

## Sockets over C/C++, Perl and Java

### Abstract

This research paper will examine the construction and use of sockets for interprocess communications over three different programming languages: C/C++, Perl and Java. The use of sockets to accomplish interprocess communications over networks is not new. However, for programmers who have never written any programs that are capable of interprocess communications, sockets are new, and can be daunting to understand. Not only do programmers need to have at least a rudimentary knowledge of networks, but programmers are also faced with the requirement that they design and construct not one, but two programs (the classic client and server) that will cooperate together in a managed and predictable exchange of data.

Brown and McDonald describe how “students frequently misuse the sockets API or have difficulty understanding where run-time and logic errors in their code occur.” Brown and McDonald conclude “Surprisingly, there are no effective tools or general techniques for debugging students’ socket programs.” Brown and McDonald are reporting a common problem, as network communication is not well understood or well represented in many of the programming texts that are available. Brown and McDonald have developed the *Socketvisualizer* “that enables students to view calls to, and assist the debugging of, functions incorporating the Berkeley sockets API.” Tools like the *Socketvisualizer* or other debugging tools such as adb or gdb (Gnu Debugger) do help,

Sockets over C/C++, Perl and Java

but they are themselves can be difficult to understand. Additional tools that can aid in the development and support of interprocess communications include Wireshark and TCPSpy.

Another simple alternative strategy is to provide programmers with working sample programs that use sockets for interprocess communications. From the sample programs, programmers can further develop their own work. It is particularly useful for students who are learning different languages to be able to see how sockets work in different languages, notably C/C++, Perl and Java.

In this research paper we will construct a pair of simple client / server programs to illustrate and test the use of sockets in C/C++, Perl and Java. Our server will listen perpetually for a client to issue a connection request. When a client makes a connection, the client will send a simple text message to our server, and our server will then echo that same message back to the client. The client will then terminate, while our server will continue to listen for the next client to request a connection. All programs were developed over the Fedora 7 operating system running on an Intel Pentium D processor, using the 4.1.2 gcc C compiler, Perl v.5.8.8 runtime, and Java 2 1.5.0 compiler and runtime. Happily, regardless of the language in which they are written, any of our clients will connect with any of our servers! And, our Perl scripts and Java class files will port and run without problem over to Windows. As we should expect, our Java class files do not have to be recompiled before they will run on the Windows Java Virtual Machine. Our Perl scripts will also run on Windows over the Perl interpreter from ActiveState

without including any new logic for Windows. On Windows we are running a Java 2 v1.4.2\_03 runtime and Active Perl v5.8.8. Active Perl is available for download from ActiveState at <http://www.activestate.com>.

## Introduction

The reader can read the **Basics** and the **Anatomy of a C/C++ Socket** sections to gain familiarity with the construction of a socket. However, if the reader already understands socket construction, then they can skip ahead to the individual sections about C, Perl and Java. Sections may be read in any order. Finally, the source code for all programs is presented in the **Appendix**.

We start by recognizing what sockets are and what they make possible. Sockets are interprocess communication connections between two programs. The two programs are usually working together in a classic client / server relationship. The two programs are usually deployed on two different machines. In turn, the two machines are frequently located on different networks. However, the two machines can be located on the same network, and the two programs can even be located on the same machine. The de-facto communications transport network mechanism is TCP/IP, most frequently over Ethernet. Sockets make it possible for two programs to send and receive data from each other using simple write and read commands. Historically, the first sockets Applications Programmer Interface (API) was shipped with the Berkeley 4.2 BSD system, which was released in 1983. The original development history of the sockets API was based on the

## Sockets over C/C++, Perl and Java

C programming language, and was inevitably supported by some flavor of the Unix operating system. Today, sockets are ubiquitous on all popular operating systems, and are supported by several programming languages, including C/C++, Perl and Java.

Toll gives us a good overall description of the socket communications process. “Simply stated sockets are a mechanism by which messages may be sent between processes on the same or different machines. If the processes are on the same machine, the sockets may be used as pipes. Internet sockets allow communication between processes running on different machines. The system calls are the same as file I/O. A typical approach to socket programming is to create a process that opens a Server socket port and listens for another process to attempt connection. A client can open a socket with the same port number as the server socket, requesting connection to the service. When the server hears the request a connection is established. Communication can now proceed with read() and write().”

## Basics

The basic C/C++ programming language socket system calls are *socket*, *bind*, *listen*, *accept*, *connect* and *close*. Let’s see how we create a socket on a server with C/C++:

- Create a new server socket with *socket*
- Assign the socket to a port number and an IP address with *bind*
- Listen to the port with *listen*
- Accept an incoming connection request from a client with *accept*
- Close the server socket with *close*

## Sockets over C/C++, Perl and Java

A client needs also needs a socket to connect to the server:

- Create a new client socket with *socket*
- Connect to the server socket port over an IP address with *connect*
- Close the client socket with *close*

Now that our client socket and server socket are connected, our client and server can send and receive data from each other with simple write and read statements. Table 1 shows the relationships of the system calls that create our client and server sockets.

Call	What it does	Client	Server
socket	Create a new local socket	✓	✓
bind	Assign the socket to a port number and an IP address		✓
listen	Notify the operating system that the socket is willing to receive connection requests		✓
accept	Receive a connection request from a client		✓
connect	Establish a connection with a server	✓	
close	Close the local socket	✓	✓

Table 1.

## Anatomy of a C/C++ Socket

Let's break down the construction of a socket and see how it is built. We will describe sockets as they are built using C/C++ system calls in Unix/Linux systems. First, we need to understand a little bit about network communications. Abdel-Wahab starts us off nicely. "There are several *domains* of communication, the most frequently used ones are the *internet* domain (*INET*) and the UNIX domain. The *INET* domain employs the DARPA standard protocols TCP/IP and UDP/IP. The UNIX domain is basically used for efficient communication between unrelated processes residing on the same machine. The *INET* domain allows processes running on different machines (as well as on the same machine) to communicate. In most cases the *INET* domain is preferable to the UNIX domain, since applications written in the *INET* domain may run on any configuration of machines." The selection of a domain is the first decision that we must make about how we will configure our socket.

Second, we have to decide on a method of communication. Abdel-Wahab describes "In each domain, there are two methods of communication: *stream* and *datagram*. In the stream method the two communicating processes have to establish a connection or *virtual circuit* before they can exchange messages. This style of communication provides a bidirectional, reliable, sequenced and unduplicated flow of data. On the other hand, datagram communication does not require any connection between the two processes, as each message is addressed individually, but there is no guarantee regarding delivery, sequencing, or duplication of messages." The network literate reader will recognize that

*stream* communication is widely known as Transport Control Protocol (TCP) and *datagram* communication is widely known as User Datagram Protocol (UDP).

## **The Server Socket**

We start the socket communications process by creating a socket on a server with the *socket* system call. When we create our server socket, we will select the *INET* domain and *stream* method of communication. This will create a server socket that is suitable for standard TCP/IP communications over a network.

Now that our server socket exists, we have to *bind* our socket to something that a client can communicate with. The binding process requires us to identify two key components in all client/server communications: a port number and an Internet Protocol (IP) address. The port and IP address will be combined together to form a logical name, and we will use the *bind* system call to bind our server socket to this logical name so that our client can locate us on a network.

With our newly bound socket, we need to *listen* for incoming connection requests from a client. When we execute the *listen* system call for our bound socket, we notify our operating system that we are ready to receive incoming connections, and we also specify how many client connection requests we will allow to be queued up and waiting for us to accept their connection request. Frequently, a default value of five (5) connection requests will be allowed to wait in the connection request queue.

Finally, our server socket will *accept* an incoming connection request from a client. The *accept* system call is different than the previous *socket*, *bind* and *listen* system calls, because *accept* is a *blocking* call. When we are *blocking*, we will not return from the *accept* system call until a client actually requests a connection. If a client never requests a connection, then we may never return from *accept*. When we do finally return from *accept*, we are able to receive and send data with our client using simple read and write system calls. When we finish our connection with our client, we can then *close* our server socket permanently.

## **The Client Socket**

The process of socket construction on the client has fewer system calls than the process on the server. Like the server, we start by creating a client socket with the *socket* system call. As we did with our server socket, we will select the *INET* domain and *stream* method of communication for our client.

Our client will then connect to our server with the *connect* system call. Our client will need to identify our server by IP address and port number, and include those values in the *connect* system call. Once connected, our client will be able to receive and send data to our server using simple read and write system calls. When we finish our connection with our server, we can then *close* our client socket permanently.

## C Socket Construction

We will start by creating a socket environment on the server, and then follow up with a matching socket on the client.

### C *Server Socket Construction*

The first thing to do is to include some standard header files to support our socket, and we also specifically need to include the `socket.h` and the `in.h` header files. The `socket` system call will create a socket in the Internet Domain for a TCP stream. The `socket` call returns a simple integer, which we will subsequently use as a file descriptor.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int    server_sock_fd;    /* server socket file descriptor */
int    client_sock_fd;   /* client socket file descriptor */
int    port_num;         /* port number */
int    client_len;       /* length of client address */
struct sockaddr_in  serv_addr; /* server address */
struct sockaddr_in  client_addr; /* client address */
int ctr;                /* Simple counter */
char buffer[256];       /* Simple buffer */

server_sock_fd = socket(AF_INET, SOCK_STREAM, 0);
```

Next we will initialize the logical name and bind to that name. We `bind` within the Internet Domain, and we will accept a connection request from a client at any IP address.

We will bind to the port number that is defined in `port_num`.

```
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
```

```
serv_addr.sin_port = htons(port_num);  
bind(server_sock_fd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

We **listen** on our socket file descriptor, and we will allow up to 5 connection requests in queue.

```
listen(server_sock_fd,5);
```

We are ready to **accept** a connection request from a client. We call **accept**, where we will block while waiting for a client to connect to us. When a client does connect to us, **accept** will return a new client\_sock\_fd file descriptor from which we will read any data that the client sends to us.

```
client_sock_fd = accept(server_sock_fd, (struct sockaddr *) &client_addr,  
&client_len);
```

Now, we can **read** up to 255 bytes of client data over the client\_sock\_fd, which we will store in our buffer. The **read** system call will return the number of bytes actually read in the ctr variable.

```
ctr = read(client_sock_fd, buffer, 255);
```

## **C Client Socket Construction**

As with our server, we start by including the required header files. Client **socket** creation is identical to socket creation on our server. The socket will be in the Internet Domain, and will conduct TCP stream traffic. The **socket** system call returns sockfd, an integer file descriptor.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int    sockfd;        /* socket file descriptor */
int    portno;        /* port number */
int    ctr;           /* simple counter */
char   buffer[256];   /* simple buffer */

struct sockaddr_in  server_addr; /* server address info */
struct hostent      *server;     /* server IP address */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Prior to issuing our connection request to the server, we will initialize several variables with the server's IP address and port number, and we will identify that this server is also in the Internet Domain.

```
bzero((char *) &server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
      (char *)&server_addr.sin_addr.s_addr,
      server->h_length);
server_addr.sin_port = htons(portno);
```

There is no need to bind, listen or accept on the client side. Instead, we just **connect** to the server.

```
connect(sockfd, &server_addr, sizeof(server_addr)) ;
```

Once connected, our client can now write data directly to the server over the sockfd file descriptor. The write system call will return the number of bytes actually written in the ctr variable.

```
ctr = write(sockfd, buffer, strlen(buffer));
```

## Perl Socket Construction

The Perl environment is deceptively simple. Because Perl is a hybrid language with significant data abstraction, there is relative little work that is needed to create a Perl socket. There is a tightly coupled relationship between Perl and C/C++, as most Perl packages are actually written in C. The reader may recognize some C like logic conventions in the Perl statements that are presented here. One difference between Perl and C is that in Perl our socket is treated as a file handle instead of a file descriptor. In Perl, we have the same *socket*, *bind*, *listen* and *accept* statements that we do in C/C++. For example, in his writing about Internet Servers in Perl, Mull describes the socket creation process as “the syntax for this function is:

```
socket SOCKET, DOMAIN, TYPE, PROTOCOL
```

SOCKET here is a Perl file handle initialized by the call to `socket`. For Internet TCP applications DOMAIN is `AF_INET` and TYPE is `SOCK_STREAM`. The Perl 5 Socket package defines the constants `AF_INET` and `SOCK_STREAM` as well as other socket-related constants and functions.”

However, in Perl we can simplify the socket creation process even further by using the `IO::Socket` package, and by instantiating a `IO::Socket::INET` object. By using the `IO::Socket` package, we can further abstract the actual socket creation process into a few simple statements.

## Perl Server Socket Construction

## Sockets over C/C++, Perl and Java

We will need to use a few Perl packages to support our socket, notably the IO:Socket package. Then, we will instantiate the socket object.

We only need one statement to create and configure a socket object in Perl when using IO:Socket. Our IO:Socket socket is really class based, so we instantiate our socket with the *new* operator on an `IO::Socket::INET` object. We can still see Perl's roots in the C language's *socket*, *bind* and *listen* system calls, as we establish our port number, protocol and number of connections in the single Perl *new* `IO::Socket::INET` statement. We establish a 'tcp' stream socket, and we listen for SOMAXCONN concurrent connection requests. SOMAXCONN is usually defined as 5. In Perl, our `$server_socket` is a file handle, not a file descriptor.

```
use IO::Socket;
use Sys::Hostname;

my $client_socket;      # client socket
my $server_socket;     # server socket
my $buf;                # simple buffer
my $parent_pidno;      # parent process id
my $server_addr;       # server address
my $port_num;          # port number

$server_socket = new IO::Socket::INET(
    LocalHost => $server_addr,
    LocalPort => $port_num,
    Proto     => 'tcp',
    Listen    => SOMAXCONN,
    Reuse     => 1);
```

We **accept** a connection request from a client with the `accept` method, which returns a `$client_socket` file handle from which we can receive data from our client.

```
$client_socket = $server_socket->accept();
```

We read client data into a simple buffer.

```
$buf = <$client_socket>;
```

## **Perl Client Socket Construction**

As with our server, we start by using the IO::Socket package. We create our socket as a new **IO::Socket::INET** object in the Internet Domain as a TCP stream, and we identify our server by IP address and port number.

```
use IO::Socket;

my $server_addr;      # server address
my $port_num;        # port number
my $client_socket;   # client socket
my $buf;              # simple buffer

$client_socket = new IO::Socket::INET(
    PeerAddr => $server_addr,
    PeerPort => $port_num,
    Proto    => 'tcp');
```

Now, our client can simply print data from a simple buffer directly to our server over the \$client\_socket file handle.

```
print $client_socket $buf;
```

## Java Socket Construction

The Java environment is significantly different than C or Perl, as both C and Perl are essentially procedural languages. Java is completely object-oriented. This makes the construction of sockets over Java much more cursory than the construction of sockets over C. Additionally, Java views its' data input and output as a *stream*.

Java also presents some unique challenges. As with the old Perl saying, there is more than one way to do it. Shay describes the decision process. “In Java there are several ways to read and write a socket. Java provides many classes for reading and writing (*Readers, BufferedReaders, InputStreams, DataInputStreams*, etc.) and not all classes work correctly under the constraint that the program at the other end of the socket may not speak Java. For example, *ObjectInputStream* and *ObjectOutputStream* are often used to serialize packets. However, as previously stated we do not assume the program at the other end understands what either of these classes means. *BufferedReader* classes are sometimes used to read from sockets and, in some of our tests, we used the *BufferedReader* class. In fact, we reasoned that *BufferedReader* classes would be easier to work with since they converted bytes to characters.”

In this paper we are going to use the *BufferedReader* class wrapped around an *InputStreamReader* class because it works well with the character data stream that we use for this exercise. However, Shay warns us that he found data corruption problems with the *BufferedReader* class “The problem was that the *BufferedReader* class was wrapped around an *InputStreamReader* class that filters the data it reads. In other words it

converts bytes to Unicode character format. In most cases this was not a problem because the default encoding was consistent with the ASCII characters being sent by a remote C program. However, when binary data was transmitted the *InputStreamReader* class interpreted certain bytes values as control characters rather than the data they were supposed to represent.” Because we are limiting ourselves to character data only, we can rely on the *InputStreamReader* class.

Gagne reminds us that “Streams are part of the java.io API and are the mechanism by which input and output is done in Java. Streams may be either character or byte oriented.” Gagne illustrates the alternative use of a byte oriented stream in the following snippet:

```
OutputStream raw = client.getOutputStream();
Writer pout = new PrintWriter(raw);
pout.write(“Hello There”);
```

## **Java Server Socket construction**

We will need to import a few Java packages to support our socket. Notably, we need the java.io.\* package and the java.net.\* package.

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

int          port;          // port number
String       clientBuffer; // client buffer

ServerSocket serverSocket; // server socket
Socket       clientSocket; // client socket

InputStreamReader inStreamFromClient; // Stream from client
BufferedReader    inFromClient;      // Reader from client
```

Sockets over C/C++, Perl and Java

We create our socket as a new instance of the **ServerSocket** class, and we designate the port that we will listen on at this time.

```
serverSocket = new ServerSocket(port);
```

We accept an incoming connection request from a client, and assign it to the clientSocket Socket class variable.

```
clientSocket = serverSocket.accept();
```

Due to our object-oriented environment, we need to instantiate a few more classes before we can read data from our client. We Set up an InputStreamReader from the clientSocket, and a BufferedReader from our InputStreamReader. Then, we read our client data into a clientBuffer from our BufferedReader.

```
inStreamFromClient = new InputStreamReader( clientSocket.getInputStream() );  
inFromClient = new BufferedReader( inStreamFromClient );  
clientBuffer = inFromClient.readLine();
```

## **Java Client Socket construction**

As with our server, we need to import the java.io.\* package and the java.net.\* package.

```
import java.io.*;  
import java.net.*;  
  
String server;           // server IP address  
int port;                // port number  
String clientBuffer;    // Data from stdin  
Socket clientSocket;    // Client socket, bi-directional  
DataOutputStream outToServer; // Stream to server
```

## Sockets over C/C++, Perl and Java

We start by creating a new instance of the Socket class, in which we define the server's IP address and port number.

```
clientSocket = new Socket(server, port);
```

With our socket built, we instantiate a new DataOutputStream to the server, and then we simply write our client's data to the server over the DataOutputStream. Don't overlook the necessary newline character '\n' to flush our stream.

```
outToServer = new DataOutputStream( clientSocket.getOutputStream() );  
outToServer.writeBytes( clientBuffer + '\n' );
```

## Conclusion

We have examined the construction of sockets over C/C++, Perl and Java. We have constructed clients and servers in each language and we have examined the similarities and differences between the constructions of sockets in those languages. Our clients and servers all exchange data with each other without regard to which language they were written in. Our Perl and Java programs will also port directly from Linux to Windows and will run without change or re-compilation. Our C programs are not directly portable to Windows, and will have to be re-written to compile or run on Windows.

This research and the programs that are shown here are good first steps for any programmer who is starting the discovery process of network programming.

References

Brown, C. and McDonald, C. (2007). Visualizing Berkeley Socket Calls in Students' Programs. *ITiCSE '07*, Dundee, Scotland, United Kingdom, 2007

Toll, W. (1995). SOCKET PROGRAMMING IN THE DATA COMMUNICATIONS LABORATORY. *Proceedings of ACM Computer Science Education Technical Symposium '95 3/95*, Nashville, TN, 1995

Abdel-Wahab, H. (1988). Experience in teaching communications software using Berkeley UNIX. *ACM-SIGCSE Bulletin Vol. 20 No. 4*, Dec. 1988

Mull, M. (1997). Internet servers with Perl. *Linux Journal*, May 1997

Shay, W. (2002). A Multiplatform/Multilanguage Client/Server Project. *ACM-SIGCSE '02*, Covington, Kentucky , February 27 – March 3, 2002

Gagne, G. (2002). To java.net and Beyond Teaching Networking Concepts Using the Java Networking API. *ACM-SIGCSE '02*, Covington, Kentucky , February 27 – March 3, 2002

## Appendix A – C Source Code

### C Server Source Code

```
/*
** server.c
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int get_message(int); /* function prototype */

int main(int argc, char *argv[])
{
    int    server_sock_fd;    /* server socket file descriptor */
    int    client_sock_fd;   /* client socket file descriptor */
    int    port_num;        /* port number */
    int    client_len;      /* length of client address */
    int    parent_pidno; /* parent process id */

    struct sockaddr_in  serv_addr;    /* server address */
    struct sockaddr_in  client_addr; /* client address */

    if (argc < 2)
    {
        fprintf(stderr, "usage: %s port\n", argv[0]);
        return 1;
    }

    port_num = atoi(argv[1]);

    /*
    **      Create a socket in the Internet Domain for a TCP stream.
    */
    server_sock_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_sock_fd < 0)
    {
        perror("ERROR: creating socket");
        return 1;
    }

    /*
```

## Sockets over C/C++, Perl and Java

```
**      Create the name information that we will bind to.
**      We will:
**      bind in the Internet Domain.
**      accept a connection request from any address.
**      listen on port_num
*/
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port_num);
if ( bind(server_sock_fd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0 )
{
    perror("ERROR: could not bind");
    return 1;
}

/*
**      Listen to our newly bound socket.
**      Allow up to 5 connection requests in queue.
*/
listen(server_sock_fd,5);

client_len = sizeof(client_addr);
while (1)
{
    client_sock_fd = accept(server_sock_fd,
        (struct sockaddr *) &client_addr, &client_len);
    if (client_sock_fd < 0)
    {
        perror("ERROR on accept");
        return 1;
    }

    /*
    **      Fork a child to read and write the data.
    */
    parent_pidno = fork();
    if (parent_pidno < 0)
    {
        perror("ERROR on fork");
        return 1;
    }

    if (parent_pidno == 0)
    {
```

## Sockets over C/C++, Perl and Java

```
        /*
        **      This is the child fork.
        */
        close(server_sock_fd);
        get_message(client_sock_fd);

        /*
        **      The child dies here.
        */
        return 0;
    }
    else
    {
        /*
        **      This is the parent
        */
        close(client_sock_fd);
    }
} /* END of while */

return 0;    /* The parent actually never returns */
}
```

```
int get_message (int sock)
{
    int ctr;    /* Simple counter */
    char buffer[256];    /* Simple buffer */

    bzero(buffer,256);

    /*
    **      Read the message from the client.
    */
    ctr = read(sock, buffer, 255);
    if (ctr < 0)
    {
        perror("ERROR reading from socket");
        return 1;
    }

    /*
    **      Echo the message back to the client.
    */
    ctr = write(sock, buffer, strlen(buffer));
    if (ctr < 0)
    {
```

## Sockets over C/C++, Perl and Java

```
        perror("ERROR writing to socket");
        return 1;
    }
    return 0;
}
```

## **C Client Source Code**

```
/*
**   client.c
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int main(int argc, char *argv[])
{
    int    sockfd;        /* socket file descriptor */
    int    portno;        /* port number */
    int    ctr;           /* simple counter */
    char   buffer[256];   /* simple buffer */

    struct sockaddr_in  server_addr; /* server address info */
    struct hostent      *server;     /* server IP address */

    if (argc < 3) {
        fprintf(stderr, "usage: %s server port\n", argv[0]);
        return(1);
    }

    /*
    **   Get the server's IP address.
    */
    server = gethostbyname(argv[1]);
    if (server == NULL)
    {
        fprintf(stderr, "ERROR server not found\n");
        return(1);
    }

    /*
    **   Get the server's port number.
    */
    portno = atoi(argv[2]);

    /*
    **   Create the client socket.
    */
}
```

## Sockets over C/C++, Perl and Java

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
    perror("ERROR opening socket");
    return(1);
}

/*
**   Set up the server's name information with:
**   Internet domain
**   server's IP address
**   server's port number
*/
bzero((char *) &server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
      (char *)&server_addr.sin_addr.s_addr,
      server->h_length);
server_addr.sin_port = htons(portno);

/*
**   Connect to the server.
*/
if (connect(sockfd, &server_addr, sizeof(server_addr)) < 0)
{
    perror("ERROR can not connect to server");
    return(1);
}

bzero(buffer, 256);
printf("enter the message: ");
fgets(buffer, 255, stdin);

/*
**   Write the message to the server.
*/
ctr = write(sockfd, buffer, strlen(buffer));
if (ctr < 0)
{
    perror("ERROR writing to socket");
    return(1);
}

/*
**   Read the echoed message from the server.
*/
```

## Sockets over C/C++, Perl and Java

```
bzero(buffer, 256);
ctr = read(sockfd, buffer, 255);
if (ctr < 0)
{
    error("ERROR reading from socket");
    return(1);
}
printf("received from server: %s\n",buffer);

return(0);
}
```

## Appendix B – Perl Source Code

### *Perl Server Source Code*

```
#!/usr/bin/perl -w
#
#   server.pl
#

use strict;
use IO::Socket;
use Sys::Hostname;
use POSIX qw(:sys_wait_h);

my $client_socket;      # client socket
my $server_socket;     # server socket
my $buf;                # simple buffer
my $parent_pidno;      # parent process id
my $server_addr;       # server address
my $port_num;          # port number

if( $#ARGV < 1 )
{
    die "usage: server.pl server port\n";
}

$server_addr = shift;
$port_num = shift;

#
#   Create a server socket file handle in the Internet domain
#   for a TCP stream.
#   We will use:
#   Maximum connections allowed by the system (usually 5)
#   Allow the system to reuse the port when this server terminates.
#
$server_socket = new IO::Socket::INET(
    LocalHost => $server_addr,
    LocalPort => $port_num,
    Proto     => 'tcp',
    Listen    => SOMAXCONN,
    Reuse     => 1);
$server_socket or die "could not create the server_socket :$!";

#
```

## Sockets over C/C++, Perl and Java

```
# Force flush socket buffer immediately.
#
STDOUT->autoflush(1);

#
# Accept an incoming connection request from a client.
#
while ($client_socket = $server_socket->accept())
{
    #
    # Fork a child to read and write the data.
    #
    $parent_pidno = fork;
    if ($parent_pidno)
    {
        #
        # Jump to parent logic
        #
        next;
    }

    die "fork: $!" unless defined $parent_pidno;

    #
    # This is the child logic.
    #
    close $server_socket;

    #
    # Read incoming data from the client socket forever
    #
    while (defined($buf = <$client_socket>))
    {
        #
        # Echo the client data back to the client
        #
        chomp($buf);
        print($client_socket $buf, "\n");
    }
    exit;
} # END while ($client_socket = $server_socket->accept())
continue
{
    #
    # This is the parent logic.
    #
}
```

## Sockets over C/C++, Perl and Java

```
        close $client_socket;  
    }
```

## **Perl Client Source Code**

```
#!/usr/bin/perl -w
#
#   client.pl
#

use strict;
use IO::Socket;

my $server_addr; # server address
my $port_num;   # port number
my $client_socket; # client socket
my $buf;        # simple buffer

if( $#ARGV < 1 )
{
    die "usage: client.pl server port\n";
}

$server_addr = shift;
$port_num = shift;

#
#   Set up the client socket file handle.
#   We will use:
#   Internet domain
#   TCP stream protocol
#
$client_socket = new IO::Socket::INET(
    PeerAddr => $server_addr,
    PeerPort => $port_num,
    Proto    => 'tcp');
$client_socket or die "no client_socket :$!";

#
#   Force flush socket buffer immediately.
#
$client_socket->autoflush(1);

#
#   Get data from STDIN.
#
print( "enter the message: ");
$buf = <STDIN>;
```

## Sockets over C/C++, Perl and Java

```
#  
#     Write data to the server.  
#  
print $client_socket $buf;  
  
#  
#     Read data from the server.  
#  
$buf = <$client_socket>;  
chomp($buf);  
print("received from server: ", $buf, "\n");  
  
close $client_socket;
```

## Appendix C – Java Source Code

### Java Server Source Code

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

class server {

public static void main( String args[]) throws Exception
{
    int          port;          // port number
    String       clientBuffer;  // client buffer
    String       serverBuffer;  // server buffer

    ServerSocket serverSocket;  // server socket
    Socket        clientSocket; // client socket

    InputStreamReader inStreamFromClient; // Stream from client
    BufferedReader    inFromClient;       // Reader from client

    DataOutputStream outToClient;          // Stream to client

    if (args.length < 1) {
        System.err.println("usage: java server port");
        System.exit(1);
    }

    port = 0;
    try {
        port = Integer.parseInt(args[0]);
    } catch (NumberFormatException e) {
        System.err.println("port must be an integer");
        System.exit(1);
    }

    if (port < 1) {
        System.err.println("port must be greater than 0");
        System.exit(1);
    }

    //
    //     Create the server socket
    //
    serverSocket = new ServerSocket(port);
```

```
while( true )
{
    //
    //      Accept an incoming client connection request
    //
    clientSocket = serverSocket.accept();

    //
    //      Set up the input stream from the client socket
    //
    inStreamFromClient = new InputStreamReader(
clientSocket.getInputStream() );
    inFromClient = new BufferedReader( inStreamFromClient );

    //
    //      Read input from client
    //
    clientBuffer = inFromClient.readLine();

    //
    //      Set up the output stream to the client socket
    //
    outToClient = new DataOutputStream( clientSocket.getOutputStream() );

    //
    //      Write output to client
    //
    serverBuffer = clientBuffer + '\n';
    outToClient.writeBytes( serverBuffer );
}

} // END public      static void main()

} // END class server
```

## **Java Client Source Code**

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

class client {

public static void main( String args[]) throws Exception
{
    String server;           // server IP address
    int    port;            // port number
    String clientBuffer;    // Data from stdin
    String serverBuffer;    // Data from server
    Socket clientSocket;    // Client socket, bi-directional

    InputStreamReader inStreamStdin;    // Stream from stdin
    BufferedReader    inFromUser;    // Reader from stdin

    DataOutputStream outToServer;      // Stream to server
    InputStreamReader inStreamFromServer; // Stream from server
    BufferedReader    inFromServer;    // Reader from server

    if (args.length < 2) {
        System.err.println("usage: java client server port");
        System.exit(1);
    }

    port = 0;
    try {
        port = Integer.parseInt(args[1]);
    } catch (NumberFormatException e) {
        System.err.println("port must be an integer");
        System.exit(1);
    }

    if (port < 1) {
        System.err.println("port must be greater than 0");
        System.exit(1);
    }

    server = args[0];

    //
    //    Set up the input stream from stdin
    //
```

## Sockets over C/C++, Perl and Java

```
inStreamStdin = new InputStreamReader( System.in);
inFromUser = new BufferedReader( inStreamStdin );

//
//      Get data from stdin
//
System.out.print("enter the message: ");
clientBuffer = inFromUser.readLine();

//
//      Create the client socket
//
clientSocket = new Socket(server, port);

//
//      Set up the output stream to the server
//
outToServer = new DataOutputStream( clientSocket.getOutputStream() );

//
//      Write output to server
//
outToServer.writeBytes( clientBuffer + '\n' );

//
//      Set up the input stream from the server
//
inStreamFromServer = new InputStreamReader( clientSocket.getInputStream() );
inFromServer = new BufferedReader( inStreamFromServer );

//
//      Read input from server
//
serverBuffer = inFromServer.readLine();
System.out.println("received from server: " + serverBuffer );

//
//      Close the socket
//
clientSocket.close();

} // END public static void main()

} // END class      TCPClient
```