A Short Note On

Socket Programming through C

By

Dr. Sukanta Das

Asst. Professor
Department of Information Technology
Bengal Engineering And Science University, Shibpur
West Bengal, India – 711103

Preface

I have been teaching *Computer Networks* to the undergraduate students in the Department of Information Technology of Bengal Engineering and Science University, Shibpur (BESUS), West Bengal, India for almost 10 years. I faced many difficulties in the laboratory works of the course. We offer this course in sixth semester. Before this course, students are offered other courses like Data Structure and Algorithms, Operating Systems, Computer Graphics etc. So the students have exposure to the basics of programming, but all of them are not expert programmer. To experiment with computer networks, students need to develop *socket* programs. Socket programming, however, is completely new to the sixth semester students, and also its little difficult for them to digest the concepts. Moreover, available books on this topic are limited. And, available books are not always helpful for the new-comers in the domain of socket programming. For example, *Unix Network Programming*, Volume 1 by Stevens is a good book on this topic, but I have observed that the students could not digest the book well. Many materials are available in Internet, but they are not well organized and could not satisfy the demand of the course.

So, I took an initiative to prepare this short note on Socket Programming using C, which would cover some basic concepts of socket programming and be brief. First draft of this note was written in 2010, which is finally modified in January, 2014. Some materials from Internet and the Steven's book were helpful in preparing this note. The note is organized according to the need of my course. I have put my effort to make this note as simple as possible. Someone may complain for over simplification. However, my target was to give some basic ideas about socket programming to my undergraduate students.

This note is primarily for beginners. It may also be helpful for others. I want to further improve this writing. I shall be very happy if you send me your comments, suggestions and criticisms. I believe, this writing is not error-free. Please let me know if any kind of error is identified.

Sukanta Das
February, 2014.
email: sukanta AT it DOT becs DOT ac DOT in OR sukanta78das AT yahoo DOT com

Contents

1 Introduction **1**

1.1 The TCP/IP and Internet 1
1.2 TCP/IP protocol architecture 2
1.3 The addresses 5
1.4 Operating system support 7

2 Accessing Datalink Layer **9**

2.1 The `tcpdump` program 9
2.2 Access to the DLL 11
2.3 Handling an interface in promiscuous mode 14
2.4 Exercises 16

3 Accessing Network layer **17**

3.1 The protocols IP, ICMP, TCP and UDP 17
3.2 Raw sockets 22
3.3 Building and injecting datagrams 23
3.4 Ping program 25
3.5 Exercise 28

4 TCP Sockets **29**

4.1 TCP Clients and Servers 29
4.2 Socket address 30
4.3 TCP Socket APIs 31
4.4 Few essential functions 35
4.5 A simple ftp client-server program 37
4.5.1 Client code 37
4.5.2 Server code 38
4.5.3 Execution of the programs 40
4.6 Error handling 40
4.7 Exercise 42

5 UDP socket **43**
5.1 Creating a UDP Socket Application 43
5.2 Day time UDP client-server 45
5.2.1 The client code 45
5.2.2 The server code 46
5.2.3 Execution of the codes 47
5.3 Broadcasting in a network 48
5.4 Excercise 50

6 Concurrent Server **51**
6.1 Writing Concurrent Server 51
6.2 Concurrent server with thread 52
6.3 Concurrent UDP server 55
6.4 Exercise 56

Chapter 1

Introduction

A computer network is a connection among a set of computers through some networking hardware like router, switch, etc. and communication channel. Primary goal of developing a computer network is to share resources of different computers. To share the resources, therefore, we need support from some programs. These programs are essential for developing any software for a networked system (and for Internet). Such programs create *socket* to send and receive data to/from the network. So, these programs are commonly known as *Socket Programs*. The sockets in computer networking can be thought as interfaces that can "plug into" each other over a network. Once so "plugged in", the programs can communicate. This note provides a short introduction on the socket programming. Languages, like C, C++, JAVA, Visual Basic etc, can be used in socket programming. In our opinion, however, socket programming through C is the most exciting way to explore the networks, specially it is true for the beginners. In this note, we use C to write socket programs.

Historically, network applications are developed in Unix environment. BSD Unix is the first Unix that provided complete networking support. Presently, almost all commercially available operating systems support networking and socket programming. Those operating systems provide some system calls (kernel APIs) to the user program to develop network applications. These APIs were originated with 4.2BSD Unix, released in 1983. The APIs, currently used in socket programs, of each operating system are either same or similar with that of BSD Unix. In this note, we develop socket programs for Unix or Unix-like (such as, Linux) platforms. All the given examples, however, are compiled and run on Linux machines.

This note is organized as follows. This chapter discusses some basic concepts about computer networks. Chapter 2 discusses about the way of accessing data link layer (DLL). Chapter 3 deals with the access of network layer. However, by socket programming, we generally refer to development of some programs that interact with two transport layer protocols - TCP and UDP. Chapter 4 and Chapter 5 discuss about TCP socket and UDP socket respectively. Chapter 6 finally deals with the design concurrent server.

