Client Server Application Programming

Week 12: Remote Method Invocation Deployment Issues, Using Remote Method Invocation to Implement Callbacks, Remote Object Activation)

By Elubu Joseph - MSc.IS
Lecturer
Department of Information Technology
Kumi University
Email: josebulinda@gmail.com
jose@kumiuniversity.ac.ug

Summary of Previous Lecture

1. Recap on Creating Stub and Skeleton Classes
2. Creating an RMI Server,
3. Creating an RMI Client,
4. Running the RMI System,
5. RMI Packages and Classes,

Agenda

1. Remote Method Invocation Deployment Issues
2. Using RMI to Implement Callbacks
3. Remote Object Activation

Remote Method Invocation Deployment Issues

Remote Method Invocation Deployment Issues

During development and testing, Remote Method Invocation runs very smoothly and with very few problems (other than perhaps out-of-date stub files if a service interface or implementation is changed). However, when it comes to deploying RMI services and clients on a network, some additional complexities may be encountered.

Class definitions for services and clients must be distributed, and there are issues with;

1. Differences between JVM implementations (such as a Microsoft JVM versus one from Sun Microsystems)
2. Differences between JDK versions.
3. Finally, applet communication with RMI services is an issue that must be considered during deployment.

Remote Method Invocation Deployment Issues+
Differences between Java Virtual Machines

There are many differences in the support of RMI between the various editions of the Java platform. These deployment issues can affect clients that use the software, as well as RMI servers that may be located across a wide variety of JVMs running on machines throughout a network.

1. Lack of Remote Method Invocation Support in Microsoft JVMs

Despite the fact that the RMI packages are part of the "core" Java API, **RMI is not generally supported by Microsoft JVMs**.

This means that RMI clients or servers cannot be easily run on these JVMs, making the use of RMI in applets difficult, as Microsoft web browsers are widely used by a large number of Internet users. An additional download is available from Microsoft to patch the Microsoft JVM to support RMI; however, it is used infrequently and thus installation on client machines can't be guaranteed.

Remote Method Invocation Deployment Issues+
Differences between Java Virtual Machines

2. Changes in Remote Method Invocation from JDK versions

For example Under JDK1.02, implementations of an RMI service would extend the java. rmi.server.UnicastRemoteServer class. This class is not available under JDK1.1, and should be replaced with java.rmi.server.UnicastRemoteObject. Support for RMI in JDK1.02 was an interim release only, and should be avoided in production systems. The availability of JDK1.1 or higher JVMs on most operating system platforms makes it a better choice for RMI.

Generally, although it has been minimized, whenever a new version of JDK is released, there may be some differences in the way RMI Applications communicates.

Remote Method Invocation Deployment Issues+

Remote Method Invocation and Applets

RMI presents some unique difficulties for applets. Aside from issues of RMI support in Microsoft web browsers (discussed earlier), there is also the problem of applet network restrictions. These restrictions can be very serious, and prevent RMI-based applets from communicating with RMI services. We look are two issues here;

