Distributed Communication

In this chapter, we discuss several different strategies that allow distributed applications to communicate. We have already provided an overview of sockets and Remote Method Invocation (RMI) in Chapter 3. In this chapter we provide further examples of socket programs written in Java as well as additional program examples using RMI. In addition, we provide coverage of CORBA which allows distributed (and possibly heterogeneous) applications to communicate.

D.1 Sockets

A **socket** is defined as an endpoint for communication and is identified by an IP address concatenated with a port number. A pair of communicating processes communicating over a network employs a pair of sockets – one for each process. This pair of sockets defines the connection between the pair of communicating processes.

In Chapter 3 we examined a date server which listens to a specified port. When it receives a connection from a client, it returns the current date and time to the client, closes the connection, and resumes listening for more client requests. This server is written using a single thread as servicing each client only requires delivering the time-of-day; the duration of the connection is brief allowing the server to service many concurrent client requests without much delay.

However consider the situation where servicing a client requires maintaining a connection with the client. As an example, consider an **echo server** which simply "echoes" the exact text it receives back to the sending client. Once a connection is established between the client and the echo server, the server waits for text input from the client, echoing any text it receives back to the client. The server can listen for subsequent client connections only when the connection between the existing client is closed.

Consider if we were to implement the echo server using a single thread as shown below:

```
while (true) {
   // listen for connections
   Socket client = sock.accept();

   /**
    * Echo whatever is received back to the client.
    * We will resume listening for subsequent
    * connections only when the current client
    * indicates it wishes to terminate the connection.
    */
}
```

In this situation, we are only able to service one client at a time as the single thread is devoted to the current connection. Subsequent clients have to wait for the existing connection to terminate before being serviced. Such a design does not provide for a responsive server if we wish to quickly service incoming client connections.

In Chapter 4 we introduced threads and this seems to address the issue of making the server more responsive. We could multithread the server and service each client connection in a separate thread which we will call `EchoThread`. Such a design appears as:

```
while (true) {
   // listen for connections
   Socket client = sock.accept();

   // pass the socket to a separate thread
   // and resume listening for more requests
   EchoThread echo = new EchoThread(client);
   echo.start();
}
```

This solution allows client requests to quickly be serviced in a new thread or by using a thread pool. However this may lead to a costly design when we consider the typical echo client - much of the time it may be idle and only infrequently send text to the server. Thus, we are using a separate thread on the server to maintain a connection with each client; a connection that may often be idle.

This design of such servers isn't considered **scalable**. We define a service as being scalable if it provides an increase in throughput corresponding to an increase in the load upon the service. Furthermore, once throughput reaches a point of saturation (i.e. we cannot achieve a higher rate of throughput), any additional load should not decrease throughput. Due to the cost of using a separate thread to maintain the state of each client connection, the server can only allow a certain number of connections before over-burdening and saturating the server thereby causing a longer delay when responding to each client.

Although we are using a simple echo application as an example, the use of a separate thread on a server to maintain the state of each client connection may inhibit the scalability of many other client–server applications such as web and file servers.

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import java.nio.charset.*;

public class NBEchoServer
{
   private static final int PORT = 6007;
   private Charset charset;
   private CharsetDecoder decoder;
   private CharsetEncoder encoder;
   private ByteBuffer buffer;
   private ServerSocketChannel server;
   private ServerSocket sock;
   private Selector acceptSelector;

   public NBEchoServer() throws IOException {
      // Figure D.2
   }

   public void startServer() throws IOException {
      // Figure D.3
   }

   public static void main(String[] args) {
      try {
         (new NBEchoServer()).startServer();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figure D.1 Echo server using non-blocking I/O.

In Chapter 13 we introduced non-blocking I/O, which allows input and output to be performed without blocking system calls. Consider how I/O is ordinarily performed: a system call (that is, `read()`) is made and the application blocks until the requested data (in its entirety) is available. With non-blocking I/O, a system call is made and it immediately returns with as much data that is currently available. It is possible that no data may be available at all, thereby requiring the application to be coded to continually perform the non-blocking system call until it reads any available data. However, the system can also provide an indication whenever at least some data is available. Using this knowledge, an application could alternatively be coded to only perform the non-blocking system call when there is an indication some data is available.

We can design more scalable servers using non-blocking I/O rather than using a separate thread to service each client connection. Such server designs often allow more concurrent client connections as we don't require the overhead of a thread to maintain the state of each client connection. Rather, the state of each connection is maintained by the server and a state change is only triggered by an I/O event. Furthermore, by minimizing the overhead required to maintain each client connection, we are minimizing response time when responding to each client.

This does not mean that that our presentation of threads was not useful. Rather, modern scalable servers such as web and chat servers, instant message servers, etc. typically rely on both threads and non-blocking I/O to service a large number of client connections with minimal response latency. In the following section we cover the non-blocking I/O features provided in Java.

D.1.1 Non-Blocking I/O in Java

In this section, we present the echo server using non-blocking I/O in Java. This program is shown in Figure D.1. There are several new issues to be covered in this program including:

1. character sets;
2. buffers;
3. channels;
4. selectors.

Character sets are sets of character representations. Example character representations include the Roman character "A" and the Greek character λ. Each character within a character set is typically assigned a numeric value and there are many different mappings of characters to numeric values; ASCII and Unicode are two such examples. As I/O over a network typically involves transmissions of byte streams, the numeric value for each character must be **encoded** and **decoded** to and from a sequence of bytes.

