Appendix B
Network Programming

In network programming, we establish a connection between two programs, which may be running on two different machines. The *client/server model* simplifies the development of network software by dividing the design process into client issues and server issues. We can draw a telephone connection analogy, where a network connection is analogous to a case of two persons talking to each other. A caller can dial a callee only if it knows the callee's phone number. This should be advertised somewhere, say in a local telephone directory. Once the caller dials the "well-known" number it can start talking if the callee is listening on this number. In the client/server terminology, the program which listens for the incoming connections is called a *server* and the program which attempts the "dialing" a well-known "phone number" is called a *client*. A server is a process that is waiting to be contacted by a client process so that it can do something for the client.

B.1 Socket APIs

Network programming is done by invoking *socket APIs* (Application Programming Interfaces). These socket API syntax is common across operating systems, although there are slight but important variations. The key abstraction of the socket interface is the *socket*, which can be thought of as a point where a local application process attaches to the network. The socket interface defines operations for creating a socket, attaching the socket to the network, sending/receiving messages through the socket, and closing the socket. Sockets are mainly used over the transport layer protocols, such as TCP and UDP; this overview is limited to TCP.

A socket is defined by a pair of parameters: the host machine's *IP address* and the application's *port number*. A port number distinguishes the programs running on the same host. It is like an extension number for persons sharing the same phone number. Internet addresses for the Internet Protocol version 4 (IPv4) are four-byte (32 bits) unsigned numbers. They are usually written as dotted quad strings, for example, `128.6.68.10`, which corresponds to the binary representation `10000000 00000110 01000100 00001010`. The port numbers are 16-bit unsigned integers, which can take values in the range 0 – 65535. Port numbers 0 – 1024 are *reserved* and can be assigned to a process only by a superuser process. For example, port number 21 is reserved for the FTP server and port number 80 is reserved for the Web server. Thus, a pair

(`128.6.68.10`, 80) defines the socket of a Web server application running on the host machine with the given IP address.

Alphanumeric names are usually assigned to machines to make IP addresses human-friendly. These names are of variable length (potentially rather long) and may not follow a strict format. For example, the above IP address quad `128.6.68.10` corresponds to the host name `eden.rutgers.edu`. The Domain Name System (DNS) is an abstract framework for assigning names to Internet hosts. DNS is implemented via *name servers*, which are special Internet hosts dedicated for performing name-to-address mappings. When a client desires to resolve a host name, it sends a query to a name server which, if the name is valid, and returns back the host's IP address[1]. Here, I will use only the dotted decimal addresses.

In Java we can deal directly with string host names, whereas in C we must perform name resolution by calling the function `gethostbyname()`. In C, even a dotted quad string must be explicitly converted to the 32-bit binary IP address. The relevant data structures are defined in the header file `netinet/in.h` as follows:

C socket address structures (defined in `netinet/in.h`)

```
struct in_addr {
    unsigned long s_addr;          /* Internet address (32 bits) */
};

struct sockaddr_in {
        sa_family_t    sin_family;  /* Internet protocol (AF_INET) */
        in_port_t      sin_port;    /* Address port (16 bits) */
        struct in_addr sin_addr;    /* Internet address (32 bits) */
        char           sin_zero[8]; /* Not used */
};
```

To convert a dotted decimal string to the binary value, we use the function `inet_addr()`:

```
struct sockaddr_in host_addr;
host_addr.sin_addr.s_addr = inet_addr("128.6.68.10");
```

[1] The name resolution process is rather complex, because the contacted name server may not have the given host name in its table, and the interested reader should consult a computer networking book for further details.