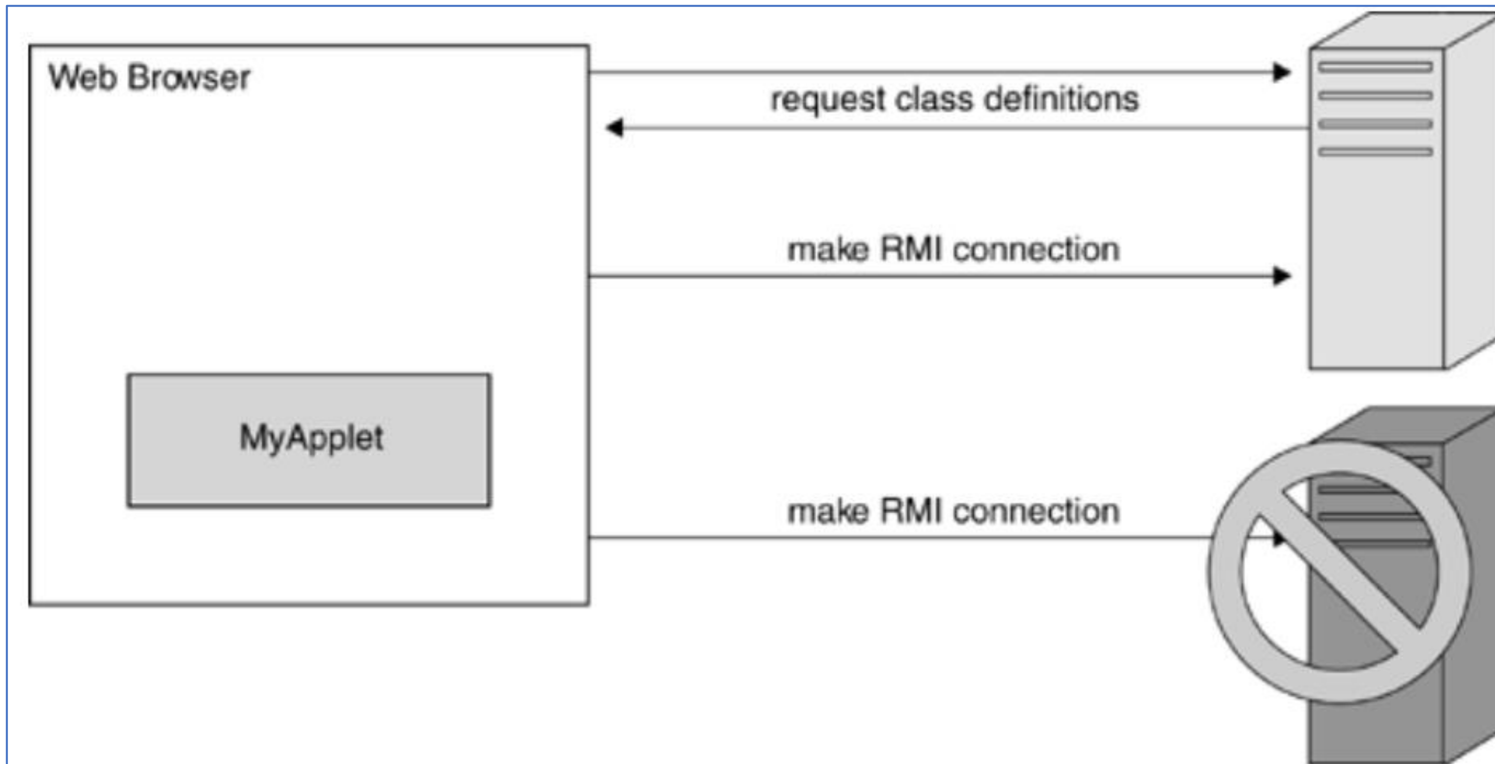
1. **Inability to Bind to TCP Ports**

   Since an applet can't bind to a TCP port, it cannot be an RMI server. While not every applet will need this functionality, it does mean that RMI services cannot be run from inside a Web browser. This means that callbacks cannot be easily implemented using RMI—a client applet can invoke methods on a server, but cannot export a remote object and have it invoked during the callback by another service.

2. **Restrictions on Network Connections**

   An applet can only connect to the host machine from which its codebase was loaded. When an applet is loaded from a machine, such as www.kumu.com, **for example,** it can only make TCP and UDP connections to that server (see Figure below).

Remote Method Invocation Deployment Issues+
Remote Method Invocation and Applets



Michael R. and David R(2002)

This is a restriction placed on applets by browsers; while the restriction is in the best interest of users, it limits developers.

Remote Method Invocation Deployment Issues+

Remote Method Invocation and Applets

3. Firewall Restrictions

A considerable number of Web users are protected by a firewall, which usually prohibits direct socket connections. Many companies, organizations, and other institutions use firewalls to protect themselves against security breaches from outside and to restrict the types of applications being run from the inside.

This means that some users will be able to use an applet that used RMI, and others will not. Even a digitally signed applet won't get through, unless RMI requests are tunneled through the firewall using a trusted protocol such as HTTP.

Tunneling allows one protocol to be piggybacked on top of another more safe protocol, to overcome the limitations of firewalls and network security.

Using Remote Method Invocation to Implement Callbacks

Using Remote Method Invocation to Implement Callbacks

A common technique used in event-driven programming is the callback. The simplest way of understanding a callback is to think of a phone call.

Suppose you want to know if the landlord of a given apartment is back, and you ask one of the landlord tenants to call you when he/she is available. When the said tenant notices that the landlord is available, he or she calls you back, to notify you of the landlords presence. That's a callback

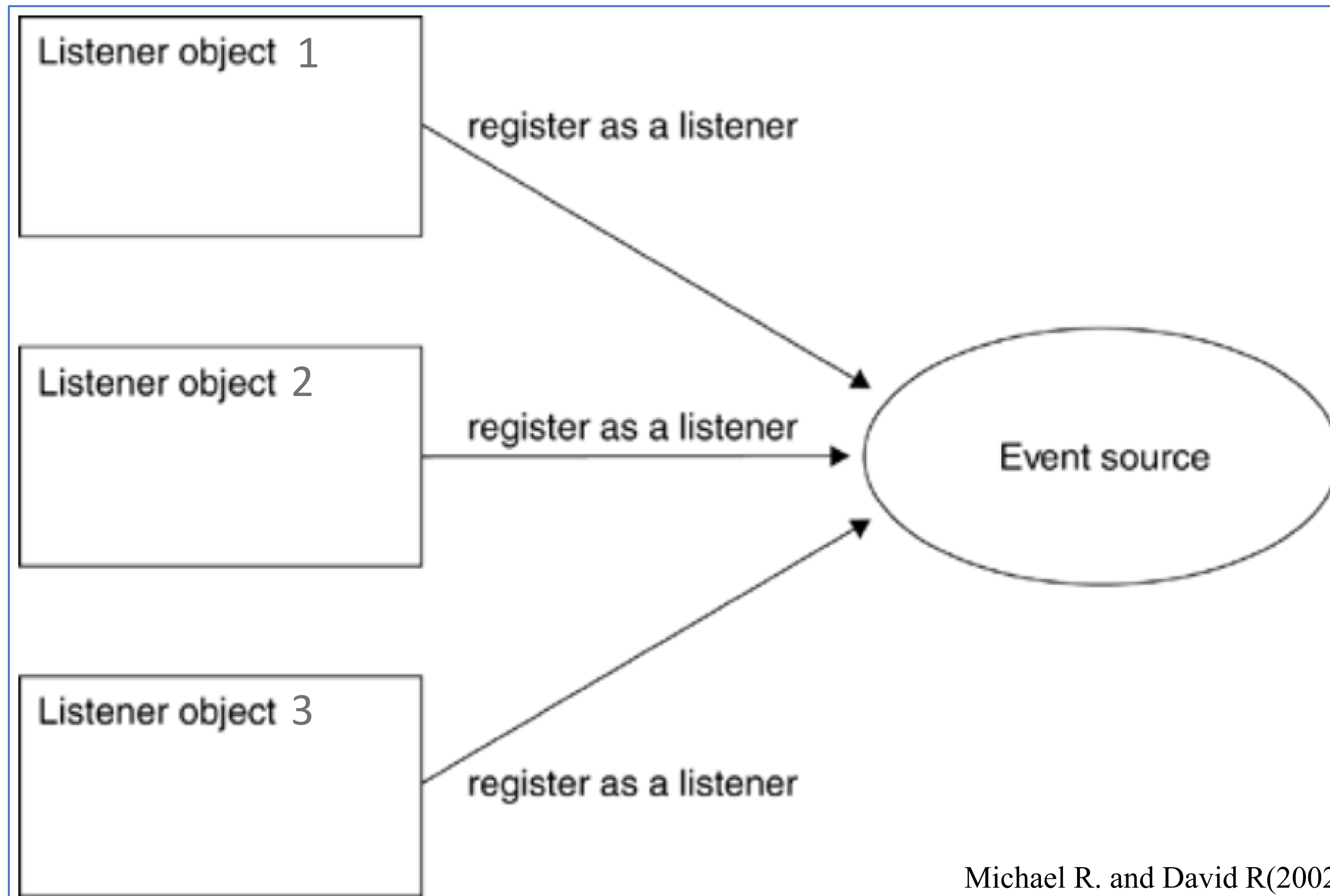
Using Remote Method Invocation to Implement Callbacks+**Object-Oriented Callbacks**

In an object-oriented system, we talk about software components or entire systems, not people. When one object needs to be notified of events as they occur (as opposed to continually polling to see if an event has happened), a callback is used.

