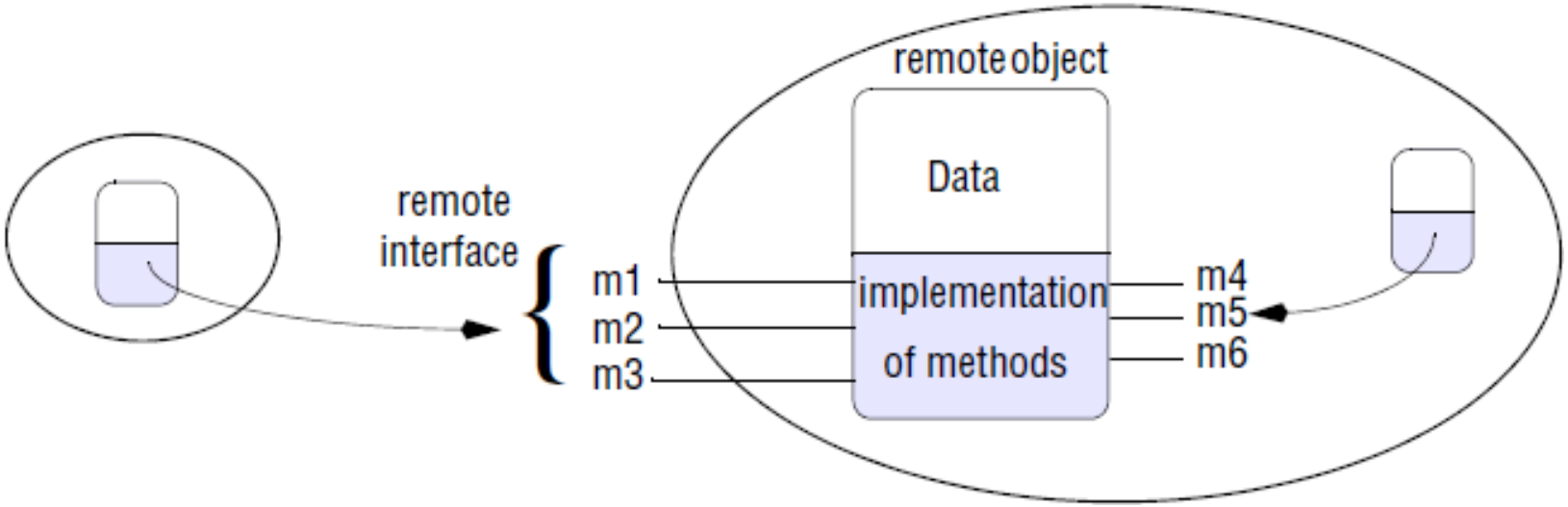
- Distributed object systems may adopt the client-server architecture. In this case, objects are managed by servers and their clients invoke their methods using remote method invocation.
- In RMI, the client's request to invoke a method of an object is sent in a message to the server managing the object. The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message.
- Distributed objects can assume other architectural models. For example, objects can be replicated in order to obtain the usual benefits of fault tolerance and enhanced performance, and objects can be migrated with a view to enhancing their performance and availability.

The distributed object model

- Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations
- We refer to objects that can receive remote invocations as ***remote objects***.
- the objects B and F are remote objects.



- **Remote object references**: Other objects can invoke the methods of a remote object if they have access to its remote object reference. For example, a remote object reference for B must be available to A.
- **Remote interfaces**: Every remote object has a remote interface that specifies which of its methods can be invoked remotely. For example, the objects B and F must have remote interfaces.



Case study: *Java RMI*

How to?

- It allows objects to invoke methods on remote objects using the same syntax as for local invocations.
- an object making a remote invocation is aware that its target is remote because it must handle *RemoteExceptions.*
- implementor of a remote object is aware that it is remote because it must implement the Remote interface.
- Remote interfaces are defined by extending an interface called *Remote* provided in the java.rmi package.

Java Remote interfaces

```
import java.rmi.RemoteException;
import java.rmi.*;
/**
 *
 * @author Ghandy Hessam
 */
public interface Square_Shape extends Remote
{

    public float area ( float side ) throws RemoteException;
        public float perimeter(float side)throws RemoteException;
}
```

- This interface has to be shared between server and client.
- Remote interfaces are defined by extending an interface called Remote provided in the java.rmi package.
- The methods must throw RemoteException, but application-specific exceptions may also be thrown.

The server

- The server program is a simplified version of a whiteboard server that implements the interface *Square_Shape*.
- This implementation called **servant**.
- In our example the servant called *Square_imp*. Using *UnicastRemoteObject* ensures that the resultant object lives only as long as the process in which it is created (an alternative is to make this an
- Activatable object that is, one that lives beyond the server instance).
- The server program consists of a main method and a servant class to implement each of its remote interfaces.

Servant class to implement the remote interface

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/**
 *
 * @author Ghandy Hessam
 */
public class Square_imp extends UnicastRemoteObject implements Square_Shape {
      public Square_imp() throws RemoteException
    {
        super();
    }
    public float area(float side) throws RemoteException {
        return side*side;
    }

    public float perimeter(float side) throws RemoteException {
        return 4*side;
    }

}
```

The *main* method of the server class

```
import java.rmi.*;
/**
 *
 * @author Ghandy Hessam
 */
public class Server{

     public static void main ( String args[] ) throws Exception
     {
         try {

             Registry  registry = LocateRegistry.createRegistry( 9999 );
             registry.rebind("Square", new Square_imp());

         } catch (Exception e) {
             e.printStackTrace();
         }
         System.out.println("SERVER is ready");


     }
}
```

The client

- It is necessary to run an instance of the *Registry* in the networked environment and then use the class *LocateRegistry*, which is in java.rmi.registry, to discover this registry.
- More specifically, this class contains a *getRegistry* method that returns an object of type *Registry* representing the remote binding service.
- Any client program needs to get started by using a binder to look up a remote object reference using the *lookup* operation of the RMIregistry.

The *main* method of the client class

```
import java.rmi.*;
/**
 *
 * @author Ghandy Hessam
 */
public class client{

     public static void main ( String args[] ) throws Exception
    {
        try {
            Registry myRegistry = LocateRegistry.getRegistry("127.0.0.1", 9999);
            // search for Server service
            Square_Shape  s =(Square_Shape) myRegistry.lookup("Square");
            // call server's method
            float f= 5.0f;
            float area = s.area(f);
           System.out.println("area is: "+area);
        } catch (Exception e) {
            System.out.println(e);
        }
     }
}
```

End of Lecture 5