1.1 The TCP/IP and Internet

TCP and IP are the abbreviations of two protocols - Transmission Control Protocol (TCP) and Internet Protocol (IP). However, the TCP/IP refers to an architecture of protocols that are designed for large networks consisting of network segments, connected by routers. The Internet, the largest computer network that covers the globe by interconnecting universities,

industries, government agencies, libraries and individuals, also utilizes the TCP/IP protocol suite. Almost all computer applications are designed considering the underlying computer network architecture follows the TCP/IP.

The roots of TCP/IP can be traced back to research conducted by the Defence Advanced Research Projects Agency (DARPA) of Department of Defence (DoD), USA in late 1960s. The DoD wanted to develop a communication system that could not be destroyed even by nuclear war. As a result of their research, world's first computer network ARPANET (Advanced Research Projects Agency NETwork) was launched in 1968. ARPANET is the predecessor of today's Internet. TCP/IP protocol suite was developed with the development of ARPANET. The following list highlights some important TCP/IP milestones:

- In 1970, ARPANET hosts started to use Network Control Protocol (NCP), a preliminary form of the Transmission Control Protocol (TCP).
- In 1972, the Telnet protocol was introduced. Telnet is used for terminal emulation to connect dissimilar systems. In the early 1970s, these systems were different types of mainframe computers.
- In 1973, the File Transfer Protocol (FTP) was introduced. FTP is used to exchange files between dissimilar systems.
- In 1974, the Transmission Control Protocol (TCP) was specified in detail. TCP replaced NCP and provided enhanced reliable communication services.
- In 1981, the Internet Protocol (IP) (also known as IP version 4 [IPv4]) was specified in detail. IP provides addressing and routing functions for end-to-end delivery.
- In 1982, the Defence Communications Agency (DCA) and ARPA established the Transmission Control Protocol (TCP) and Internet Protocol (IP) as the TCP/IP protocol suite.
- In 1983, ARPANET switched from NCP to TCP/IP.
- The 4.2BSD was released in 1983 that implemented TCP/IP (and socket API) for the first time.

The name of BSD (Berkeley Software Distribution) Unix is intimately associated with the unix networking. The 4.2BSD did not only implement the TCP/IP protocol suite, but also originated the socket APIs, required to develop network programs. Later versions of 4.2BSD modified the protocol suit in different way to improve the performance. The entire 4.xBSD project was financed by DARPA.

1.2 TCP/IP protocol architecture

The TCP/IP protocols are organized in the form of a stack. The protocols are dependent on each other. Based on the dependency, one or more protocols can form a layer. The TCP/IP protocol architecture refers 4 such layers. The architecture with major protocols of the layers are noted in Figure 1.1. A layer performs some specific task to provide service to its upper layer protocols. For example, TCP (a transport layer protocol) takes service of IP to provide service to the protocols of application layer.

TCP/IP is most commonly associated with the Unix operating system. 4.2BSD Unix started bundling TCP/IP protocols with the operating system. Nevertheless, TCP/IP protocols are

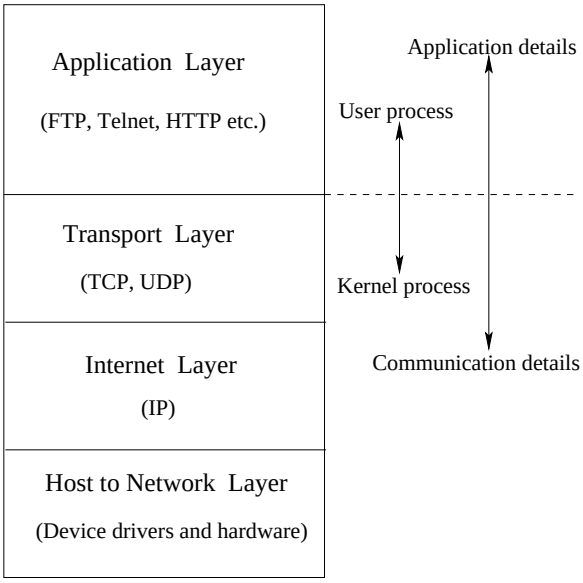

Figure 1.1: The TCP/IP architecture

available for all widely-used operating systems today and native TCP/IP support is provided in OS/2, OS/400, and Windows, as well as most Unix variants.

The protocols of Transport layer, Internet layer and Host-to-Network layer are implemented and maintained by the kernel (Figure 1.1). The network application programs are developed by taking the required service from the kernel. Here, the communication details are hidden from the users, and that are dealt by kernel only.

A brief discussion on the layers are provided next.

Host-to-Network layer

The Host-to-Network layer is the lowest layer of the protocol suite. This layer is hardware dependent, whereas the above layers of TCP/IP are hardware independent. The Host-to-Network deals with the local network connection and the interface between host and network. A number of network interface protocols are designed such as, Ethernet, Token Ring, FDDI, PPP etc. A network card can be found that implements the layer.

The task of this layer is to move raw data bits (frame) from one host (or router) to another. The processes of transmitting and receiving packets through a given link can be controlled by the software (device driver) for the network card, as well as by the firmware or specialized chip-sets. They perform data link functions, such as adding a frame header to prepare it for transmission, and transmitting the frame over a physical medium. This layer, however, is generally considered as the composition of two layers - Data Link layer and Physical layer. For details about them, consult any standard text book on *Computer Networks*.