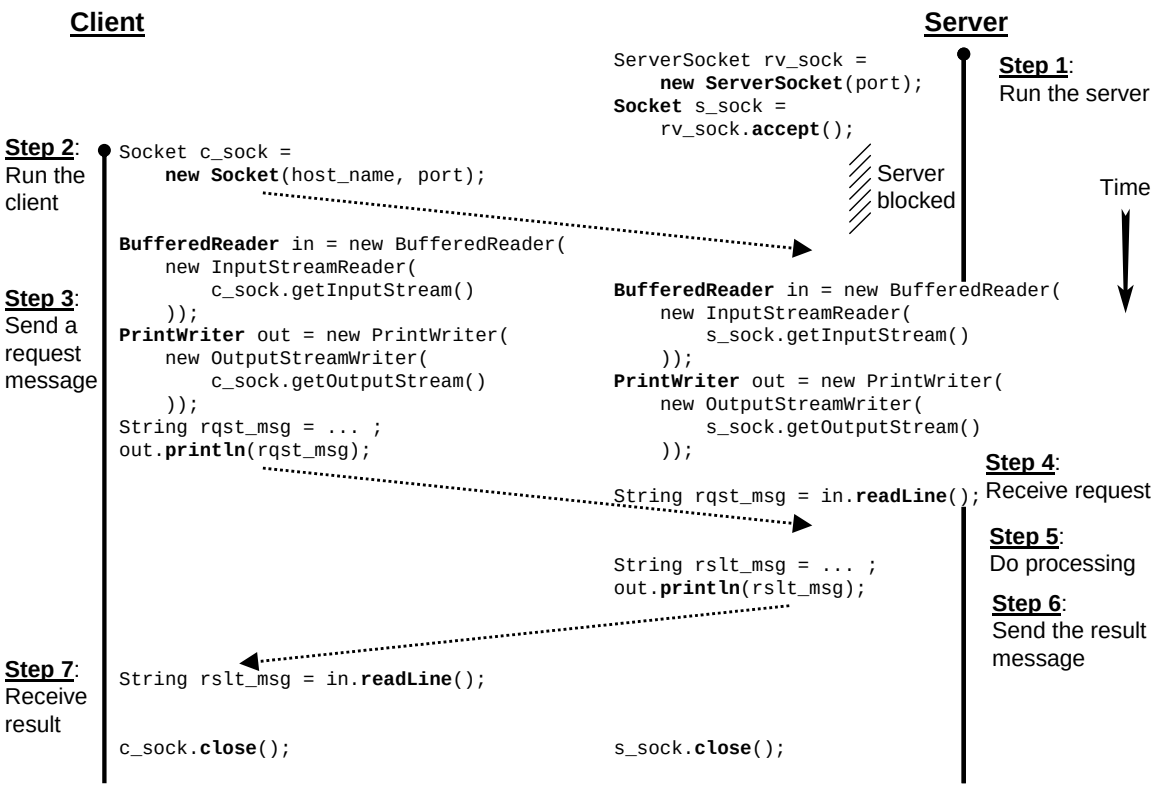

Figure B-1: Summary of network programming in the Java programming language.

Figure B-1 summarizes the socket functions in Java for a basic client-server application. Similarly, Figure B-2 summarizes the socket functions in C. Although this figure may appear simpler than that for Java, this is deceptive because the Java constructs incorporate much more than necessary for this simple example.

The first step is to create a socket, for which in C there is only one function: `socket()`. Java distinguishes two types of sockets that are implemented over the TCP protocol: *server sockets* from *client sockets*. The former are represented by the class `java.net.ServerSocket` and the latter by the class `java.net.Socket`. In addition, there is `java.net.DatagramSocket` for sockets implemented over the UDP protocol. The following example summarizes the Java and C actions for opening a (TCP-based) server socket:

Opening a TCP SERVER socket in Java vs. C ("Passive Open")	
`import java.net.ServerSocket;` `import java.net.Socket;`	`#include <arpa/inet.h>` `#include <sys/socket.h>`
`public static final int` `  PORT_NUM = 4999;`	`#define PORT_NUM 4999`

`ServerSocket rv_sock =`
`  new ServerSocket(PORT_NUM);`
	`int rv_sock, s_sock, cli_addr_len;`
	`struct sockaddr_in serv_addr, cli_addr;`
	`rv_sock = socket(`
	`  PF_INET, SOCK_STREAM, IPPROTO_TCP);`
	`serv_addr.sin_family = AF_INET;`
	`serv_addr.sin_addr.s_addr =`
	`  htonl(INADDR_ANY);`
	`serv_addr.sin_port = htons(PORT_NUM);`
	`bind(rv_sock,`