This is often a more efficient way of implementing a system, as the object doesn't need to periodically check the state of another object—it is instead notified if an event is triggered. Multiple listeners can register with the same event source, as shown in the figure below.

Using Remote Method Invocation to Implement Callbacks+Object-Oriented Callbacks

Multiple listeners can register with one or more event sources



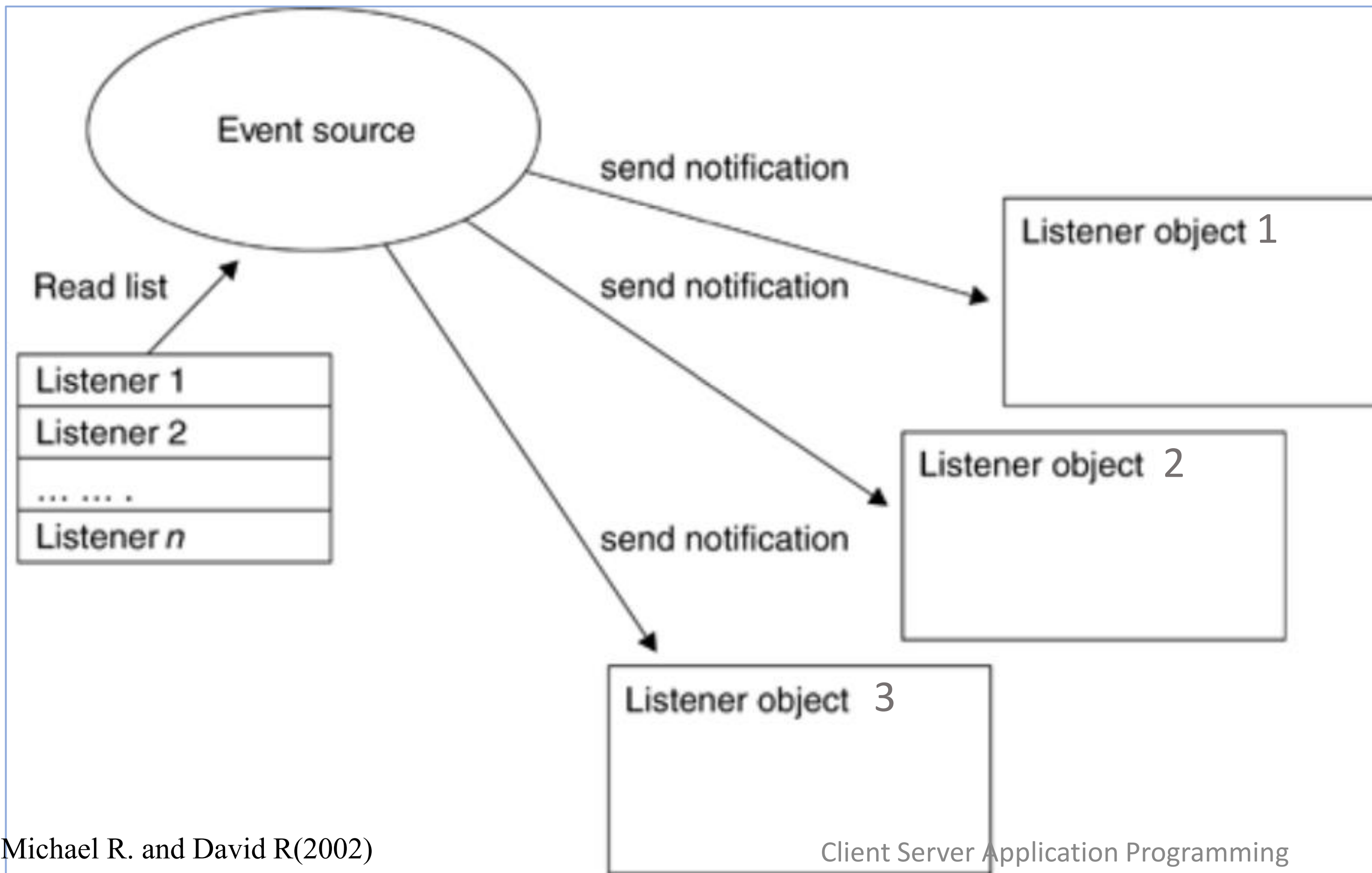
Michael R. and David R(2002)

When an event is triggered (either as a result of an interaction with external objects or systems, or an internal process such as completing a work unit or interacting with a user), the event source will notify each and every registered listener (see Figure on the next slide).

This means that the event source must maintain a list of active listeners, and then pass a message back to them that an event has occurred, and optionally give some details.

Using Remote Method Invocation to Implement Callbacks+**Object-Oriented Callbacks+**

Callback notification of event, for every registered listener



The next issue to understand about callbacks is **how the notification is achieved.** Some systems are designed to send and receive messages, but how can one object pass information to another (and hence inform the object that an event has occurred)?