Internet layer

The Internet layer solves the problem of sending packets across one or more networks. Internetworking requires sending data from the source network to the destination network. This

process is called routing. The basic protocol of this layer is the Internet Protocol (IP). It uses a special addressing scheme, called IP address, to transport packets. In TCP/IP architecture, all the protocols utilize the IP to send and receive data. This layer, however, is also known as Network layer.

Transport layer

The Transport layer's responsibilities include end-to-end message transfer capabilities without depending on the underlying network, along with the error control, segmentation, flow control, congestion control, and application addressing (port numbers). End to end message transmission can be categorized as either connection-oriented, implemented in Transmission Control Protocol (TCP), or connectionless, implemented in User Datagram Protocol (UDP).

The connection-oriented services are modelled after telephony system, where connection is established first before sending the data. Here a virtual tube is formed to make the communication reliable. On the other hand, no such connection is established in connection-less service. So the connectionless services are unreliable.

The Transport layer provides the service of connecting applications through some port numbers (described in next section). Since IP is connectionless and so provides only a best effort delivery, the Transport layer is the first layer of TCP/IP stack to offer reliability. The Application layer protocols (like FTP) that take service from TCP, a reliable Transport layer protocol, can also provide reliability. The TCP addresses following reliability issues:

- data arrives in-order
- data has minimal error (that is, correctness)
- duplicate data is discarded
- lost/discarded packets are resent
- traffic congestion control

User Datagram Protocol (UDP), on the other hand, is a connectionless protocol. Like IP, it is a best effort, unreliable protocol. Reliability is addressed through error detection using a weak checksum algorithm. UDP is typically used for applications such as streaming media (audio, video, Voice over IP, etc.) where on-time arrival is more important than reliability, or for simple query/response applications like DNS lookups, where the overhead of setting up a reliable connection is disproportionately large. Real-time Transport Protocol (RTP) is another connectionless protocol that is designed for real-time data such as streaming audio and video.

TCP and UDP are used to carry the application data. The appropriate transport protocol is chosen based on the higher-layer protocol application. For example, the File Transfer Protocol (FTP) expects a reliable connection, so it uses TCP as Transport layer protocol.

Application layer

The Application layer is the highest layer which implements the higher-level protocols used by the applications for network communication. Examples of application layer protocols include the File Transfer Protocol (FTP), the Simple Mail Transfer Protocol (SMTP), the Hypertext Transfer Protocol (HTTP), etc. The user data are then coded according to application layer protocols, and finally passed to some Transport layer protocol (like TCP, UDP, etc).

Application layer protocols generally assume that the transport layer (and lower layer) protocols provide a stable network connection across which to communicate. However, the applications are usually aware of key qualities of the transport layer connection such as the end point IP addresses and port numbers (discussed in next section). Application layer protocols are most often associated with client-server applications.

Transport and lower layers are commonly unconcerned with the specifics of application layer protocols. Routers and switches are typically unaware of encapsulated application data. However, some firewall and bandwidth throttling applications do try to determine what's inside, as with the Resource Reservation Protocol (RSVP). It is also sometime necessary for Network Address Translation (NAT) facilities to take account of the needs of particular application layer protocols. (NAT allows hosts on private networks to communicate with the outside world via a single visible IP address using port forwarding, and is an almost ubiquitous feature of modern domestic broadband routers).

1.3 The addresses

The computers, connected in Internet, use three types of addresses for communication – physical address (used in Host-to-Network layer), IP address (used in Internet layer) and port number (used in Transport layer).

Physical or MAC address

Every network interface has an 6-byte long physical or MAC (Media Access Control) address. This is the hardware address that the lowest layer of the network architecture uses to communicate. Hence, to forward data using a physical link, the MAC address is used. The MAC address is sometime used to assign the IP address by DHCP (Dynamic Host Configuration Protocol).

IP address

The IP addresses are the logical addresses which are used by Internet Protocol (IP). In internetworking, where we connect many networks, the IP address guides to reach the data packets to the correct host of the correct network. There are two versions of IP (Internet Protocol): IPv4 (IP version 4) and IPv6 (IP version 6). Each version defines the IP address differently. However, the generic term IP address typically still refers to the addresses defined by IPv4.

The IP uses 32-bit (four-byte) addresses, which limits the address space to 4,294,967,296 (2^{32}) possible unique addresses. However, some are reserved for special purposes such as private networks (about 18 million addresses) or multicast addresses (about 270 million addresses). This reduces the number of addresses that can potentially be allocated for routing on the public Internet. As addresses are being incrementally delegated to end users, an IP address shortage has been developing. This limitation has stimulated the development of IPv6 (128-bit long addresses), which is currently in the early stages of deployment, and may be the long-term solution.

IPv4 addresses are usually represented in dot-decimal notation (four numbers, each ranging from 0 to 255, separated by dots, e.g. 208.77.188.166). Each part represents 8 bits of the address, and is therefore called an octet. In less common cases of technical writing, IPv4 addresses may be presented in hexadecimal, octal, or binary representations. In most representations each octet is converted individually.

