Chapter 6

Distributed Programming

6.1 Introduction

In this chapter, we will learn primitives provided in the Java programming language for building distributed applications. We will see primarily two programming styles: sockets and remote method invocations. Sockets provide a lower-level interface for building distributed programs but are more efficient and flexible. Remote method invocations (RMI) are easier to use.

In this chapter we first describe the class `InetAddress`, which is useful for network programming no matter which style of primitives are used. Then we discuss primitives for programming using sockets. These sockets may use either the Universal Datagram Protocol (UDP), or the Transmission Control Protocol (TCP). We give an example of an echo server using sockets based on the UDP protocol and a simple name server using sockets based on the TCP protocol. Finally, we discuss programming using remote method invocations.

6.2 InetAddress Class

For any kind of distributed application, we need the notion of an Internet address. Any computer connected to the Internet (called a *host*) can be uniquely identified by an address called an *IP address*. Since addresses are difficult to remember, each host also has a hostname. It is the task of a domain name system (DNS) server to provide the mapping from a hostname to its address. Java provides a class `Java.net.Inetaddress`, which can be used for this translation. The relevant methods for the class InetAddress are given below:

```
public byte[] getAddress()
  Returns the raw IP address of this InetAddress object.
public static InetAddress getByName(String)
  Determines the IP address of a host, given the host's name.
public String getHostAddress()
  Returns the IP address string "%d.%d.%d.%d"
public String getHostName()
  Returns the fully qualified host name for this address.
public static InetAddress getLocalHost()
  Returns the local host.
```

6.3 Sockets based on UDP

Sockets are useful in writing programs based on communication using messages. A Socket is an object that can be used to send and receive messages. There are primarily two protocols used for sending and receiving messages: Universal Datagram Protocol (UDP) and Transmission Control Protocol (TCP). The UDP provides a low-level connectionless protocol. This means that packets sent using UDP are not guaranteed to be received in the order sent. In fact, the UDP protocol does not even guarantee reliability, that is, packets may get lost. The protocol does not use any handshaking mechanisms (such as acknowledgments) to detect loss of packets. Why is UDP useful, then? Because, even though UDP may lose packets, in practice, this is rarely the case. Since there are no overheads associated with error checking, UDP is an extremely efficient protocol.

The TCP protocol is a reliable connection-oriented protocol. It also guarantees ordered delivery of packets. Needless to say, TCP is not as efficient as UDP.

6.3.1 Datagram Sockets

The first class that we use is `DatagramSocket` which is based on the UDP protocol. This class represents a socket for sending and receiving datagram packets. A *datagram socket* is the sending or receiving point for a connectionless packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from a machine to another may be routed differently, and may arrive in any order. This class provides a very low level interface for sending and receiving messages. There are few guarantees associated with datagram sockets. An advantage of datagram sockets is that it allows fast data transmission.

The details for the methods in this class are given below. To construct a DatagramSocket, we can use one of the following constructors:

```
public DatagramSocket()
public DatagramSocket(int port)
public DatagramSocket(int port, InetAddress laddr)
```

The first constructor constructs a datagram socket and binds it to any available port on the local host machine. Optionally, a port may be specified as in the second constructor. The last constructor creates a datagram socket, bound to the specified local address. These constructors throw `SocketException` if the socket could not be opened, or if the socket could not bind the specified local port.

The other important methods of this class are as follows:

1. `public void close()`: This method closes a datagram socket.
2. `public int getLocalPort()`: To get the information about the socket, one can use this method, which returns the port number on the local host to which this socket is bound.
3. `public InetAddress getLocalAddress()`: This method gets the local address to which the socket is bound.
4. `public void receive(DatagramPacket p)`: This method `receive` receives a datagram packet from this socket. When this method returns, the DatagramPacket's buffer is filled with the data received. The datagram packet also contains the sender's IP address and the port number on the sender's machine. Note that this method blocks until a datagram is received. The length field of the datagram packet object contains the length of the received message. If the message is longer than the buffer length, the message is truncated. It throws IOException if an I/O error occurs. The blocking can be avoided by setting the timeout.
5. `public void send(DatagramPacket p)`: This method sends a datagram packet from this socket. The DatagramPacket includes information indicating the data to be sent, its length, the IP address of the remote host, and the port number on the remote host.