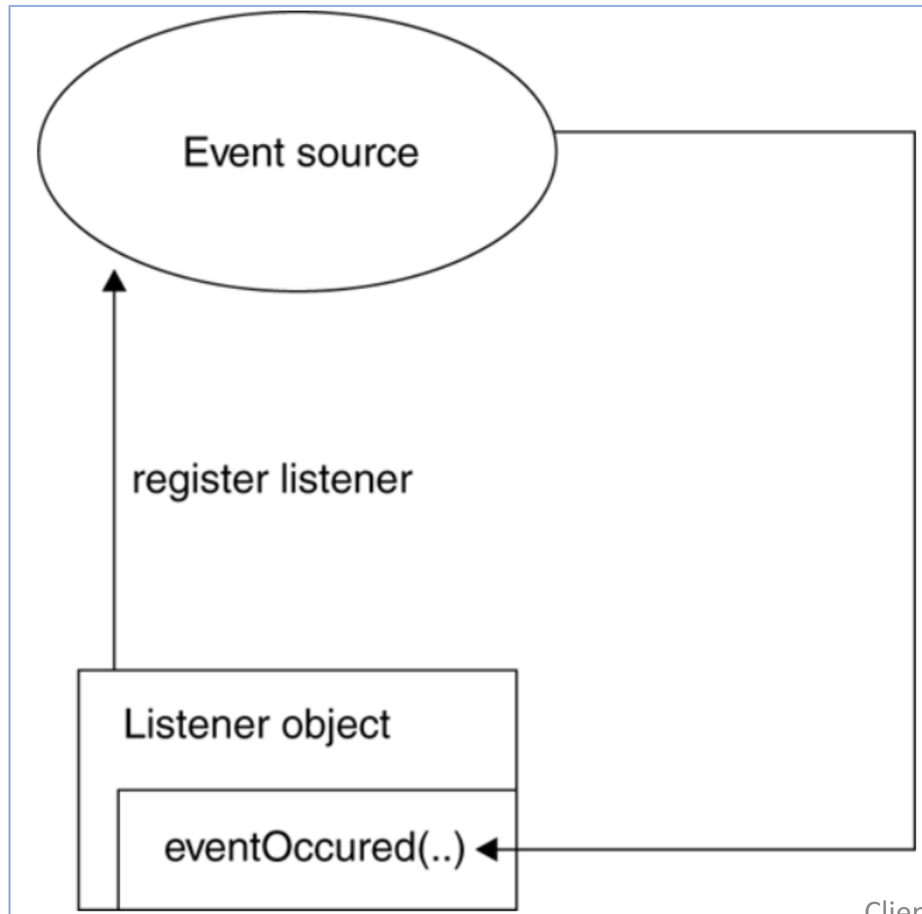
While you may not realize it, objects do this all the time, **by invoking methods**.

In object-oriented systems, callbacks are achieved by passing an object reference (the listener) to the object that is the event source.

Michael R. and David R(2002)

Using Remote Method Invocation to Implement Callbacks+**Object-Oriented Callbacks++**

Callback implemented by invoking a method on a listening object



Michael R. and David R(2002)

This type of system is widely used in object-oriented programming, and in the context of Java, the Abstract Windowing Toolkit. The AWT event-handling model requires application developers to implement a listener interface and have that listener register with each component that needs to be monitored.

Multiple listeners can register with the same component, and listeners can also register with multiple components (although due care must be taken to check which component was the source in this case).

When building a system, you can create your own listener interfaces, and add for each type of event relevant methods to register and deregister listeners. Well, implementing callbacks in local objects is one thing, what about callbacks using RMI?

RMI Callbacks

Callbacks can be easily implemented using RMI. The **major difference** between a **local callback** and a **remote callback** is that some **extra initialization must occur**, and that both the **listener interface** and the **event source** must be **implemented as an RMI service**.

For the listener to register itself with the remote event source, it must invoke a remote method and pass an object reference to the remote listener interface it defines. This sounds complex in theory, but is reasonably straightforward in practice, requiring only a small amount of additional code.

RMI Callbacks+

Example Temperature Change monitoring system

This next example works through implementing a callback using RMI, and shows how a listener can be a client and a server.

For the purpose of this example, the server supplies information about changes in **temperature**, which can be accessed remotely.

However, it would be inefficient to check the monitor continually, polling for changes, as the time at which a change will occur is not known. **A temperature monitor** registers with a **temperature sensor service**, and passes a reference to a **remote temperature listener object**. When the temperature does change, a callback is made, notifying registered listeners.

RMI Callbacks+Example

Defining the Listener Interface called TemperatureListener

The listener interface defines a **remote object** with a **single method**. This method should be invoked by an **event source** whenever an event occurs, so as to act as notification that the event occurred. The method signifies a change in temperature, and allows the new temperature to be passed as a parameter.

```
Import java.rmi.*;
interface TemperatureListener extends Remote{
	public void temperatureChanged(double temperature)
	throws RemoteException;
}
```

RMI Callbacks+

Defining the Event source Interface called TemperatureSensor

The **event source** must allow a listener to be registered and unregistered, and may optionally provide **additional methods.** In this case, a method to request the temperature on demand is offered.

```
Import java.rmi.*;
interface TemperatureSensor extends Remote{
   public double getTemperature() throws RemoteException;
   public void addTemperatureListener
   (TemperatureListener listener ) throws RemoteException;
   public void removeTemperatureListener (TemperatureListener
   listener )throws RemoteException;


   }
```

RMI Callbacks+ Implementing the Event Source Interface + TemperatureSensorServer class

Once interfaces have been defined, the next step is to implement them. A TemperatureSensorServer class is defined, which acts as an RMI server.

This server will also notify registered listeners as a client. The server must extend UnicastRemoteObject, to offer a service, and implement the TemperatureSensor interface defined earlier.

After creating an instance of the service and registering it with the rmiregistry, the server launches a new thread, responsible for updating the value of the temperature, based on randomly generated numbers.

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureSensorServer class+

Both the amount (plus or minus 0.5 degrees) and the time delay between changes are generated randomly. As each change occurs, registered listeners are notified, by reading from a list of listeners stored in a java.util.Vector object.

This list is modified by the remote addTemperatureListener(TemperatureListener) and removeTemperatureListener(Tempera tureListener) methods. Additionally, if a remote exception occurs while notifying a listener (indicating the temperature client is inaccessible or has terminated), it will be dropped.