A **buffer** is a linear sequence of elements containing primitive types (i.e. `byte`, `char`, `int`, etc.) A buffer has associated with it a *capacity* and *position*. The capacity refers to the number of elements the buffer may hold. The buffer's capacity is established when the buffer is first allocated and it cannot change. The position of a buffer refers to the logical index of the next element to be read or written. After a number of elements have been placed into a buffer, reading the data from the buffer first requires setting the position back to zero.

Channels serve as conduits for performing I/O from files, hardware devices, and sockets. Typically channels and buffers work in tandem whereby the contents of a buffer may be written to or read from a channel.

Selectors provide the mechanism for performing non-blocking I/O in Java. Selectors are used to multiplex many I/O channels concurrently. Rather than blocking while waiting for input from one channel (thereby ignoring the remaining channels), a selector can simultaneously wait for I/O readiness from any number of channels. When I/O is available on one (or more) channels, the selector then indicates which channel(s) are ready. I/O can then be performed on available channels.

```
public NBEchoServer() throws IOException {
   charset = Charset.forName("ISO-8859-1");
   CharsetDecoder decoder = charset.newDecoder();
   CharsetEncoder encoder = charset.newEncoder();

   ByteBuffer buffer = ByteBuffer.allocate(256);

   server = ServerSocketChannel.open();
   sock = server.socket();
   sock.bind(new InetSocketAddress(PORT));

   server.configureBlocking(false);
   acceptSelector = Selector.open();
   server.register(acceptSelector, SelectionKey.OP_ACCEPT);
}
```

Figure D.2 Constructor for `NBEchoServer`.

Before we proceed with the necessary details of non-blocking I/O in Java, let's first illustrate how to run the echo server shown in Figure D.1. For the purposes of this discussion, let's assume the server is running on the localhost with IP address `127.0.0.1`.

The server is started with the command

```
java NBEchoServer
```

and is now listening for client connection requests to port 6007. A client may connect to the server using telnet by invoking the command

```
telnet 127.0.0.1 6007
```

A connection has now been established between the client and echo server. When the client enters text, it is sent to the server and echoed back to the client. When the client wishes to terminate the connection, it enters a period.

With this necessary material covered, we are now ready to present the details in Figure D.1. We will first examine the constructor shown in Figure D.2.

The character set to be used is `ISO-8859-1` as this is suitable for most Western character sets. We then create a character set encoder and decoder used to encode and decode between byte streams and the specified character set.

The statement

```
buffer = ByteBuffer.allocate(256);
```

creates a buffer containing `byte` elements with a capacity of 256 bytes. We then create a `ServerSocketChannel` with the

```
server = ServerSocketChannel.open();
```

statement. The socket associated with this channel is then retrieved by invoking the `socket()` method on the `ServerSocketChannel`. Initially the `ServerSocket` is unbound, the `bind()` method is used to bind this socket to the default port.

```
public void startServer() throws IOException {
   while (true) {
      acceptSelector.select();
      Set keys = acceptSelector.selectedKeys();

      for (Iterator itr = keys.iterator(); itr.hasNext();) {
         SelectionKey key = (SelectionKey)itr.next();
         itr.remove();
         if (key.isAcceptable()) {
            SocketChannel client = server.accept();
            client.configureBlocking(false);
            SelectionKey clientKey = client.register
               (acceptSelector, SelectionKey.OP_READ);
         }
         else if (key.isReadable()) {
            SocketChannel clientChannel =
               (SocketChannel)key.channel();
            if (clientChannel.read(buffer) == -1) {
               key.cancel();
               clientChannel.close();
            }
            else {
               buffer.flip();
               String message
                  = decoder.decode(buffer).toString();
               if (message.trim().equals(".")) {
                  clientChannel.write(encoder.encode
                     (CharBuffer.wrap("BYE"));
                  key.cancel();
                  clientChannel.close();
               }
               else {
                  buffer.flip();
                  clientChannel.write(buffer);
               }
            }
            buffer.clear();
         }
      }
   }
}
```

Figure D.3 `startServer()` method for the non-blocking echo server.

The statement `server.configureBlocking(false);` configures the socket channel as non-blocking.

We then open a selector that we will use to manage the multiplexed channels using the `open()` method which is a `static` method in the `Selector` class. Channels that are to be managed by the selector must first register with

the selector and include the type of readiness the selector will wait upon. The statement

```
server.register(acceptSelector, SelectionKey.OP_ACCEPT);
```

registers the socket channel `server` indicating the type of readiness the selector can expect from this channel is `SelectionKey.OP_ACCEPT` - a connection request from a client.

Once we have created an instance of `NBEchoServer`, we invoke the `startServer()` method in Figure D.3. The echo server first invokes the `select()` method of the selector. Even though we are performing non-blocking I/O, the call to `select()` will block until at least one of the registered channels is ready. Recall we initially only register the socket channel `server`; the only readiness we can expect at first from this selector is connect requests from clients.

After returning from `select()`, we then extract a set of `SelectionKeys` with the call to `acceptSelector.selectedKeys()`; This set of keys represents all the I/O that is available on all channels registered with the selector. It is possible to test the type of readiness each selection key is available to perform. For the purposes of our echo server, each selection key may be either (1) a connection request (`isAcceptable()` returns `true`), or (2) data is available to be read from the channel (`isReadable()` returns `true`.)

Let's consider the case where our echo server has received its first connection request. In this instance, we then call the `accept()` method of the server socket channel to accept the client connection. Ordinarily `accept()` is a blocking call but since we have configured the channel as non-blocking, `accept()` will return immediately, possibly returning a `null` value. However, we *know* `accept()` will return a socket connection as the channel readiness indicated a connection request; that is, `isAcceptable()` for the `SelectionKey` returned `true`. We then retrieve the channel associated with the client socket connection with the statement `client = server.accept();`, configure this channel non-blocking as well, and finally register the `client` channel with the selector. In this situation, this channel will register its readiness as `SelectionKey.OP_READ` indicating to the selector the type of readiness it can expect from this channel is that data is available to be read. Because we have configured the channel associated with the client connection as non-blocking, subsequent reads from this channel will return with whatever data is available on the channel. However we will only read from this channel when the selector indicates I/O readiness of the channel.