6.3.2 DatagramPacket Class

The `DatagramSocket` class required data to be sent as datagram packets. The class `java.net.DatagramPacket` is used for that. Its definition is given below.

```
public  final  class  java.net.DatagramPacket
    extends  java.lang.Object {
    public DatagramPacket(byte  ibuf[], int  ilength);
```

```
    public DatagramPacket(byte  ibuf[], int  ilength,
                          InetAddress  iaddr, int iport);
    public InetAddress getAddress();
    public byte[] getData();
    public int getLength();
    public int getPort();
    public void setAddress(InetAddress)
    public void setData(byte[])
    public void setLength(int)
    public void setPort(int)
}
```

The first constructor

```
public DatagramPacket(byte ibuf[], int ilength)
```

constructs a DatagramPacket for receiving packets of length `ilength`. The parameter `ibuf` is the buffer for holding the incoming datagram, and `ilength` is the number of bytes to read.

The constructor for creating a packet to be sent is

```
public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int
iport)
```

It constructs a DatagramPacket for sending packets of length `ilength` to the specified port number on the specified host. The parameters `iaddr` and `iport` are used for the destination address and the destination port number, respectively. The method `getAddress` returns the IP address of the machine to which this datagram is being sent, or from which the datagram was received. The method `getData` returns the data received, or the data to be sent. The method `getLength` returns the length of the data to be sent, or the length of the data received. Similarly, the method `getPort` returns the port number on the remote host to which this datagram is being sent, or from which the datagram was received. The `set` methods are used to set the IP address, port number, and other elements appropriately.

6.3.3 Example Using Datagrams

We give a simple example of a program that uses datagrams. This example consists of two processes—a server and a client. The client reads input from the user and sends it to the server. The server receives the datagram packet and then echoes back the same data. The program for the server is given in Figure 6.1.

```
import java.net.*;
import java.io.*;
public class DatagramServer {
    public static void main(String[] args) {
        DatagramPacket datapacket, returnpacket;
        int port = 2018;
        int len = 1024;
        try {
            DatagramSocket datasocket = new DatagramSocket(port);
            byte[] buf = new byte[len];

            while (true) {
                try {
                    datapacket = new DatagramPacket(buf, buf.length);
                    datasocket.receive(datapacket);
                    returnpacket = new DatagramPacket(
                    datapacket.getData(),
                    datapacket.getLength(),
                    datapacket.getAddress(),
                    datapacket.getPort());
                    datasocket.send(returnpacket);
                } catch (IOException e) {
                    System.err.println(e);
                }
            }
        } catch (SocketException se) {
            System.err.println(se);
        }
    }
}
```

Figure 6.1: A datagram server

The client process reads a line of input from `System.in`. It then creates a datagram packet and sends it to the server. On receiving a response from the server it displays the message received. The program for the client is given in Figure 6.2.

6.4 Sockets Based on TCP

The second style of interprocess communication is based on the notion of streams. In this style, a connection is set up between the sender and the receiver. This style allows better error recovery and guarantees on the delivery of packets. Thus, in a stream the packets are received in the order they are sent.

The `socket` class in Java extends the `Object` class. We will give only a subset of constructors and methods available for `Socket`.

The constructor `public Socket(String host, int port)` creates a stream socket and connects it to the specified port number on the named host. It throws `UnknownHostException`, and `IOException`.

Here we have used the name of the host. Alternatively, IP address can be used in the form of the class InetAddress as below:

```
public Socket(InetAddress address, int port)
```

The methods for the socket are

- `public InetAddress getInetAddress()`, which returns the remote IP address to which this socket is connected.
- `public InetAddress getLocalAddress()`, which returns the local address to which the socket is bound.
- `public int getPort()`, which returns the remote port to which this socket is connected.
- `public InputStream getInputStream()`, which returns an input stream for reading bytes from this socket.
- `public OutputStream getOutputStream()`, which returns an output stream for writing bytes to this socket.
- `public synchronized void close()`, which closes this socket.