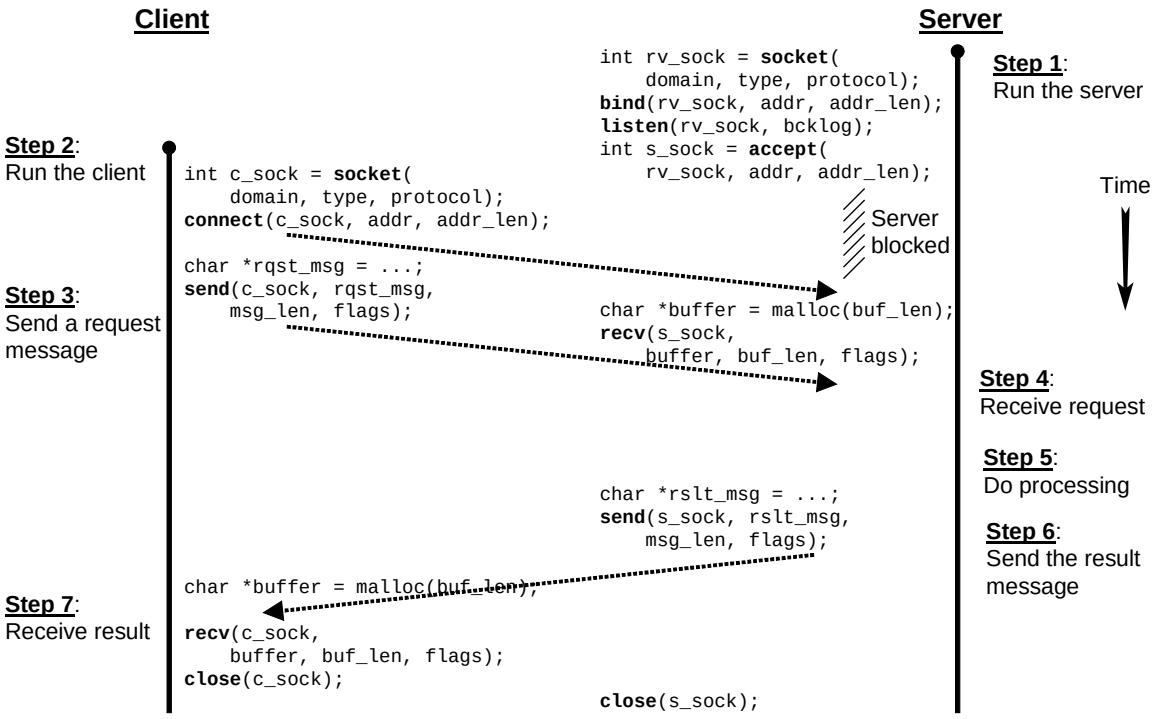

Figure B-2: Summary of network programming in the C programming language.

	`&serv_addr, sizeof(serv_addr));` `listen(rv_sock, 5);`

`Socket s_sock =`
`rv_sock.accept();`
	`cli_addr_len = sizeof(cli_addr);`
	`s_sock = accept(rv_sock,`
	`&cli_addr, &cli_addr_len);`

The above code is simplified, as will be seen in the example below, but it conveys the key points. Notice that in both languages we deal with two different socket descriptors. One is the so-called *well-known* or *rendezvous* socket, denoted by the variable `rv_sock`. This is where the server listens, blocked and inactive in the operation `accept()`, waiting for clients to connect. The other socket, `s_sock`, will be described later. In Java, you simply instantiate a new `ServerSocket` object and call the method `accept()` on it. There are several different constructors for `ServerSocket`, and the reader should check the reference manual for details. In C, things are a bit more complex.

The operation `socket()` takes three arguments, as follows. The first, *domain*, specifies the protocol *family* that will be used. In the above example, I use `PF_INET`, which is what you would use in most scenarios[2]. The second argument, *type*, indicates the semantics of the communication. Above, `SOCK_STREAM` is used to denote a byte stream. An alternative is `SOCK_DGRAM` which stands for a message-oriented service, such as that provided by UDP. The last argument, *protocol*, names the specific protocol that will be used. Above, I state `IPPROTO_TCP` but I could have used `UNSPEC`, for "unspecified," because the combination of `PF_INET` and `SOCK_STREAM` implies TCP. The return value is a *handle* or *descriptor* for the

[2] Note that `PF_INET` and `AF_INET` are often confused, but luckily both have the same numeric value (2).

newly created socket. This is an identifier by which we can refer to the socket in the future. As can be seen, it is given as an argument to subsequent operations on this socket.