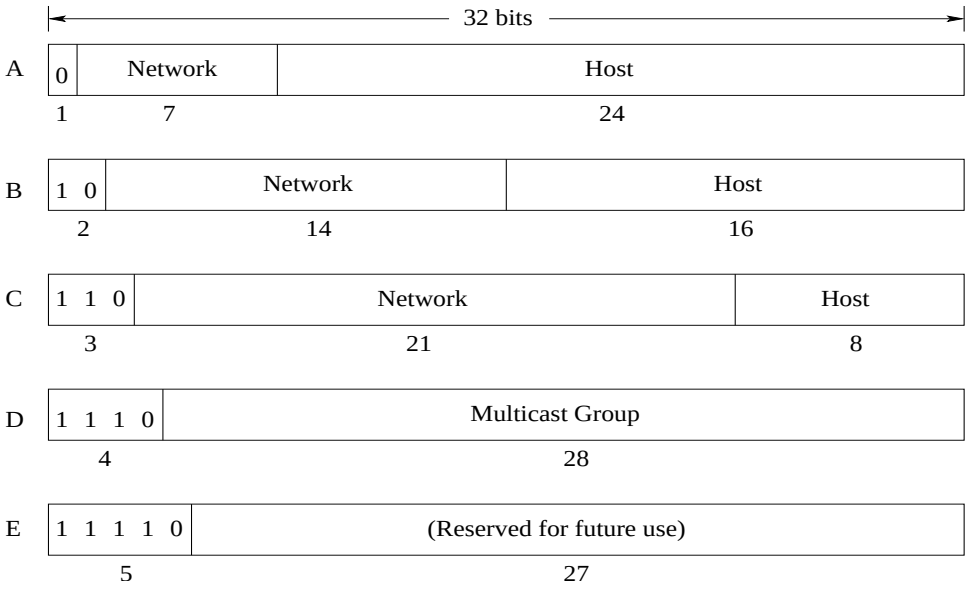

Figure 1.2: Five classes of IP addresses

In the early stages of development of the Internet Protocol (RFC[1] 760), an IP address was interpreted as the combination of network number portion and host number portion. The highest order octet (most significant eight bits) in an address was designated the network number and the rest of the bits were called the rest field or host identifier which were used for host numbering within a network. This method soon proved inadequate as additional networks developed that were independent from the existing networks. In 1981, the Internet addressing specification was revised with the introduction of classful network architecture.

Classful network design allows a larger number of individual network assignments. Presently the classful addressing scheme is used. There are 5 classes of IP addresses – A, B, C, D, E and F (Figure 1.2). Out of five, three classes (A, B, and C) are defined for universal unicast addressing. Class E is defined for multicast address and class F is reserved for future use. To identify the class of an IP address, first few bits of the most significant octet are used (Figure 1.2). For example, the most significant bit of an IP address as 0 signifies that the IP address is from class A. The IP addresses of class A, B and C have two parts – network and host. However, unlike previous, the number of bits in network and host vary from class to class. The IP addresses in dotted decimal form of different classes are noted in Table 1.1. As it is mentioned earlier, some of the IP addresses (like 0.0.0.0) are reserved for some special purpose.

Therefore, network part (*net-id*) of a Class A, Class B or Class C IP address identifies the network, whereas host part of the IP address identifies the host within the network. It is observed that all host numbers of a particular network are not always used. For example, in Class B IP address, 2^{16} number of hosts can be addressed with a given *net-id*. However, the network may not have that number of hosts. So, host part is again divided (RFC 950) into two parts – *subnet* and *host*. Number of bits for hosts is now determined based on the maximum

[1]RFC stands for Request For Comments, which is followed by an integer number. Memos in the RFC document series contain technical and organizational notes about the Internet. They cover many aspects of computer networking, including protocols, procedures, programs, and concepts, as well as meeting notes, opinions, and sometimes humor.

Table 1.1: Range of IP addresses of different classes

Class	No. of Networks	No. of Hosts	Range of IP address (in dotted decimal format)
A	2^7	2^{24}	0.0.0.0 to 127.255.255.255
B	2^{14}	2^{16}	128.0.0.0 to 191.255.255.255
C	2^{21}	2^8	192.0.0.0 to 223.255.255.255
D			224.0.0.0 to 239.255.255.255
E			240.0.0.0 to 247.255.255.255

possible hosts in a network. To extract subnet part (*subnet-id*), we use *subnet mask*. Now, *net-id* and *subnet-id* combinedly identify a network.

Port number

An IP address in a network (say Internet) can uniquely identify a host. A packet received by a host can not be delivered to proper application layer protocol (that is, an application process) using the IP address only, because the host generally run a number of such application protocols such as FTP, HTTP, SMTP, etc. To distinguish those packets based on their target applications, 16 bit integer numbers are introduced in Transport layer. These numbers are called Port number. Each network application has a unique port number. The Transport layer delivers the incoming packets to proper application based on the port number.