Note that many of these methods throw `IOException` if an I/O error occurs when applying the method to the socket.

```
import java.net.*;
import java.io.*;
public class DatagramClient {
    public static void main(String[] args) {
        String hostname;
        int port = 2018;
        int len = 1024;
        DatagramPacket sPacket, rPacket;
        if (args.length > 0)
            hostname = args[0];
        else
            hostname = "localhost";
        try {
            InetAddress ia = InetAddress.getByName(hostname);
            DatagramSocket datasocket = new DatagramSocket();
            BufferedReader stdinp = new BufferedReader(
            new InputStreamReader(System.in));
            while (true) {
                try {
                    String echoline = stdinp.readLine();
                    if (echoline.equals("done")) break;
                    byte[] buffer = new byte[echoline.length()];
                    buffer = echoline.getBytes();
                    sPacket = new DatagramPacket(buffer,
                                        buffer.length, ia, port);
                    datasocket.send(sPacket);
                    byte[] rbuffer = new byte[len];
                    rPacket = new DatagramPacket(rbuffer, rbuffer.length);
                    datasocket.receive(rPacket);
                    String retstring = new String(rPacket.getData(), 0,
                    rPacket.getLength());
                    System.out.println(retstring);
                } catch (IOException e) {
                    System.err.println(e);
                }
            } // while
        } catch (UnknownHostException e) {
            System.err.println(e);
        } catch (SocketException se) {
            System.err.println(se);
        }
    } // end main
}
```

Figure 6.2: A datagram client

6.4.1 Server Sockets

On the server side the class that is used is called `ServerSocket`. A way to create a server socket is `public ServerSocket(int port)`
This call creates a server socket on a specified port. Various methods on a server socket are as follows:

- `public InetAddress getInetAddress()`, which returns the address to which this socket is connected, or null if the socket is not yet connected.
- `public int getLocalPort()`, which returns the port on which this socket is listening.
- `public Socket accept()`, which listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.
- `public void close()`, which closes this socket.

6.4.2 Example 1: A Name Server

We now give a simple name server implemented using server sockets. The name server maintains a table of `(name, hostName, portNumber)` to give a mapping from a process `name` to the host and the port number. For simplicity, we assume that the maximum size of the table is 100 and that there are only two operations on the table: `insert` and `search`. This table is kept by the object `NameTable` shown in Figure 6.3.

Now let us look at the name server. The name server creates a server socket with the specified port. It then listens to any incoming connections by the method `accept`. The `accept` method returns the socket whenever a connection is made. It then handles the request that arrives on that socket by the method `handleclient`. We call `getInputStream` and `getOutputStream` to get input and output streams associated with the socket. Now we can simply use all methods associated for reading and writing input streams to read and write data from the socket.

In our implementation of the name server shown in Figure 6.4, at most one client is handled at a time. Once a request is handled, the main loop of the name server accepts another connection. For many applications this may be unacceptable if the procedure to handle a request takes a long time. For these applications, it is quite common for the server to be multithreaded. The server accepts a connection and then spawns a thread to handle the request. However, it must be observed that since the data for the server is shared among multiple threads, it is the responsibility of the programmer to ensure that the data is accessed in a safe manner (for example, by using `synchronized` methods).

The client program in Figure 6.5 can be used to test this name server.

```
import java.util.*;
public class NameTable {
    final int maxSize = 100;
    private String[] names = new String[maxSize];
    private String[] hosts = new String[maxSize];
    private int[] ports = new int[maxSize];
    private int dirsize = 0;
    int search(String s) {
        for (int i = 0; i < dirsize; i++)
            if (names[i].equals(s)) return i;
        return -1;
    }
    int insert(String s, String hostName, int portNumber) {
        int oldIndex = search(s); // is it already there
        if ((oldIndex == -1) && (dirsize < maxSize)) {
            names[dirsize] = s;
            hosts[dirsize] = hostName;
            ports[dirsize] = portNumber;
            dirsize++;
            return 1;
        } else // already there, or table full
            return 0;
    }
    int getPort(int index) {
        return ports[index];
    }
    String getHostName(int index) {
        return hosts[index];
    }
}
```