On a server machine, the application process performs a *passive open*—the server says that it is prepared to accept connections, but it does not actually establish a connection. The server's address and port number should be known in advance and, when the server program is run, it will ask the operating system to associate an (address, port number) pair with it, which is accomplished by the operation `bind()`. This resembles a "server person" requesting a phone company to assign to him/her a particular phone number. The phone company would either comply or deny if the number is already taken. The operation `listen()` then sets the capacity of the queue holding new connection attempts that are waiting to establish a connection. The server completes passive open by calling `accept()`, which blocks and waits for client calls.

When a client connects, `accept()` returns a new socket descriptor, `s_sock`. This is the actual socket that is used in client/server exchanges. The well-known socket, `rv_sock`, is reserved as a meeting place for associating server with clients. Notice that `accept()` also gives back the clients' address in `struct sockaddr_in cli_addr`. This is useful if server wants to decide whether or not it wants to talk to this client (for security reasons). This is optional and you can pass `NULL` for the last two arguments (see the server C code below).

The reader should also notice that in C data types may be represented using different byte order (most-significant-byte-first, vs. least-significant-byte-first) on different computer architectures (e.g., UNIX vs. Windows). Therefore the auxiliary routines `htons()`/`ntohs()` and `htonl()`/`ntohl()` should be used to convert 16- and 32-bit quantities, respectively, between network byte order and host byte order. Because Java is platform-independent, it performs these functions automatically.

Client application process, which is running on the client machine, performs an *active open*—it proactively establishes connection to the server by invoking the `connect()` operation:

Opening a TCP CLIENT socket in Java vs. C ("Active Open")

```
import java.net.Socket;
```

```
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netdb.h>
```

```
public static final String HOST =
"eden.rutgers.edu";
public static final int PORT_NUM =
4999;
```

```
#define HOST "eden.rutgers.edu"
#define PORT_NUM 4999
```

```
Socket c_sock =
  new Socket(HOST, PORT_NUM);
```

```
int c_sock;
struct hostent *serverIP;
struct sockaddr_in serv_addr;
serverIP =
    gethostbyname(HOST);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr =
// ... copy from: serverIP->h_addr ...
serv_addr.sin_port =
    htons(port_num);
c_sock = connect(c_sock,
    (struct sockaddr *) &serv_addr,
    sizeof(serv_addr));
```

Notice that, whereas a server listens for clients on a well-known port, a client typically does not care which port it uses for itself. Recall that, when you call a friend, you should know his/her phone number to dial it, but you need not know your number.

B.2 Example Java Client/Server Application

The following client/server application uses TCP protocol for communication. It accepts a single line of text input at the client side and transmits it to the server, which prints it at the output. It is a *single-shot connection*, so the server closes the connection after every message. To deliver a new message, the client must be run anew. Notice that this is a *sequential server*, which serves clients one-by-one. When a particular client is served, any other client attempting to connect will be placed in a waiting queue. To implement a *concurrent server*, which can serve multiple clients in parallel, you should use threads (see Section 4.3). The reader should consult Figure B-1 as a roadmap to the following code.

Listing B-1: A basic SERVER application in Java

```
 1 import java.io.BufferedReader;
 2 import java.io.IOException;
 3 import java.io.InputStreamReader;
 4 import java.io.OutputStreamWriter;
 5 import java.io.PrintWriter;
 6 import java.net.ServerSocket;
 7 import java.net.Socket;
 8
 9 public class BasicServer {
10   public static void main(String[] args) {
11     if (args.length != 1) { // Test for correct num. of arguments
12         System.err.println( "ERROR server port number not given");
13         System.exit(1);
14     }
15     int port_num = Integer.parseInt(args[0]);
16     ServerSocket rv_sock = null;
17     try {
18         new ServerSocket(port_num);
19     } catch (IOException ex) { ex.printStackTrace(); }
20
21     while (true) { // run forever, waiting for clients to connect
22         System.out.println("\nWaiting for client to connect...");
23         try {
24             Socket s_sock = rv_sock.accept();
25             BufferedReader in = new BufferedReader(
26                 new InputStreamReader(s_sock.getInputStream())
27             );
28             PrintWriter out = new PrintWriter(
29                 new OutputStreamWriter(s_sock.getOutputStream()),
30                 true);
31             System.out.println(
```

```
32                  "Client's message: " + in.readLine());
33              out.println("I got your message");
34              s_sock.close();
35          } catch (IOException ex) { ex.printStackTrace(); }
36      }
37    }
39  }
```

The code description is as follows:

Lines 1–7: make available the relevant class files.

