CHAPTER 5

DISTRIBUTED COMPUTING

Some of the most powerful and important concepts of modern computing come from the simple idea of moving computation (or storage) to a remote machine. This simple yet powerful idea of *distributed computing* gives rise to a myriad of new computing paradigms. In this chapter, we take a look at two of these. First, we will talk about *Remote Method Invocation* and then look at an *Apache* project that allows you to write powerful server-side programs with little boilerplate code.

distributed computing

Remote Method Invocation

Apache

5.1 Remote Method Invocation

Remote Method Invocation – or *RMI* [7] – enables Java programmers to easily execute code on a remote machine. The basic idea is very simple from a programmer's perspective. We execute a function on a local machine, the actual function call gets transported to a remote machine, executed there and the results returned to our local function as if the function was executed locally. This is depicted in Figure **??**.

RMI

This adds immensely to the ease of deploying distributed applications. If you have a processing-intensive task that needs to be run many times, you might want to move it to a more powerful machine. Another use case might be when you have a data store in a central location and want to retrieve data easily without having to worry about writing network programming code.

Below is an example of how to use RMI. First, we will create an interface that acts as a connection that both the server and client will share. This will allow us to write client code that can be compiled without any errors. The following interface goes in a new project `w05-rmi-server`.

```java
package org.csrdu.java.ex;

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

```

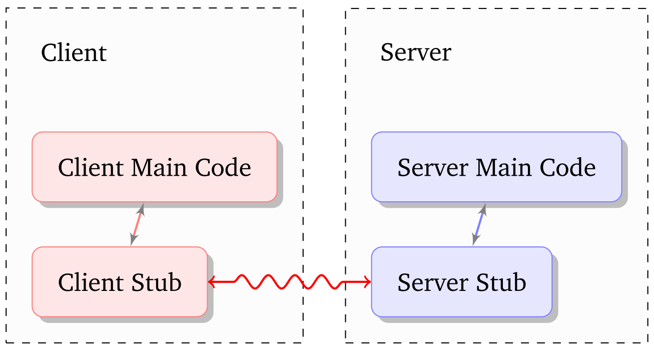

Figure 5.1: Java RMI Architecture

```java
public interface Hello extends Remote {
  String sayHello() throws RemoteException;
  String concatStrings(String s1, String s2) throws RemoteException;
}
```

5.1.1 Writing the Server

Now, we will create a server that will serve requests coming in for this interface.

```java
1  package org.csrdu.java.ex;
2  
3  import java.rmi.RemoteException;
4  import java.rmi.registry.Registry;
5  import java.rmi.registry.LocateRegistry;
6  import java.rmi.server.UnicastRemoteObject;
7  
8  public class Server implements Hello {
9    public String sayHello() {
10     return "Hello, world!";
11   }
12 
13   public static void main(String args[]) {
14     try {
15       Server obj = new Server();
16       Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
17 
18       // Bind the remote object's stub in the registry
19       Registry registry = LocateRegistry.getRegistry();
20       registry.bind("Hello", stub);
21       System.err.println("Server ready");
22     } catch (Exception e) {
23       System.err.println("Server exception: " + e.toString());
24       e.printStackTrace();
25     }
26   }
27 
28   @Override
29   public String concatStrings(String s1, String s2)
30     throws RemoteException {
31     return s1.concat(s2);
32   }
33 }
```

There are a couple of things to note here:

stub

1. Line 16 exports a *stub* method based on the `Server` class. RMI works around the concept of stubs (see the architecture figure above). This stub had to be created manually prior to Java 5 but this line automates this process.

bind
registry

2. We *bind* our `Hello` server to the local *registry*. This registry acts as a directory listing for the services offered by the current machine.