Application layer protocols are most often associated with client-server applications, and the common servers have specific ports assigned to them by the IANA (Internet Assigned Numbers Authority – the IANA also oversees the IP address assignment). For example, HTTP has port 80, Telnet has port 23, FTP has port 21, etc. The port numbers are divided into three ranges: the well known ports, the registered ports, and the dynamic and/or private ports. The well known ports are those from 0 through 1023. The registered ports are those from 1024 through 49151. The rest are dynamic and/or private ports. It is advised to use these dynamic ports in the socket programs.

1.4 Operating system support

Network applications are developed above network operating system. All the protocols except application layer protocols of TCP/IP architecture, like TCP, UDP, IP, ICMP, Ethernet etc. are developed in and maintain by the kernel. Figure 1.3 shows the relationship between the network application programs and the kernel. Like other application programs, network application programs reside in application area. To develop network applications, the programs utilize few system calls provided by the operating system.

These system calls are commonly referred as API (application program interface) implemented by operating system.[2] These APIs are programming language independent and can be utilized by any programming language. Example of such APIs in Linux are: *socket*, *connect*, *accept*, *read*, etc.

Generally user processes interact with Transport layer protocols – TCP and UDP. However, Unix, Unix-like and other operating systems provide powerful APIs through which a user pro-

[2]APIs can also be implemented by an application program. APIs, implemented by kernel, are sometime called as kernel API.

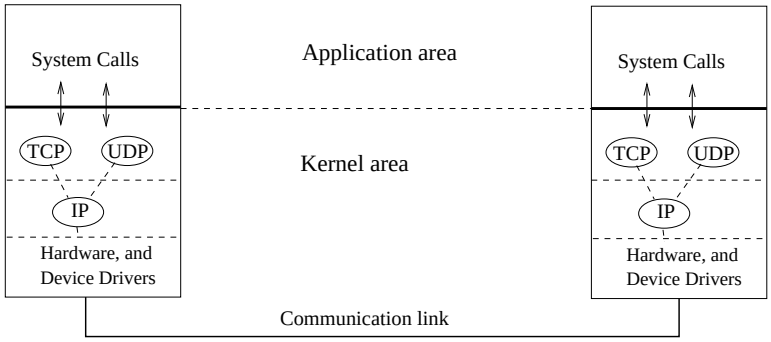

Figure 1.3: Role of operating system in developing networking applications

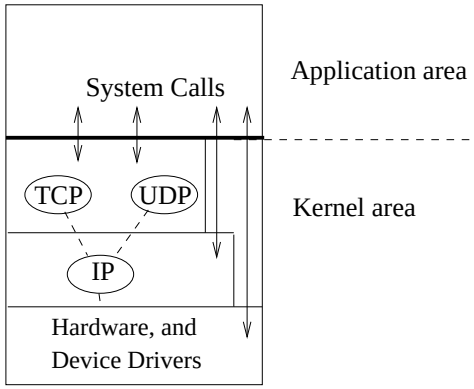

Figure 1.4: Interaction between user processes and network protocols

cess can interact with Internet/Network layer protocols (that is, with IP) bypassing Transport layer protocols (such as TCP, UDP). A user process can even directly talk with the Data Link layer bypassing Network and Transport layers. These facilities are needed to develop powerful application programs (such as `tcpdump`, `ping`, etc.), which are generally used to monitor and administer networks. Therefore, Figure 1.1 and Figure 1.3 show the most common view of user process interaction with network protocols. Figure 1.4 shows more perfect view of such interaction.

Chapter 2

Accessing Datalink Layer

The datalink layer (DLL), closely related with physical layer, notices all the packets destined to the computer. The data successfully received by the DLL is forwarded to the upper layer for further action. When a computer sends some data, then also the DLL is trapped. So, accessing DLL directly (just bypassing the upper layers) provides an application developer to develop powerful applications. For example, the `tcpdump` (available in almost all versions of Unix operating system) that targets to capture all the packets destined to the computer, accesses the DLL directly.

When a packet is received by the NIC (network interface card), the packet is transferred to the kernel by DMA (direct memory access) controller. The packet is put in a buffer. However, the NIC can be operated in *promiscuous mode* and *non-promiscuous mode*. The promiscuous mode allows a network device to intercept and read each network packet that arrives in its entirety. This mode of operation (which is to be supported by the NIC and the host operating system) is sometimes used to capture and save all the packets for analyzing and monitoring networks. In non-promiscuous mode, the NIC *listens to* the packets to determine if the packet is destined to the computer. If it is so, then the packet is forwarded to the kernel.

To access the DLL, an application program has to get access of the buffer, where the received packet is put by the DMA controller. After getting access to the buffer, the application program analyzes the packets to report the required results. Figure 2.1 shows the access of data link layer packets bypassing the TCP/IP protocol stack of kernel.

In this chapter, the `tcpdump` program, used to dump the network traffic, is first introduced. How to directly access the DLL is then presented.

2.1 The `tcpdump` program

The `tcpdump` is a packet analyzer that runs in command line. It allows the users to intercept and display all packets being transmitted to or received from a network to which the computer is attached. The tcpdump is found in almost all versions of Unix operating system – Linux, Solaris, BSD, Mac OS X, etc. The packet capture library, `libpcap`[1] that provides implementation-independent access to the underlying packet capture facility, is used by the tcpdump program.