Lines 9–14: define the server class with only one method, `main()`. The program accepts a single argument, the server's port number (1024 – 65535, for non-reserved ports).

Line 15: convert the port number, input as a string, to an integer number.

Lines 16–19: create the well-known server socket. According to the javadoc of `ServerSocket`, the default value for the backlog queue length for incoming connections is set to 50. There is a constructor which allows you to set different backlog size.

Line 24: the server blocks and waits indefinitely until a client makes connection at which time a `Socket` object is returned.

Lines 25–27: set up the input stream for reading client's requests. The actual TCP stream, obtained from the `Socket` object by calling `getInputStream()` generates a stream of binary data from the socket. This can be decoded and displayed in a GUI interface. Because our simple application deals exclusively with text data, we wrap a `BufferedReader` object around the input stream, in order to obtain buffered, character-oriented output.

Lines 28–30: set up the output stream for writing server's responses. Similar to the input stream, we wrap a `PrintWriter` object around the binary stream object returned by `getOutputStream()`. Supplying the `PrintWriter` constructor with a second argument of `true` causes the output buffer to be flushed for every `println()` call, to expedite the response delivery to the client.

Lines 31–32: receive the client's message by calling `readLine()` on the input stream.

Line 33: sends acknowledgement to the client by calling `println()` on the output stream.

Line 34: closes the connection after a single exchange of messages. Notice that the well-known server socket `rv_sock` *remains open*, waiting for new clients to connect.

The following is the client code, which sends a single message to the server and dies.

Listing B-2: A basic CLIENT application in Java

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4  import java.io.OutputStreamWriter;
5  import java.io.PrintWriter;
6  import java.net.Socket;
7
8  public class BasicClient {
```

```
 9   public static void main(String[] args) {
10     if (args.length != 2) { // Test for correct num. of arguments
11         System.err.println(
12             "ERROR server host name AND port number not given");
13         System.exit(1);
14     }
15     int port_num = Integer.parseInt(args[1]);
16
17     try {
18         Socket c_sock = new Socket(args[0], port_num);
19         BufferedReader in = new BufferedReader(
20             new InputStreamReader(c_sock.getInputStream())
21         );
22         PrintWriter out = new PrintWriter(
23             new OutputStreamWriter(c_sock.getOutputStream()),
24             true);
25         BufferedReader userEntry = new BufferedReader(
26             new InputStreamReader(System.in)
27         );
28         System.out.print("User, enter your message: ");
29         out.println(userEntry.readLine());
30         System.out.println("Server says: " + in.readLine());
31         c_sock.close();
32     } catch (IOException ex) { ex.printStackTrace(); }
33     System.exit(0);
34   }
35 }
```

The code description is as follows:

Lines 1–6: make available the relevant class files.

Lines 9–14: accept two arguments, the server host name and its port number.

Line 15: convert the port number, input as a string, to an integer number.

Line 18: simultaneously opens the client's socket and *connects* it to the server.

Lines 19–21: create a character-oriented input socket stream to read server's responses. Equivalent to the server's code lines 25–27.

Lines 22–24: create a character-oriented output socket stream to write request messages. Equivalent to the server's code lines 28–30.

Lines 25–27: create a character-oriented input stream to read user's keyboard input from the standard input stream `System.in`.

Line 29: sends request message to the server by calling `println()` on the output stream.

Line 30: receives and displays the server's response by `readLine()` on the input stream.

Line 31: closes the connection after a single exchange of messages.

Line 33: client program dies.

B.3 Example Client/Server Application in C

Here I present the above Java application, now re-written in C. Recall that the TCP protocol is used for communication; a UDP-based application would look differently. The reader should consult Figure B-2 as a roadmap to the following code.

Listing B-3: A basic SERVER application in C on Unix/Linux