We now have two channels registered with the selector: (1) the server socket channel awaiting further client connections, and (2) the channel associated with the first client connection. Subsequent calls to the `select()` method of the selector may indicated readiness on either of these channels.

If the selection key is readable (`isReadable()` returns `true`), this indicates a client has written data to the socket. The echo server must read this data from the available channel and echo it back to the client. We first retrieve the channel with available data with the statement:

```
clientChannel = (SocketChannel)key.channel();
```

Recall our earlier discussion that channels and buffers work in tandem; available data from a channel must be read into a buffer. We read the data from

the channel into the buffer with the `clientChannel.read(buffer);` statement. However, after reading the contents of the channel into the buffer, the position of the buffer is set to the next element to be read into the buffer. To extract the contents of the buffer, we must set the position of the buffer back to zero with the statement `buffer.flip();` We then decode the contents of the buffer into a `String` object so we may test if the client wishes to terminate the connection (by entering a period "."). If they do not wish to terminate the connection, we write (i.e. echo) the contents of the buffer back to the client channel with the statement `clientChannel.write(buffer);`. However it is important to note that we must `flip()` the buffer again before writing it back to the channel. This is necessary because the position of the buffer was modified when we decoded the buffer into a `String` object. Before we resume testing the selector, we must first clear the buffer to prepare it for new reads.

Lastly, notice how we handle the situation if the client indicates it wishes to terminate the connection (by entering a period ".") or if the client has closed its channel. This is detected by the server when the `read()` method from the channel returns -1. In either of these situations, we must first cancel the registration of the channel with the selector by invoking the statement `key.cancel();` followed by closing the channel on the server side.

D.2 UDP Sockets

Our coverage of sockets so far has only considered connection-oriented TCP sockets. When we say TCP sockets are **connection-oriented**, we are referring to the connection established between the socket pair. This connection behaves much like a telephone call taking place between two people which we will name Bob and Alice. Suppose Bob wishes to call Alice; he dials her telephone number and waits for her to pick up the receiver. When she does so, there is a connection established between the two and is only terminated when Bob or Alice hangs up. We see evidence of the connection in our Java TCP socket programs with the `InputStream` and `OutputStream` that is available with a socket.

Furthermore, TCP sockets are considered **reliable** meaning that all data will be reliably transmitted across the socket. TCP provides reliability through acknowledgments - upon correct receipt of data, the receiver sends an acknowledgement to the sender. TCP also maintains **flow control** between communicating hosts which prevents a fast sender from overloading a slow receiver with data. Lastly, TCP provides **congestion control** which prevents a sender from delivering too much data on the network thereby saturating overall network performance. The fundamental distinction between flow and congestion control is that flow control is an issue between a sender and receiver, congestion control is a network-wide issue.

TCP is used for a majority of Internet protocols including HTTP, FTP, SMTP, and Telnet which require the reliability provided by TCP sockets. For example, web clients use HTTP to reliably download a file from a web server, electronic mail users require the reliability of SMTP to receive and deliver email. However, the reliability, flow, and congestion control provided by TCP comes at the price of increased overhead. This overhead typically results in the data being

delivered more slowly than a delivery mechanism that does not provide any reliability guarantees.

UDP sockets are connection-less and unreliable: there is no connection established between communicating hosts and there is no guarantee data will be reliably delivered across the socket. Because UDP does not incur the overhead of TCP, data delivered using UDP arrives more quickly than with TCP. An unreliable protocol such as UDP is appealing to applications that can tolerate a certain level of packet loss such as streaming audio and video where a few lost packets may not even be noticeable to the viewer.

D.2.1 UDP Date Server

In this section we construct an application that returns the current date and time using UDP sockets similar to the TCP-based solution first shown in Chapter 3. The server will listen to port 6013 (which we chose arbitrarily) for client requests. The two key Java classes in use with UDP are `DatagramSocket` and `DatagramPacket`. A `DatagramSocket` is used to establish a UDP socket. Unlike TCP-based sockets which distinguish between server and client sockets (`ServerSocket` and `Socket`), a `DatagramSocket` is used for both servers and clients.

As described above, UDP sockets are connection-less. This means there is no connection - hence no `InputStream` or `OutputStream` - available to communicate across. All data - including the address and port it is being delivered to - is encapsulated into a `DatagramPacket`. The `DatagramPacket` is delivered using the `DatagramSocket`. Earlier in this Section we used a telephone conversation as an analogy of connection-oriented TCP sockets where addressing (i.e. a telephone number) is only necessary to establish the connection. Once the connection is established, it can be used to send and receive data without further addressing. The equivalent analogy for UDP sockets is the postal system where communication is connection-less and takes place using letters. Unlike in a connection-oriented system where an address is only necessary during connection establishment, each `Datagrampacket` must contain the address of the receiver.

With this basics of UDP established, let's first look at the Date client shown in Figure D.4.

The client first establishes a `DatagramSocket` with the statement:

```
server = new DatagramSocket();
```

This establishes a socket on the client system binding the socket to an arbitrary port number. Because UDP is unreliable, it is possible the client may wait for a response from the server that it may never receive. We will set the timeout of the socket to 5000 milliseconds (5 seconds) with the statement `theSocket.setSoTimeout(5000);`. If the socket times out, an `IOException` will be thrown.