Figure 6.3: Simple name table

```
import java.net.*;
import java.io.*;
import java.util.*;
public class NameServer {
    NameTable table;
    public NameServer() {
        table = new NameTable();
    }
    void handleclient(Socket theClient) {
        try {
            BufferedReader din = new BufferedReader
            (new InputStreamReader(theClient.getInputStream()));
            PrintWriter pout = new PrintWriter(theClient.getOutputStream());
            String getline = din.readLine();
            StringTokenizer st = new StringTokenizer(getline);
            String tag = st.nextToken();
            if (tag.equals("search")) {
                int index = table.search(st.nextToken());
                if (index == -1) // not found
                    pout.println(-1 + " " + "nullhost");
                else
                    pout.println(table.getPort(index) + " "
                    + table.getHostName(index));
            } else if (tag.equals("insert")) {
                String name = st.nextToken();
                String hostName = st.nextToken();
                int port = Integer.parseInt(st.nextToken());
                int retValue = table.insert(name, hostName, port);
                pout.println(retValue);
            }
            pout.flush();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
    public static void main(String[] args) {
        NameServer ns = new NameServer();
        System.out.println("NameServer started:");
        try {
            ServerSocket listener = new ServerSocket(Symbols.ServerPort);
            while (true) {
                Socket aClient = listener.accept();
                ns.handleclient(aClient);
                aClient.close();
            }
        } catch (IOException e) {
            System.err.println("Server aborted:" + e);
        }
    }
}
```

Figure 6.4: Name server

```
import java.lang.*; import java.util.*;
import java.net.*; import java.io.*;
public class Name {
    BufferedReader din;
    PrintStream pout;
    public void getSocket() throws IOException {
        Socket server = new Socket(Symbols.nameServer,
                                                Symbols.ServerPort);
        din = new BufferedReader(
                    new InputStreamReader(server.getInputStream()));
        pout = new PrintStream(server.getOutputStream());
    }
    public int insertName(String name, String hname, int portnum)
            throws IOException {
        getSocket();
        pout.println("insert " + name + " " + hname + " " + portnum);
        pout.flush();
        return Integer.parseInt(din.readLine());
    }
    public PortAddr searchName(String name) throws IOException {
        getSocket();
        pout.println("search " + name);
        pout.flush();
        String result = din.readLine();
        StringTokenizer st = new StringTokenizer(result);
        int portnum = Integer.parseInt(st.nextToken());
        String hname = st.nextToken();
        return new PortAddr(hname, portnum);
    }
    public static void main(String[] args) {
        Name myClient = new Name();
        try {
            myClient.insertName("hello1", "oak.ece.utexas.edu", 1000);
            PortAddr pa = myClient.searchName("hello1");
            System.out.println(pa.getHostName() + ":" + pa.getPort());
        } catch (Exception e) {
            System.err.println("Server aborted:" + e);
        }
    }
}
```

Figure 6.5: A client for name server

6.4.3 Example 2: A Linker

We now show a java class `Linker` that allows us to link a given set of processes with each other. Assume that we want to start n processes $P_1, P_2, \ldots, P_n$ in a distributed system and establish connections between them such that any of the process can send and receive messages with any other process. We would like to support direct naming to send and receive messages; that is, processes are unaware of the host addresses and port numbers. They simply use process identifiers $\{1 \ldots n\}$ to send and receive messages.

We first read the topology of the underlying network. This is done by the method `readNeighbors` in the class `Topology` shown in Figure 6.6. The list of neighbors of P_i are assumed to be enumerated in the file "topology$i$." If such a file is not found, then it is assumed that all other processes are neighbors.

```
import java.io.*;
import java.util.*;
public class Topology {
    public static void readNeighbors(int myId, int N,
                                     IntLinkedList neighbors) {
        Util.println("Reading topology");
        try {
            BufferedReader dIn = new BufferedReader(
                                 new FileReader("topology" + myId));
            StringTokenizer st = new StringTokenizer(dIn.readLine());
            while (st.hasMoreTokens()) {
                int neighbor = Integer.parseInt(st.nextToken());
                neighbors.add(neighbor);
            }
        } catch (FileNotFoundException e) {
            for (int j = 0; j < N; j++)
                if (j != myId) neighbors.add(j);
        } catch (IOException e) {
            System.err.println(e);
        }
        Util.println(neighbors.toString());
    }
}
```