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureSensorServer class+Code

```
1    package cs;
2    import java.util.*; import java.rmi.*;
3    import java.rmi.server.*; import java.rmi.registry.LocateRegistry;
4    public class TemperatureSensorServer extends UnicastRemoteObject implements
5            TemperatureSensor, Runnable {
6        private volatile double temp;
         private Vector<TemperatureListener> list = new Vector<>();
8        public TemperatureSensorServer() throws java.rmi.RemoteException {...4 line
         public double getTemperature() throws java.rmi.RemoteException {...3 lines
15
         public void addTemperatureListener(TemperatureListener listener) throws
17               java.rmi.RemoteException {...4 lines }
         public void removeTemperatureListener(TemperatureListener listener) throws
22               java.rmi.RemoteException {...4 lines }
         public void run() {...23 lines }
49
50       private void notifyListeners() {...14 lines }
64
65       public static void main(String args[]) {...23 lines }
88   }
```

RMI Callbacks+ Implementing the Event Source Interface+
TemperatureSensorServer class+Code+

Lest expand the methods of the class.

1. Constructor

```
private volatile double temp;
private Vector list = new Vector();
public TemperatureSensorServer() throws java.rmi.RemoteException{
// Assign a default setting for the temperature
temp = 98.0;
}
```

2. getTemperature() method

```
public double getTemperature() throws java.rmi.RemoteException{
     return temp;
}
```

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureSensorServer class+Code++

3. addTemperatureListener() method

```
public void addTemperatureListener ( TemperatureListener
                    listener )throws java.rmi.RemoteException{
    System.out.println ("adding listener -" + listener);
    list.add (listener);
}
```

4. removeTemperatureListener() method

```
public void removeTemperatureListener ( TemperatureListener
listener )throws java.rmi.RemoteException {
System.out.println ("removing listener -" + listener);
list.remove (listener);
}
```

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureSensorServer class+Code+++

5. run() method

```
public void run(){
Random r = new Random();
for (;;){
try{
    // Sleep for a random amount of time
    int duration = r.nextInt() % 10000 + 2000;
    // Check to see if negative, if so, reverse
    if (duration < 0) duration = duration * -1;
    Thread.sleep(duration);
    }
```

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureSensorServer class+Code++

5. run() method

```
catch (InterruptedException ie) { }
    // Get a number, to see if temp goes up or down
    int num = r.nextInt();
    if (num < 0){
    temp += 0.5;
}
else{
    temp -= 0.5;
}
// Notify registered listeners
notifyListeners();
}
}
```

RMI Callbacks+ Implementing the Event Source Interface+
TemperatureSensorServer class+Code+++++

7. The main() method

```
public static void main(String args[]) {
    System.out.println("Loading temperature service");
    try {
        // Load the service
        TemperatureSensorServer sensor = new TemperatureSensorServer();

        // Start the RMI registry on port 1099
          LocateRegistry.createRegistry(1099);

        // Register with service so that clients can find us
        Naming.rebind("rmi://localhost:1099/TemperatureSensor", sensor);
        System.out.println("Temperature service registered and running.");
```

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureSensorServer class+Code+++++

7. The main() method+

```
            // Create a thread, and pass the sensor server.This will
            // activate the run()and trigger regular temperature changes.
            Thread thread = new Thread(sensor);
            thread.start();
        } catch (RemoteException re) {
            System.err.println("Remote Error - " + re);
        } catch (Exception e) {
            System.err.println("Error - " + e);
        }
    }
}
```

RMI Callbacks+ Implementing the Event Source Interface+
TemperatureSensorServer Code Explained

The code provided above is an implementation of a temperature sensor server using Java's RMI (Remote Method Invocation) technology. Let's go through the code step by step:

1. The code starts with the package declaration package cs;, indicating that the classes in this file belong to the cs package.
2. The necessary imports are made to include the required Java classes and interfaces: **java.util.***, **java.rmi.*,** and **java.rmi.server.*.** These imports provide the necessary classes for RMI functionality.
3. The class **TemperatureSensorServer** is declared. It extends the **UnicastRemoteObject** class and implements the **TemperatureSensor** and **Runnable** interfaces.
4. The class contains two member variables: **temp** and **list**. temp is a volatile double variable that represents the current temperature value. list is a Vector that holds the registered **TemperatureListener** objects.

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureSensorServer Code Explained+

5. The constructor **TemperatureSensorServer()** initializes the temp variable with a default **temperature** value of **98.0**.

6. The **getTemperature()** method is implemented as part of the **TemperatureSensor** interface and returns the current temperature value.

7. The **addTemperatureListener()** method is implemented to add a **TemperatureListener** to the list vector.

8. The **removeTemperatureListener()** method is implemented to remove a **TemperatureListener** from the list vector.

RMI Callbacks+ Implementing the Event Source Interface+
TemperatureSensorServer Code Explained++

9. The **run()** method is implemented from the **Runnable** interface. This method runs an **infinite loop** that simulates **temperature** changes. It generates a random sleep duration, **adds** or **subtracts 0.5** from the temperature value, and then calls the **notifyListeners()** method to inform the registered listeners about the temperature change.

10. The **notifyListeners()** method iterates over the registered TemperatureListener objects in the list vector and calls the **temperatureChanged()** method on each listener. If a listener throws a **RemoteException**, indicating a communication error, it is removed from the list vector.

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureSensorServer Code Explained+++

11. The **main()** method is the entry point of the application. It performs the following tasks:

i. Prints a message to indicate that the temperature service is being loaded.

ii. Creates an instance of the **TemperatureSensorServer** class.

iii. Retrieves the registry hostname (defaulting to "localhost") from the command-line arguments, if provided.

iv. Constructs the registration string for the RMI service using the specified registry hostname.

v. Binds the sensor object to the RMI registry using the **Naming.rebind()** method.

vi. Creates a new thread and starts it, passing the sensor object. This activates the **run()** method and begins simulating temperature changes.