3. Everything else is taken care of automatically by the stub methods.

5.1.2 Writing the Client

To use the server, we need to write a client. For now, we will be running the client on the same machine but we will create a separate project for the client so that it can be moved independently of the server. First, create a new project called w05-rmi-client. Add the same code for the Hello interface as the server. Then, add a client class with the following code:

```java
package org.csrdu.java.ex;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

  public static void main(String[] args) {

  String host = (args.length < 1) ? null : args[0];
    try {
      Registry registry = LocateRegistry.getRegistry(host);
      Hello stub = (Hello) registry.lookup("Hello");
      String response = stub.sayHello();
      System.out.println("response sayHello: " + response);
      response = stub.concatStrings("First", "Second");
      System.out.println("response concatStrings: " + response);
    } catch (Exception e) {
      System.err.println("Client exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

As with the server, there are a couple of things to note here:

1. We can provide the host of the registry to the client. If our server were on a separate machine, we could provide that as the host and our client code need not be changed at all.
2. We use the registry to get a stub for our client. We can then use this stub for calling any method on the remote machine. It is the responsibility of the stubs and the registry to communicate our calls to the server virtual machine and object.

5.1.3 Running the Example

Running the example requires only a little bit of configuration. We need to do a couple of things:

1. **Start the registry:** We need to have the registry service running on our machine so that the server can *register* with it and the client can *look up* services through it. To do this, we can issue the following command on our terminal:

   register

   look up

   ```
   rmiregistry
   ```

2. To start the server, we need to set the *code base location*. Do this by setting a VM argument in your server's run configuration:

   code base location

   ```
   -Djava.rmi.server.codebase=file:///[workspace location]/w05-rmi-server/bin/
   ```

   This is the location of your compiled files. When a stub is created, it will be placed automatically in the correct location and this parameter will allow the VM to read it correctly.

   After setting this option, you can start the server.

3. Starting the client is very easy. Just run the client and everything will be taken care of by the code. You should see the following output if everything goes as planned:

```
response sayHello: Hello, World!
response concatStrings: FirstSecond
```

You can now extend the RMI example to pass all sorts of parameters and get all manners of responses from the server. The only issue is that the client needs to be a Java program as well. We'll take care of this limitation in the following section.

5.2 Apache Mina

Apache MINA

According to the official homepage, "*Apache MINA* is a network application framework which helps users develop high performance and high scalability network applications easily." [2]. In essence, it allows you to write distributed applications that can communicate over the usual *TCP/IP* or *UDP/IP* protocols. You can, for example, write an FTP server using this infrastructure.

TCP/IP

UDP/IP

We demonstrate the usefulness of MINA through an example. Begin by downloading the binaries for MINA from `http://mina.apache.org/downloads.html`. From the downloaded archive, you need to get two files `dist/mina-core-*.jar` and `lib/slf4j-api-*.jar`. Create a new project and put these in the `referenced` folder of the project. Add both jars to the build path. Now, create the following class that acts as our server:

```java
package org.csrdu.java.ex;

import java.io.IOException;

import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.mina.core.service.IoAcceptor;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaTimeServer {
  private static final int PORT = 9123;

  public static void main(String[] args) throws IOException {
    IoAcceptor acceptor = new NioSocketAcceptor();

    acceptor.getFilterChain().addLast("logger", new LoggingFilter());
    acceptor.getFilterChain().addLast("codec",
      new ProtocolCodecFilter(new TextLineCodecFactory(
        Charset.forName("UTF-8")
      ))
    );

    acceptor.setHandler(new TimeServerHandler());
    acceptor.getSessionConfig().setReadBufferSize(2048);
    acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);
    acceptor.bind(new InetSocketAddress(PORT));
  }
}
```

The main method begins by creating an `IoAcceptor`. This acceptor receives data from the network interface (on the port specified on Line 31). We set the logger for the acceptor and add a *codec*. This codec will decode all incoming messages. *codec* Since we will be using a command-line interface, this is a `TextLineCodecFactory` object which works on *UTF-8* character set. We then set the handler which will be *UTF-8* passed all messages that are received – this is a custom class that we define below. Finally, we set the buffer size and idle timeout and start the server.

The `TimeServerHandle` class used above is defined as:

```java
package org.csrdu.java.ex;

import java.util.Date;

import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IoSession;

public class TimeServerHandler extends IoHandlerAdapter {
  @Override
  public void exceptionCaught(IoSession session, Throwable cause)
    throws Exception {
    cause.printStackTrace();
  }

  @Override
  public void messageReceived(IoSession session, Object message)
  throws Exception {
    String str = message.toString();
    if (str.trim().equalsIgnoreCase("quit")) {
    session.close(true);
      return;
    }

    Date date = new Date();
    session.write(date.toString());
    System.out.println("Message written...");
  }

  @Override
  public void sessionIdle(IoSession session, IdleStatus status)
  throws Exception {
    System.out.println("IDLE " + session.getIdleCount(status));
  }
}
```

The code is very straight-forward. Whenever a message is received, it checks whether the command is `quit`. If so, it closes the session. Otherwise, it returns the current date. To run this example, we can initiate a session through telnet on the given port and issue some commands. A sample session follows:

```
nam@temp:~$ telnet localhost 9123
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
Tue Feb 14 10:29:58 PKT 2012
another command
Tue Feb 14 10:30:02 PKT 2012
quit
Connection closed by foreign host.
```