Figure 6.6: Topology class

Now we discuss the `Connecter` class, which establishes connections between processes. Since processes may start at different times and at different locations, we use the `NameServer` to help processes locate each other. Any process P_i that starts up first creates a `ServerSocket` for itself. It uses the `ServerSocket` to listen

for incoming requests for communication with all small numbered processes. It then contacts the `NameServer` and inserts its entry in that table. All the smaller numbered processes wait for the entry of P_i to appear in the `NameServer`. When they get the port number from the `NameServer`, they use it to connect it to P_i. Once P_i has established a TCP connection with all smaller number processes, it tries to connect with higher-number processes. This class is shown in Figure 6.7. For simplicity, it is assumed that the underlying topology is completely connected.

Once all the connections are established, the `Linker` provides methods to send and receive messages from process P_i to P_j. We will require each message to contain at least four fields: source identifier, destination identifier, message type (or the message tag), and actual message. We implement this in the Java class shown in Figure 6.8.

The Linker class is shown in Figure 6.9. It provides methods to send and receive messages based on process identifiers. Different `send` methods have been provided to facilitate sending messages of different types. Every message is assumed to have a field `tag` that corresponds to the message tag (or the message type).

6.5 Remote Method Invocations

A popular way of developing distributed applications is based on the concept of remote procedure calls (RPCs) or remote method invocations (RMIs). Here the main idea is that a process can make calls to methods of a remote object as if it were on the same machine. The process making the call is called a *client* and the process that serves the request is called the *server*. In RMI, the client may not even know the location of the remote object. This provides *location transparency* to the client. In Java, for example, the remote object may be located using `rmiregistry`. Alternatively, references to remote objects may be passed around by the application as references to local objects.

A call to a method may have some arguments, and the execution of the method may return some value. The arguments to the method when the object is remote are sent via a message. Similarly, the return value is transmitted to the caller via a message. All this message passing is hidden from the programmer, and therefore RMI can be viewed as a higher-level programming construct than sending or receiving of messages.

Although the idea behind RMI is quite simple, certain issues need to be tackled in implementing and using RMI. Since we are passing arguments to the method, we have to understand the semantics of the parameter passing. Another issue is that of a failure. What happens when the messages get lost? We will look at such issues in this section.

```
import java.util.*; import java.net.*; import java.io.*;
public class Connector {
    ServerSocket listener; Socket [] link;
    public void Connect(String basename, int myId, int numProc,
    BufferedReader[] dataIn, PrintWriter[] dataOut) throws Exception {
        Name myNameclient = new Name();
        link = new Socket[numProc];
        int localport = getLocalPort(myId);
        listener = new ServerSocket(localport);

        /* register in the name server */
        myNameclient.insertName(basename + myId,
        (InetAddress.getLocalHost()).getHostName(), localport);

        /* accept connections from all the smaller processes */
        for (int i = 0; i < myId; i++){
            Socket s = listener.accept();
            BufferedReader dIn = new BufferedReader(
                    new InputStreamReader(s.getInputStream()));
            String getline = dIn.readLine();
            StringTokenizer st = new StringTokenizer(getline);
            int hisId = Integer.parseInt(st.nextToken());
            int destId = Integer.parseInt(st.nextToken());
            String tag = st.nextToken();
            if (tag.equals("hello")) {
                link[hisId] = s;
                dataIn[hisId] = dIn;
                dataOut[hisId] = new PrintWriter(s.getOutputStream());
            }
        }
        /* contact all the bigger processes */
        for (int i = myId + 1; i < numProc; i++){
            PortAddr addr;
            do {
                addr = myNameclient.searchName(basename + i);
                Thread.sleep(100);
            } while (addr.getPort() == -1);
            link[i] = new Socket(addr.getHostName(), addr.getPort());
            dataOut[i] = new PrintWriter(link[i].getOutputStream());
            dataIn[i] = new BufferedReader(new
            InputStreamReader(link[i].getInputStream()));
            /* send a hello message to P_i */
            dataOut[i].println(myId +" "+ i +" "+ "hello" + " " + "null");
            dataOut[i].flush();
        }
    }
    int getLocalPort(int id) { return Symbols.ServerPort + 10 + id; }
    public void closeSockets(){
        try {
            listener.close();
            for (int i=0;i<link.length; i++) link[i].close();
        } catch (Exception e) {System.err.println(e);}
    }
}
```