The client will then construct a `DatagramPacket` with zero bytes of data, the datagram will also include the IP address of the server (which we will use the localhost or `127.0.0.1`) and the port the server is listening to (6013). This is accomplished with the statement:

```
sendPacket = new DatagramPacket(new byte[0],0,server,PORT);
```

```
import java.net.*;
import java.io.*;

public class UDPDateClient {
   public static void main(String[] args) throws IOException {
      final int PORT = 6013;
      final int BUFFER_SIZE = 256;
      DatagramSocket theSocket = null;

      try {
         theSocket = new DatagramSocket();
         theSocket.setSoTimeout(5000);
         InetAddress server = InetAddress.getByName("127.0.0.1");

         // construct a zero-length packet to send to the server
         DatagramPacket sendPacket =
             new DatagramPacket(new byte[0],0,server,PORT);

         // construct a packet to receive a response from the server
         byte[] buffer = new byte[BUFFER_SIZE];
         DatagramPacket receivePacket =
             new DatagramPacket(buffer, buffer.length);

         // send the packet over the socket
         theSocket.send(sendPacket);

         // wait for a response
         theSocket.receive(receivePacket);

         // extract the date from the datagram
         byte[] data = receivePacket.getData();
         int length = receivePacket.getLength();

         String currentDate = new String(data, 0, length);
         System.out.println(currentDate);
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
      finally {
         if (theSocket != null)
            theSocket.close(); }}
}
```

Figure D.4 Date client using UDP sockets.

Why zero bytes of data? Well, the client is simply sending an empty datagram to the server that indicates it would like the current date and time. The server will send back to the client a `DatagramPacket` with the date and time as its data.

Before sending the empty packet to the server with the statement `theSocket.send(sendPacket);`, the client will first construct a `DatagramPacket` of size `BUFFER_SIZE` that will be used to store the date and time it receives from the server. This is accomplished with the statements:

```
byte[] buffer = new byte[BUFFER_SIZE];
receivePacket = new DatagramPacket(buffer, buffer.length);
```

of parbox

The client then delivers the datagram `sendPacket` to the socket and waits to receive the result with `theSocket.receive(receivePacket);`. The response from the server will be encapsulated into the `receivePacket` datagram. The data portion of the `DatagramPacket` is delivered as an array of bytes which the client must extract using the `getData()` method of the `DatagramPacket` class. The client will also determine the length of the data field with the `getLength()` also in the `DatagramPacket` class. Using the data and its length, the client then constructs a `String` object of the current date and time.

With the role of the client established, let's now look at the server side of the application as shown in Figure D.5. The server first establishes a `DatagramSocket` and listens to port 6013 for client requests with the statement:

```
server = new DatagramSocket(PORT);
```

The server then constructs a 1-byte `DatagramPacket` for receiving client requests. As noted earlier, the client will not send any data in the datagram it delivers to the server so it might make sense for the server to also construct a datagram with zero bytes of data to receive client requests. However, some implementations of the Java virtual machine (JVM) require a data field containing at least 1 byte when receiving data and therefore we adopt this conservative approach in this text. The server then awaits client requests with the statement:

```
server.receive(receivePacket);
```

When the server receives a client request, it will respond with the current date and time. However, the server must know the IP address and port of the client to send this data to. (Recall with TCP connection-oriented sockets this was not an issue as there was an `InputStream` and `OutputStream` associated with the socket.) The server will extract this information with the statements

```
InetAddress remoteAddr = receivePacket.getAddress();
int remotePort = receivePacket.getPort();
```

mbox

The server then determines the current date (as a `String` object) and gets the byte equivalent using the `getBytes()` method of the `String` class. With the address and port of the remote system and the current date as data, the server then creates a new `DatagramPacket` and delivers it to the sender.

D.3 Remote Method Invocation

Remote method invocation, or (**RMI**), is a Java feature that allows a thread to invoke a method on a remote object. Objects are considered remote if they

```
import java.net.*;
import java.io.*;

public class UDPDateServer
{
   public static void main(String[] args) throws IOException {
      final int PORT = 6013;
      DatagramSocket server = null;

      try {
         // create a Datagramsocket listening to the default port
         server = new DatagramSocket(PORT);

         // the packet for receiving empty datagrams
         DatagramPacket receivePacket =
            new DatagramPacket(new byte[1],1);

         while (true) {
            server.receive(receivePacket);

            // get the IP address and the port of the sender
            InetAddress remoteAddr = receivePacket.getAddress();
            int remotePort = receivePacket.getPort();

            // record the current date and extract its byte equivalent
            String currentDate = (new java.util.Date()).toString();
            byte[] data = currentDate.getBytes("ISO-8859-1");

            DatagramPacket sendPacket = new
            DatagramPacket(data, data.length, remoteAddr, remotePort);
            server.send(sendPacket);
          }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
      finally {
         if (server != null)
           server.close();
      }
   }
}
```

Figure D.5 Date server using UDP sockets.

reside in a different **Java Virtual Machine** (JVM). Therefore, the remote object may be in a different JVM on the same computer or on a remote host connected by a network. By allowing a Java program to invoke methods on remote objects, RMI makes it possible for users to develop Java applications that are distributed across a network.

Recall from Chapter 3 that RMI implements the remote object using stubs and skeletons. A **stub** is a proxy for the remote object; it resides with the client. When a client invokes a remote method, it is this stub for the remote object that is called.