[1]The packet capture library (`pcap.h`) provides a set of functions that can be called from a C program. Interested readers may consult the manual of the library.

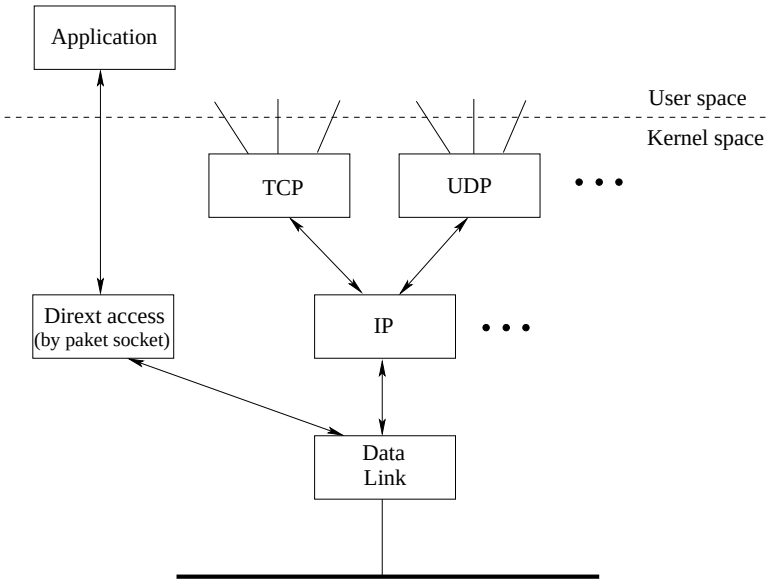

Figure 2.1: Accessing datalink layer bypassing TCP/IP stack

The tcpdump program is independent of DLL protocols and network hardware. It can work with Ethernet, as well as PPP (point-to-point protocol). The program prints the header of each captured packet. The header tells about the packet type and the protocol to which the data is forwarded.

In some versions of Unix (eg. Linux), a user must have superuser privileges to use tcpdump because the packet capturing mechanisms on those systems require elevated privileges. However, the -Z option may be used to drop privileges to a specific unprivileged user. In other Unix-like operating systems, the packet capturing mechanism can be configured to allow non-privileged users to use it; if that is done, superuser privileges are not required. A sample output of the program in Linux is given below. The program has run with root privilege.

```
# /usr/sbin/tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
17:24:30.042138 arp who-has 10.24.8.233 tell 10.24.1.107
17:24:30.042952 IP 10.24.247.6.32771 > itbox.it.becs.ac.in.domain:
                              14830+ PTR? 233.8.24.10.in-addr.arpa. (42)
17:24:30.044313 IP itbox.it.becs.ac.in.domain > 10.24.247.6.32771:
                                              14830 NXDomain* 0/0/0 (42)
17:24:30.044475 IP 10.24.247.6.32771 > itbox.it.becs.ac.in.domain:
                             44910+ PTR? 107.1.24.10.in-addr.arpa. (42)
17:24:30.045136 IP itbox.it.becs.ac.in.domain > 10.24.247.6.32771:
                                              44910 NXDomain* 0/0/0 (42)
17:24:30.045333 IP 10.24.247.6.32771 > itbox.it.becs.ac.in.domain:
                                4230+ PTR? 1.0.24.10.in-addr.arpa. (40)
17:24:30.046112 IP itbox.it.becs.ac.in.domain > 10.24.247.6.32771:
                                                4230* 1/0/0 PTR[|domain]
17:24:30.046252 IP 10.24.247.6.32771 > itbox.it.becs.ac.in.domain:
```

```
                            47204+ PTR? 6.247.24.10.in-addr.arpa. (42)
17:24:30.046847 IP itbox.it.becs.ac.in.domain > 10.24.247.6.32771:
                                              47204 NXDomain* 0/0/0 (42)
17:24:31.118525 arp who-has 10.24.8.206 tell 10.24.1.108
17:24:31.118724 IP 10.24.247.6.32771 > itbox.it.becs.ac.in.domain:
                            35453+ PTR? 206.8.24.10.in-addr.arpa. (42)
17:24:31.120859 IP itbox.it.becs.ac.in.domain > 10.24.247.6.32771:
                                             35453 NXDomain* 0/0/0 (42)
17:24:31.120991 IP 10.24.247.6.32771 > itbox.it.becs.ac.in.domain:
                            18703+ PTR? 108.1.24.10.in-addr.arpa. (42)
17:24:31.121715 IP itbox.it.becs.ac.in.domain > 10.24.247.6.32771:
                                             18703 NXDomain* 0/0/0 (42)
17:24:31.198918 arp who-has 10.24.8.234 tell 10.24.1.107
17:24:31.199108 IP 10.24.247.6.32771 > itbox.it.becs.ac.in.domain:
                            42027+ PTR? 234.8.24.10.in-addr.arpa. (42)
17:24:31.201024 IP itbox.it.becs.ac.in.domain > 10.24.247.6.32771:
                                             42027 NXDomain* 0/0/0 (42)

17 packets captured
17 packets received by filter
0 packets dropped by kernel
#
```

The last three lines of the above output shows the summery. Kernel may drop few captured packets due to lack of buffer space. Two types of packets (IP and arp) are found during the run of tcpdump. It is, however, possible to use tcpdump for the specific purpose of intercepting and displaying the communications of another user or computer. A user with the necessary privileges on a system acting as a router or gateway through which unencrypted traffic such as Telnet or HTTP passes can use tcpdump to view login IDs, passwords, the URLs and content of websites being viewed, or any other unencrypted information.

