Unit 4A. Java Remote Method Invocation: Basics

Outcomes Summary. You will learn

- how to make multiple object systems (eg JVMs) work as a single system
- to program simple Java RMI applications
- how remote method invocation differs from ordinary method invocation
- how the RMI network model differs from the JMS network model and the implications these differences have for DS developers

Further Reading: CDK2005 5.1, 5.5.

So far our only example of middleware was the Erlang language and virtual machine. Erlang allows for succinct distributed programming, allowing the programmer to focus on the essence of the system's logic: distribution and communication of data and synchronisation of threads. We have seen only a fraction of its advanced features. Erlang has won respect in the industry and has been used to develop very large and critical DSs not only by Ericsson, the creators of the language, but also eg Amazon.com, Nortel and T-Mobile. Nevertheless, the language is far from widely adopted and nowhere near close in popularity to Java. As Java is the main PL taught in Aston, we will switch our attention to various middleware that are well integrated with Java. Among these we start with Java RMI — a Java-only middleware — because it is the simplest one to learn and use.

4A.1 Why distributed objects

Java RMI is a Java implementation of a more general idea called Remote Method Invocation (RMI). The main idea of RMI is to unite multiple object-oriented virtual machines so that they can be seen as one distributed object-oriented virtual machine. An executing thread within one virtual machine should be able to execute methods for objects stored on the same computer as well as objects stored on a remote computer.

One could go as far as achieving complete object access and location transparency for the programmer but this is not always desirable. Some differences between local and remote method invocations should not be hidden from the programmer because they can critically affect the performance of the system. Java RMI makes the programmer aware of which objects are local and which ones are remote but allows them to call their methods using the same syntax irrespective of their location.

The main benefit of a distributed objects middleware is that it makes it possible to take the largely successful, tried-and-tested, well-understood, well-supported and widely-adopted object-oriented methodology and apply it to distributed systems. This does not mean that DS development suddenly becomes a matter of deciding which objects from a non-distributed object-oriented system will live on which node. To make the

DS perform acceptably, the design needs to be careful about the amount of RMIs and the size of the parameters and results that are being transferred. Also, the possibility of network and node failures make it more difficult to make the systems reliable.

4A.2 Accessing remote objects

Let us now look at how a Java thread can access an object (let us call it R for example) residing in a remote JVM using Java RMI. This typically involves the following steps:

- The remote object R is registered by its JVM under some global name, say *"Chatter"*.
- The thread connects to a registry service that knows about this object.
- The thread looks up the name "Chatter" and gets a reference to a local object, called *proxy*, that represents the remote object R.
- The thread calls a method of the proxy object (thinking of R) and the call gets transferred to R and the thread waits until the method is finished and gets the result of the call, if any.

Additionally, both JVMs have to have an instance of `RMISecurityManager` setup to supervise the communication as shown in Figure 4A.1.

4A.2.1 Locating a remote object

Each remote object has a globally unique remote reference (similar to Erlang process ID) that can be sent to any JVM and thus enable the JVM to perform RMI on the object. Such IDs are transparent to programmers in the same way that references to local objects are transparent to the programmers in ordinary Java. Once a JVM has a reference to one remote object, it can fetch from that remote object references to other remote objects as return values of RMIs (see Figure 4A.2). Nevertheless, there is a need to establish an initial contact with at least one remote object using a registry of global names (called rmiregistry) as shown in Figure 4A.1.

In a client-server scenario, there is usually a registry on the server computer and clients use it to get the reference of the main server object. During the interaction, the client may be passed references to other server objects by the main server object.

4A.2.2 Using a remote interface

Java RMI places certain limitations on the use of remote objects. For example, one cannot access fields (ie instance variables) of remote objects even if they are public. To make it clear and explicit what methods are remotely available, remote users cannot treat the object as an instance of its class but as an instance of a remote interface. Eg in Figure 4A.1 computer A creates an instance of `ChatServer` but computer B sees it as an instance of `ChatServerInterface`. Thus some methods of the object may