```
 1 #include <stdio.h>      /* perror(), fprintf(), sprintf() */
 2 #include <stdlib.h>     /* for atoi() */
 3 #include <string.h>     /* for memset() */
 4 #include <sys/socket.h> /* socket(), bind(), listen(), accept(),
 5                            recv(), send(), htonl(), htons() */
 6 #include <arpa/inet.h>  /* for sockaddr_in */
 7 #include <unistd.h>     /* for close() */
 8
 9 #define MAXPENDING 5    /* Max outstanding connection requests */
10 #define RCVBUFSIZE 256  /* Size of receive buffer */
11 #define ERR_EXIT(msg) { perror(msg); exit(1); }
12
13 int main(int argc, char *argv[]) {
14     int rv_sock, s_sock, port_num, msg_len;
15     char buffer[RCVBUFSIZE];
16     struct sockaddr_in serv_addr;
17
18     if (argc != 2) {  /* Test for correct number of arguments */
19         char msg[64];  memset((char *) &msg, 0, 64);
20         sprintf(msg, "Usage: %s server_port\n", argv[0]);
21         ERR_EXIT(msg);
22     }
23
24     rv_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
25     if (rv_sock < 0) ERR_EXIT("ERROR opening socket");
26     memset((char *) &serv_addr, 0, sizeof(serv_addr));
27     port_num = atoi(argv[1]);  /* First arg: server port num. */
28     serv_addr.sin_family = AF_INET;
29     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
30     serv_addr.sin_port = htons(port_num);
31     if (bind(rv_sock,
32         (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
33         ERR_EXIT("ERROR on binding");
34     if (listen(rv_sock, MAXPENDING) < 0)
35         ERR_EXIT("ERROR on listen");
36
37     while ( 1 ) {                     /* Server runs forever */
38         fprintf(stdout, "\nWaiting for client to connect...\n");
39         s_sock = accept(rv_sock, NULL, NULL);
40         if (s_sock < 0) ERR_EXIT("ERROR on accept new client");
41         memset(buffer, 0, RCVBUFSIZE);
42         msg_len = recv(s_sock, buffer, RCVBUFSIZE - 1, 0);
43         if (msg_len < 0)
44             ERR_EXIT("ERROR reading from socket");
45         fprintf(stdout, "Client's message: %s\n", buffer);
```

```
46          msg_len = send(s_sock, "I got your message", 18, 0);
47          if (msg_len < 0) ERR_EXIT("ERROR writing to socket");
48          close(s_sock);
49      }
50      /* NOT REACHED, because the server runs forever */
51  }
```

The code description is as follows:

Lines 1–7: import the relevant header files.

Lines 9–10: define the relevant constants.

Line 11: defines an inline function to print error messages and exit the program.

Line 13: start of the program.

Line 14: declares the variables for two socket descriptors, well-known (`rv_sock`) and client-specific (`s_sock`), as well as server's port number (`port_num`) and message length, in bytes (`msg_len`) that will be used below.

Line 15:

Line 39: accepts new client connections and returns the socket to be used for message exchanges with the client. Notice that `NULL` is passed for the last two arguments, because this server is not interested in the client's address.

Line 42: receive up to the buffer size (minus 1 to leave space for a null terminator) bytes from the sender. The input parameters are: the active socket descriptor, a char buffer to hold the received message, the size of the receive buffer (in bytes), and any flags to use. If no data has arrived, `recv()` blocks and waits until some arrives. If more data has arrived than the receive buffer can hold, `recv()` removes only as much as fits into the buffer.
NOTE: This simplified implementation may not be adequate for general cases, because `recv()` may return a partial message. Remember that TCP connection provides an illusion of a virtually infinite stream of bytes, which is randomly sliced into packets and transmitted. The TCP receiver may call the application immediately upon receiving a packet.
Suppose a sender sends M bytes using `send( · , · , M, · )` and a receiver calls `recv( · , · , N, · )`, where $M \leq N$. Then, the actual number of bytes K returned by `recv()` may be less than the number sent, i.e., $K \leq M$.
A simple solution for getting complete messages is for sender to preface all messages with a "header" indicating the message length. Then, the receiver finds the message length from the header and may need to call `recv()` repeatedly, while keeping a tally of received fragments, until the complete message is read.

Line 46: sends acknowledgement back to the client. The return value indicates the number of bytes successfully sent. A return value of −1 indicates locally detected errors only (not network ones).

Line 48: closes the connection after a single exchange of messages. Notice that the well-known server socket `rv_sock` *remains open*, waiting for new clients to connect.

Notice that this is a *sequential server*, which serves clients one-by-one. When a particular client is served, any other client attempting to connect will be placed in a waiting queue. The capacity of

the queue is limited by `MAXPENDING` and declared by invoking `listen()`. To implement a *concurrent server*, which can serve multiple clients in parallel, you should use threads (see Section 4.3).

Listing B-4: A basic CLIENT application in C on Unix/Linux