Now, we shall write our own program to access the DLL.

2.2 Access to the DLL

There are three common methods in Unix systems for accessing datalink layer – the *BPF* (BSD Packet Filter) for 4.4BSD and many other Berkley-derived implementations, the *DLPI* (SVR4 Data Link Provider Interface) for SVR4 systems and the *packet socket* for the Linux systems. The underlying principles of these three are similar. However, we shall discuss here only the datalink access capabilities of Linux through packet socket.

To receive packets from the datalink layer in Linux, a socket of type *SOCK_RAW* is created.

```
#include <sys/socket.h>
#include <netpacket/packet.h>
#include <net/Ethernet.h>     /* For DLL protocols */

packet_socket = socket(PF_PACKET, int socket_type, int protocol);
```

Packet sockets are used to receive or send raw packets at the device driver (datalink layer) level. They allow the user to implement protocol modules in user space on top of the physical layer.

The `socket()` returns a socket descriptor (`packet_socket`). First parameter of `socket()` is the *protocol family*, which is set to `PF_PACKET` for creating packet socket. Here, the *socket_type* is either `SOCK_RAW` for raw packets including the link level header or `SOCK_DGRAM` for *cooked* packets with the link level header removed. SOCK_RAW packets are passed to and from the device driver without any changes in the packet data. The network byte order actually refers the big-endian byte ordering. But the host machine may not always follow big-endian byte ordering. So, we convert host byte order to network byte order by using `htons()` and `htonl` (see Section 4.4 for further details).

The parameter *protocol* refers the IEEE 802.3 protocol number in network byte order. The list of allowed protocols is noted in the include file `<linux/if_ether.h>`. However, we consider that the underlying link layer follows the Ethernet protocol. When *protocol* field is set to ETH_P_ALL then the packets of all protocols to which Ethernet forwards data, are received[2]. All incoming packets of that protocol type will be passed to the packet socket before they are passed to the protocols implemented in the kernel. However, a process that implements the packet socket needs the superuser privilege.

To receive packets from the socket, just created, the `recvfrom()` is generally called.

```
#include <sys/socket.h>

int   recvfrom(int packet_socket, void *buf, unsigned int len, int flags,
               struct sockaddr *from, unsigned int *fromlen);
```

The *packet_socket* is socket descriptor, data received from the socket is stored in `buf` of size `len`. If *from* is not NULL, and the underlying protocol provides the source address, which is filled in a proper address structure. However, the *from* is always typecast to the generic socket structure `struct sockaddr`[3]. The argument *fromlen* is a value-result parameter, initialized to the size of the buffer associated with from, and modified on return to indicate the actual size of the address stored there. When receiving a packet from packet socket, the address is passed in a standard `sockaddr_ll` address structure. The `sockaddr_ll` is a device independent address structure. When transmitting a packet, the user supplied buffer should contain the physical layer header. That packet is then queued unmodified to the network driver of the interface defined by the destination address.

```
struct sockaddr_ll {
      unsigned short  sll_family;    /* Always AF_PACKET */
      unsigned short  sll_protocol;  /* Protocol */
      int             sll_ifindex;   /* Interface number */
      unsigned short  sll_hatype;    /* Header type */
      unsigned char   sll_pkttype;   /* Packet type */
      unsigned char   sll_halen;     /* Length of address */
      unsigned char   sll_addr[8];   /* Physical layer address */
};
```

The `sll_protocol` refers the type of data (in network byte order format) of the Ethernet frame is carrying. List of protocols are defined in the `linux/if_ether.h` include file. It defaults to the sockets protocol, that is, the *protocol* parameter specified during the

[2]To understand the *protocol* field, consult any text book for Ethernet frame format.

[3]Many socket APIs like *accept*, *bind*, *connect*, *sendto*, etc. use this generic socket structure `struct sockaddr`. Other socket structures like `sockaddr_ll`, `sockaddr_in`, etc. are typecast to this generic structure.

call of socket(). If we specify ETH_P_IP as the *protocol* during the call of socket() (that is, we want to get those Ethernet packets which carry IP data), then sll_protocol is automatically set to ETH_P_IP. The sll_ifindex is the interface index of the interface; 0 matches any interface (only legal for binding). sll_hatype is an ARP type as defined in the linux/if_arp.h include file. sll_pkttype contains the packet type. Valid types are PACKET_HOST for a packet addressed to the local host, PACKET_BROADCAST for a physical layer broadcast packet, PACKET_MULTICAST for a packet sent to a physical layer multicast address, PACKET_OTHERHOST for a packet to some other host that has been caught by a device driver in promiscuous mode, and PACKET_OUTGOING for a packet originated from the local host that is looped back to a packet socket. These types make only sense for receiving. The sll_addr and sll_halen contain the physical layer (e.g. IEEE 802.3) address and its length. The exact interpretation depends on the device.