In this section, we build a distributed message-passing solution to the producer–consumer problem that uses RMI. The message passing solution to the producer–consumer was first presented in Chapter 4. This distributed solution will consist of a remote object for messages that will be passed by reference, allowing the producer and consumer threads to invoke the `send()` and `receive()` methods remotely.

D.3.1 Remote Objects

We define remote objects by first declaring an interface that specifies the methods that may be invoked remotely. In Chapter 3 we defined the `Channel` interface which specifies the `send()` and `receive()` methods for message passing. To provide for remote objects, this interface must also extend the `java.rmi.Remote` interface, which identifies objects implementing this interface as being remote. Further, each method declared in the interface must throw the exception `java.rmi.RemoteException`. For remote objects, we provide the `RemoteChannel` interface shown in Figure D.6.

The class that defines the remote object must implement the `RemoteChannel` interface (Figure D.7). In addition to defining the methods of the interface, the class must also extend `java.rmi.server.UnicastRemoteObject`. Extending `UnicastRemoteObject` allows the creation of a single remote object that listens for network requests using RMI's default scheme of sockets for network communication. This object also includes a `main()` method. The `main()` method creates an instance of the object and registers with the RMI registry running on the server with the `rebind()` method. In this case, the object instance registers itself with the name `MessageQueue`. Also note that the constructor for the `MessageQueue` class must throw a `RemoteException` if a communication or network failure prevents RMI from exporting the remote object.

Note that the implementations of the `send()` and `receive()` methods are declared as `synchronized` to ensure thread-safety. We can also use Java thread synchronization, described in Chapter 6, to control concurrent access to remote objects.

D.3.2 Access to the Remote Object

Once an object is registered on the server, a client (as shown in Figure D.9) can get a proxy reference to this remote object from the RMI registry running on the

```
import java.rmi.*;

public interface RemoteChannel
     extends Remote
{
    public abstract void send(Object item) throws
        RemoteException;

    public abstract Object receive() throws
        RemoteException;
}
```

Figure D.6 The `RemoteChannel` interface.

server by using the static method `lookup()` in the `Naming` class. RMI provides a URL-based lookup scheme using the form `rmi://host/objectName`, where `host` is the IP name (or address) of the server on which the remote object `objectName` resides. `objectName` is the name of the remote object specified by the server in the `rebind()` method (in this case, `MailBox`.) Once the client has a proxy reference to the remote object, it creates separate producer and consumer threads, passing to each thread a proxy reference to the remote object. The source-code fragments for the producer and consumer threads are shown in Figures D.10 and D.11. Note that the producer and consumer threads invoke the remote methods `send()` and `receive()` as routine method invocations. However, as these methods can throw a `java.rmi.RemoteException`, they

```
import java.util.*;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.*;

public class MessageQueue UnicastRemoteObject
     implements RemoteChannel
{
   private Vector queue;

    public MessageQueue() throws RemoteException {
       queue = new Vector();
    }

    public synchronized void send(Object item)
        throws RemoteException {
       // Figure D.8
    }

    public synchronized Object receive()
        throws RemoteException {
       // Figure D.8
    }

    public static void main(String args[]) {
       try {
          RemoteChannel server = new MessageQueue();

          // Bind this object instance to the name "MessageQueue"
          Naming.rebind("MessageQueue", server);
System.out.println("Server Bound");
       }
       catch(Exception e) {
          System.err.println(e);
       }
    }
}
```

Figure D.7 Implementation of the `RemoteChannel` interface.

must be placed in `try-catch` blocks. Otherwise applications invoking remote methods are designed as if they are invoking ordinary local methods.

D.3.3 Running of the Programs

We now demonstrate the steps necessary to run the example programs. For simplicity, we are assuming that all programs are running on the local host. However, communication is still considered remote, because the client and server programs are each running in their own JVM.

1. *Compile all source files.*
2. *Generate the stub and skeleton.* The tool RMIC is used to generate the stub and skeleton class files, which is done by the user entering

   ```
   rmic MessageQueue
   ```

   on the command line; this create the files `MessageQueue_Skel.class` and `MessageQueue_Stub.class`. (If you are running this example on two different computers, make sure that all the class files—including the stub classes—are available on each computer. It is possible to load classes dynamically using RMI, a topic beyond the scope of this text but covered in texts mentioned in the Bibliographical Notes.)
3. *Start the registry and create the remote object.* To start the registry on UNIX platforms, the user may type

   ```
   rmiregistry &
   ```

   For Windows, the user may type

   ```
   start rmiregistry
   ```

   This command starts the registry with which the remote object will register. Next, create an instance of the remote object with

   ```
   java MessageQueue
   ```

   This remote object will register using the name `MessageQueue`.

```
// This implements a non-blocking send
public synchronized void send(Object item)
     throws RemoteException {
    queue.addElement(item);
}

// This implements a non-blocking receive
public synchronized Object receive()
     throws RemoteException {
    if (queue.size() == 0)
       return null;
    else
       return queue.remove(0);
}
```

Figure D.8 `send()` and `receive()` methods.

```
import java.rmi.*;

public class Factory
{
    public Factory() {
       // remote object
       RemoteChannel mailBox;

RMISecurityManager());

       try {
          mailBox = (RemoteChannel)Naming.lookup
             ("rmi://127.0.0.1/MessageQueue");

          Thread producer = new Thread(new Producer(mailBox));
          Thread consumer = new Thread(new Consumer(mailBox));

          producer.start();
          consumer.start();
       }
       catch (Exception e) {
          System.err.println(e);
       }
    }

    public static void main(String args[]) {
       Factory client = new Factory();
    }
}
```

Figure D.9 The client factory.