```
 1 #include <stdio.h>      /* for perror(), fprintf(), sprintf() */
 2 #include <stdlib.h>     /* for atoi() */
 3 #include <string.h>     /* for memset(), memcpy(), strlen() */
 4 #include <sys/socket.h> /* for sockaddr, socket(), connect(),
 5                            recv(), send(), htonl(), htons() */
 6 #include <arpa/inet.h>  /* for sockaddr_in */
 7 #include <netdb.h>      /* for hostent, gethostbyname() */
 8 #include <unistd.h>     /* for close() */
 9
10 #define RCVBUFSIZE 256 /* Size of receive buffer */
11 #define ERR_EXIT(msg) { perror(msg); exit(1); }
12
13 int main(int argc, char *argv[]) {
14     int c_sock, port_num, msg_len;
15     struct sockaddr_in serv_addr;
16     struct hostent *serverIP;
17     char buffer[RCVBUFSIZE];
18
19     if (argc != 3) {   /* Test for correct number of arguments */
20         char msg[64];  memset((char *) &msg, 0, 64);  /* erase */
21         sprintf(msg, "Usage: %s serv_name serv_port\n", argv[0]);
22         ERR_EXIT(msg);
23     }
24
25     serverIP = gethostbyname(argv[1]); /* 1st arg: server name */
26     if (serverIP == NULL)
27         ERR_EXIT("ERROR, server host name unknown");
28     port_num = atoi(argv[2]);  /* Second arg: server port num. */
29     c_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
30     if (c_sock < 0)  ERR_EXIT("ERROR opening socket");
31     memset((char *) &serv_addr, 0, sizeof(serv_addr));
32     serv_addr.sin_family = AF_INET;
33     memcpy((char *) &serv_addr.sin_addr.s_addr,
34         (char *) &(serverIP->h_addr), serverIP->h_length);
35     serv_addr.sin_port = htons(port_num);
36     if (connect(c_sock,
37         (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
38         ERR_EXIT("ERROR connecting");
39
40     fprintf(stdout, "User, enter your message: ");
41     memset(buffer, 0, RECVBUFSIZE);  /* erase */
42     fgets(buffer, RECVBUFSIZE, stdin); /* read input */
43     msg_len = send(c_sock, buffer, strlen(buffer), 0);
44     if (msg_len < 0) ERR_EXIT("ERROR writing to socket");
45     memset(buffer, 0, RECVBUFSIZE);
46     msg_len = recv(c_sock, buffer, RECVBUFSIZE - 1, 0);
47     if (msg_len < 0) ERR_EXIT("ERROR reading from socket");
48     fprintf(stdout, "Server says: %s\n", buffer);
50     close(c_sock);
51     exit(0);
```

```
52 }
```

The code description is as follows:

Lines 1–8: import the relevant header files.

Line 43: writes the user's message to the socket; equivalent to line 46 of server code.

Line 46: reads the server's response from the socket; equivalent to line 42 of server code.

I tested the above programs on Linux 2.6.14-1.1637_FC4 (Fedora Core 4) with GNU C compiler gcc version 4.0.1 (Red Hat 4.0.1-5), as follows:

Step 1: Compile the server using the following command line:

```
% gcc -o server server.c
```

On Sun Microsystems's Solaris, I had to use:

```
% gcc -g -I/usr/include/ -lsocket -o server server.c
```

Step 2: Compile the client using the following command line:

```
% gcc -o client client.c
```

On Sun Microsystems's Solaris, I had to use:

```
% gcc -g -I/usr/include/ -lsocket -lnsl -o client client.c
```

Step 3: Run the server on the machine called caipclassic.rutgers.edu, with server port 5100:

```
% ./server 5100
```

Step 4: Run the client

```
% ./client caipclassic.rutgers.edu 5100
```

The server is silently running, while the client will prompt you for a message to type in. Once you hit the **Enter** key, the message will be sent to the server, the server will acknowledge the receipt, and the client will print the acknowledgment and die. Notice that the server will continue running, waiting for new clients to connect. Kill the server process by pressing simultaneously the keys **Ctrl** and **c**.

B.4 Windows Socket Programming

Finally, I include also the server version for Microsoft Windows:

Listing B-5: A basic SERVER application in C on Microsoft Windows