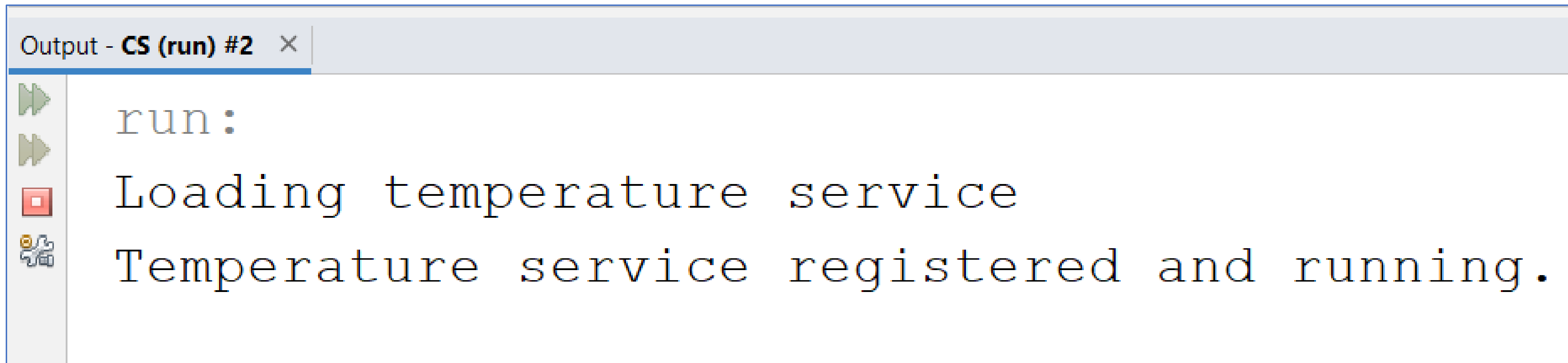
RMI Callbacks+ Implementing the Event Source Interface+
TemperatureSensorServer Code Explained++++

12. The code includes exception handling to catch and display any errors that occur during the RMI setup or execution.

In summary, this code sets up a temperature sensor server using RMI, allowing clients to register as listeners and receive notifications about temperature changes. The server continuously updates the temperature and notifies all registered listeners about the changes.

RMI Callbacks+ Implementing the Event Source Interface+
TemperatureSensorServer Code Output

Temperature Server-Side output before connection the client



TemperatureMonitor class

RMI Callbacks+ Implementing the Listener Interface+ Code

TemperatureMonitor class

The **temperature monitor** client must implement the TemperatureListener interface, and register itself with the remote temperature sensor service, by invoking the TemperatureSensor.addTemperatureListener (Temperature Listener) method.

By registering as a listener, the monitor client will be notified of changes as they occur, using a remote callback.

The client waits patiently for any changes, and though it does not ever remove itself as a listener, functionality to achieve this is supplied by the TemperatureSensor.removeTemper atureListener(TemperatureListener)method.

RMI Callbacks+ Implementing the TemperatureListener Interface+

TemperatureMonitor class

Has three methods: its own constructor, **main()** method and **temperatureChanged()** method.

```
1    package cs;
2    import java.rmi.*;  import java.rmi.server.*;
3    public class TemperatureMonitor extends UnicastRemoteObject implements
4            TemperatureListener {
5        // Default constructor throws a RemoteException
6        public TemperatureMonitor() throws RemoteException {...3 lines }
9
         public void temperatureChanged(double temperature) throws
11               java.rmi.RemoteException {...3 lines }
14
15       public static void main(String args[]) {...25 lines }
40   }
```

RMI Callbacks+ Implementing the TemperatureListener Interface+

TemperatureMonitor class+

We will now expand this class methods.

1. TemperatureMonitor() constructor and 2 temperatureChanged() method

```
// Default constructor throws a RemoteException
public TemperatureMonitor() throws RemoteException{
// no code req'd
}

public void temperatureChanged(double temperature)
    throws java.rmi.RemoteException{
System.out.println ("Temperature change event : " +temperature);
}
```

RMI Callbacks+ Implementing the TemperatureListener Interface+

TemperatureMonitor class++

3. The main() method

```
public static void main(String args[]) {
    System.out.println("Looking for temperature sensor");
    try {

    // Lookup the service in the registry, and obtain a remote service
    TemperatureSensor remoteService = (TemperatureSensor)Naming.lookup("rmi"
            + "://localhost:1099/TemperatureSensor");

        // Cast to a TemperatureSensor interface
        TemperatureSensor sensor = (TemperatureSensor) remoteService;
        // Get and display current temperature
        double reading = sensor.getTemperature();
        System.out.println("Original temp: " + reading);
```

RMI Callbacks+ Implementing the TemperatureListener Interface+

TemperatureMonitor class+++

3. The main() method+

```
            // Create a new monitor and register it as a listener with the
            //remote sensor
            TemperatureMonitor monitor = new TemperatureMonitor();
            sensor.addTemperatureListener(monitor);
        } catch (NotBoundException nbe) {
            System.out.println("No sensors available");
        } catch (RemoteException re) {
            System.out.println("RMI Error - " + re);
        } catch (Exception e) {
            System.out.println("Error - " + e);
        }
}
```

RMI Callbacks+ Implementing the Listener Interface+

TemperatureMonitor class code Explained

The code provided above is an implementation of a temperature monitor client that interacts with a temperature sensor server using Java's RMI (Remote Method Invocation) technology.

Let's go through the code step by step:

1. The code starts with the package declaration package cs;, indicating that the classes in this file belong to the cs package.
2. The necessary imports are made to include the required Java classes and interfaces: **java.rmi.*** and **java.rmi.server.***. These imports provide the necessary classes for RMI functionality.
3. The class **TemperatureMonitor** is declared. It extends the **UnicastRemoteObject** class and implements the **TemperatureListener** interface.

RMI Callbacks+ Implementing the Listener Interface+

TemperatureMonitor class code Explained+

4. The constructor TemperatureMonitor() is implemented, which does not have any code inside it. It is declared to throw a RemoteException.
5. The temperatureChanged() method is implemented from the TemperatureListener interface. This method is called by the temperature sensor server to notify the monitor client about a temperature change. In this implementation, it simply prints a message with the new temperature value.
6. The **main()** method is the entry point of the application. It performs the following tasks:
   i. Prints a message to indicate that the client is looking for a temperature sensor.
   ii. Retrieves the registry hostname (defaulting to "localhost") from the command-line arguments, if provided.

RMI Callbacks+ Implementing the Listener Interface+

TemperatureMonitor class code Explained++

iii. Constructs the registration string for the RMI service using the specified registry hostname.

iv. Looks up the remote service in the RMI registry using the **Naming.lookup()** method and obtains a reference to it as a Remote object.

v. Casts the Remote object to the **TemperatureSensor** interface to access the **temperature-related** methods.

vi. Retrieves and displays the current temperature value by calling the **getTemperature()** method on the sensor object.

vii. Creates a new instance of the **TemperatureMonitor** class.

viii. Registers the monitor object as a listener with the sensor object by calling the **addTemperatureListener()** method.

RMI Callbacks+ Implementing the Listener Interface+

TemperatureMonitor class code Explained+++

7. The code includes exception handling to catch and display any errors that occur during the RMI lookup or execution.

In summary, this code represents a temperature monitor client that looks for a temperature sensor server using RMI. Once it finds the server, it retrieves the current temperature and registers itself as a listener to receive temperature change events. The client then prints the original temperature and displays any subsequent temperature changes received from the server.

Running the RMI Callback Example

When developing an RMI callback system, both parties (listener and event source) will act as servers and as clients. This means that both parties need to have a copy of the class definitions, or access to a Web server if dynamic class loading is needed.

Running the RMI Callback Example Steps

The following steps should be performed, in order, to run the example.

1. Compile the applications and generate stub/skeleton files for both TemperatureSensorServer and TemperatureSensorMonitor.(Only for those)

2. Run the rmiregistry application.

3. Run the TemperatureSensorServer.

4. Run the TemperatureSensorMonitor.

When first connected, the monitor will display the current temperature. It registers itself with the server and is notified of changes by a callback.

Terminating the client will cause it to be unregistered in the TemperatureSensorServer, when the sensor is unable to notify the client. In addition, the callback will be unregistered if the server is unable to notify the client due to a network error.

RMI Callbacks+ Implementing the Event Source Interface+

TemperatureMonitor Code Output

Temperature Client Side output

```
Output ×
CS (run) #2 ×   CS (run) #3 ×
run:
Looking for temperature sensor
Original temp: 99.0
Temperature change event: 98.5
Temperature change event: 98.0
Temperature change event: 98.5
Temperature change event: 99.0
Temperature change event: 98.5
Temperature change event: 99.0
Temperature change event: 98.5
Temperature change event: 98.0
Temperature change event: 98.5
Temperature change event: 98.0
Temperature change event: 97.5
Temperature change event: 97.0
```

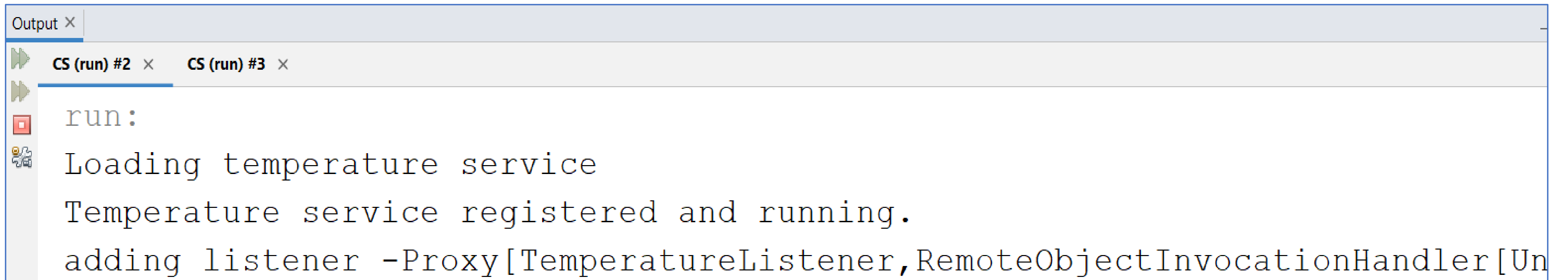
```
Temperature change event: 89.0
Temperature change event: 88.5
Temperature change event: 88.0
Temperature change event: 88.5
Temperature change event: 89.0
Temperature change event: 89.5
Temperature change event: 90.0
Temperature change event: 89.5
```

Note Infinite loop in action. Generating Temperature change simulation.

```
Temperature change event: 90.0
Temperature change event: 90.5
Temperature change event: 91.0
Temperature change event: 91.5
Temperature change event: 92.0
Temperature change event: 92.5
```

RMI Callbacks+ Implementing the Event Source Interface+
TemperatureSensorServer Code Output

Temperature Server-Side output during execution/ communication with the Client Side(TemperatureSensor)



Remote Object Activation

Remote Object Activation

Remote object activation is a technique that solves the problem of running a large number of idle RMI services.

It allows services to be registered with the rmiregistry, but not instantiated. Instead, they remain inactive, until called upon by a client, when they will awaken and perform their operation.

A special daemon process called the *remote method invocation activation system daemon* (rmid) listens for these requests and instantiates RMI services on demand.

Using some special trickery behind the scenes, a request is forwarded onto the activation daemon process, which creates the service. Requests are then forwarded onto the newly created service, transparently to the client, just as if it were a normal RMI service that had been running indefinitely. This means that services will lie dormant until invoked, and activated just in time (JIT) for use by a RMI client.

This technique controls wastage of memory and CPU circles.

How Remote Object Activation Work

Remote object activation works a little differently from normal RMI servers. To be locatable, a service must be available through an RMI registry, but to do that normally requires an object to be instantiated.

Since the whole point is to avoid instantiating RMI servers and instead to activate them as required, **a faulting remote reference** is registered in its place.