Figure 6.7: Connector class

```
import java.util.*;
public class Msg {
    int srcId, destId;
    String tag;
    String msgBuf;
    public Msg(int s, int t, String msgType, String buf) {
        this.srcId = s;
        destId = t;
        tag = msgType;
        msgBuf = buf;
    }
    public int getSrcId() {
        return srcId;
    }
    public int getDestId() {
        return destId;
    }
    public String getTag() {
        return tag;
    }
    public String getMessage() {
        return msgBuf;
    }
    public int getMessageInt() {
        StringTokenizer st = new StringTokenizer(msgBuf);
        return Integer.parseInt(st.nextToken());
    }
    public static Msg parseMsg(StringTokenizer st){
        int srcId = Integer.parseInt(st.nextToken());
        int destId = Integer.parseInt(st.nextToken());
        String tag = st.nextToken();
        String buf = st.nextToken("#");
        return new Msg(srcId, destId, tag, buf);
    }
    public String toString(){
        String s = String.valueOf(srcId)+" " +
                   String.valueOf(destId)+ " " +
                   tag + " " + msgBuf + "#";
        return s;
    }
}
```

Figure 6.8: Message class

```
import java.util.*;
import java.io.*;
public class Linker {
    PrintWriter[] dataOut;
    BufferedReader[] dataIn;
    BufferedReader dIn;
    int myId, N;
    Connector connector;
    public IntLinkedList neighbors = new IntLinkedList();
    public Linker(String basename, int id, int numProc) throws Exception {
        myId = id;
        N = numProc;
        dataIn = new BufferedReader[numProc];
        dataOut = new PrintWriter[numProc];
        Topology.readNeighbors(myId, N, neighbors);
        connector = new Connector();
        connector.Connect(basename, myId, numProc, dataIn, dataOut);
    }
    public void sendMsg(int destId, String tag, String msg) {
        dataOut[destId].println(myId + " " + destId + " " +
                                    tag + " " + msg + "#");
        dataOut[destId].flush();
    }
    public void sendMsg(int destId, String tag) {
        sendMsg(destId, tag, " 0 ");
    }
    public void multicast(IntLinkedList destIds, String tag, String msg){
        for (int i=0; i<destIds.size(); i++) {
            sendMsg(destIds.getEntry(i), tag, msg);
        }
    }
    public Msg receiveMsg(int fromId) throws IOException {
        String getline = dataIn[fromId].readLine();
        Util.println(" received message " + getline);
        StringTokenizer st = new StringTokenizer(getline);
        int srcId = Integer.parseInt(st.nextToken());
        int destId = Integer.parseInt(st.nextToken());
        String tag = st.nextToken();
        String msg = st.nextToken("#");
        return new Msg(srcId, destId, tag, msg);
    }
    public int getMyId() { return myId; }
    public int getNumProc() { return N; }
    public void close() {connector.closeSockets();}
}
```

Figure 6.9: Linker class

An RMI is implemented as follows. With each remote object there is an associated object at the client side and an object at the server side. An invocation to a method to a remote object is managed by using a local surrogate object at the client called the *stub* object. An invocation of a method results in packing the method name and the arguments in a message and shipping it to the server side. This is called *parameter marshaling*. This message is received on the server side by the server skeleton object. The skeleton object is responsible for receiving the message, reconstructing the arguments, and then finally calling the method. Note that a RMI class requires compilation by a RMI compiler to generate the stub and the skeleton routines.