4. *Reference the remote object.* The statement

```
java Factory
```

is entered on the command line to start the client factory. This program will get a proxy reference to the remote object `MessageQueue`, and will create the producer and consumer threads, passing to each thread the proxy reference to the remote object. The producer and consumer threads can now invoke the methods `send()` and `receive()` on the remote object.

D.4 Other Aspects of Distributed Communication

Distributed communications is an important and evolving area of computing. Therefore, there are many aspects to distributed communications, and many solutions to the common challenges found there. This section discusses CORBA, an alternative to RMI and how remote objects can register themselves with services such as RMI and CORBA.

```
import java.util.*;

class Producer implements Runnable
{
    private RemoteChannel mbox;

    public Producer(RemoteChannel m) {
       mbox = m;
    }

    public void run() {
       Date message;

       while (true) {
          // nap for awhile
         SleepUtilities.nap();

          // produce an item and enter
          // it into the buffer
          message = new Date();
          System.out.println("Producer produced " +
             message);
          try {
             mbox.send(message);
          } catch (java.rmi.RemoteException re) {
             System.err.println(re);
          }
       }
    }
}
```

Figure D.10 Producer thread.

D.4.1 CORBA

RMI is a technology that allows threads to invoke methods on distributed objects. RMI is native Java technology, however, so it requires all distributed applications to be written in Java. Many existing systems that we might also want to be distributed are written in C, C++, COBOL, or Ada. No communication protocol discussed so far provides a convenient mechanism for these disparite applications to communicate.

The **Common Object Request Broker Architecture** (**CORBA**) is **middleware**—an intermediate software layer—that allows heterogeneous client and server applications to communicate. For example, a C++ program could use CORBA to access a database service written in COBOL. CORBA allows applications written in different languages to communicate using an **interface-definition language** (**IDL**) and an **Object Request Broker** (**ORB**). An IDL allows a distributed object—such as a database—to describe an interface to the services that it provides. The IDL is a generic programming language; it allows a service to describe itself independently of any specific language. To communicate with a server, a client needs to communicate with only the

```
import java.util.*;

class Consumer implements Runnable
{
    private RemoteChannel mbox;

    public Consumer(RemoteChannel m) {
       mbox = m;
    }

    public void run() {
       Date message;

       while (true) {
          // nap for awhile
         SleepUtilities.nap();

          // consume an item from the buffer
          try {
           message = (Date)mbox.receive();
           if (message != null)
              System.out.println("Consumer consumed "
                + message);
          } catch (java.rmi.RemoteException re) {
           System.err.println(re);
          }
       }
    }
}
```

Figure D.11 Consumer thread.

interface specified with the IDL. The server object implements the interface specified by the IDL. The ORB forms the backbone of CORBA, allowing clients to invoke methods on distributed objects and accept return values. There is an ORB on both the client and the server side, and a protocol, the **Internet InterORB Protocol** (**IIOP**), that specifies how the ORBs can communicate.

When a client application requests a reference for a remote object, it is the responsibility of the client ORB to find the remote object in the distributed system. The client ORB is also responsible for routing the remote method calls to the appropriate server, and for accepting results from the server. The server ORB allows the server to register new CORBA objects and is also responsible for accepting requests from client ORBs for invoking methods on the server. The server ORB also passes return values back to the client.

A CORBA system works as follows. Once a client has a reference for a remote object, any method invocations for this object are made through the client-side stub. This stub uses the client's ORB to communicate with the server. When the server ORB receives a client request for invoking a method, it calls the appropriate skeleton, which in turn invokes the implementation of the remote method on the server. Return values are passed back along the same path. This situation is shown in Figure D.12.

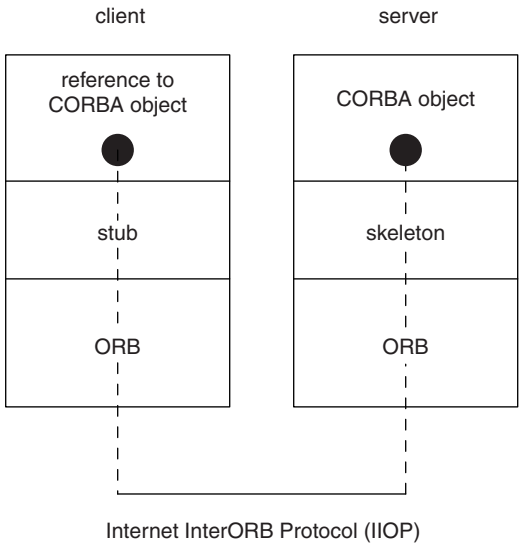

Figure D.12 CORBA model.

In many ways, the ORB behaves much like RMI. However RMI is Java-to-Java technology. CORBA provides interoperability between heterogeneous clients and servers, allowing applications that are written in different languages to communicate.

D.4.2 Object Registration

Both RMI and CORBA are **distributed object systems:** Objects are distributed across a network and applications can communicate with the distributed objects. An important feature of such systems is an **object-registration service,** with which an object registers itself. Applications that wish to use a distributed object obtain a reference to the object through this service. The **rmiregistry** acts as the registration service for RMI. Once the registry is started, a remote object registers itself with a unique name using the `Naming.rebind()` method. Clients use this name when they request a reference to the remote object from the registration server. Using RMI, a client obtains a reference to a remote object using the `Naming.lookup()` method. In a CORBA system, the ORB on the server side is responsible for providing the registration service.

D.5 Web Services

One of the most rapidly expanding and changing computing technologies is **web services**. These services were created to work over the Internet, but in fact are used throughout LANs and WANs, within companies and even within houses. These services include a wide range of functions and features, but their key aspects are that they use a standardized XML-based messaging system, and that they are operating system independent. Most of them are **self-describing** and **self-discovering**. That is, they publish a description of the service they provide, such that other web services can interoperate with them without any

customization or programming. They can also find one another. For example, a portal service could need a stock ticker service and could locate it and integrate it automatically. This section discusses the technologies that compose web services.