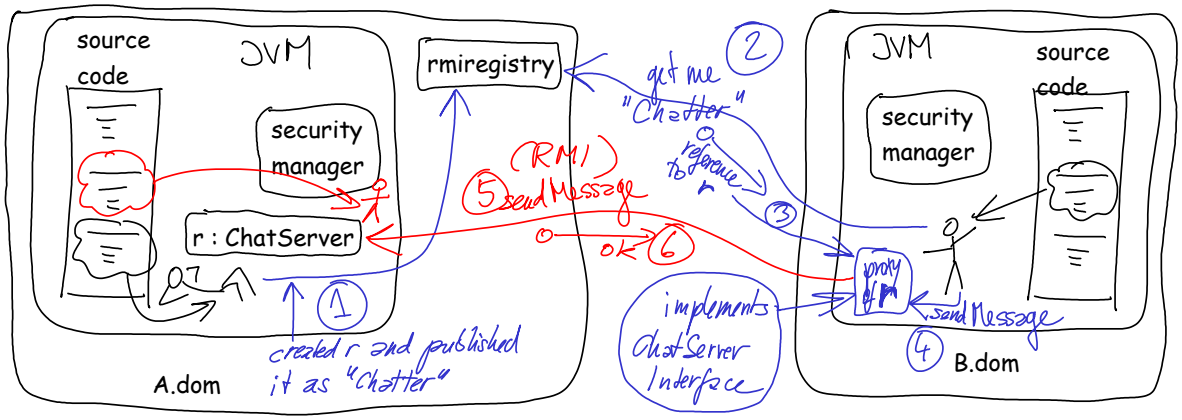

Figure 4A.1: Steps of Java RMI

Extract of code at node A holding remote object of class `ChatServer`:

```
1  import java.rmi.Naming;
2  import java.rmi.RMISecurityManager;
3
4      ...
5
6      // start security manager to allow and monitor connections:
7      System.setSecurityManager(new RMISecurityManager());
8
9      // create chat server object:
10     ChatServer chatServer = new ChatServer();
11
12     // publish the server object using global name:
13     Naming.rebind("Chatter", chatServer);
```

Extract of code at node B using the remote object created above:

```
1  import java.rmi.Naming;
2  import java.rmi.RMISecurityManager;
3
4      ...
5
6      // start security manager to allow and monitor connections:
7      System.setSecurityManager(new RMISecurityManager());
8
9      // connect to node A's RMI registry
10     // + look-up remote object "Chatter"
11     // + create local proxy object for it:
12     ChatServerInterface server =
13       (ChatServerInterface) Naming.lookup("rmi://A.domain/Chatter");
14
15     // invoke a remote method:
16     server.sendMessage("hello there...");
```

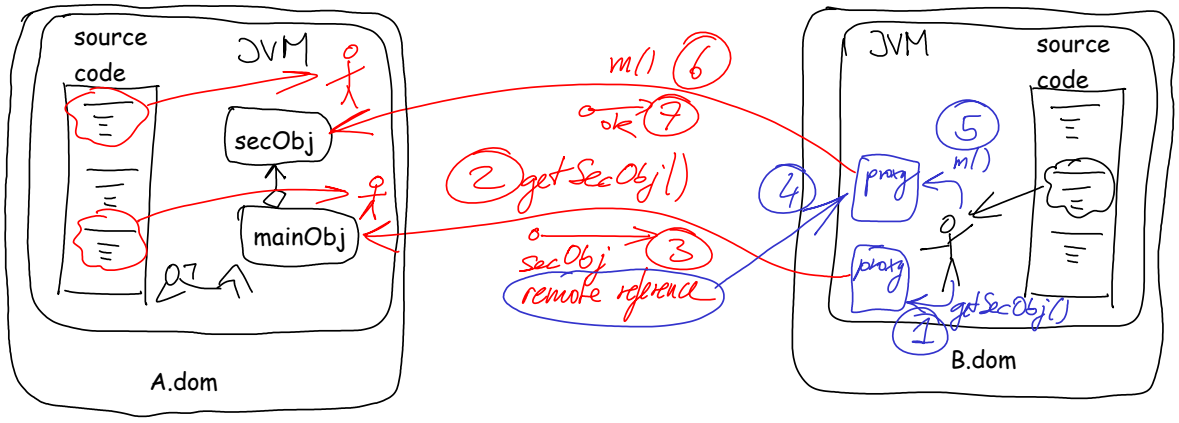

Figure 4A.2: Fetching a remote reference

```
import java.rmi.Remote;
import java.rmi.RemoteException;

import client.ChatClientInterface;

public interface ChatServerInterface extends Remote
{
    void subscribe(ChatClientInterface client) throws RemoteException;

    void unsubscribe(ChatClientInterface client) throws RemoteException;

    void sendMessage(Object msg) throws RemoteException;
}
```

Figure 4A.3: A remote interface for a chat server.

be hidden because they do not appear in the interface. This is convenient when we want to give objects some methods that break the RMI rules — such methods will be available locally but not remotely.