6.5.1 Remote Objects

An object is called *remote object* if its methods can be invoked from another Java virtual machine running on the same host or a different host. Such an object is described using a `remote` interface. An interface is remote if it extends java.rmi.Remote. The remote interface serves to identify all remote objects. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a remote interface are available remotely. Figure 6.10 gives a remote interface for a name service.

```
import java.rmi.*;
public interface NameService extends Remote {
    public int search(String s) throws RemoteException;
    public int insert(String s, String hostName, int portNumber)
           throws RemoteException;
    public int getPort(int index) throws RemoteException;
    public String getHostName(int index) throws RemoteException;
}
```

Figure 6.10: Remote interface

Any object that implements a remote interface and extends `UnicastRemoteObject` is a remote object. Remote method invocation corresponds to invocation of one of the methods on a remote object. We can now provide a class that implements the `NameService` as shown in Figure 6.11.

To install our server, we first compile the file `NameServiceImpl.java`. Then, we need to invoke the RMI compiler to generate the stub and skeleton associated with the server. On a UNIX machine, one may use the following commands to carry out these steps:

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class NameServiceImpl extends UnicastRemoteObject
        implements NameService {
    final int maxSize = 100;
    private String[] names = new String[maxSize];
    private String[] hosts = new String[maxSize];
    private int[] ports = new int[maxSize];
    private int dirsize = 0;
    public NameServiceImpl() throws RemoteException {
    }
    public int search(String s) throws RemoteException {
        for (int i = 0; i < dirsize; i++)
            if (names[i].equals(s)) return i;
        return -1;
    }
    public int insert(String s, String hostName, int portNumber)
            throws RemoteException {
        int oldIndex = search(s); // is it already there
        if ((oldIndex == -1) && (dirsize < maxSize)) {
            names[dirsize] = s;
            hosts[dirsize] = hostName;
            ports[dirsize] = portNumber;
            dirsize++;
            return 1;
        } else
            return 0;
    }
    public int getPort(int index) throws RemoteException {
        return ports[index];
    }
    public String getHostName(int index) throws RemoteException {
        return hosts[index];
    }
    public static void main(String args[]) {
        // create security manager
        System.setSecurityManager(new RMISecurityManager());
        try {
            NameServiceImpl obj = new NameServiceImpl();
            Naming.rebind("MyNameServer", obj);
            System.out.println("MyNameServer bound in registry");
        } catch (Exception e) {
            System.out.println("NameServiceImpl err: " + e.getMessage());
        }
    }
}
```

Figure 6.11: A name service implementation

```
> javac NameServiceImpl.java
> rmic NameServiceImpl
> rmiregistry &
```

Now assuming that the `rmiregistry` service is running on the machine, we can start our server. There is just one last thing that we need to take care of: security. We need to specify who can connect to the server. This specification is done by a security policy file. For example, consider a file called `policy` as follows:

```
grant {
permission java.net.SocketPermission "*:1024-65535",
"connect,accept";
permission java.net.SocketPermission "*:80", "connect";
};
```

This policy allows downloaded code, from any code base, to do two things: (1) connect to or accept connections on unprivileged ports (ports greater than 1024) on any host, or (2) connect to port 80 [the port for HTTP(Hypertext Transfer Protocol)].

Now we can start the NameServiceImpl server as follows:

```
> java -Djava.security.policy=policy NameServiceImpl
```

6.5.2 Parameter Passing

If a local object is passed as an argument to a *local* method on a local object, then in Java we simply pass the reference to the object. However, if the method is to a remote object, then reference to a local object is useless at the other side. Therefore, arguments to remote methods are handled differently.

There are three ways of passing arguments (and returning results) in remote method invocations. The primitive types in Java (e.g., `int` and `boolean`) are passed by values.

Objects that are not remote are passed by value using *object serialization*, which refers to the process of converting the object state into a stream of bytes. Any object that implements the interface `Serializable` can be communicated over the Internet using serialization. The object is written into a stream of bytes at one end ("serialized") and at the other end it is reconstructed from the stream of bytes received ("deserialized"). An interesting question is what happens if the object has references to other objects. In this case, those objects also need to be serialized; otherwise references will be meaningless at the other side. Thus, all objects that are reachable from that object get serialized. The same mechanism works when

a nonremote object is returned from a remote method invocation. Java supports referential integrity, that is, if multiple references to the same object are passed from one Java Virtual Machine (JVM) to the other, then those references will refer to a single copy of the object in the receiving JVM.

Finally, references to objects that implement `remote` interface are passed as remote references. In this case, the stub for the remote object is passed.

6.5.3 Dealing with Failures

One difference between invoking a local method and a remote method is that more things can go wrong when a remote method is invoked. The machine that contains the remote object may be down, the connection to that machine be down, or the message sent may get corrupted or lost. In spite of all these possible problems, Java system guarantees *at-most-once* semantics for a remote method invocation: any invocation will result in execution of the remote method at most once.

6.5.4 Client Program

The client program first needs to obtain a reference for the remote object. The `java.rmi.Naming` class provides methods to do so. It is a mechanism for obtaining references to remote objects based on Uniform Resource Locator (URL) syntax. The URL for a remote object is specified using the usual host, port, and name:

`rmi://host:port/name`

where `host` is the host name of registry (defaults to current host), `port` is the port number of registry (defaults to the registry port number), and `name` is the name for the remote object.

The key methods in this class are

```
bind(String, Remote)
  Binds the name to the specified remote object.
list(String)
  Returns an array of strings of the URLs in the registry.
lookup(String)
  Returns the remote object for the URL.
rebind(String, Remote)
  Rebind the name to a new object; replaces any existing binding.
unbind(String)
  Unbind the name.
```

We now show how a client can use `lookup` to get a reference of the remote object and then invoke methods on it (see the program in Figure 6.12).

```
import java.rmi.*;
public class NameRmiClient {
    public static void main(String args[]) {
        try {
            NameService r = (NameService)
                            Naming.lookup("rmi://linux02/MyNameServer");
            int i = r.insert("p1", "tick.ece", 2058);
            int j = r.search("p1");
            if (j != -1)
               System.out.println(r.getHostName(j) +":" + r.getPort(j));
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Figure 6.12: A RMI client program

6.6 Other Useful Classes

In this chapter, we have focused on classes that allow you to write distributed programs. For cases when a process simply needs data from a remote location, Java provides the Uniform Resource Locator (URL) class. A URL consists of six parts: protocol, hostname, port, path, filename, and document section. An example of a URL is

```
http://www.ece.utexas.edu:80/classes.html#distributed
```

The java.net.URL class allows the programmer to read data from a URL by methods such as

```
public final InputStream openStream()
```

This method returns a `InputStream` from which one can read the data. For different types of data such as images and audio clips there are methods such as
`public Image getImage(URL u, String filename)`
and
`public void play(URL u).`

We will not concern ourselves with these classes and methods.

6.7 Problems

6.1. Make the `NameServer` class fault-tolerant by keeping two copies of the server process at all times. Assume that the client chooses a server at random.

If that server is down (i.e., after the timeout), the client contacts the other server. You may assume that at most one server goes down. When the server comes up again, it would need to synchronize with the other server to ensure consistency.

6.2. Message passing can also be employed for communication and synchronization among threads. Implement a Java monitor library that provides message passing primitives for threads in a single Java Virtual Machine (JVM).

6.3. Develop a `Linker` class that provides *synchronous* messages. A message is synchronous if the sender of the message blocks until the message is received by the receiver.

6.4. Give advantages and disadvantages of using synchronous messages (see Problem 6.3) over asynchronous messages for developing distributed applications.

6.5. Write a Java program to maintain a large linked list on multiple computers connected by a message passing system. Each computer maintains a part of the linked list.

6.6. List all the differences between a local method invocation and a remote method invocation.

6.7. How will you provide semaphores in a distributed environment?

6.8. Solve the producer consumer problem discussed in Chapter 3 using messages.

6.9. Give advantages and disadvantages of using RMI over TCP sockets for developing distributed applications.

6.8 Bibliographic Remarks

Details on the Transmission Control Protocol can be found in the book by Comer [Com00]. Remote procedure calls were first implemented by Birrell and Nelson [BN84].