```
1 #include <stdio.h>
2 #include <winsock2.h>     /* for all WinSock functions */
3
4 #define MAXPENDING 5    /* Max outstanding connection requests */
5 #define RCVBUFSIZE 256  /* Size of receive buffer */
6 #define ERR_EXIT { \
```

```
 7      fprintf(stderr, "ERROR: %ld\n", WSAGetLastError()); \
 8      WSACleanup();  return 0; }
 9
10  int main(int argc, char *argv[]) {
11      WSADATA wsaData;
12      SOCKET rv_sock, s_sock;
13      int port_num, msg_len;
14      char buffer[RCVBUFSIZE];
15      struct sockaddr_in serv_addr;
16
17      if (argc != 2) {  /* Test for correct number of arguments */
18          fprintf(stdout, "Usage: %s server_port\n", argv[0]);
19          return 0;
20      }
21      WSAStartup(MAKEWORD(2,2), &wsaData);/* Initialize Winsock */
22
23      rv_sock = WSASocket(PF_INET, SOCK_STREAM, IPPROTO_TCP,
24          NULL, 0, WSA_FLAG_OVERLAPPED);
25      if (rv_sock == INVALID_SOCKET) ERR_EXIT;
26      memset((char *) &serv_addr, 0, sizeof(serv_addr));
27      port_num = atoi(argv[1]);  /* First arg: server port num. */
28      serv_addr.sin_family = AF_INET;
29      serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
30      serv_addr.sin_port = htons(port_num);
31      if (bind(rv_sock, (SOCKADDR*) &serv_addr,
32          sizeof(serv_addr)) == SOCKET_ERROR) {
33          closesocket(rv_sock);
34          ERR_EXIT;
35      }
36      if (listen(rv_sock, MAXPENDING) == SOCKET_ERROR) {
37          closesocket(rv_sock);
38          ERR_EXIT;
39      }
40
41      while ( 1 ) {                       /* Server runs forever */
42          fprintf(stdout, "\nWaiting for client to connect...\n");
43          if (s_sock = accept(rv_sock, NULL, NULL)
44              == INVALID_SOCKET) ERR_EXIT;
45          memset(buffer, 0, RCVBUFSIZE);
46          msg_len = recv(s_sock, buffer, RCVBUFSIZE - 1, 0);
47          if (msg_len == SOCKET_ERROR) ERR_EXIT;
48          fprintf(stdout, "Client's message: %s\n", buffer);
49          msg_len = send(s_sock, "I got your message", 18, 0);
50          if (msg_len == SOCKET_ERROR) ERR_EXIT;
51          closesocket(s_sock);
52      }
53      return 0;
54  }
```

The reader should seek further details on Windows sockets here:
http://msdn.microsoft.com/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp.

B.5 Bibliographical Notes

[Stevens *et al*., 2004] remains the most authoritative guide to network programming. A good quick guides are [Donahoo & Calvert, 2001] for network programming in the C programming language and [Calvert & Donahoo, 2002] for network programming in the Java programming language.

There are available many online tutorials for socket programming. Java tutorials include

- Sun Microsystems, Inc., http://java.sun.com/docs/books/tutorial/networking/sockets/index.html
- Qusay H. Mahmoud, "Sockets programming in Java: A tutorial," http://www.javaworld.com/jw-12-1996/jw-12-sockets.html

and C tutorials include

- Sockets Tutorial, http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html
- Peter Burden, "Sockets Programming," http://www.scit.wlv.ac.uk/~jphb/comms/sockets.html
- Beej's Guide to Network Programming – Using Internet Sockets, http://beej.us/guide/bgnet/ also at http://mia.ece.uic.edu/~papers/WWW/socketsProgramming/html/index.html
- Microsoft Windows Sockets "Getting Started With Winsock," http://msdn.microsoft.com/library/en-us/winsock/winsock/getting_started_with_winsock.asp

Davin Milun maintains a collection of UNIX programming links at http://www.cse.buffalo.edu/~milun/unix.programming.html. UNIX sockets library manuals can be found at many websites, for example here: http://www.opengroup.org/onlinepubs/009695399/mindex.html. Information about Windows Sockets for Microsoft Windows can be found here: http://msdn.microsoft.com/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp.