Network Programming
Experiment Guide

Utkarsh Kumar Raut

Sputnik — perhaps the reason why Internet exists today.

Preface

While I was preparing for an event — for which I was asked to study how the Internet was born. I went through pages of history which gave me very fascinating facts regarding how *Internet* came into existence; how *an experiment in CERN physics lab* – gave birth to WWW.

This experiment guide consist of programs which are basically meant for talking to each other via SOCKETS. I used UNIX environment and `gcc` and tried to program CLIENT – SERVER APPLICATIONS.

This guide is presented as lab manual for subject NETWORK PROGRAMMING LAB (322762(22)). Any errors, suggestions or feedback can be sent me via email.

Check out event about which I was talking — `http://ieeesstc.org/8085/`.

Have happy Socket Programming with me!

UTKARSH KUMAR RAUT
`hello@utkarshkumarraut.in`
Department of Computer Science & Technology
Shri Shankaracharya Group of Institute — Faculty of Engineering
Junwani, Bhilai, INDIA

Contents

1

Sockets

"The internet could be a very positive step towards education, organisation and participation in a meaningful society."
– Noam Chomsky, *Cognitive Scientist*

1.1 What is a Socket

Sockets are just a way to speak to other programs using standard UNIX file descriptors. UNIX performs I/O by reading or writing to a file descriptor. A file descriptor is simply an integer associated with an open file. And, that file can be a network connection, a FIFO, a pipe, a terminal, a real on-the-disk file, or just about anything else. Everything in UNIX is a file.

So when we want to communicate with another program over the Internet we are going do it through a file descriptor.

We make a call to the `socket()` system routine. It returns the socket descriptor, and you communicate through it using the specialized `send()` and `recv()` socket calls.

1.2 Types of Internet Sockets

1. Datagram Sockets, also known as connectionless sockets, which use User Datagram Protocol (UDP).

2. Stream Sockets, also known as connection-oriented sockets, which use Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP).

3. Raw Sockets (or Raw IP Sockets), typically available in routers and other network equipment. Here the transport layer is bypassed, and the packet headers are made accessible to the application.

We however are going to be concentrated on *first two types of Internet Sockets*

1.3 Stream Socket

In computer networking, a stream socket is a type of a internet socket which provides a connection-oriented, sequenced, and unique flow of data without record boundaries, with well-defined mechanisms for creating and destroying connections and for detecting errors.

This internet socket type transmits data on a reliably, in order, and with out-of-band capabilities.

Traditionally, stream sockets are implemented on top of `TCP` so that applications can run across any networks using `TCP/IP` protocol. `SCTP` can also be used for stream sockets.

Stream Sockets — Delivery in a networked environment is guaranteed. If you send through the stream socket three items `"A, B, C"`, they will arrive in the same order — `"A, B, C"`. These sockets use **TCP (Transmission Control Protocol)** for data transmission. If delivery is impossible, the sender receives an *error indicator*.

'Telnet' uses stream sockets. All the characters we type need to arrive in the same order we type them. Also, web browsers use the `HTTP` protocol which uses stream sockets to get pages. Indeed, if we telnet to a web site on port 80, and type `"GET / HTTP/1.0"` and hit `RETURN` twice, it'll dump the HTML back.

1.4 Datagram Socket

A *datagram socket* is a type of connectionless network socket, which is the point for sending or receiving of packet delivery services. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may arrive in any order and might not arrive at the receiving computer.

`UDP` broadcasts sends are always enabled on a datagram socket. Datagram sockets are Basically connection-less, it's because we don't have to maintain an open connection as we do with stream sockets. We just build a packet, slap an IP header on it with destination information, and send it out. No connection needed.

TCP v/s UDP— If we are making Chat application — sending chat message TCP is great; but we are streaming some live feed; may be it doesn't matter so much if one or two frames skipped — we use UDP for such application. UDP is all about speed.

1.5 Network Theory — Let's break packet

It's time to talk about *how networks really work*, and to show some examples of how `SOCK_DGRAM` packets are built.

SOCK_STREAM & SOCK_DGRAM — TCP almost always uses SOCK_STREAM and UDP uses SOCK_DGRAM.

TCP / SOCK_STREAM is a connection-based protocol. The connection is established and the two parties have a conversation until the connection is terminated by one of the parties or by a network error.

UDP / SOCK_DGRAM is a datagram-based protocol. You send one datagram and get one reply and then the connection terminates.

- If you send multiple packets, TCP promises to deliver them in order. UDP does not, so the receiver needs to check them, if the order matters.
- If a TCP packet is lost, the sender can tell. Not so for UDP.
- TCP is a bit more robust and makes more checks. UDP is a shade lighter weight (less computer and network stress).

Data Encapsulation — Data Packet is born – then it's wrapped ('encapsulated') in a header by first protocol (in our case TFTP protocol), then the whole thing (Data + TFTP header included) is again encapsulated by next protocol (that is 'UDP'); then again by next (which is 'IP' protocol). Then finally, by protocol on hardware (physical) layer (say 'Ethernet').

TFTP (Trivial File Transfer Protocol) — Today, TFTP is virtually unused for Internet transfers. It is a simple protocol for transferring files, implemented on top of the UDP/IP protocols using well-known port number 69. TFTP was designed to be small and easy to implement, and therefore it lacks most of the advanced features offered by more robust file transfer protocols. TFTP only reads and writes files from or to a remote server. It cannot list, delete, or rename files or directories and it has no provisions for user authentication.

When another computer receives the packet, the hardware strips the Ethernet header, the kernel strips the IP and UDP headers, the TFTP program strips the TFTP header, and it finally has the data.

1.6 Layered Network Model

The *Open Systems Interconnection model (OSI Model)* is a conceptual model that characterizes and standardizes the communication functions of a telecommunication or computing system without regard to their underlying internal structure and technology.

This Network Model describes a system of network functionality that has many advantages over other models. For instance, you can write sockets programs that are exactly the same without caring how the data is physically transmitted (serial, thin Ethernet, AUI, whatever) because programs on lower levels deal with it for you. The actual network hardware and topology is transparent to the socket programmer.

A *layered model* more consistent with UNIX might be —

- Application Layer — *(telnet, ftp, http)*
- Host-to-Host Transport Layer — *(TCP, UDP)*
- Internet Layer — *(IP)*
- Network Access Layer — *(Ethernet, Wi-Fi)*

The kernel builds the Transport Layer and Internet Layer on for us and the hardware does the Network Access Layer.

1.7 IPv4 & IPv6

IPv4 (Internet Protocol Version 4) — It is the fourth revision of the Internet Protocol (IP) used to to identify devices on a network through an addressing system. The Internet Protocol is designed for use in interconnected systems of *packet-switched computer communication networks.*

IPv4 is the most widely deployed Internet protocol used to connect devices to the Internet. IPv4 uses a 32-bit address scheme allowing for a total of 2^{32} addresses (just over 4 billion addresses). With the growth of the Internet it is expected that the number of unused IPv4 addresses will eventually run out because every device – including computers, smart-phones, game consoles, tv, microwaves – that connects to the Internet requires an address.

A new Internet addressing system Internet Protocol version 6 (IPv6) is being deployed to fulfill the need for more Internet addresses.

IPv6 (Internet Protocol Version 6) — It is the newest version of the Internet Protocol (IP) reviewed in the IETF standards committees to replace the current version of IPv4 (Internet Protocol Version 4). IPv6 is the successor to Internet Protocol Version 4 (IPv4). IPv6 uses a 128-bit address scheme and this 128 bits represents about *340 trillion trillion trillion* numbers (for real, 2^{128}). That's like a million IPv4 Internets for every single star in the Universe.

Illustration

```
IPv4   117.223.172.195
IPv6   0:0:0:0:0:ffff:75df:acc3
```

1.8 Port Numbers

Think of the IP address as the street address of a hotel, and the port number as the room number. A port number is a way to identify a specific process to which an Internet or other network message is to be forwarded when it arrives at a server. For the Transmission Control Protocol and the User Datagram Protocol, a port number is a 16-bit integer that is put in the header appended to a message unit. This port number is passed logically between client and server transport layers and physically between the transport layer and the Internet Protocol layer and forwarded on.

Well-known Port Numbers

The well-known port numbers are the port numbers that are reserved for assignment by the Internet Corporation for Assigned Names and Numbers (ICANN) for use by the application end points that communicate using the Internet's Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). Each kind of application has a designated (and thus "well-known") port number.

The well-known ports cover the range of possible port numbers from 0 through 1023. The registered ports are numbered from 1024 through 49151. The remaining ports, referred to as dynamic ports or private ports, are numbered from 49152 through 65535.

For example, the Hypertext Transfer Protocol (HTTP) application has the port number of 80; and the Post Office Protocol Version 3 (POP3) application, commonly used for e-mail delivery, has the port number of 110. Port 21 is used for FTP control.

There is list of all well-known ports for TCP and UDP available here — `https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`

1.9 Endianness — Byte Order

Endianness refers to the order of the bytes, comprising a digital word, in computer memory. It also describes the order of byte transmission over a digital link. Words may be represented in big-endian or little-endian format. Both forms of endianness are widely used in digital electronics. The choice of endianness for a new design is often arbitrary.

Big-endian is the most common format in data networking; fields in the protocols of the Internet protocol suite, such as IPv4, IPv6, TCP, and UDP, are transmitted in big-endian order. For this reason, big-endian byte order is also referred to as network byte order.

Little-endian storage is popular for microprocessors, in part due to significant influence on microprocessor designs by Intel Corporation.

1. Big-endian — In this scheme, high-order byte is stored on the starting address (A) and low-order byte is stored on the next address (A + 1).

2. Little-endian — In this scheme, low-order byte is stored on the starting address (A) and high-order byte is stored on the next address (A + 1).

To allow machines with different byte order conventions communicate with each other, the Internet protocols specify a canonical byte order convention for data transmitted over the network. This is known as Network Byte Order.

Basically, we want to convert the numbers to Network Byte Order before they go out on the wire, and convert them to Host Byte Order as they come in off the wire.

2

Programming Sockets

It's time to talk about programming. In this section, We'll cover various data types used by the sockets interface.

Before we start — let me give an opportunity to illustrate what we are going to cover through this chapter. Assume there are two console terminals, from one we send something to other terminal and expect some output from it. In this case console which is sending data can be assumed to be *client* and other console which serves to our client and give an appropriate output can be assumed to be a *server*.

So what actually we need to do in order to achieve that. *We need to program, of-course!* Socket programing is this only — making application talk to each other over the network. In our illustration as both applications (*consoles*) were on same machine — they shared same network also called as `localhost`.

> **localhost** — On most computer systems, `localhost` resolves to the IP address `127.0.0.1`, which is the most commonly used IPv4 loopback address, and to the IPv6 loopback address `::1`. The name `localhost` is also a reserved top-level domain name (cf. .localhost), set aside. to avoid confusion with the definition as a hostname.

2.1 Understanding Server application

Before we start from base, lets deduce an abstract model of our illustration. Server application is something which *listens* and once someone *sends a request* to it, server produces *response* to that particular request. Server listens through a binded socket, it can do both reading and writing through that binded socket. So lets plan our server application.

2.1.1 Steps in order to make Server application

Below are the steps we need to follow in order to create socket— based server application in UNIX environment using `gcc`.

1. Create Socket — We need to create socket using `socket()` system call.
2. Bind Socket — As soon as we have created socket, we now need to bind that socket with address using `bind()` system call. On internet–based server socket address is consist of *hostname* and *port number*.
3. Listen to Connection — Once server–socket is binded to particular address; it listens to that in order to check if something has happened. This is done by using `listen()` system call.
4. Accept the Connection — Using `accept()` system call socket accept the connection from client and this call typically blocks until a client connects with the server.
5. Sends & Receive Data — Data flows through socket, we use `read()` and `write()` system calls for reading and writing purpose respectively.

2.1.2 Create Socket using `socket()` system call

In UNIX environment to perform network I/O, the first thing a process must do is, call the socket function, specifying the type of communication protocol desired and protocol family, etc.

We will focus on three basic things that are —

1. family — We need to specify the family type of socket we want to create. For our illustration – in order to create server socket we specify family as `AF_INET`. Alongside, `AF_INET` there are few more family type we can specify while creating socket. These are —
   - `AF_INET` — IPv4 protocols
   - `AF_INET6` — IPv6 protocols
   - `AF_LOCAL` — UNIX domain protocols
   - `AF_ROUTE` — Routing sockets
2. type — It specifies the kind of socket you want. It can take one of the following values —
   - `SOCK_STREAM` — Stream socket
   - `SOCK_DGRAM` — Datagram socket
   - `SOCK_SEQPACKET` — Sequenced packet socket

- `SOCK_RAW` — Raw socket

In TCP we prefer `SOCK_STREAM`; while in UDP we choose `SOCK_DGRAM` *socket type.*

3. PROTOCOL — This argument should be set to the specific protocol type as given below, or 0 to select the system's default for the given combination of family and type.

   - `IPPROTO_TCP` — TCP transport protocol
   - `IPPROTO_UDP` — UDP transport protocol
   - `IPPROTO_SCTP` — SCTP transport protocol

SYNTAX —

```c
#include <sys/types.h>
#include <sys/socket.h>
/* Socket function */
int socket (int family, int type, int protocol);
```

This call returns a `socket descriptor` that we can use in later system calls or -1 in case of error.

Let's start creating out first socket program, create `server.c` for implementing server application.

> **GCC Compiler—** We done all the programming in UNIX environment, all C programs are compiled using gcc compiler. If you are using Windows/Mac operating systems — you need to install UNIX–emulation tools in order to work alongside with examples.

STEP 00 — CREATING `SERVER.C`

```c
/************************
* SERVER.C
* UTKARSH KUMAR RAUT
*************************/

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

 /*******************
 * CODES HERE
 *******************/

```

```
15  return 0;
16 }
```

STEP 01 — DEFINE SOCKET DESCRIPTOR

`socket()` system call returns socket descriptor of `int` type. So we will need one `int` type variable to store that descriptor.

```
1  /************************
2  * SERVER.C
3  * UTKARSH KUMAR RAUT
4  ************************/
5
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  /* Headers for Socket system call */
10 #include <sys/types.h>
11 #include <sys/socket.h>
12
13 int main(int argc, char *argv[]){
14
15  int sockfd;
16
17  sockfd = socket(AF_INET, SOCK_STREAM, 0);
18
19  if(sockfd < 0){
20    printf("CAN'T CREATE SOCKET!");
21    exit(1);
22  }
23
24  printf("SOCKET MADE!");
25
26  return 0;
27 }
```

Below is the standard `gcc` compilation step. We will use this same on all illustrations —

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "server.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "server" "server.c" -lm
```

And, lets execute `server.c` as follows —

```
karshe@karshe-dell:~/root\$ ./server
```

Output

```
SOCKET MADE!
```

2.1.3 Binding socket using `bind()` system call

We created Socket and now we are going to *bind* that with network address and port – in order to bind socket with host network we need to initialize `socket structure`.

STEP 02 — INITIALIZE SOCKET STRUCTURE & BIND IT

Socket structure is defined using `struct sockaddr_in` in C. This structure is composed of members like `sa_family`, `sin_port`, `sin_addr`, `sin_zero`.

Before initialize socket structure lets define this variable. Remember to include header file `<netinet/in.h>` and this header shall define the `sockaddr_in` structure.

The `sockaddr_in` structure is used to store addresses for the Internet address family. Values of this type shall be cast by applications to `sockaddr` for use with socket functions.

REMEMBER — bind() of `INADDR_ANY` does not *generate a random network address (IP)*. It binds the socket to all available interfaces and for a server program, you typically want to bind to all interfaces — not just `'localhost'`.

```
/************************
* SERVER.C
* UTKARSH KUMAR RAUT
*************************/

#include <stdio.h>
#include <stdlib.h>

/* Headers for Socket system call */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h> /* HEADER FOR SOCKET STRUCT */
#include <unistd.h>

#include <string.h>

int main(int argc, char *argv[]){

 int sockfd;
 struct sockaddr_in serv_addr; /* SOCKET STRUCTURE */
 int portno; /* USED WHILE BINDING */

 sockfd = socket(AF_INET, SOCK_STREAM, 0);

 if(sockfd < 0){
   printf("CAN'T CREATE SOCKET!");
   exit(1);
 }

  /* SOCKET IS CREATED! */
 printf("SOCKET MADE! \n");
 printf("NOW BIND IT WITH PORT! \n");

 /* INIT SERVER SOCKET STRUCTURE */
   bzero((char *) &serv_addr, sizeof(serv_addr));
   portno = 5001; /*PORT NUMBER*/

```

```
38    serv_addr.sin_family = AF_INET; /*IPV4*/
39    serv_addr.sin_addr.s_addr = INADDR_ANY;
40    serv_addr.sin_port = htons(portno); /* PORT */
41
42  /* BIND THE SOCKET */
43  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(
      serv_addr)) < 0) {
44       printf("CAN'T BIND THE SOCKET!");
45       exit(1);
46    }
47
48   printf("SOCKET HAS BEEN BINDED WITH PORT %d\n", portno);
49
50  return 0;
51 }
```

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "server.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "server" "server.c" -lm
```

Executing `server.c` —

```
karshe@karshe-dell:~/root\$ ./server
```

Output

```
SOCKET MADE!
NOW BIND IT WITH PORT!
SOCKET HAS BEEN BINDED WITH PORT 5001
```

Port 80 — If we change `portno` on `line` 36 to 80 we expect following result.
Output

```
SOCKET MADE!
NOW BIND IT WITH PORT!
CAN'T BIND THE SOCKET!
```

As to bind with well-know ports or ports (<1024) we need root privileges.

2.1.4 Listen socket using `listen()` system call

We binded *(perhaps!)* socket with address and port. Now we need to use `listen()` function and it performs these two actions —

1. The `listen()` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket.

2. The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

This call returns `0` on success, otherwise it returns `-1` on error.
STEP 03 — CONVERSION INTO PASSIVE SOCKET

```
listen(sockfd, 5); /* INTO PASSIVE SOCKET */
```

REMEMBER — `listen()` function is called only by a `TCP server`.

2.1.5 Accept connection using `accept()` system call

It basically accept an incoming connection on a listening socket, Once we've gone through the trouble of getting a `SOCK_STREAM` socket and setting it up for incoming connections with `listen()`, then you call `accept()` to actually get a new socket descriptor to use for subsequent communication with the newly connected client.

The old socket that we were using for listening is still there, and will be used for further `accept()` calls as they come in.

The socket descriptor returned by `accept()` is a bona fide socket descriptor, open and connected to the remote host. You have to `close()` it when you're done with it.

RETURN VALUE — `accept()` returns the newly connected socket descriptor, or `-1` on error.

```
/************************
* SERVER.C
* UTKARSH KUMAR RAUT
*************************/

#include <stdio.h>
#include <stdlib.h>

/* Headers for Socket system call */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h> /* HEADER FOR SOCKET STRUCT */
#include <unistd.h>

#include <string.h>

int main(int argc, char *argv[]){

 int sockfd; /* TO CREATE A SOCKET WE NEED */
 struct sockaddr_in serv_addr; /*SERVER ADDRESS STRUCTURE -
    STORE SERVER THING*/

 /* CLIENT LISTENER */
 int clientfd;
 struct sockaddr_in client_addr;
```

```
25 socklen_t clilen;
26
27 int portno; /* WE NEED PORT! */
28 int t;
29
30 char buffer[256];
31
32 /* CREATE SOCKET */
33 /* IT IS IPV4 + TCP */
34 sockfd = socket(AF_INET, SOCK_STREAM, 0);
35
36 /* CHECK IF SOCKET IS MADE */
37 if(sockfd < 0){
38   printf("ERROR IN CREATING SOCKET!");
39   exit(1);
40 }
41
42 /* SOCKET IS CREATED! */
43 printf("SOCKET HAS BEEN CREATED, NOW BIND IT WITH PORT! \n");
44
45 /* INIT SERVER SOCKET STRUCTURE */
46   bzero((char *) &serv_addr, sizeof(serv_addr));
47   portno = 5001; /*PORT NUMBER*/
48
49   serv_addr.sin_family = AF_INET; /*TCP*/
50   serv_addr.sin_addr.s_addr = INADDR_ANY;
51   serv_addr.sin_port = htons(portno); /* PORT */
52
53 /* BIND THE SOCKET */
54 if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(
     serv_addr)) < 0) {
55       printf("CAN'T BIND THE SOCKET!");
56       exit(1);
57   }
58
59  printf("SOCKET HAS BEEN BINDED WITH PORT %d\n", portno);
60
61 /* LISTEN TO CONNECTION */
62 listen(sockfd, 5);
63
64 clilen = sizeof(client_addr); /* SIZE OF CLIENT SOCKET STRUCT
     */
65
66 /* ACCEPT THE CONNCETION */
67 clientfd = accept(sockfd, (struct sockaddr *)&client_addr, &
     clilen);
68
69 if(clientfd < 0){
70   printf("CAN'T ACCEPT CONNECTION FROM CLIENT!");
71   exit(1);
72 }
73
74 /* CLOSE CONNECTION */
75 close(sockfd);
```

```
76  close(clientfd);
77
78  return 0;
79 }
```

2.1.6 Send and Receive data

In *server-side console* — we will use `read()` and `write()` system call for sending and receive data over socket.

Below code sample is complete TCP Based Server-side console.

```
1  /*************************************
2   * Server using TCP.
3   * by https://github.com/karshe
4   * PROJECT : NETWORK PROGRAMMING IN UNIX
5   * Compiled using gcc
6   * ***********************************/
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
11 #include <netdb.h>
12 #include <netinet/in.h>
13
14 #include <sys/socket.h>
15 #include <unistd.h>
16
17 #include <string.h>
18 int main(int argc, char *argv[]){
19
20  int sockfd; /* TO CREATE A SOCKET WE NEED */
21  struct sockaddr_in serv_addr; /*SERVER ADDRESS STRUCTURE -
       STORE SERVER THING*/
22
23  /* CLIENT LISTENER */
24  int clientfd;
25  struct sockaddr_in client_addr;
26  socklen_t clilen;
27
28  int portno; /* WE NEED PORT! */
29  int t;
30
31  char buffer[256];
32
33  /* CREATE SOCKET */
34  /* IT IS IPV4 + TCP */
35  sockfd = socket(AF_INET, SOCK_STREAM, 0);
36
37  /* CHECK IF SOCKET IS MADE */
38  if(sockfd < 0){
39    printf("ERROR IN CREATING SOCKET!");
40    exit(1);
41  }
```

```
42
43 /* SOCKET IS CREATED! */
44 printf("SOCKET HAS BEEN CREATED, NOW BIND IT WITH PORT! \n");
45
46 /* INIT SERVER SOCKET STRUCTURE */
47   bzero((char *) &serv_addr, sizeof(serv_addr));
48   portno = 6000; /*PORT NUMBER*/
49
50   serv_addr.sin_family = AF_INET; /*TCP*/
51   serv_addr.sin_addr.s_addr = INADDR_ANY;
52   serv_addr.sin_port = htons(portno); /* PORT */
53
54 /* BIND THE SOCKET */
55 if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(
     serv_addr)) < 0) {
56      printf("CAN'T BIND THE SOCKET!");
57      exit(1);
58   }
59
60  printf("SOCKET HAS BEEN BINDED WITH PORT %d\n", portno);
61
62 /* LISTEN TO CONNECTION */
63 listen(sockfd, 5);
64
65 clilen = sizeof(client_addr); /* SIZE OF CLIENT SOCKET STRUCT
     */
66
67 /* ACCEPT THE CONNCETION */
68 clientfd = accept(sockfd, (struct sockaddr *)&client_addr, &
     clilen);
69
70 if(clientfd < 0){
71   printf("CAN'T ACCEPT CONNECTION FROM CLIENT!");
72   exit(1);
73 }
74
75 /* READ OR WRITE HERE */
76 bzero(buffer, 256);
77 t = read(clientfd, buffer, 255); /* READ FROM CLIENT SOCKET!
     */
78
79 if(t < 0){
80   printf("CAN'T READ FROM CLIENT!");
81   exit(1);
82 }
83
84 printf("CLIENT SAYS: %s\n", buffer);
85
86 t = write(clientfd, "RODGER THAT!", 18); /* WRITE ON CLIENT
     SOCKET */
87 if(t < 0){
88   printf("I CAN'T WRITE ON SOCKET!");
89   exit(1);
90 }
```

```
91
92  /* CLOSE CONNECTION */
93  close(sockfd);
94  close(clientfd);
95
96  return 0;
97 }
```

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "server.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "server" "server.c" -lm
```

Executing `server.c` —

```
karshe@karshe-dell:~/root\$ ./server
```

Output

```
SOCKET HAS BEEN CREATED, NOW BIND IT WITH PORT!
SOCKET HAS BEEN BINDED WITH PORT 6000
```

2.2 Creating Client application

We created `SERVER APPLICATION` which listens to particular port — Now, its time we are going to program *client application.*

Below code sample is complete TCP Based Client-side console.

```
1  /**************************************
2   * Client using TCP.
3   * by https://github.com/karshe
4   * PROJECT : NETWORK PROGRAMMING IN UNIX
5   * Compiled using gcc
6   * ************************************/
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
11 #include <netdb.h>
12 #include <netinet/in.h>
13
14 #include <sys/socket.h>
15 #include <unistd.h>
16
17 #include <string.h>
18 int main(int argc, char *argv[]){
19
20   int socketfd;
21   struct sockaddr_in client_addr;
22
23   int portno;
24   struct hostent *server;
25
```

```
26  int t;
27
28  char buffer[256];
29  portno = 6000;
30
31  /* CREATE A SOCKET TO TALK TO SERVER */
32  socketfd = socket(AF_INET, SOCK_STREAM, 0);
33
34  if(socketfd < 0){
35    printf("CAN'T CREATE SOCKET!");
36    exit(1);
37  }
38
39  server = gethostbyname("localhost");
40
41  if(server == NULL){
42    printf("CAN'T CONNCET TO SPECIFIED SERVER");
43    exit(0);
44  }
45
46  /* CONNCET TO SERVER */
47  bzero((char *) &client_addr, sizeof(client_addr)); /*
     NULLIFIED! */
48
49  client_addr.sin_family = AF_INET;
50  bcopy((char *)server->h_addr, (char *)&client_addr.sin_addr.
     s_addr, server->h_length);
51
52  client_addr.sin_port = htons(portno);
53
54  if (connect(socketfd, (struct sockaddr*)&client_addr, sizeof(
     client_addr)) < 0) {
55      printf("WE FAILED TO CONNECTED WITH SERVER!");
56     exit(1);
57   }
58
59  printf("WE ARE SUCCESSFULLY CONNECTED WITH SERVER!\n");
60
61  /* READ/WRITE TO SERVER */
62  printf("YOUR MESSAGE - ");
63  bzero(buffer,256);
64  fgets(buffer,255,stdin);
65
66  /* SEND BUFFER TO SERVER */
67  t = write(socketfd, buffer, strlen(buffer));
68   if (t < 0) {
69    printf("\nCLIENT WAS UNABLE TO WRITE ON SERVER SOCKET!");
70    exit(1);
71   }
72
73   /* NULIFIED THE BUFFER */
74   bzero(buffer,256);
75
76   /* READ FROM SERVER */
```

```
77    t = read(socketfd, buffer, 255);
78
79    if (t < 0) {
80       printf("\nCLIENT WAS UNABLE TO READ FROM SERVER SOCKET!")
      ;
81       exit(1);
82    }
83
84   /* IF EVERY THING GOES FINE! */
85   printf("%s\n",buffer);
86
87   /* CLSOE SOCKET */
88   close(socketfd);
89
90   return 0;
91 }
```

Above code is same as we used for `writing our server application` but instead of `bind()`, `listen()` and `accept()` — we used `connect()` system call and connected to remote–server by specifying its `hostname` and `listeing port`.

Lets Execute the Client

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "client.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "client" "client.c" -lm
```

Executing `client.c` —

```
karshe@karshe-dell:~/root\$ ./client
```

Output

```
WE FAILED TO CONNECTED WITH SERVER!

------------------
(program exited with code: 1)
```

Failed! — This happened because client was expecting a server running on `localhost:6000` but we never ran `server.c` while executing this script. In order to run Client-Server application we need both application up and running. So lets try again and execute both `server.c` and `client.c` in different console.

Executing `server.c` in console 1—

```
karshe@karshe-dell:~/root\$ ./server
```

Executing `client.c` in console 2—

```
karshe@karshe-dell:~/root\$ ./client
```

Below are the expected output — after we send some bits from client to server.

Console 1 — server.c

```
karshe@karshe-dell:~/root\$ ./server
SOCKET HAS BEEN CREATED, NOW BIND IT WITH PORT!
SOCKET HAS BEEN BINDED WITH PORT 6000
CLIENT SAYS: Hello Socket!

------------------
(program exited with code: 0)
```

Console 2 — client.c

```
karshe@karshe-dell:~/root\$ ./client
WE ARE SUCCESSFULLY CONNECTED WITH SERVER!
YOUR MESSAGE - Hello Socket!
RODGER THAT!

------------------
(program exited with code: 0)
```

3

Experiments

In following chapter we will cover most of the experiments *(perhaps!)* using `gcc` in our `UNIX` environment.

List of Experiments we are going to cover[1] —

1. Write an echo program with client and iterative server using TCP.
2. Write an echo program with client and concurrent server using TCP.
3. Write a client and server program for chatting.
4. Write a program to retrieve date and time using TCP.
5. Write a program to retrieve date and time using UDP.
6. Write a client and server routines showing I/O multiplexing.
7. Write an echo client and server program using UNIX domain stream socket.
8. Write a client and server program to implement the remote command execution.
9. Write a client program that gets a number from the user and sends the number to server for conversion into hexadecimal and gets the result from the server.

[1] As described in CSVTU BE-CSE VIIth semester syllabus

3.1 Echo program with client & iterative server using TCP.

Below are the source code for both *server-application* and *client-application*.

We will execute *server* first and then two instances of *client* which would be served iteratively by *server application*..

3.1.1 Iterative server based on TCP

TCP_ITERATIVE_SERVER.C

```
/**************************************
 * An iterative server using TCP.
 * by https://github.com/karshe
 * PROJECT : NETWORK PROGRAMMING IN UNIX
 * ************************************/

/**************************************
 * ONE CLIENT AT A TIME, MAX OF DEFINED IN
 * client_conn VARIABLE
 * RECEIVES DATA FROM CLIENT n AND SAYS
 * BYE AND WAIT FOR n+1 CLIENT
 * ************************************/

#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>

#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char *argv[]){

  int listen_sock; /* SOCKET TO LISTEN */
  int accept_sock; /* CONNECTION FROM CLIENT */
  char buffer[256]; /* BUFFER */
  struct sockaddr_in addr; /* SOCKET STRUCTURE */
  struct sockaddr_in listnerclient_addr;

  int server_port; /* SERVER PORT */
  int client_conn = 2;
  int i;
  int chunk;
  socklen_t accept_socksize;

  /* SOCKET CREATION */
  listen_sock = socket(AF_INET, SOCK_STREAM, 0);

  if(listen_sock < 0){
    perror("CAN'T CREATE SOCKET! \n");
```

```
42      exit(1);
43    }
44    /* SOCKET CREATED SUCCESSFULLY */
45    printf("SOCKET CREATED! WAITING FOR BIND \n");
46
47    /* WE NEED TO BIND IT ON SOME PORT */
48    /* YOU CAN USE BZERO ALSO! NULIFIED EVERYTHING*/
49    memset(&addr, 0, sizeof(addr));
50    server_port = 6000;
51
52    addr.sin_family = AF_INET;
53    addr.sin_addr.s_addr = INADDR_ANY;
54    addr.sin_port = htons(server_port);
55
56    if( bind(listen_sock, (struct sockaddr *)&addr, sizeof(addr))
        < 0 ){
57      /* UNSUCCESSFUL BINDING */
58      perror("BINDING ERROR! \n");
59      close(listen_sock); /* CLOSE SOCKET */
60      exit(1);
61    }
62
63    /* BINDIND SUCCESSFULLY */
64    printf("SOCKET BINDED TO PORT %d \n", server_port);
65
66    if( listen(listen_sock, 5) < 0 ){
67      /* UNSUCCESSFUL LISTENING ATTEMPT */
68      perror("LISTEN ERROR! \n");
69      close(listen_sock); /* CLOSE SOCKET */
70      exit(1);
71    }
72
73    /* LISTENING SUCCESSFULLY */
74    printf("SERVER READY \n");
75
76    for(i=0; i<client_conn; i++){
77      accept_socksize = sizeof(listnerclient_addr);
78      accept_sock = accept(listen_sock, (struct sockaddr *)&
       listnerclient_addr, &accept_socksize);
79
80        if(accept_sock < 0){
81          perror("ACCEPT SOCK ERROR!");
82          exit(1);
83        }
84        printf("WAITING FOR CLIENT %d \n", i+1);
85
86        bzero(buffer, sizeof(buffer));
87        chunk = read(accept_sock, buffer, sizeof(buffer));
88
89        if(chunk < 1){
90          perror("CHUNK CAN'T BE READ FROM CLIENT");
91          exit(1);
92        }
93
```

```
94        printf("CLIENT %d SAID : %s", i+1, buffer);
95
96        bzero(buffer, sizeof(buffer));
97        strcpy(buffer, "THANKS FOR CONNECTING ME!");
98
99        if(write(accept_sock, buffer, sizeof(buffer)) < 1){
100         printf("CAN'T SEND TO CLIENT %d", i+1);
101         exit(1);
102       }
103
104       printf("BYE CLIENT %d\n", i+1);
105       close(accept_sock);
106   }
107   printf("TURNING OFF SERVER NO MORE REQUESTS!");
108   close(listen_sock);
109   return 0;
110 }
```

#include <networking.h> — In order to use socket programming with more ease, I just created one standard header file (`networking.h`) which included all the necessary header files with standard routines which we are going to use through out this chapter and in coming experiments.

3.1.2 Client based on TCP

TCP_Client.c

```
1  /**************************************
2   * Client using TCP.
3   * by https://github.com/karshe
4   * PROJECT : NETWORK PROGRAMMING IN UNIX
5   * USING "networking.h"
6   * ************************************/
7
8  #include "networking.h"
9
10 int main(void){
11  char host[] = "localhost";
12  int port_no = 6000;
13  char buffer[256];
14
15  int sockfd = TCP_SOCKET_CLIENT();
16   if( TCP_CONNECT_SERVER(host, port_no, sockfd) ){
17        printf("CONNECTED WITH %s ON PORT %d \n", host, port_no)
       ;
18        printf("YOUR MESSAGE - ");
19        bzero(buffer,256);
20        fgets(buffer,255,stdin);
21        TCP_SEND_SERVER(sockfd, buffer);
```

```
22          printf("SERVER REPLIED - %s\n", TCP_RECEIVE_SERVER(
    sockfd));
23          TCP_CLOSE_SOCKET(sockfd);
24        }else{
25          printf("NO CONNECTION MADE WITH SERVER ON PORT %d ",
    port_no);
26        }
27   return 0;
28  }
```

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "TCP_Iterative_Server.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "TCP_Iterative_Server"
"TCP_Iterative_Server.c" -lm

karshe@karshe-dell:~/root\$ gcc -Wall -c "TCP_Client.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "TCP_Client"
"TCP_Client.c" -lm
```

Executing TCP_Iterative_Server.c —

```
karshe@karshe-dell:~/root\$ ./TCP_Iterative_Server
```

Output

```
SOCKET CREATED! WAITING FOR BIND
SOCKET BINDED TO PORT 6000
SERVER READY
WAITING FOR CLIENT 1
CLIENT 1 SAID : Hello
BYE CLIENT 1
WAITING FOR CLIENT 2
CLIENT 2 SAID : World
BYE CLIENT 2
TURNING OFF SERVER NO MORE REQUESTS!

------------------
(program exited with code: 0)
```

Executing TCP_Client.c —

```
karshe@karshe-dell:~/root\$ ./TCP_Client
```

Output of Instance 1

```
CONNECTED WITH localhost ON PORT 6000
YOUR MESSAGE - Hello
SERVER REPLIED - THANKS FOR CONNECTING ME!

------------------
(program exited with code: 0)
```

Output of Instance 2

```
CONNECTED WITH localhost ON PORT 6000
YOUR MESSAGE - World
SERVER REPLIED - THANKS FOR CONNECTING ME!

------------------
(program exited with code: 0)
```

3.2 Echo program with client & concurrent server using TCP.

Below are the source code for both *server-application* and *client-application*.

We will execute *server* first and then n instances of *client* which would be served concurrently by *server application*..

3.2.1 Concurrent server based on TCP

TCP_Concurrent_Server.c

```
/*****************************************
 * TCP_Concurent_Server.c
 * A concurrent server using TCP.
 * by https://github.com/karshe
 * PROJECT : NETWORK PROGRAMMING IN UNIX
 * ***************************************/

/*****************************************
 * FORKING IS AWESOME!
 * ***************************************/

#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>

#include <sys/socket.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>

void doprocessing (int sock) {
   int n;
   char buffer[256];
   bzero(buffer,256);
   n = read(sock,buffer,255);

   if (n < 1) {
      printf("ERROR IN READING CLIENT \n");
      exit(1);
   }

   printf("CLIENT SAYS: %s\n", buffer);

   n = write(sock,"THANKS!",18);

   if (n < 1) {
      printf("CAN'T WRITE ON SOCKET! \n");
      exit(1);
   }
```

```
42
43 }
44
45
46 int main(int argc, char *argv[]){
47
48   int listen_sock; /* SOCKET TO LISTEN */
49   int accept_sock; /* CONNECTION FROM CLIENT */
50   struct sockaddr_in addr; /* SOCKET STRUCTURE */
51   struct sockaddr_in listnerclient_addr;
52
53   int server_port; /* SERVER PORT */
54   int pid;
55   socklen_t accept_socksize;
56
57   /* SOCKET CREATION */
58   listen_sock = socket(AF_INET, SOCK_STREAM, 0);
59
60   if(listen_sock < 0){
61     perror("CAN'T CREATE SOCKET! \n");
62     exit(1);
63   }
64   /* SOCKET CREATED SUCCESSFULLY */
65   printf("SOCKET CREATED! WAITING FOR BIND \n");
66
67   /* WE NEED TO BIND IT ON SOME PORT */
68   /* YOU CAN USE BZERO ALSO! NULIFIED EVERYTHING*/
69   memset(&addr, 0, sizeof(addr));
70   server_port = 6000;
71
72   addr.sin_family = AF_INET;
73   addr.sin_addr.s_addr = INADDR_ANY;
74   addr.sin_port = htons(server_port);
75
76   if( bind(listen_sock, (struct sockaddr *)&addr, sizeof(addr))
       < 0 ){
77     /* UNSUCCESSFUL BINDING */
78     perror("BINDING ERROR! \n");
79     close(listen_sock); /* CLOSE SOCKET */
80     exit(1);
81   }
82
83   /* BINDIND SUCCESSFULLY */
84   printf("SOCKET BINDED TO PORT %d \n", server_port);
85
86   if( listen(listen_sock, 5) < 0 ){
87     /* UNSUCCESSFUL LISTENING ATTEMPT */
88     perror("LISTEN ERROR! \n");
89     close(listen_sock); /* CLOSE SOCKET */
90     exit(1);
91   }
92
93   /* LISTENING SUCCESSFULLY */
94   printf("SERVER READY \n");
```

```
95
96    while(1){
97      /* WAITING FOR NEW CONNECTIONS */
98      printf("SERVER WAITING NOW CLIENT TO GET CONNECTED! \n");
99      accept_socksize = sizeof(listnerclient_addr);
100     accept_sock = accept(listen_sock, (struct sockaddr *)&
        listnerclient_addr, &accept_socksize);
101     printf("CONNECTED TO - %s\n", inet_ntoa(listnerclient_addr.
        sin_addr));
102
103     /* CREATE CHILD PROCESS TO SERVE CLIENT */
104     pid = fork();
105
106     if(pid < 0){
107       perror("FORKING ERROR!");
108       exit(0);
109     }
110
111     if(pid == 0){
112       close(listen_sock);
113       doprocessing(accept_sock);
114       exit(0);
115     }else{
116       close(accept_sock);
117     }
118   }
119
120   printf("TURNING OFF SERVER NO MORE REQUESTS!");
121   close(listen_sock);
122   return 0;
123 }
```

We can run generic TCP Client and connect it to `localhost:6000` — we will use below code for running such client.

3.2.2 Client based on TCP

TCP_Client.c

```
1  /**************************************
2   * Client using TCP.
3   * by https://github.com/karshe
4   * PROJECT : NETWORK PROGRAMMING IN UNIX
5   * USING "networking.h"
6   * ************************************/
7
8  #include "networking.h"
9
10 int main(void){
11   char host[] = "localhost";
12   int port_no = 6000;
13   char buffer[256];
14
```

```
15 int sockfd = TCP_SOCKET_CLIENT();
16  if( TCP_CONNECT_SERVER(host, port_no, sockfd) ){
17       printf("CONNECTED WITH %s ON PORT %d \n", host, port_no)
     ;
18        printf("YOUR MESSAGE - ");
19        bzero(buffer,256);
20        fgets(buffer,255,stdin);
21        TCP_SEND_SERVER(sockfd, buffer);
22        printf("SERVER REPLIED - %s\n", TCP_RECEIVE_SERVER(
    sockfd));
23        TCP_CLOSE_SOCKET(sockfd);
24      }else{
25        printf("NO CONNECTION MADE WITH SERVER ON PORT %d ",
    port_no);
26      }
27  return 0;
28 }
```

Output of server console

```
SOCKET CREATED! WAITING FOR BIND
SOCKET BINDED TO PORT 6000
SERVER READY
SERVER WAITING NOW CLIENT TO GET CONNECTED!
CONNECTED TO - 127.0.0.1
SERVER WAITING NOW CLIENT TO GET CONNECTED!
CONNECTED TO - 127.0.0.1
SERVER WAITING NOW CLIENT TO GET CONNECTED!
CLIENT SAYS: Hello World

CLIENT SAYS: Are you there?
```

1st client console

```
CONNECTED WITH localhost ON PORT 6000
YOUR MESSAGE - Hello World
SERVER REPLIED - THANKS!
YOUR MESSAGE -
```

2nd client console

```
CONNECTED WITH localhost ON PORT 6000
YOUR MESSAGE - Are you there?
SERVER REPLIED - THANKS!
YOUR MESSAGE -
```

Both server and clients are in infinite loop!

3.3 Client and Server program for chatting.

Below is chat agent written in C; which comprises both Chat Server and Chat Client. When user runs himself as server, he allotted with random port and he than accept other *client user* on that port. In same fashion if he runs himself as client – he need to use port of some server in order to initiate chat with respective server. We used `networking.h` in order to get ease while developing.

3.3.1 Chatting Client based on TCP

Chat_Agent.c

```
/*****************************************
 * Chat_Agent.c
 * CHAT AGENT
 * BASED ON TCP
 * DEVELOPED IN UNIX BY SOCKET PROGRAMMING
 * ****************************************/

#include "networking.h"

int ChatAgent(){

 /*CREATE SOCKET AND BIND IT TO SOME PORT*/
 int sockfd;
 int port_no;

 int doingChat = 0;

 /* CHAT COMMANDS */
 char cmd[10];

 char buffer[256];
 char choice;

 do{

   /* HE IS NOT DOING CHAT RIGHT NOW */
   if(!doingChat){
     printf("YOU ARE NOT CONNECTED TO ANY USER.");
     printf("USE (Chat) COMMAND TO START CHAT OR USE (Help) TO
    HELP \n");
     printf("CMD (chat-agent)>> ");
     fgets(cmd, sizeof(cmd), stdin);

     if(strcmp(cmd, "Chat\n") == 0 || strcmp(cmd, "chat\n") ==
    0){
       printf("SPECIFY HIS PORT (chat-agent)>> ");
       scanf("\n%d%*c", &port_no);
       sockfd = TCP_SOCKET_CLIENT();

```

```
38        do{
39          if( TCP_CONNECT_SERVER("localhost", port_no, sockfd) )
    {
40            printf("CONNECTED WITH %s ON PORT %d \n", "localhost
    ", port_no);
41            while(1){
42             printf("MESSAGE (chat-agent)>> ");
43             bzero(buffer,256);
44             fgets(buffer,255,stdin);
45
46             TCP_SEND_SERVER(sockfd, buffer);
47             printf("REPLY (chat-agent)>> %s\n",
    TCP_RECEIVE_SERVER(sockfd));
48           }
49           if(sockfd > 0) { TCP_CLOSE_SOCKET(sockfd); }
50          }else{
51            printf("NO CONNECTION MADE WITH SERVER. DO YOU WANT
    TO RETRY (Y/N) : ");
52            scanf("\n%c%*c", &choice);
53          }
54        }while(choice == 'y' || choice == 'Y');
55
56        printf("Command (Bye, Quit) ? : ");
57        fgets(cmd, sizeof(cmd), stdin);
58        //printf("COMPARE VALUE : %d", strcmp(cmd, "quit\n"));
59
60        if(strcmp(cmd, "Bye\n") == 0 || strcmp(cmd, "bye\n") ==
    0){
61          printf("WE DISCONNECTED YOU FROM %d PORT. \n", port_no
    );
62        }else if(strcmp(cmd, "Quit\n") == 0 || strcmp(cmd, "quit
    \n") == 0){
63          printf("SIGNING OFF \n");
64          break;
65        }else{
66          printf("Didn't got you! Try again! \n");
67        }
68
69       }else if(strcmp(cmd, "Help\n") == 0 || strcmp(cmd, "help\n
    ") == 0){
70          printf("HELP MENU. \n");
71          continue;
72       }else if(strcmp(cmd, "Quit\n") == 0 || strcmp(cmd, "quit\n
    ") == 0){
73          printf("SIGNING OFF. \n");
74          break;
75       }else{
76        printf("YOU ARE ALONE! WAIT UNTIL SOMEONE CONNECTS YOU!
    \n");
77        printf("... \n");
78       }
79     }
80
81
```

```
 }while(1);

 printf("Thanks for using Chat Application!");
 return 0;
}

int ChatServer(int port_no){
  int sockfd = TCP_SOCKET_BIND(port_no);
  int listenfd;
  char buffer[256];

  if(sockfd < 0) exit(1);
  printf("SERVER RUNNING WITH CLIENT ON PORT %d \n",port_no);

  listenfd = TCP_SOCKET_LISTNER(sockfd, 5);
  TCP_SOCKET_CONNECTED_CLIENT(sockfd);

  while(1){
   printf("CLIENT (chat-agent)>> ");
   printf("%s", TCP_READ_LISTENER(listenfd));

   printf("YOU (chat-agent)>> ");
   bzero(buffer,256);
   fgets(buffer,255,stdin);
   TCP_WRITE_LISTENER(listenfd, buffer);
  }

  TCP_CLOSE_SOCKET(listenfd);
  TCP_CLOSE_SOCKET(sockfd);

  return 0;
}

int rand_range(int min_n, int max_n)
{
    return rand() % (max_n - min_n + 1) + min_n;
}

int main(int argc, char *argv[]){
  char s;
  int rand_port;

  printf("CHAT AGENT v0.0.1 \n");

  do{
    printf("WHAT YOU WANT TO SERVE! \n");
    printf("1. CHAT CLIENT (TALK TO PORT) \t 2. CHAT SERVER (
    LISTEN TO PORT) \t 9. QUIT\n");
    printf("OPTION (chat-app)>> ");
    fflush(stdin);
    scanf(" %c%*c", &s);

    if(s == '1'){
      printf("... \n");
```

```
135       printf("ACTIVATED CHAT CLIENT\n");
136       printf("... \n");
137       ChatAgent();
138
139     }else if(s == '2'){
140       printf("... \n");
141       printf("ACTIVATED CHAT SERVER\n");
142       printf("... \n");
143       rand_port = rand_range(2000, 5000);
144       printf("YOUR PORT IS (chat-app)>> %d \n", rand_port);
145       ChatServer(rand_port);
146
147     }else{
148       continue;
149     }
150   }while(s != '9');
151
152   return 0;
153 }
```

Running Chat Agent — We will initiate two console and run one as *chat server* while other one as *chat client*. Chat server would be provided with port — the same port later used by Chat Client and socket communication is established between them.

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "Chat_Agent.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "Chat_Agent" "Chat_Agent.c" -lm
```

Executing `Chat_Agent.c` —

```
karshe@karshe-dell:~/root\$ ./Chat_Agent
```

Running Chat Server

```
CHAT AGENT v0.0.1
WHAT YOU WANT TO SERVE!
1. CHAT CLIENT (TALK TO PORT)
2. CHAT SERVER (LISTEN TO PORT)
9. QUIT
OPTION (chat-app)>> 2
...
ACTIVATED CHAT SERVER
...
YOUR PORT IS (chat-app)>> 3154
SERVER RUNNING WITH CLIENT ON PORT 3154
CLIENT CONNECTED AT PORT 3154
CLIENT (chat-agent)>> Hello Server!
YOU (chat-agent)>> Hello Client!
```

Running Chat Client

```
CHAT AGENT v0.0.1
WHAT YOU WANT TO SERVE!
1. CHAT CLIENT (TALK TO PORT)
2. CHAT SERVER (LISTEN TO PORT)
9. QUIT
OPTION (chat-app)>> 1
...
ACTIVATED CHAT CLIENT
...
YOU ARE NOT CONNECTED TO ANY USER.
USE (Chat) COMMAND TO START CHAT OR USE (Help) TO HELP
CMD (chat-agent)>> Chat
SPECIFY HIS PORT (chat-agent)>> 3154
CONNECTED WITH localhost ON PORT 3154
MESSAGE (chat-agent)>> Hello Server!
REPLY (chat-agent)>> Hello Client!

MESSAGE (chat-agent)>>
```

Now both agents can talk infinitely, unless one of them hung up!

3.4 Retrieve Date and Time using TCP server.

We will create one *time server* and one *generic TCP-based client* — where our client will request server for time and expect formatted output from it. Lets code both agents.

3.4.1 TCP based Time Server

TIME_SERVER.C

```c
/**************************************
 * TCP BASED TIME SERVER
 * by https://github.com/karshe
 * PROJECT : NETWORK PROGRAMMING IN UNIX
 * USING "networking.h"
 * ************************************/


#include "networking.h"
#include <time.h>

int main(void){
 int port_no = 6000;
 int sockfd = TCP_SOCKET_BIND(port_no);
 int listenfd;
 time_t tick;
 char time_str[255];

 if(sockfd < 0) exit(1);
 printf("TIME SERVER ACTIVATED ON PORT %d \n",port_no);

 listenfd = TCP_SOCKET_LISTNER(sockfd, 5);
 TCP_SOCKET_CONNECTED_CLIENT(sockfd);

 tick = time(NULL);
 snprintf(time_str, sizeof(time_str), "%s", ctime(&tick));
 TCP_WRITE_LISTENER(listenfd, time_str);

 TCP_CLOSE_SOCKET(listenfd);
 TCP_CLOSE_SOCKET(sockfd);

 return 0;
}
```

And below is client who wants to know time at server.

3.4.2 TCP based Client

TIME_CLIENT.C

```c
/**************************************
 * TCP BASED TIME CLIENT
 * by https://github.com/karshe
```

```
4  * PROJECT : NETWORK PROGRAMMING IN UNIX
5  * USING "networking.h"
6  * ************************************/
7
8 #include "networking.h"
9
10 int main(void){
11  char host[] = "localhost";
12  int port_no = 6000;
13
14  int sockfd = TCP_SOCKET_CLIENT();
15   if( TCP_CONNECT_SERVER(host, port_no, sockfd) ){
16        printf("CONNECTED WITH %s ON PORT %d \n", host, port_no)
     ;
17         printf("TIME AT SERVER - %s\n", TCP_RECEIVE_SERVER(
     sockfd));
18         TCP_CLOSE_SOCKET(sockfd);
19      }else{
20        printf("NO CONNECTION MADE WITH TIME SERVER ON PORT %d "
     , port_no);
21      }
22  return 0;
23 }
```

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "Time_Server.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "Time_Server" "Time_Server.c" -lm
karshe@karshe-dell:~/root\$
karshe@karshe-dell:~/root\$ gcc -Wall -c "Time_Client.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "Time_Client" "Time_Client.c" -lm
karshe@karshe-dell:~/root\$
karshe@karshe-dell:~/root\$ ./Time_Server
TIME SERVER ACTIVATED ON PORT 6000
CLIENT CONNECTED AT PORT 6000
```

Lets fire client and know what is the time at server.

Executing `Time_Client.c` —

```
karshe@karshe-dell:~/root\$ ./Time_Client
CONNECTED WITH localhost ON PORT 6000
TIME AT SERVER - Tue Nov  3 22:44:09 2015

------------------
(program exited with code: 0)
```

3.5 Client and Server routines showing I/O multiplexing.

Before we start to code our server supporting synchronous I/O multiplexing we will try to cover some theory in order to ease our understanding.

Blocking — We probably noticed that when we run *listener*, it just sits there until a packet arrives. What happened is that it called *recvfrom()* or `read()` and there was no data, and so *recvfrom()* or `read()` is said to **"block"** (that is, sleep there) until some data arrives.

By setting a socket to non-blocking, we can effectively **"poll"** the socket for information. If we try to read from a non-blocking socket and there's no data there, it's not allowed to block—it will return `-1` and `errno` will be set to `EAGAIN` or `EWOULDBLOCK`.

We use `select()` system call for synchronous I/O Multiplexing. It gives us the power to monitor several sockets at the same time. It'll tell us which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions.

Below is TCP Server based on I/O multiplexing —

3.5.1 TCP server based on I/O Multiplexing

Chat_Server_IO_Mult.c

```
/*********************************
 * IO BASED MULTIPLEXING
 * NO NEED FOR WAIT
 * USES SELECT()
 * ********************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include "networking.h"

#define PORT "4000"   // port we're listening on

/* HELPER FUNCTION FOR RETURNING IPv4 or IPv6 HOST ADDRESS */
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }
```

```
28
29      return &(((struct sockaddr_in6*)sa)->sin6_addr);
30  }
31
32  int main(void)
33  {
34      fd_set master;    /* ALL CLIENT WILL BE SAVED HERE */
35      fd_set read_fds;  /* TEMP FD LIST FOR SELECT CALL */
36      int fdmax;        /* MAXIMUM CLIENT FD */
37
38      int listener;     /* LISTENER SOCKET */
39      int newfd;        /* CLIENT SOCKET */
40      struct sockaddr_storage remoteaddr; /* CLIENT ADDRESS */
41      socklen_t addrlen;
42
43      char buf[256];    /* BUFFER FOR CLIENT DATA */
44      int nbytes;
45
46      char remoteIP[INET6_ADDRSTRLEN];
47
48      int yes=1;
49      /* i,j LOOPER, rv DATA CHUNKS RECIVIED COUNTER */
50      int i, j, rv;
51
52      struct addrinfo hints, *ai, *p;
53
54      /* STEP 00 : CLEAT ALL FDS, MASTER + TEMP SETS */
55      FD_ZERO(&master);
56      FD_ZERO(&read_fds);
57
58      /* STEP 01 : GET US A SOCKET & BIND IT */
59      memset(&hints, 0, sizeof hints);
60      hints.ai_family = AF_UNSPEC; /* IPV4 OR IPV6 */
61      hints.ai_socktype = SOCK_STREAM;
62      hints.ai_flags = AI_PASSIVE;
63
64      if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
65          fprintf(stderr, "ERROR: %s\n", gai_strerror(rv));
66          exit(1);
67      }
68
69      /* GETTING ALL LISTENERS */
70      for(p = ai; p != NULL; p = p->ai_next) {
71          listener = socket(p->ai_family, p->ai_socktype, p->
    ai_protocol);
72
73          // IF SOCKET CREATION FAILS
74          if (listener < 0) {
75              continue;
76          }
77
78          // SUPRESS "address already in use" ERROR
79          setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
    sizeof(int));
```

```
80
81          // IF BINDING FAILS
82          if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
83              close(listener);
84              continue;
85          }
86
87          break;
88      }
89
90      // IN CASE WE DIDN'T BINDED AT ALL
91      if (p == NULL) {
92          fprintf(stderr, "FAILED TO BIND!\n");
93          exit(2);
94      }
95
96      freeaddrinfo(ai);
97
98      /* LISTENING... */
99      if (listen(listener, 10) == -1) {
100         perror("listen");
101         exit(3);
102     }
103
104     printf("SERVER UP AND RUNNING \n");
105
106     /* ADD IT TO MASTER SET */
107     FD_SET(listener, &master);
108
109     /* TRACK MAX_LISTENER */
110     fdmax = listener;
111
112     /* LOOP */
113     for(;;) {
114
115         /* COPY OF MASTER SET */
116         read_fds = master;
117
118         /* SELECT FD FROM SET */
119         if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1)
    {
120             perror("select");
121             exit(4);
122         }
123
124         /* RUNNING THROUGH EXISTING CONNECTIONS */
125         /* LOOKING FOR DATA! */
126         for(i = 0; i <= fdmax; i++) {
127
128             if (FD_ISSET(i, &read_fds)) {
129                 /* LOOK, WE GOT ONE! */
130                 if (i == listener) {
131
132                     /* ACCEPT THE CLIENT CONNECTION */
```

```
133                     addrlen = sizeof(remoteaddr);
134                     newfd = accept(listener,(struct sockaddr *)
    &remoteaddr,&addrlen);
135 
136                     if (newfd == -1) {
137                         perror("accept");
138                     } else {
139                         /* CONNECTION ESTABLISHED */
140                         FD_SET(newfd, &master); /* ADDED TO
    MASTER DATA */
141 
142                         if (newfd > fdmax) {    /* KEEP TRACK
    OF MAX */
143                             fdmax = newfd;
144                         }
145 
146                         printf("NEW CONNECTION FROM %s SOCKET %
    d\n",inet_ntop(remoteaddr.ss_family, get_in_addr((struct
    sockaddr*)&remoteaddr), remoteIP, INET6_ADDRSTRLEN), newfd)
    ;
147 
148                     }
149                 } else {
150 
151                     /* HANDLE DATA FROM CLIENT */
152                     if ((nbytes = recv(i, buf, sizeof buf, 0))
    <= 0) {
153 
154                         /* EITHER CONNECTION IS CLOSED OR
    CLIENT SHUT DOWN */
155                         if (nbytes == 0) {
156                             /* CONNECTION TERMINATED */
157                             printf("INFO: CLIENT AT SOCKET %d
    HUNGED UP\n", i);
158                         } else {
159                             perror("recv");
160                         }
161                         close(i);
162                         FD_CLR(i, &master);
163                     } else {
164                         /* GOT DATA FROM CLIENT */
165                         /* j = ALL CONNECTION TO SERVER */
166                         /* i = WE! THE SERVER */
167                         /* listener = CLIENT WHO SEND THE DATA!
     */
168 
169                         for(j = 0; j <= fdmax; j++) {
170                             /* BROADCAST IT, EVERYONE! */
171                             if (FD_ISSET(j, &master)) {
172                                 /* TO ALL OTHER CLIENTS */
173                                 if (j != listener && j != i) {
174                                     if (send(j, buf, nbytes, 0)
    == -1) {
175                                         perror("send");
```

```
                                        }
                                    }
                                }
                            }
            }

                    }
                }
            }
    }/* EVERYTHING HAS END, EVEN OUR SUN HAS! */

    return 0;
}
```

Below is our generic TCP Client, lets put code for TCP Client.

3.5.2 TCP Client

TCP_CLIENT.C

```
#include "networking.h"

int main(void){
 char host[] = "localhost";
 int port_no = 4000;
 char buffer[256];
 char choice;

 int sockfd = TCP_SOCKET_CLIENT();

 do{
   if( TCP_CONNECT_SERVER(host, port_no, sockfd) ){
     printf("CONNECTED WITH %s ON PORT %d \n", host, port_no);
     while(1){
      printf("YOUR MESSAGE - ");
      bzero(buffer,256);
      fgets(buffer,255,stdin);

      TCP_SEND_SERVER(sockfd, buffer);
      printf("SERVER REPLIED - %s", TCP_RECEIVE_SERVER(sockfd))
    ;
    }
    if(sockfd > 0) { TCP_CLOSE_SOCKET(sockfd); }
   }else{
     printf("NO CONNECTION MADE WITH SERVER. DO YOU WANT TO
    RETRY (Y/N) : ");
     scanf("\n%c%*c", &choice);
   }
 }while(choice == 'y' || choice == 'Y');

 return 0;
}
```

Lets compile and run both, *server* and `client` terminals.

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "Chat_Server_IO_Mult.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "Chat_Server_IO_Mult"
"Chat_Server_IO_Mult.c" -lm
karshe@karshe-dell:~/root\$
```

We will run server first then initiate two instances of client terminals. On successful connection between all nodes and server; we will send data from `client #1` to `client #2` and *vice versa*.

Executing `Chat_Server_IO_Mult.c` —

```
karshe@karshe-dell:~/root\$ ./Chat_Server_IO_Mult
SERVER UP AND RUNNING
NEW CONNECTION FROM 127.0.0.1 SOCKET 4
NEW CONNECTION FROM 127.0.0.1 SOCKET 5
```

Executing Client #1 —

```
karshe@karshe-dell:~/root\$ ./TCP_Client
CONNECTED WITH localhost ON PORT 4000
YOUR MESSAGE - Hello
SERVER REPLIED - World
YOUR MESSAGE -
```

Executing Client #2 —

```
karshe@karshe-dell:~/root\$ ./TCP_Client
CONNECTED WITH localhost ON PORT 4000
YOUR MESSAGE - World
SERVER REPLIED - Hello
YOUR MESSAGE -
```

As one of the client closes its terminal we get following on server termnial —

```
karshe@karshe-dell:~/root\$ ./TCP_Client
SERVER UP AND RUNNING
NEW CONNECTION FROM 127.0.0.1 SOCKET 4
NEW CONNECTION FROM 127.0.0.1 SOCKET 5
INFO: CLIENT AT SOCKET 5 HUNGED UP
```

When all of its nodes are closed, there are no entries in `master` list. Hence server console waits for any new connection to get establish. Lets create client one more time — and we see following output on terminal.

```
karshe@karshe-dell:~/root\$ ./TCP_Client
SERVER UP AND RUNNING
NEW CONNECTION FROM 127.0.0.1 SOCKET 4
NEW CONNECTION FROM 127.0.0.1 SOCKET 5
INFO: CLIENT AT SOCKET 5 HUNGED UP
INFO: CLIENT AT SOCKET 4 HUNGED UP
NEW CONNECTION FROM 127.0.0.1 SOCKET 4
```

3.6 Echo client and server program using UNIX domain stream socket.

A UNIX domain socket or IPC socket *(inter-process communication socket)* is a data communications endpoint for exchanging data between processes executing on the same host operating system.

In below illustration we will use another header file `n_headers.h` which is defined for handling header file.[2]

3.6.1 UNIX domain stream socket based server

UNIX_Domain_Socket_Server.c

```
#include "n_headers.h"

#define NAME "UN_SOCK"

int main(int argc, char *argv[]){
    int sock, msgsock, rval;
    struct sockaddr_un server;
    char buf[1024];
    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("CAN'T OPEN UNIX STREAM SOCKET");
        exit(1);
    }

    server.sun_family = AF_UNIX;
    strcpy(server.sun_path, NAME);

    if (bind(sock, (struct sockaddr *) &server, sizeof(struct
   sockaddr_un))) {
        perror("ERROR IN BINDING SOCKET!");
        exit(1);
    }

    printf("SOCKET NAME : %s\n", server.sun_path);
    listen(sock, 5);

    for (;;) {
        msgsock = accept(sock, 0, 0);
        if (msgsock == -1)
            perror("ERROR IN ACCEPTING");
        else do {
            bzero(buf, sizeof(buf));
            if ((rval = read(msgsock, buf, 1024)) < 0)
                perror("READING CLIENT ERROR!");
            else if (rval == 0)
                printf("CONNECTION TERMINATED\n");
            else
```

[2]Source codes of both header files are given at the end of chapter.

```
37                  printf("CLIENT SAYS - %s\n", buf);
38          } while (rval > 0);
39          close(msgsock);
40      }
41
42      close(sock);
43      unlink(NAME);
44      return 0;
45  }
```

In same way we will create client program based on UNIX Domain Socket.

3.6.2 UNIX domain stream socket based client

UNIX_Domain_Socket_Client.c

```
1   #include "n_headers.h"
2
3   #define DATA "NETWORK PROGRAMMING IS FUN!"
4
5   int main(int argc, char *argv[]){
6       int sock;
7       struct sockaddr_un server;
8
9       sock = socket(AF_UNIX, SOCK_STREAM, 0);
10      if (sock < 0) {
11          perror("CAN'T OPEN UNIX STREAM SOCKET");
12          exit(1);
13      }
14
15      server.sun_family = AF_UNIX;
16      strcpy(server.sun_path, "UN_SOCK"); /* FROM SERVER */
17
18      if (connect(sock, (struct sockaddr *) &server, sizeof(
    struct sockaddr_un)) < 0) {
19          close(sock);
20          perror("ERROR IN CONNECTION WITH SERVER! ");
21          exit(1);
22      }
23      if (write(sock, DATA, sizeof(DATA)) < 0){
24          perror("WRITING ERROR ON SERVER SOCKET! ");
25          exit(1);
26    }
27
28    printf("WROTE TO SERVER!");
29
30      close(sock);
31      return 0;
32  }
```

Lets compile and run both server and client — and *enjoy inter process communication!*

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "UNIX_Domain_Socket_Server.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "UNIX_Domain_Socket_Server"
"UNIX_Domain_Socket_Server.c" -lm
karshe@karshe-dell:~/root\$
karshe@karshe-dell:~/root\$ gcc -Wall -c "UNIX_Domain_Socket_Client.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "UNIX_Domain_Socket_Client"
"UNIX_Domain_Socket_Client.c" -lm
karshe@karshe-dell:~/root\$
```

Executing server terminal —

```
karshe@karshe-dell:~/root\$ ./UNIX_Domain_Socket_Server
SOCKET NAME : UN_SOCK
```

As soon as we execute server console, we can see a file name `UN_SOCK` has been created in working directory. We can confirm using following command in UNIX —

```
karshe@karshe-dell:~/root\$ ls -l UN_SOCK
srwxrwxr-x 1 karshe karshe 0 Nov  3 23:53 UN_SOCK
```

Now execute client terminal which will communicate with server using `UN_SOCK` —

```
karshe@karshe-dell:~/root\$ ./UNIX_Domain_Socket_Client
WROTE TO SERVER!
```

And we can confirm communication by observing server console.

```
karshe@karshe-dell:~/root\$ ./UNIX_Domain_Socket_Server
SOCKET NAME : UN_SOCK
CLIENT SAYS - NETWORK PROGRAMMING IS FUN!
CONNECTION TERMINATED
```

3.7 Implement the Remote Command Execution.

We will try to demonstrate above objective using `system()` call defined in `gcc` on UNIX.

Below code shows server running at `localhost` and waiting client to send commands via socket and then executing the same in its terminal (i.e. *server's terminal*).

3.7.1 Remote Server

CONSOLE_SERVER.C

```c
/**************************************
 * LETS RUN COMMANDS!
 * by https://github.com/karshe
 * PROJECT : NETWORK PROGRAMMING IN UNIX
 * ************************************/

#include "networking.h"

int main(void){
 int port_no = 6000;
 int sockfd = TCP_SOCKET_BIND(port_no);
 int listenfd;
 int firstRun = 1;
 char cmd[10];

 if(sockfd < 0) exit(1);

 printf("FILE READER SERVER READY! \n");

 listenfd = TCP_SOCKET_LISTNER(sockfd, 5);
 TCP_SOCKET_CONNECTED_CLIENT(sockfd);

 while(1){
    /* FIRST RUN SCRIPT */
    if(firstRun){
       TCP_WRITE_LISTENER(listenfd, "WELCOME TO SERVER");
       firstRun = 0;

    }
    /* ACCEPT COMMANDS HERE*/
    strcpy(cmd, TCP_READ_LISTENER(listenfd));
    system(cmd);
    TCP_WRITE_LISTENER(listenfd, "EXECUTED!");
   }

 TCP_CLOSE_SOCKET(listenfd);
 TCP_CLOSE_SOCKET(sockfd);

 return 0;
}
```

3.7.2 Client who commands

Console_Client.c

```
/**************************************
 * GIVE SERVER ORDER
 * by https://github.com/karshe
 * PROJECT : NETWORK PROGRAMMING IN UNIX
 * USING "networking.h"
 * ************************************/

#include "networking.h"

int main(void){
 char host[] = "localhost";
 int port_no = 6000;
 char buffer[256];

 int sockfd = TCP_SOCKET_CLIENT();
  if( TCP_CONNECT_SERVER(host, port_no, sockfd) ){
       printf("CONNECTED WITH SERVER ON PORT %d \n", port_no);


        while(1){
          printf("REPLY (SERVER) >>> %s\n", TCP_RECEIVE_SERVER(
   sockfd));
          printf("COMMAND (CLIENT) >>> ");
          bzero(buffer,256);
          fgets(buffer,255,stdin);
          TCP_SEND_SERVER(sockfd, buffer);

        }
        TCP_CLOSE_SOCKET(sockfd);
     }else{
       printf("NO CONNECTION MADE WITH SERVER ON PORT %d ",
   port_no);
     }
 return 0;
}
```

Lets compile both server and client.

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "Console_Server.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "Console_Server"
"Console_Server.c" -lm
karshe@karshe-dell:~/root\$
karshe@karshe-dell:~/root\$ gcc -Wall -c "Console_Client.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "Console_Client"
"Console_Client.c" -lm
karshe@karshe-dell:~/root\$
```

Executing server and waiting for client —

```
karshe@karshe-dell:~/root\$ ./Console_Server
FILE READER SERVER READY!
```

Now we will execute client and try to send `who me i` command. Lets try —

Executing client —

```
karshe@karshe-dell:~/root\$ ./Console_Client
CONNECTED WITH SERVER ON PORT 6000
REPLY (SERVER) >>> WELCOME TO SERVER
COMMAND (CLIENT) >>> who me i
REPLY (SERVER) >>> EXECUTED!
COMMAND (CLIENT) >>>
```

We got acknowledgement from server that command has been successfully executed. Lets see what's on server's console —

```
karshe@karshe-dell:~/root\$ ./Console_Server
FILE READER SERVER READY!
CLIENT CONNECTED AT PORT 6000
karshe   pts/2        2015-11-04 00:11 (:0.0)
```

Enjoy the commanding server!

3.8 Hexadecimal Converter

Write a client program that gets a number from the user and sends the number to server for conversion into hexadecimal and gets the result from the server.

Below is the code for server side application —

3.8.1 Hexadecimal Conversion Server

TCP_Calc_Server.c

```c
/**************************************
 * SERVER TO CALCULATE HEX
 * by https://github.com/karshe
 * PROJECT : NETWORK PROGRAMMING IN UNIX
 * ************************************/

#include "networking.h"

int main(void){
 int port_no = 6000;
 int sockfd = TCP_SOCKET_BIND(port_no);
 int listenfd;
 int num;
 char hex[10];
 char integer[10];

 if(sockfd < 0) exit(1);

 printf("SERVER READY! \n");

 listenfd = TCP_SOCKET_LISTNER(sockfd, 5);
 TCP_SOCKET_CONNECTED_CLIENT(sockfd);

 strcpy(integer, TCP_READ_LISTENER(listenfd));

 printf("REQUESTED NUMBER TO CONVERT : %s\n", integer);

 num = atoi(integer);
 sprintf(hex, "%x", num);

 TCP_WRITE_LISTENER(listenfd, hex);

 TCP_CLOSE_SOCKET(listenfd);
 TCP_CLOSE_SOCKET(sockfd);

 return 0;
}
```

And below is the code for client — who is just asking for help from server.

3.8.2 Innocent Client

TCP_Calc_Client.c

```
/**************************************
 * LETS DO HEXA HEXA!
 * by https://github.com/karshe
 * PROJECT : NETWORK PROGRAMMING IN UNIX
 * USING "networking.h"
 * ************************************/

#include "networking.h"

int main(void){
 char host[] = "localhost";
 int port_no = 6000;
 char buffer[256];

 int sockfd = TCP_SOCKET_CLIENT();
  if( TCP_CONNECT_SERVER(host, port_no, sockfd) ){
       printf("CONNECTED WITH CALCULATOR ON PORT %d \n",
    port_no);

       /* ASK FOR NUMBER */
       printf("GIVE NUMBER TO CONVERT : ");
       bzero(buffer,256);
       fgets(buffer,255,stdin);
       TCP_SEND_SERVER(sockfd, buffer);

       printf("CONVERTED HEX IS - %s\n", TCP_RECEIVE_SERVER(
    sockfd));
       TCP_CLOSE_SOCKET(sockfd);
     }else{
       printf("NO CONNECTION MADE WITH SERVER ON PORT %d ",
    port_no);
     }
 return 0;
}
```

Lets bake the cake — I mean lets compile both server and client —

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "TCP_Calc_Server.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "TCP_Calc_Server"
"TCP_Calc_Server.c" -lm
karshe@karshe-dell:~/root\$
karshe@karshe-dell:~/root\$ gcc -Wall -c "TCP_Calc_Client.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "TCP_Calc_Client"
"TCP_Calc_Server.c" -lm
karshe@karshe-dell:~/root\$
```

Server up and running — give him number!

```
karshe@karshe-dell:~/root\$ ./TCP_Calc_Server
```

```
SERVER READY!
```

Giving number to server —

```
karshe@karshe-dell:~/root\$ ./TCP_Calc_Client
CONNECTED WITH CALCULATOR ON PORT 6000
GIVE NUMBER TO CONVERT : 2356
CONVERTED HEX IS - 934


------------------
(program exited with code: 0)
```

Server also said something —

```
karshe@karshe-dell:~/root\$ ./TCP_Calc_Server
SERVER READY!
CLIENT CONNECTED AT PORT 6000
REQUESTED NUMBER TO CONVERT : 2356


------------------
(program exited with code: 0)
```

And that's how friend — *"How I converted hexadecimal numbers."*

3.9 Lets find IP of Google.com

Below is code written in C which basically converts Domain Name into its IP. So here is code —

3.9.1 DNS Agent

DNS_Agent.c

```
/*****************************
 * WHERE IS GOOGLE.COM
 * ***************************/

#include <stdio.h>
#include <netdb.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>

int main(int argc, char*argv[]) {
  char *host = "www.google.com";
  struct hostent *h=gethostbyname(host);
  if (h)
   printf("GOOGLE.COM LIVES HERE : %s\n", inet_ntoa(*(struct
    in_addr*)h->h_addr)); /* DNS! */
  else
   printf("%s\n", hstrerror(h_errno));
  return 0;
}
```

Compile & Run —

```
karshe@karshe-dell:~/root\$ gcc -Wall -c "DNS_Agent.c"
karshe@karshe-dell:~/root\$ gcc -Wall -o "DNS_Agent"
"DNS_Agent.c" -lm
karshe@karshe-dell:~/root\$
karshe@karshe-dell:~/root\$ ./DNS_Agent
GOOGLE.COM LIVES HERE : 216.58.196.100
karshe@karshe-dell:~/root\$
```

3.10 Mysteries of Header files

Below are source code for `networking.h`

3.10.1 NETWORKING.H — Helper for TCP Sockets

NETWORKING.H

```
/*********************************************************
 * NETWORKING.H
 * COLLECTION OF TCP/UDP BASED SERVER-CLIENT SOCKET CODES
 * *******************************************************/

#ifndef NETWORKING_H_
#define NETWORKING_H_

#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

#define author "https://github.com/karshe/unixnetworkprograms"

/*************************************
 * DO INCLUDE FOLLOWING HEADER FILES
 * #include <stdio.h>
 * #include <stdlib.h>
 * #include <netdb.h>
 * #include <netinet/in.h>
 * #include <sys/socket.h>
 * #include <unistd.h>
 * #include <string.h>
 * ***********************************/

/* TCP FAMILY IPV4 TYPE */

/*************************************
 * CREATE SOCKET BINDED TO PORT - SERVER
 * RETURN SOCKET FILE DESC
 * ***********************************/

int TCP_SOCKET_BIND(int portno){
    int sockfd;
    struct sockaddr_in serv_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if(sockfd < 0){
       perror("ERROR IN CREATING SOCKET!");
       exit(1);
     }
     bzero((char *) &serv_addr, sizeof(serv_addr));
```

```
47      serv_addr.sin_family = AF_INET; /*TCP*/
48      serv_addr.sin_addr.s_addr = INADDR_ANY;
49      serv_addr.sin_port = htons(portno); /* PORT */
50
51      if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(
        serv_addr)) < 0) {
52          perror("CAN'T BIND THE SOCKET!");
53          exit(1);
54         }
55
56       return sockfd;
57   }
58
59
60 /*************************************
61  * CREATE LISTENER TO SOCKET WITH BACKLOG - SERVER
62  * RETURN LISTENER SOCKET FILE DESC
63  * ***********************************/
64
65 int TCP_SOCKET_LISTNER(int sockfd, int backlog){
66    int listnerfd;
67    struct sockaddr_in listnerclient_addr;
68    socklen_t listnersize;
69
70    listen(sockfd, backlog);
71    listnersize = sizeof(listnerclient_addr);
72    listnerfd = accept(sockfd, (struct sockaddr *)&
       listnerclient_addr, &listnersize);
73
74    if(listnerfd < 0){
75      perror("CAN'T ACCEPT CONNECTION FROM CLIENT!");
76      exit(1);
77    }
78    return listnerfd;
79 }
80
81 /*************************************
82  * READ FROM LISTENER SOCKET - SERVER
83  * RETURN BUFFER READ FROM CLIENT
84  * ***********************************/
85
86 char *TCP_READ_LISTENER(int lisenfd){
87   int t;
88   static char buffer[256];
89
90   bzero(buffer, 256);
91   t = read(lisenfd, buffer, 255);
92
93   if(t < 1){
94      printf("CAN'T READ FROM CLIENT!");
95      exit(1);
96   }
97   return(buffer);
98 }
```

```
99
100 /*************************************
101  * WRITE ON SOCKET - SERVER
102  * SENDS DATA TO CLIENT
103  * ***********************************/
104
105 void TCP_WRITE_LISTENER(int listenfd, char buff[]){
106     int t;
107     char buffer[256];
108     bzero(buffer, 256);
109     strcpy(buffer, buff);
110     t = write(listenfd, buffer, strlen(buffer)); /* WRITE ON
        CLIENT SOCKET */
111      if(t < 1){
112        printf("I CAN'T WRITE ON SOCKET!");
113        exit(1);
114      }
115 }
116
117 /*************************************
118  * GET INFO OF CLIENT CONNECTED - SERVER
119  * PRINTS CLIENT CONNECTED TO WHICH PORT
120  * ***********************************/
121
122 void TCP_SOCKET_CONNECTED_CLIENT(int sockfd){
123   struct sockaddr_in sin;
124   socklen_t len = sizeof(sin);
125   if (getsockname(sockfd, (struct sockaddr *)&sin, &len) == -1)
        {
126     perror("I WAS UNABLE TO CALL getsockname");
127     exit(1);
128   }else{
129     printf("CLIENT CONNECTED AT PORT %d\n", ntohs(sin.sin_port)
        );
130   }
131 }
132
133 /*************************************
134  * CUSTOM COMMAND FOR CHAT SERVER
135  * ***********************************/
136  int TCP_SOCKET_CLIENT_CONNECTED(int sockfd){
137   struct sockaddr_in sin;
138   socklen_t len = sizeof(sin);
139   if (getsockname(sockfd, (struct sockaddr *)&sin, &len) == -1)
        {
140     perror("I WAS UNABLE TO CALL getsockname");
141     exit(1);
142   }else{
143     printf("CLIENT CONNECTED AT PORT %d\n", ntohs(sin.sin_port)
        );
144     return 0;
145   }
146 }
147
```

```
148 /*************************************
149  * CLOSES SOCKET
150  * ************************************/
151
152 void TCP_CLOSE_SOCKET(int sockid){
153   shutdown(sockid, 1);
154   }
155
156 /*************************************
157  * CREATE SOCKET FOR - CLIENT
158  * RETURN SOCKET FILE DESC
159  * ************************************/
160 int TCP_SOCKET_CLIENT(){
161   int sockfd;
162   sockfd = socket(AF_INET, SOCK_STREAM, 0);
163    if(sockfd < 0){
164      perror("ERROR IN CREATING SOCKET!");
165      exit(1);
166    }
167    return sockfd;
168 }
169
170 /*************************************
171  * CONNECTS TO SERVER (LOCATION, PORT NO, CLIENT SOCKET)
172  * RETURN 1 IF CONNECTION MADE SUCCESSFULL
173  * ************************************/
174
175 int TCP_CONNECT_SERVER(char loc[], int portno, int socketfd){
176   struct sockaddr_in client_addr;
177   struct hostent *server;
178   char hostloc[256];
179   bzero(hostloc, 256);
180   strcpy(hostloc, loc);
181
182   server = gethostbyname(hostloc);
183
184   if(server == NULL){
185     printf("CAN'T CONNCET TO SPECIFIED SERVER");
186     exit(0);
187   }
188
189   bzero((char *) &client_addr, sizeof(client_addr)); /*
      NULLIFIED! */
190   client_addr.sin_family = AF_INET;
191   bcopy((char *)server->h_addr, (char *)&client_addr.sin_addr.
      s_addr, server->h_length);
192   client_addr.sin_port = htons(portno);
193
194   if (connect(socketfd, (struct sockaddr*)&client_addr, sizeof(
      client_addr)) < 0) {
195       return 0;
196    }
197
198    return 1;
```

```
199 }
200
201 /*************************************
202  * SENDS DATA TO SPECIFIED SOCKET - FROM CLIENT TO SERVER
203  * RETURNS NOTHING
204  * ***********************************/
205
206 void TCP_SEND_SERVER(int socketfd, char buff[]){
207     int t;
208     char buffer[256];
209     bzero(buffer, 256);
210     strcpy(buffer, buff);
211     t = write(socketfd, buff, strlen(buff));
212
213
214     if (t < 0) {
215       printf("\nCLIENT WAS UNABLE TO WRITE ON SERVER SOCKET!");
216       exit(1);
217     }
218
219 }
220
221 /*************************************
222  * RECEIVE DATA FROM SERVER - CLIENT
223  * RETURNS DATA
224  * ***********************************/
225
226 char *TCP_RECEIVE_SERVER(int socketfd){
227     int t;
228     static char buffer[256];
229     bzero(buffer,256);
230
231     /* READ FROM SERVER */
232     t = read(socketfd, buffer, 255);
233
234     if (t < 1) {
235       printf("CONNECTION BETWEEN SERVER AND CLIENT LOST!");
236       exit(1);
237     }
238
239     return(buffer);
240 }
241
242 #endif //
243 /**** https://github.com/karshe ***/
```

And of-course, below is *infamous* — `n_headers.h`

3.10.2 Collection

N__HEADERS.H

```
#ifndef NHEADERS_H_
#define NHEADERS_H_

#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/un.h> /* UNIX domain sockets */
#include <unistd.h>
#include <string.h>

#define author "https://github.com/karshe/unixnetworkprograms"

/* I KNOW ITS NOT NO MAGIC HERE */

#endif //
```

3.11 Windows Socket Programming

3.11.1 Winsock Agent

HELLO_WINSOCK.CPP

```
/*********************************************
*** WINSOCK BASED TCP CLIENT/SERVER APPLICATION
*** https://github.com/karshe
*********************************************/

// HEADER FILES
#include <stdio.h>

/* SERVER APP */
int ServerApp();
/* CLIENT APP */
int ClientApp();


/* MAIN - Hello_Winsock */
int main(){

    /** VARIABLES **/
    int choice;

    printf("HELLO WINSOCK!\n");

    do{
        printf("SELECT BELOW (1-3)\n");
        printf("1. SERVER \t 2. CLIENT \t 3.EXIT\n");
        scanf_s("%d", &choice);

        if(choice == 1){
            ServerApp();
        }else if(choice == 2){
            ClientApp();
        }

    }while(choice != 3);


    return 0;
}
```

3.11.2 Winsock Client

HELLO_CLIENT.CPP

```
/*********************************************
*** CLIENT APPLICATION
*** https://github.com/karshe
*********************************************/
#include <winsock2.h>
```

```
6  #include <ws2tcpip.h>
7  #include <stdio.h>
8
9  /* #pragma comment indicates to the linker that the Ws2_32.lib
       file is needed. */
10 #pragma comment(lib, "Ws2_32.lib")
11
12 /* Every socket is binded to some port */
13 #define DEFAULT_PORT "27015"
14 #define DEFAULT_SERVERHOST "127.0.0.1"
15 #define DEFAULT_BUFLEN 512
16
17 int ClientApp(){
18     printf("CLIENT APPLICATION STARTED!\n");
19
20     WSADATA wsaData; /* Create WSADATA object */
21
22     int iResult;
23
24     /* Initialize Winsock */
25     /* MAKEWORD(2,2)
26     ** parameter of WSAStartup makes a request
27     ** for version 2.2 of Winsock on the system
28     */
29     iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
30     if (iResult != 0) {
31         printf("WSAStartup FAILED WITH ERROR: %d\n", iResult);
32         return 1;
33     }
34
35     printf("INITIALIZATION SUCCESSFUL WITH WSAStartup = %d\n",
       iResult);
36     struct addrinfo *result = NULL, *ptr = NULL, hints;
37
38     ZeroMemory( &hints, sizeof(hints) );
39     hints.ai_family = AF_UNSPEC;
40     hints.ai_socktype = SOCK_STREAM;
41     hints.ai_protocol = IPPROTO_TCP;
42
43     // Resolve the server address and port
44     iResult = getaddrinfo(DEFAULT_SERVERHOST, DEFAULT_PORT, &
       hints, &result);
45     if (iResult != 0) {
46         printf("getaddrinfo FAILED WITH CODE : %d\n", iResult);
47         WSACleanup();
48         return 1;
49     }
50     SOCKET ConnectSocket = INVALID_SOCKET;
51
52     // Attempt to connect to the first address returned by
53     // the call to getaddrinfo
54     ptr=result;
55
56     // Create a SOCKET for connecting to server
```

```
57      ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,ptr
        ->ai_protocol);
58
59      if (ConnectSocket == INVALID_SOCKET) {
60          printf("ERROR at socket() WITH CODE: %ld\n",
        WSAGetLastError());
61          freeaddrinfo(result);
62          WSACleanup();
63          return 1;
64      }
65
66      // Connect to server.
67      iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->
        ai_addrlen);
68      if (iResult == SOCKET_ERROR) {
69          closesocket(ConnectSocket);
70          ConnectSocket = INVALID_SOCKET;
71      }
72
73      // Should really try the next address returned by
        getaddrinfo
74      // if the connect call failed
75      // But for this simple example we just free the resources
76      // returned by getaddrinfo and print an error message
77
78      freeaddrinfo(result);
79
80      if (ConnectSocket == INVALID_SOCKET) {
81          printf("UNABLE TO CONNECT TO SERVER!\n");
82          WSACleanup();
83          return 1;
84      }
85
86      /*** SEND & RECEIVE FROM CLIENT ***/
87      int recvbuflen = DEFAULT_BUFLEN;
88
89      char *sendbuf = "HELLO SERVER!";
90      char recvbuf[DEFAULT_BUFLEN];
91
92      // Send an initial buffer
93      iResult = send(ConnectSocket, sendbuf, (int) strlen(sendbuf
        ), 0);
94      if (iResult == SOCKET_ERROR) {
95          printf("SEND FAILED: %d\n", WSAGetLastError());
96          closesocket(ConnectSocket);
97          WSACleanup();
98          return 1;
99      }
100
101     printf("BYTES SENT : %ld\n", iResult);
102
103     // Receive data until the server closes the connection
104     do {
105         iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
```

```
106         if (iResult > 0){
107             printf("BYTES RECEIVED : %d\n", iResult);
108             printf("SERVER SAID :  %s\n", recvbuf);
109         }
110         else if (iResult == 0)
111             printf("CONNECTION CLOSED!\n");
112         else
113             printf("recv FAILED WITH CODE : %d\n",
    WSAGetLastError());
114     } while (iResult > 0);
115     /** END LOGIC  ***/
116 
117     // shutdown the send half of the connection since no more
    data will be sent
118     iResult = shutdown(ConnectSocket, SD_SEND);
119     if (iResult == SOCKET_ERROR) {
120         printf("SHUTDOWN FAILED WITH CODE : %d\n",
    WSAGetLastError());
121         closesocket(ConnectSocket);
122         WSACleanup();
123         return 1;
124     }
125 
126     closesocket(ConnectSocket);
127     WSACleanup();
128 
129     return 0;
130 
131 }
```

3.11.3 Winsock Server

HELLO_SERVER.CPP

```
1  /*********************************************
2  *** SERVER APPLICATION
3  *** https://github.com/karshe
4  *********************************************/
5  #include <winsock2.h>
6  #include <ws2tcpip.h>
7  #include <stdio.h>
8  
9  /* #pragma comment indicates to the linker that the Ws2_32.lib
       file is needed. */
10 #pragma comment(lib, "Ws2_32.lib")
11 
12 #define DEFAULT_BUFLEN 512
13 #define DEFAULT_PORT "27015"
14 
15 int ServerApp(){
16     printf("SERVER APPLICATION STARTED!\n");
17 
18     /** STEP 0 : VARIABLES **/
```

```
19      WSADATA wsaData; //Information of Windows Socket
20      int iResult;
21
22      //Socket addrinfo structure
23      struct addrinfo *result = NULL, *ptr = NULL, hints;
24
25      //Sockets
26      SOCKET ListenSocket = INVALID_SOCKET; /* SOCKET object for
        the server to listen for client connections. */
27      SOCKET ClientSocket = INVALID_SOCKET; /* CLIENT LISTENER */
28
29      //During communication
30      int iSendResult;
31      char recvbuf[DEFAULT_BUFLEN];
32      int recvbuflen = DEFAULT_BUFLEN;
33
34      /** STEP 1 : INIT WINSOCK
35      Call WSAStartup and return its value as an integer and
        check for errors.
36      WSAStartup function is called to initiate use of WS2_32.dll
        .
37      **/
38
39      iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
40      if (iResult != 0) {
41          printf("WSAStartup FAILED WITH ERROR: %d\n", iResult);
42          return 1;
43      }
44
45      /** STEP 2 : SOCKET ADDR INIT
46      After initialization, a SOCKET object must be instantiated
        for use by the server.
47      AF_INET is used to specify the IPv4 address family.
48      SOCK_STREAM is used to specify a stream socket.
49      IPPROTO_TCP is used to specify the TCP protocol .
50      AI_PASSIVE flag indicates the caller intends to use the
        returned socket address structure in a call to the bind
        function.
51      **/
52      ZeroMemory(&hints, sizeof (hints));
53      hints.ai_family = AF_INET;
54      hints.ai_socktype = SOCK_STREAM;
55      hints.ai_protocol = IPPROTO_TCP;
56      hints.ai_flags = AI_PASSIVE;
57
58      /** Resolve the local address and port to be used by the
        server **/
59      iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
60      if (iResult != 0) {
61          printf("RESOLUTION FAILED! RETURN CODE : %d\n", iResult
        );
62          WSACleanup();
63          return 1;
64      }
```

```
65
66      /** STEP 3 : SOCKET OBJECT CREATION
67      Call the socket function and return its value to the
68      ListenSocket variable. For this server application,
69      use the first IP address returned by the call to
        getaddrinfo
70      that matched the address family, socket type,
71      and protocol specified in the hints parameter.
72      **/
73      ListenSocket = socket(result->ai_family, result->
        ai_socktype, result->ai_protocol);
74      if (ListenSocket == INVALID_SOCKET) {
75          printf("ERROR IN SOCKET CREATION!\n %ld\n",
        WSAGetLastError());
76          freeaddrinfo(result);
77          WSACleanup();
78          return 1;
79      }
80
81      printf("SOCKET CREATED, NOW BINDING TO PORT "DEFAULT_PORT"\
        n");
82
83      /** STEP 4 : BIND SOCKET
84      The sockaddr structure holds information regarding the
        address family,
85      IP address, and port number.
86      Call the bind function, passing the created socket and
        sockaddr structure
87      returned from the getaddrinfo function as parameters.
88      **/
89      iResult = bind( ListenSocket, result->ai_addr, (int)result
        ->ai_addrlen);
90      if (iResult == SOCKET_ERROR) {
91          printf("BINDING UNSUCCESSFUL! ERROR: %d\n",
        WSAGetLastError());
92          freeaddrinfo(result);
93          closesocket(ListenSocket);
94          WSACleanup();
95          return 1;
96      }
97      /** Once the bind function is called, the address
        information
98      returned by the getaddrinfo function is no longer needed.
99      **/
100     freeaddrinfo(result);
101     printf("BINDED TO "DEFAULT_PORT"\n");
102
103     /** STEP 5 : START LISTENING
104     Call the listen function, passing as parameters the
105     created socket and a value for the backlog, maximum
106     length of the queue of pending connections to accept.
107     In this example, the backlog parameter was set to SOMAXCONN
        .
108     This value is a special constant that instructs the
```

```
109     Winsock provider for this socket to allow a maximum
110     reasonable number of pending connections in the queue.
111     **/
112     if ( listen( ListenSocket, SOMAXCONN ) == SOCKET_ERROR ) {
113         printf( "LISTEN FAILED WITH ERROR : %ld\n",
    WSAGetLastError() );
114         closesocket(ListenSocket);
115         WSACleanup();
116         return 1;
117     }
118 
119     /** STEP 6 : ACCEPT CONNECTION **/
120 
121     // Accept a client socket
122     ClientSocket = accept(ListenSocket, NULL, NULL);
123     if (ClientSocket == INVALID_SOCKET) {
124         printf("ACCEPT FAILED WITH CODE : %d\n",
    WSAGetLastError());
125         closesocket(ListenSocket);
126         WSACleanup();
127         return 1;
128     }
129 
130     /** STEP 7 : LETS TALK!
131     ** Receive until the peer shuts down the connection
132     **/
133     do {
134 
135         iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
136         if (iResult > 0) {
137             printf("BYTES GOT FROM CLIENT: %d\n", iResult);
138             printf("CLIENT SAID :  %s\n", recvbuf);
139 
140             // Echo the buffer back to the sender
141             iSendResult = send(ClientSocket, "THANK YOU CLIENT!
    ", sizeof("THANK YOU CLIENT!"), 0);
142             if (iSendResult == SOCKET_ERROR) {
143                 printf("SEND FAILED: %d\n", WSAGetLastError());
144                 closesocket(ClientSocket);
145                 WSACleanup();
146                 return 1;
147             }
148             printf("BYTES SEND TO CLIENT : %d\n", iSendResult);
149         } else if (iResult == 0)
150             printf("CONNECTION CLOSING...\n");
151         else {
152             printf("LOOKS LIKE CLIENT HUNGED UP!\nSTATUS CODE :
     %d\n", WSAGetLastError());
153             closesocket(ClientSocket);
154             WSACleanup();
155             return 1;
156         }
157 
158     } while (iResult > 0);
```

```
159
160     /** STEP 9 : SHUTDOWN SOCKET **/
161      iResult = shutdown(ClientSocket, SD_SEND);
162     if (iResult == SOCKET_ERROR) {
163         printf("SHUTDOWN FAILED : %d\n", WSAGetLastError());
164         closesocket(ClientSocket);
165         WSACleanup();
166         return 1;
167     }
168
169     /** STEP 10 : CLEANUP! **/
170     closesocket(ClientSocket);
171     WSACleanup();
172
173     return 0;
174 }
```

Compile & Run — For Server

```
/winsock/bin/Debug>Winsock_Programs.exe
HELLO WINSOCK!
SELECT BELOW (1-3)
1. SERVER        2. CLIENT       3.EXIT
1
SERVER APPLICATION STARTED!
SOCKET CREATED, NOW BINDING TO PORT 27015
BINDED TO 27015
BYTES GOT FROM CLIENT: 13
CLIENT SAID :  HELLO SERVER!
BYTES SEND TO CLIENT : 18
LOOKS LIKE CLIENT HUNGED UP!
STATUS CODE : 10054
SELECT BELOW (1-3)
1. SERVER        2. CLIENT       3.EXIT
3

Process returned 0 (0x0)   execution time : 65.984 s
Press any key to continue.
```

For Client

```
/winsock/bin/Debug>Winsock_Programs.exe
HELLO WINSOCK!
SELECT BELOW (1-3)
1. SERVER        2. CLIENT       3.EXIT
2
CLIENT APPLICATION STARTED!
INITIALIZATION SUCCESSFUL WITH WSAStartup = 0
```

```
BYTES SENT : 13
BYTES RECEIVED : 18
SERVER SAID :  THANK YOU CLIENT!
^Z
```

3.12 Java Network Programming — Client/Server

3.12.1 Java Network Agent

JavaNetworking.java

```
import java.util.*;

public class JavaNetworking{

  public static void main(String args[]){
    System.out.println("What you want to be?");
    System.out.println("1. Server");
    System.out.println("2. Client");

    Scanner in = new Scanner(System.in);
    int ch = in.nextInt();

    if(ch == 1){
      /* SERVER APPLICATION */
      System.out.println("SERVER APPLICATION");
      System.out.println("Specify port");
      int p_n = in.nextInt();

      SimpleServer s = new SimpleServer(p_n);
      s.bindServer();
      s.startServer();


    }else{
      /* CLIENT APPLICATION */
      System.out.println("CLIENT APPLICATION");

      System.out.println("Specify server port");
      int p_n = in.nextInt();

      SimpleClient c = new SimpleClient(p_n);
      c.initClient();
      c.sendServer("Hello Server!");
      c.closeClinet();
    }

    System.out.println("!");

  }

}
```

3.12.2 Java–based Client

SimpleClient.java

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

```
3
4  public class SimpleClient{
5    String remote_host = "localhost";
6    int port_no;
7    Socket clientSocket;
8
9    SimpleClient(int p_n, String r_host){
10     port_no = p_n;
11     remote_host = r_host;
12   }
13
14   SimpleClient(int p_n){
15     port_no = p_n;
16   }
17
18   public void initClient(){
19     try{
20       clientSocket = new Socket(remote_host, port_no);
21       System.out.println("Binded successfully with Server on
     Port : "+port_no);
22     }catch(Exception e){
23       System.err.println("Can't bind with port! \n"+e);
24     }
25
26   }
27
28   public void sendServer(String msg){
29     try{
30       DataOutputStream outToServer = new DataOutputStream(
     clientSocket.getOutputStream());
31       outToServer.writeBytes(msg + '\n');
32
33       BufferedReader inFromServer = new BufferedReader(new
     InputStreamReader(clientSocket.getInputStream()));
34       String modifiedSentence = inFromServer.readLine();
35       System.out.println("FROM SERVER: " + modifiedSentence);
36     }catch(Exception e){
37       System.err.println("Something went wrong! \n"+e);
38     }
39
40   }
41
42   public void closeClinet(){
43     try{
44       clientSocket.close();
45     }catch(Exception e){
46       System.err.println("Something went wrong! \n"+e);
47     }
48   }
49
50 }
```

3.12.3 Java–based Server

SimpleServer.java

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

public class SimpleServer{

  /* CREATE NEW SERVER WITH GIVEN ADDRESS AND PORT */
  private String server_address;
  private int port_no;
  private ServerSocket welcomeSocket;
  private Socket connectionSocket;

  SimpleServer(int p_n){
    port_no = p_n;
  }

  /* BIND SERVER WITH PORT */
  public void bindServer(){
    try{
      welcomeSocket = new ServerSocket(port_no);
      System.out.println("Binded successfully with Port : "+
    port_no);
    }catch(Exception e){
      System.err.println("Can't bind server with port! \n"+e);
    }
  }

  public void startServer(){
    String clientSentence;
    String capitalizedSentence;
    while(true)
       {
        try{
          connectionSocket = welcomeSocket.accept();

          /* READY FOR CLIENT ! */


          System.out.println("I guess client is connected!");
          BufferedReader inFromClient =
           new BufferedReader(new InputStreamReader(
    connectionSocket.getInputStream()));

          DataOutputStream outToClient = new DataOutputStream(
    connectionSocket.getOutputStream());
          if( (clientSentence = inFromClient.readLine()) !=
    null){
            System.out.println("He said: " + clientSentence);
          }else{
            System.out.println("I guess client is closed!");
            break;
```

```
47            }
48
49            System.out.println("Lets reply him with: " +
    clientSentence.toUpperCase());
50            capitalizedSentence = clientSentence.toUpperCase() +
    '\n';
51            outToClient.writeBytes(capitalizedSentence);
52
53            break;
54
55          }catch(Exception e){
56            System.err.println("Something went wrong! \n"+e);
57          }
58         }
59      System.out.println("Say bye to him!");
60      closeServer();
61    }
62
63    public void closeServer(){
64      try{
65        connectionSocket.close();
66        welcomeSocket.close();
67      }catch(Exception e){
68        System.err.println("Something went wrong! \n"+e);
69      }
70    }
71
72
73 }
```

Compile & Run —

```
 karshe@karshe-dell:~/root\$ javac JavaNetworking.java
 karshe@karshe-dell:~/root\$
 karshe@karshe-dell:~/root\$ java JavaNetworking
What you want to be?
1. Server
2. Client
1
SERVER APPLICATION
Specify port
6000
Binded successfully with Port : 6000
I guess client is connected!
He said: Hello Server!
Lets reply him with: HELLO SERVER!
Say bye to him!
!

 karshe@karshe-dell:~/root\$ java JavaNetworking
```

```
What you want to be?
1. Server
2. Client
2
CLIENT APPLICATION
Specify server port
6000
Binded successfully with Server on Port : 6000
FROM SERVER: HELLO SERVER!
!
```

3.13 Java Network Programming — Accessing remote URL

3.13.1 Java URL agent — accessing URL

JavaURLCalling.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;
import java.nio.charset.Charset;

/**
 * @author https://github.com/karshe/
 * PROJECT : https://github.com/karshe/networkprogramming
 */

public class JavaURLCalling {

  public static void main(String[] args) {
    /* Lets poke localhost at port 6001 */
    System.out.println("\nOutput : \n" + callURL("http
    ://127.0.0.1:6001/"));
  }

  public static String callURL(String myURL) {
    System.out.println("Requeted URL : " + myURL);

    /* Buffer for Output from requested URL */
    StringBuilder sb = new StringBuilder();
    URLConnection urlConn = null;
    InputStreamReader in = null;
    try {
      URL url = new URL(myURL);
      urlConn = url.openConnection();
      if (urlConn != null)
        urlConn.setReadTimeout(60 * 1000); /* 1 MIN TIMEOUT */
      if (urlConn != null && urlConn.getInputStream() != null)
    {
        in = new InputStreamReader(urlConn.getInputStream(),
            Charset.defaultCharset());
        BufferedReader bufferedReader = new BufferedReader(in);
        if (bufferedReader != null) {
          int cp;
          while ((cp = bufferedReader.read()) != -1) {
            sb.append((char) cp);
          }
          bufferedReader.close();
        }
      }
    in.close();
    } catch (Exception e) {
```

```
45          throw new RuntimeException("Exception while calling URL:"
        + myURL, e);
46        }
47
48        /* Return output! */
49        return sb.toString();
50      }
51  }
```

3.13.2 Content at localhost

INDEX.HTML

```
1  Hello Java program, I am python powered SimpleHTTPServer at
       port 6001!
```

Compile & Run —

```
 karshe@karshe-dell:~/root\$ javac JavaURLCalling.java
 karshe@karshe-dell:~/root\$ java JavaURLCalling
Requeted URL : http://127.0.0.1:6001/

Output :
Hello Java program, I am python powered SimpleHTTPServer at port 6001!
 karshe@karshe-dell:~/root\$
```

We used Python Server to host file at `localhost@6001` —

```
 karshe@karshe-dell:~/root\$ python -m SimpleHTTPServer 6001
Serving HTTP on 0.0.0.0 port 6001 ...
localhost - - [26/Nov/2015 20:56:20] "GET / HTTP/1.1" 200 -
```

3.14 Java Network Programming — Using HTTP GET method

3.14.1 HTTP GET Agent — How many days are there for Christmas

JavaHttpGetConnection.java

```
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class JavaHttpGetConnection {

  private final String USER_AGENT = "Mozilla/5.0";

  public static void main(String[] args) throws Exception {

    JavaHttpGetConnection http = new JavaHttpGetConnection();

    System.out.println("Send HTTP GET request.");
    http.sendGet();

  }

  // HTTP GET request
  private void sendGet() throws Exception {

    String url = "http://127.0.0.1:8080/host/php/
    christmas_calculator.php";

    URL obj = new URL(url);
    HttpURLConnection con = (HttpURLConnection) obj.
    openConnection();

    // optional default is GET
    con.setRequestMethod("GET");

    //add request header
    con.setRequestProperty("User-Agent", USER_AGENT);

    int responseCode = con.getResponseCode();
    System.out.println("\nSENDING 'GET' REQUEST TO CHRISTMAS
    CALCULATOR : " + url);
    System.out.println("RESPONSE CODE : " + responseCode);

    BufferedReader in = new BufferedReader(
            new InputStreamReader(con.getInputStream()));
    String inputLine;
    StringBuffer response = new StringBuffer();

```

```
43      while ((inputLine = in.readLine()) != null) {
44        response.append(inputLine);
45      }
46      in.close();
47
48      //print result
49      System.out.println(response.toString());
50
51    }
52
53  }
```

3.14.2 Christmas Calculator

CHRISTMAS_CALCULATOR.PHP

```
1  <?php
2    header('Content-Type: text/plain; charset=utf-8');
3
4    $EveDay = strtotime("2015-12-25");
5    $futureEve = date('Y-m-d', strtotime('+1 year',$EveDay));
6    $Today = strtotime("today");
7
8    if($EveDay > $Today) {
9      $diff = $EveDay - $Today;
10     echo "Hey Java Agent, you know only ";
11     echo floor($diff / 86400);
12     echo " days left for Christmas!";
13   }else if($Today == strtotime("2015-12-25") || $Today ==
       $futureEve ){
14     echo "Today is Christmas, Java Agent! Merry Christmas!";
15   }else{
16     $diff = strtotime($futureEve) - $Today;
17     echo "Hey Java Agent, you know only ";
18     echo floor($diff / 86400);
19     echo " days left for Christmas!";
20   }
21 ?>
```

Compile & Run —

```
karshe@karshe-dell:~/root\$ javac JavaHttpGetConnection.java
karshe@karshe-dell:~/root\$ java JavaHttpGetConnection
Send Http GET request.

SENDING 'GET' REQUEST TO CHRISTMAS CALCULATOR :
http://127.0.0.1:8080/host/php/christmas_calculator.php
RESPONSE CODE : 200
Hey Java Agent, you know only 29 days left for Christmas!
karshe@karshe-dell:~/root\$
```

3.15 Java Network Programming — Using HTTP POST method

3.15.1 HTTP POST Agent — MD5 Encryption

JavaHttpPostConnection.java

```
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class JavaHttpPostConnection {

  private final String USER_AGENT = "Mozilla/5.0";

  public static void main(String[] args) throws Exception {

    JavaHttpPostConnection http = new JavaHttpPostConnection();

    System.out.println("Send HTTP POST request.");
    http.sendPost();

  }

  // HTTP POST request
  private void sendPost() throws Exception {

    String url = "http://127.0.0.1:8080/host/php/md5_encoder.
    php";
    URL obj = new URL(url);
    HttpURLConnection con = (HttpURLConnection) obj.
    openConnection();

    //add reuqest header
    con.setRequestMethod("POST");
    con.setRequestProperty("User-Agent", USER_AGENT);
    con.setRequestProperty("Accept-Language", "en-US,en;q=0.5")
    ;


    String msgString = "Hello World";
    String urlParameters = "msg="+msgString;

    // Send post request
    con.setDoOutput(true);
    DataOutputStream wr = new DataOutputStream(con.
    getOutputStream());
    wr.writeBytes(urlParameters);
    wr.flush();
    wr.close();

    int responseCode = con.getResponseCode();
```

```
44      System.out.println("\nSENDING 'POST' REQUEST TO : " + url);
45      System.out.println("Post parameters : " + urlParameters);
46      System.out.println("Response Code : " + responseCode);
47
48      BufferedReader in = new BufferedReader(
49              new InputStreamReader(con.getInputStream()));
50      String inputLine;
51      StringBuffer response = new StringBuffer();
52
53      while ((inputLine = in.readLine()) != null) {
54        response.append(inputLine);
55      }
56      in.close();
57
58      //print result
59      System.out.println("\nmd5 OF "+msgString+" IS\n");
60      System.out.println(response.toString());
61
62    }
63
64  }
```

3.15.2 MD5 Encoder

MD5_ENCODER.PHP

```
1  <?php
2    header('Content-Type: text/plain; charset=utf-8');
3    echo md5($_POST['msg']);
4  ?>
```

Compile & Run —

```
 karshe@karshe-dell:~/root\$ javac JavaHttpPostConnection.java
 karshe@karshe-dell:~/root\$ java JavaHttpGetConnection
Send HTTP POST request.

SENDING 'POST' REQUEST TO : http://127.0.0.1:8080/host/php/md5_encoder.php
Post parameters : msg=Hello World
Response Code : 200

md5 OF Hello World IS

b10a8db164e0754105b7a99be72e3fe5
 karshe@karshe-dell:~/root\$
```

4

Tools

"Give me six hours to chop down a tree and I will spend the first four sharpening the axe."

– Abraham Lincoln, 16^{th} *U.S. President*

Below are tools I used while making this report —

1. Document Markup Language — Used LaTeX— it is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation.
2. Wrote using — ShareLaTex engine
3. C Compiler — None other than `gcc`. You can find instruction on how to download `gcc` from here `https://gcc.gnu.org/install/download.html`
4. IDE — For most of the time Geany IDE on Ubuntu. Check out Geany IDE at `http://www.geany.org/`
5. Source Code Management — Used GitHub for most of the time. Download the source code of this book from here — `https://github.com/karshe/unixnetworkprograms`
6. Essential — LibreOffice for drafting & Clementine for music.

5

Books & References

Below are the books and online literature I followed while making the report and preparing the source code.

1. UNIX Network Programming *by W. Richard Stevens* (ISBN : 9780131411555)
2. The UNIX Programming Environment *by Brian Kernighan and Rob Pike* (ISBN : 9780139376818)
3. Advanced Network Programming – Principles and Techniques: Network Application Programming with Java *by Bogdan Ciubotaru* (ISBN : 781447152910)
4. Beej's Guide to Network Programming – Using Internet Sockets — `http://beej.us/guide/bgnet/`