A remote interface must extend the predefined empty interface `Remote` and each method must declare that it may throw a `RemoteException`. Figure 4A.3 shows a complete example remote interface for a chat server analogous to the one studied in earlier units.

Apart from providing a useful abstract view of the remote objects, the interface is used by Java RMI to construct a local object that acts like a *proxy* for the remote object, as shown in Figure 4A.1, step 3. The proxy implements the remote interface without understanding any of its methods, simply forwarding all method calls it receives to the remote object over the network and then delivering the return values back from the remote object. The class that defines these proxy objects is written automatically by the Java RMI mechanism.

A remote interface is usually *shared* between all users of the remote object. The interface shown in Figure 4A.3 is used by both the server and the client code. The server class must declare that it implements this interface and the client must use this interface as the type for all references it has to the server (eg see line 12 in the client code in Figure 4A.1).

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ChatClientInterface extends Remote
{
    void newMessage(Object msg) throws RemoteException;

    String getName() throws RemoteException;
}
```

Figure 4A.4: A remote interface for a chat client.

Notice that the parameter of `subscribe` (as well as `unsubscribe`) has the interface type `ChatClientInterface` shown in full in Figure 4A.4. This hints at the fact that the client objects are remotely accessed by the server whenever a new message has arrived. Thus both server and client code share both Java interfaces, one for the server object and one for the client objects. The server implements the server interface and uses the client interface and, vice versa: the client implements the client interface and uses the server interface.

Messages are instances of a class `Message`. If either of the interfaces would mention this class, the class source would need to be present on both the server and client so that the compiler would accept the interfaces. To avoid such duplication of code, the messages are treated as instances of `Object` in the interfaces. The class `Message` is not required by the server because it does not need to know anything about messages except that they are objects — it passes them as parameters and prints them to its debugging log using their `toString()` method. It is the client who creates messages from their components and thus the definition of the `Message` class belongs to the client side.

4A.2.3 Method parameters

Java RMI puts restrictions on the types of parameters and return values used in methods that are invoked remotely. There is no problem with value types such as **`int`** and **`char`**. Nevertheless, objects are passed by memory reference in local calls, which is not possible to do remotely. There are 2 options here:

- The passed objects are remote objects and they can be passed using their remote references.
- Java knows how to serialise the objects and passes them by value as if they were **`int`**, creating a new copy instance of the object on the remote JVM.

We have already illustrated the first option as a return value in Figure 4A.2. (This is also what happens when a chat client enquires the rmiregistry about the server object.) Analogously, one can send a remote reference to a local remote-enabled object C as a parameter of a remote method invocation and then the temporary thread dealing with this RMI at the remote site will be able to make remote calls back to object C. Alternatively, the call can store the reference somewhere in the remote JVM to be used

```
import java.io.Serializable;

public class Message implements Serializable
{
    private static final long serialVersionUID = 220112709756253576L;

    private String sender;
    private String content;

    public Message(String sender, String content)
    {
        this.sender = sender;
        this.content = content;
    }

    @Override
    public String toString()
    {
        return "From " + sender + ": " + content;
    }
}
```

Figure 4A.5: A chat message class.

by other local or remote calls in the future. (An example of this is the `subscribe` method taking a parameter referring to a chat client.)

The second option is used by the example chat server/client to transfer chat messages to each other. We can conclude that `Message` has to be a serialisable class (see Figure 4A.5). Recall that a class is made serialisable by declaring that it implements the `Serialisable` interface and giving it an integer constant called `serialVersionUID`. This UID (Unique ID) should be changed to a different random integer each time the class is changed in a way that makes it impossible to treat instances of the old version as instances of the new version of the class. For example, adding or removing instance variables require a change in the `serialVersionUID`. (IDEs such as Eclipse make it easier to manage these numbers.)

4A.3 Defining remote objects

The easiest way to define a class of remote objects is to extend the standard class `UnicastRemoteObject` as shown in Figure 4A.6. Each constructor of the class must call one of the constructors of the parent class `UnicastRemoteObject` using the **`super`** keyword (see lines 12–15). This is because the constructor of `UnicastRemoteObject` does the hard work of making it available over the network.

As is evident from Figure 4A.2, with each remote access the JVM that hosts the object is likely to start a new thread to act on behalf of the initiating thread at the other end of the connection. Thus we cannot rule out that the remotely available methods of

```
1  public class ChatServer
2      extends UnicastRemoteObject
3      implements ChatServerInterface
4  {
5
6      private static final long serialVersionUID =
7          -1140073548213973798L;
8
9      private Set<ChatClientInterface> listeners =
10         new HashSet<ChatClientInterface>();
11
12     public ChatServer() throws RemoteException
13     {
14         super();
15     }
16
17     @Override
18     public synchronized void sendMessage(Object msg)
19         throws RemoteException
20     {
21         System.out.printf("New message: %s\n", msg.toString());
22
23         // for each listening client:
24         for ( ChatClientInterface client : listeners )
25         {
26             // notify this client of the message:
27             client.newMessage(msg);
28         }
29     }
30
31     @Override
32     public synchronized void subscribe(ChatClientInterface client)
33         throws RemoteException
34     {
35         listeners.add(client);
36         System.out.printf("subscribed client: %s\n", client.getName());
37     }
38
39     @Override
40     public synchronized void unsubscribe(ChatClientInterface client)
41         throws RemoteException
42     {
43         listeners.remove(client);
44         System.out.printf("unsubscribed client: %s\n", client.getName());
45     }
46
47     public static void main(String[] args)
48     {
49         (code left out — mostly shown in Figure 4A.1)
50         // start security manager to allow and monitor connections
51         // create chat server object:
52         // publish the server object using a global name:
53     }
54 }
```

Figure 4A.6: A class defining a chat server remote object.

the object will be invoked concurrently by multiple threads as responses to concurrent remote invocations. To be safe, one could make all remotely available methods `synchronized` as shown in the chat server code. For better efficiency, one would enclose only the portions of the methods that access the instance variable `listeners` so that the hash set of listeners would not be corrupted by concurrent access.

4A.4 Listings of simple chat system

4A.4.1 Server interface

```
1  package server;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5
6  import client.ChatClientInterface;
7
8  public interface ChatServerInterface extends Remote
9  {
10     public static final String DEFAULT_SERVER_NAME = "CS3250ChatServer";
11
12     void subscribe(ChatClientInterface client) throws RemoteException;
13
14     void unsubscribe(ChatClientInterface client) throws RemoteException;
15
16     void sendMessage(Object msg) throws RemoteException;
17 }
```

4A.4.2 Server class

```
1  package server;
2
3  import java.net.MalformedURLException;
4  import java.rmi.Naming;
5  import java.rmi.RMISecurityManager;
6  import java.rmi.RemoteException;
7  import java.rmi.registry.LocateRegistry;
8  import java.rmi.server.UnicastRemoteObject;
9  import java.util.HashSet;
10 import java.util.Set;
11
12 import client.ChatClientInterface;
13
14 public class ChatServer extends UnicastRemoteObject implements
15         ChatServerInterface
16 {
17
18     private static final long serialVersionUID = -1140073548213973798L;
19
20     private Set<ChatClientInterface> listeners =
21         new HashSet<ChatClientInterface>();
22
23     public ChatServer(int port) throws RemoteException
24     {
25         super(port);
26     }
```

```
28      public void subscribe(ChatClientInterface client) throws RemoteException
29      {
30          synchronized (this)
31          {
32              listeners.add(client);
33          }
34          System.out.printf("subscribed client: %s\n", client.getName());
35      }
36
37      public void unsubscribe(ChatClientInterface client) throws RemoteException
38      {
39          synchronized (this)
40          {
41              listeners.remove(client);
42          }
43          System.out.printf("unsubscribed client: %s\n", client.getName());
44      }
45
46      public void sendMessage(Object msg) throws RemoteException
47      {
48          System.out.printf("New message: %s\n", msg.toString());
49
50          synchronized (this)
51          {
52              // for each listening client:
53              for (ChatClientInterface client : listeners)
54              {
55                  try
56                  {
57                      // notify this client of the message:
58                      client.newMessage(msg);
59                  }
60                  catch (RemoteException e)
61                  {
62                      ;// ignore client exceptions - they may have died or gone
63                       // crazy...
64                  }
65              }
66          }
67      }
68
69      public static void main(String[] args)
70        throws RemoteException, MalformedURLException
71      {
72          // start security manager to allow and monitor connections:
73          System.setSecurityManager(new RMISecurityManager());
74
75          // create chat server object:
76          ChatServer chatServer = new ChatServer(51515);
77
78          LocateRegistry.createRegistry(1099);
79          System.out.println("Java RMI registry created");
80
81          // publish the server object at the registry using a global name:
82          Naming.rebind(DEFAULT_SERVER_NAME, chatServer);
83          System.out.printf("chat server %s now available\n", DEFAULT_SERVER_NAME);
84      }
85  }
```

4A.4.3 Client interface

```
1  package client;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5
6  public interface ChatClientInterface extends Remote
7  {
8      void newMessage(Object msg) throws RemoteException;
9
10     String getName() throws RemoteException;
11 }
```

4A.4.4 Client class

```
19 public class ChatClient extends UnicastRemoteObject implements
20         ChatClientInterface
21 {
22     private static final long serialVersionUID = -6438525179496519739L;
23     private ChatServerInterface server;
24     private String clientname;
25
26     public ChatClient(String name, int port, String serverHost,
27             String serverName) throws RemoteException, MalformedURLException,
28             NotBoundException, UnknownHostException
29     {
30         super(port);
31
32         clientname = name + "@" + InetAddress.getLocalHost().getHostName();
33
34         // locate the server:
35         server = (ChatServerInterface) Naming.lookup("rmi://" + serverHost
36                 + "/" + serverName);
37
38         // subscribe to the server:
39         server.subscribe(this);
40     }
41
42     private void sendMessage(String msgContent) throws RemoteException,
43             ServerNotActiveException
44     {
45         server.sendMessage(new Message(clientname, msgContent));
46     }
47
48     private void unsubscribe() throws RemoteException
49     {
50         server.unsubscribe(this);
51     }
52
53     public synchronized void newMessage(Object msg) throws RemoteException
54     {
55         System.out.printf("New message:\n%s\n", msg.toString());
56     }
```

```
58      public String getName() throws RemoteException
59      {
60          return clientname;
61      }
62
63      public static void main(String[] args)
64        throws NotBoundException, IOException, ServerNotActiveException
65      {
66          String serverName = ChatServerInterface.DEFAULT_SERVER_NAME;
67
68          // determine client name from command-line argument:
69          String name = "client"; // default name
70          if (args.length > 0)
71          {
72              name = args[0];
73          }
74          // determine client name from command-line argument:
75          String serverHost = "localhost"; // by default look for server locally
76          if (args.length > 1)
77          {
78              serverHost = args[1];
79          }
80          // determine client name from command-line argument:
81          int port = 51000; // by default look for server locally
82          if (args.length > 2)
83          {
84              port = Integer.parseInt(args[2]);
85          }
86          // start security manager to allow and monitor connections:
87          System.setSecurityManager(new RMISecurityManager());
88
89          // create a client object that will act as a listener:
90          ChatClient chatClient = new ChatClient(name, port, serverHost, serverName);
91          // keep asking user for messages:
92          BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
93          String msgContent;
94          while (true)
95          {
96              // get a message from user:
97              System.out.printf("msg: ");
98              msgContent = stdin.readLine();
99
100             // check whether the user wants to quit:
101             if (msgContent != null && (!msgContent.equals("q")))
102             {
103                 // no, user gave a message, send it out:
104                 chatClient.sendMessage(msgContent);
105             }
106             else
107             {
108                 // unsubscribe and quit:
109                 chatClient.unsubscribe();
110                 System.exit(0);
111             }
112         }
113     }
114 }
```