At the core of these services is the eXtensible Markup Language (**XML**), which is at once simple and powerful. It is similar to the HTML language in which most web pages are written. HTML has a defined syntax and structure. In contrast, XML is self-describing. A given XML document starts by defining what it contains, and then providing the contents. The resulting flexibility has made XML a communication standard for entities (such as companies) wishing to exchange data with others. Previously, a program at each end would have had to know the details of the data to be exchanged, and would have been specifically coded to exchange that data. Now, an entity can send XML and without prior communication, and the receiver can decipher the format and extract the data.

D.5.1 Web Services Sub-layers

Fundamentally, systems communications is made up of layers, much like a parfait. The physical layer is below the data-link layer, which is below the network layer, and so on up to the application layer. At this top layer applications communicate with each other, and they leave it to the other layers to assure that the communication happens.

The application layer can be further divided into sub-layers, as new services are created for higher-level functionality. Web services are of this model. They are at the application layer, and they depend on the underlying layers to provide a transport. The transport can be over HTTP and SMTP on top of TCP/IP for example.

Within the web-services application layer is the **XML messaging layer**. This layer encodes messages in standard XML format, for receipt by an XML-understanding service. Currently, eXtensible Markup Language - Remote Procedure Call (XML-RPC) and Simple Object Request Protocol (SOAP) implement this sub-layer.

The next sub-layer is **service description**. It provides a public description of the web service. This description is intended for other web services. If they use the same web-services technologies, they will be able to understand the service description and use the service that it describes. The Web Services Description Language (**WSDL**) currently provides this function.

The final sub-layer is **service discovery**, handled by the Universal Description, Discovery, and Integration (**UDDI**) protocol. By using service discover, a web service can find others that provide services it needs. It can then use the service description to understand how to talk to the other services, and the XML messaging layer to execute those other services, remotely. In this manner, a new application, based on the Internet and its composite technologies, can be designed and developed quickly and cleanly, building on services already written and deployed by others.

Several companies have announced and are deploying competing web services frameworks. Microsoft has .Net, IBM has its Web Services, and Sun has Sun Open Net Environment (Sun ONE). Also, a coalition has created Project Liberty to compete with .Net. Fortunately for programmers and web-services

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: test.mydomain.com
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
   <methodName>method-to-call</methodName>
   <params>
   <param>
   <value><i4>my-value</i4></value>
   </param>
   </params>
</methodCall>
```

Figure D.13 An example of a simple XML-RPC call.

providers, all of these superstructures are based on the same core sub-layers and technologies (SOAP, WSDL, and UDDI. The rest of this section explores these technologies.

D.5.2 XML-RPC and SOAP

XML-RPC can be thought of as a subset of SOAP. Therefore, XML-RPC is described first, and then differences between it and SOAP are discussed.

XML-RPC is a method for calling remote procedures via XML. The call is done via an HTTP POST call which is transported over HTTP. The call is in XML format, as is the response. In essence, a remote web server can be asked to execute a procedure, rather than just return a web page. As with standard RPC, XML-RPC can call a specific procedure, pass it parameters, and receive an XML-formatted response. XML-RPC is not a proper replacement for CORBA or other complex communications frameworks, because it has no idea of objects or other complex data types. It can be used to glue services together, or to fairly trivially turn a web server into a web services provider (by XML-RPC enabling its functions).

In Figure D.13 we show an example of a simple XML-RPC call routed to the "RPC2" responder on server "test.mydomain.com". The RPC being called is "method-to-call" and the parameter being passed is the four-byte integer ("i4") "my-value":
The above call might result in the response encapsulating the "Return Value" answer as shown in Figure D.14.

As should be clear from the example, there is flexibility built into the XML-RPC interactions, allowing multiple parameters of various types to be sent and varying responses to be returned.

XML-RPC libraries are available for all of the common programming languages. Further, support is built into Apache and other web servers. Because of its wide-spread support, XML-RPC-based services can quickly be deployed, and can be called easily from almost any program that has web access.

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: test.mydomain.com Frontier/5.1.2-WinNT

<?xml version="1.0"?>
<methodResponse>
   <params>
   <param>
   <value><string>Return Value</string> </value>
   </param>
   </params>
</methodResponse>
```

Figure D.14 The response encapsulating the "Return Value" answer.

SOAP takes XML-RPC one step further, expanding the features and functionality toward a more full-fledged remote procedure mechanism. SOAP is under consideration to become a standard by the W3C organization. Even as a draft standard, it has garnered widespread adoption. SOAP has been implemented by many languages and there are hundreds of SOAP-based web services on the Internet (and many more within companies).

Like XML-RPC, SOAP is XML-based. It has more data types that XML-RPC, and adds fundamental features like compound types, referencing external namespaces (for pre-defined data schemas for example), and fault management. It can be transported over any protocol, but HTTP is the most common one. SOAP has the advantage of being platform, operating system, and language independent, and as such is a very functional way of connecting programs and services together.

D.5.3 WSDL

For web services to be more than the sum of their parts, they need to work together seamlessly. **WSDL** is a specification for describing web services, using XML as the description language. In this way, web services can understand each other via XML descriptions and can communicate via XML messages (such as SOAP calls).