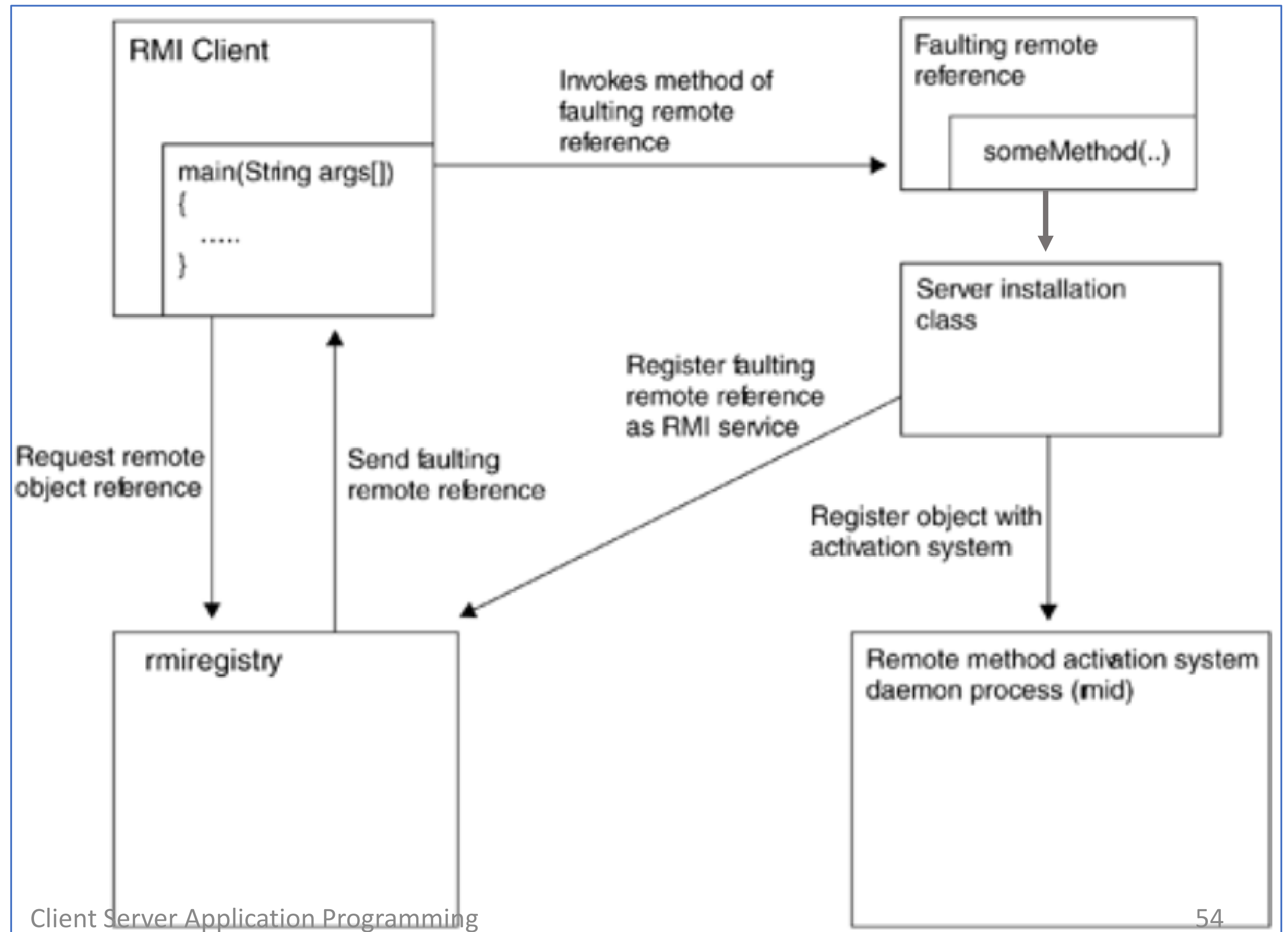
A faulting remote reference is a remote reference that acts as a proxy between the remote client and the as-yet-unactivated server.

Unlike a UnicastRemoteObject server, which **runs indefinitely**, a server installation program runs for a short duration. Its purpose is to notify the activation system of an activatable remote object, and to register the faulting remote reference with the rmiregistry (see the figure in the next slide).

Once the registration is complete, the server installation program can terminate; creating the remote object is now the responsibility of the faulting reference and the remote method activation system daemon.

How Remote Object Activation Work+

Server installation program registers activatable object with rmid, and
faulting reference with rmiregistry.



Michael R. and David R(2002)

Assignment

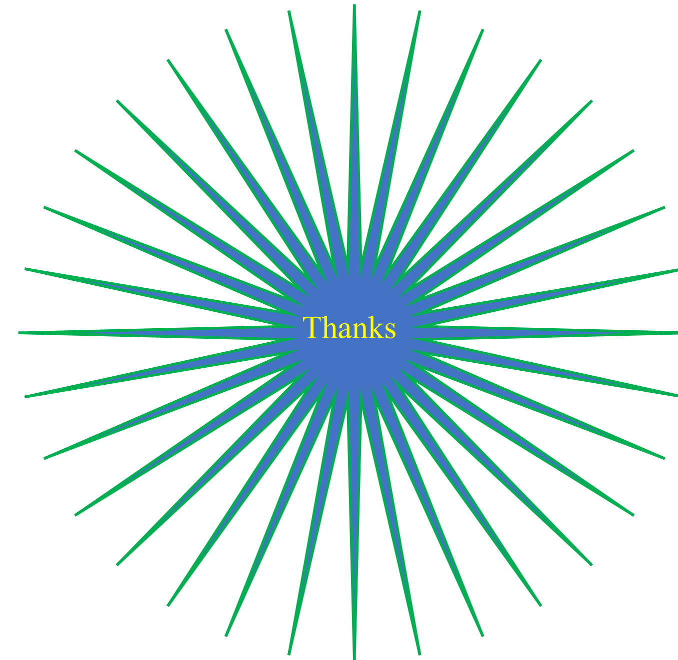
Find out how to create an Activatable Remote Object .

Summary

Summary

1. Remote Method Invocation Deployment Issues
2. Using RMI to Implement Callbacks
3. Remote Object Activation

Thank you for Listening



References

Java™ Network Programming and Distributed Computing, **(**David R.,Michael R. 2002), Publisher : Addison Wesley; ISBN: 0201710374