WSDL needs to provide several bits of information about a web service to enable the services use by others. These bits include:

1. the name of the web service,
2. the data types used between the client and the server,
3. details of the messages that could be sent or received by the server,
4. the "portType" that combines multiple messages into a complete communication,

5. the binding of the service (i.e. what protocol to use to call it and how to do so),
6. and the address (i.e. URL) of the service.

This definition of a service is carefully designed to be able to be automatically generated based on the SOAP implementation of the service. In total, an existing web site could have SOAP services layered on top of its current functions, and then have WSDL definitions of those services automatically generated such that other web services understand how to call it.

Like SOAP, WSDL has been submitted to the W3C organization for consideration as a standard. WSDL is being used widely to describe web services which are then called via SOAP.

D.5.4 UDDI

Once the web services are written, and their interfaces are published, there needs to be a mechanism for them to be located. Humans or computers can then search for a web service that best meets their needs and can then make use of that best-fitting service.

UDDI is both a specification for building a distributed directory of WSDL-described services, and an operational registry. The registry allows any web service to be published, and all web services published there to be searched.

The UDDI directory is divided into three categories. The "white pages" list per-business information such as name, description, and contact information. The "yellow pages" lists companies and web services available based on their category and type, industry, product, and so on. The "green pages" has the technical information about the web services listed in the yellow pages. These details include the address of the web service and a pointer to the external specification of the service. UDDI is not limited to SOAP-based services. Other services can be registered there as well, including CORBA and Java RMI services.

Although the idea of UDDI is straight-forward, distributed directory services are anything but that. The UDDI specification covers such complexities as replication, security, schema versioning, and internationalization. Using UDDI is simplified by use of the libraries provided by the many languages that support it.

Humans can search for UDDI-registered web services via the web at http://uddi.microsoft.com and http://www-3.ibm.com/services/uddi. Programatically, web service lookup is available via SOAP calls to the web sites http://uddi.microsoft.com/inquire and http://www-3.ibm.com/services/uddi/inquiryapi.

As a simplistic programming example, consider one web service trying to locate another that provides weather information. The first call is to locate any business providing this service, asking the directory at uddi-org:

```
<find_business generic=''1.0'' xmlns=''urn:uddi-org:api''>
<name>weather</name>
</find_business>
```

This call could return zero or more matches (in XML format). The returned information includes the full service name, a **servicekey** and **businesskey** for

each. These keys are unique identifiers for the business which is providing the service, and the service itself.

Next, our program could get the white pages entry for the business by using the UDDI "get_businessDetail" call and giving it the businesskey. Also, a given returned-service-key could be used to get the service details:

```
<get_serviceDetail generic''1.0'' xmlns=''urn:uddi-org:api''>
<servicekey>returned-service-key</servicekey>
</get_serviceDetail>
```

The information returned from this call is enough for our program to execute a remote procedure call to the web service.

The combination of web service remote procedure calls, service descriptions, and service registry and lookup provides a very powerful platform for arbitrary web-enabled programs to find one another and use each others services. This power has lead to the rapid rise of web services-based solutions and growing momentum for companies to both design new web services applications and web services-enable existing applications.

D.6 Summary

In this chapter, we presented several methods that allow distributed applications to communicate. The first, a socket, is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel. RPCs are another form of distributed communication. An RPC occurs when a process or thread calls a procedure on a remote application. RMI is the Java version of an RPC. RMI allows a thread to invoke a method on a remote object just as it would invoke a method on a local object. The primary distinction between RPCs and RMIs is that the data being passed to a remote procedure in the latter case are in the form of an ordinary data structure. RMI allows objects to be passed in remote method calls. We also discussed CORBA, a technology that allows communication among distributed applications written in different languages. RMI and CORBA are both distributed object systems. An important feature of these systems is an object-registration service that registers distributed objects. Clients contact the registration service when they wish to locate a distributed object.

Web services are similar to RPCs, but are platform, operating system, and language independent. They are XML-based. They include not only allow for remote procedure calls, but also for service description and service discovery.

Exercises

D.1 Write a socket-based *Fortune Teller* server using non-blocking I/O. Your program should create a server that listens to a specified port. When a connection is received by a client, the server should respond with a random fortune chosen from its database of fortunes.

D.2 Write a socket-based multithreaded echo server **without** using non-blocking I/O. Create a new thread to service each incoming connection. Contrast this design to the program shown in Figure D.1.

D.3 Design an HTTP web server using non-blocking I/O.

D.4 Design a chat client-server application using non-blocking I/O.

D.5 Describe why servicing client connections in a thread pool may still limit the scalability of server designs.

D.6 Implement Algorithm 3 from Chapter 6 using RMI.

D.7 This chapter provided a distributed solution to the bounded-buffer problem that used message passing. Another solution to the bounded-buffer problem used Java synchronization; it was shown in Chapter 6. Modify this second solution using RMI, so that the solution works in a distributed environment.

D.8 We solved the readers–writers problem by using Java synchronization in Chapter 6. Modify that solution so that it works in a distributed environment using RMI.

D.9 Implement a distributed solution to the dining-philosophers problem using RMI.

D.10 Explain the differences and similarities between RMI and CORBA.

Bibliographical Notes

Hitchens [2002] presents a thorough discussion of non-blocking I/O in Java. Welsh et al. [2001] and Nian-Min et al. [2002] discuss architectures of high performance Internet servers. Kurose and Ross [2005] and Harold [2000] describe socket programming with an emphasis on Java. Grosso [2002] covers RMI. Farley [1998] focuses on using RMI, and also contains a section describing how to use CORBA with Java. The RMI homepage at [http://www.javasoft.com/products/jdk/rmi] lists up-to-date resources. The OMG homepage [http://www.omg.org] is a good starting point on the Web for information regarding CORBA.