However, to send packets, it is enough to specify sll_family, sll_addr, sll_halen, sll_ifindex. The other fields should be 0. sll_hatype and sll_pkttype are set on received packets. For bind only sll_protocol and sll_ifindex are used.

Now we present a program to capture the datalink layer packets.

```
#include<sys/socket.h>
#include<netpacket/packet.h>
#include<net/ethernet.h>

int main()
{
        int sockfd, len;
        char buffer[2048];
        struct sockaddr_ll pla;

        sockfd=socket(PF_PACKET,SOCK_RAW,htons(ETH_P_ALL));
        if(sockfd<0)
        {
                perror("packet_socket");
                exit(0);
        }
        printf("Types of the captured packets are...\n"
        while(1)
        {
                len = sizeof(struct sockaddr_ll);
                recvfrom(sockfd,buffer,sizeof(buffer),0,
                                (struct sockaddr *)&pla,&len);
                switch(pla.sll_pkttype)
                {
                        // these constant values are taken from linux/if_packet.h
                        case 0://PACKET_HOST
                                printf("For_Me\n");
                                break;
                        case 1://PACKET_BROADCAST
                                printf("Broadcast\n");
                                break;
                        case 2://PACKET_MULTICAST
                                printf("Multicast\n");
```

```
                        break;
                case 3://PACKET_OTHERHOST
                        printf("Other_Host\n");
                        break;
                case 4://PACKET_OUTGOING
                        printf("Outgoing\n");
                        break;
                case 5://PACKET_LOOPBACK
                        printf("Loop_Back\n");
                        break;
                case 6://PACKET_FASTROUTE
                        printf("Fast_Route\n");
                        break;
        }
    }
}
```

Above program runs in superuser mode and one can observe the packet types received by the datalink layer.

By default all packets of a specified protocol type are passed to a packet socket. However, there is an alternative way to get a packet socket – by calling `socket(PF_INET, SOCK_PACKET, protocol)` (which was designed for Linux 2). The main difference between the two methods is that SOCK_PACKET uses the old `struct sockaddr_pkt` to specify an interface, which doesnt provide physical layer independence. This structure is obsolete and should not be used in new code.

2.3 Handling an interface in promiscuous mode

While packet socket is created, all the packets, destined to the computer, for a specified protocol are received. If protocol field of `socket()` is set to ETH_P_ALL, the Ethernet accepts all the broadcast packets and unicast packets addressed to it. Now to get all the packets, which are not even intended to the host but reached somehow to the interface, the interface is to be set in promiscuous mode.

Before setting an interface in promiscuous mode, however, we need information regarding the interface. To get the informations, `ioctl()`, the function of controlling I/O (input output) devices, is called.

```
#include <sys/ioctl.h>

int ioctl(int d, int request, ...);
```

In our case, the first argument is a valid packet socket descriptor, and second argument, a device-dependent *request* code, is set to SIOCGIFINDEX. The third argument is a pointer whose type depends on the *request*. The requested data is stored in a memory location, pointed by the pointer.

The structure `ifreq` is used to store the interface related data. Suppose, the interface name is *eth0*[4]. Following lines are required to get the interface information. The header file `<linux/if.h>` is to be included for the use of `struct ifreq`.

[4]To know about active interfaces in a computer, use `ifconfig` (or `/sbin/ifconfig`) command.

```
struct ifreq req;
strcpy(req.ifr_name,"eth0");
io = ioctl(sockfd,SIOCGIFINDEX,&req);
if(io<0)
        printf("The interface eth0 does not exist");
```

While `ioctl()` is called with SIOCGIFINDEX request, the `req.ifr_ifindex` field for interface id is filled up. With other request options in `ioctl()`, other informations of the interface (like hardware address, broadcast address) can be received.

However, one may be interested to get packets only from the specified interface. To do this, the packet socket is to be attached with the interface. The function `bind()` [5] is used to attach the interface. To bind the packet socket to an interface, one has to specify an address in a `struct sockaddr_ll`. Only the `sll_protocol` and `sll_ifindex` address fields are used for purposes of binding.

```
struct sockaddr_ll pla;

pla.sll_family=PF_PACKET;
pla.sll_protocol=htons(ETH_P_ALL);
pla.sll_ifindex=req.ifr_ifindex;
b=bind(sockfd,(struct sockaddr *)&pla,sizeof(pla));
if(b<0)
      perror("bind");
```

Finally, the `setsockopt()` function is called to set the socket in promiscuous mode. For this purpose, `setsockopt()` expects *packet_mreq* structure as argument:

```
struct packet_mreq {
       int             mr_ifindex;    /* interface index */
       unsigned short  mr_type;       /* action */
       unsigned short  mr_alen;       /* address length */
       unsigned char   mr_address[8]; /* physical layer address */
 };
```

The `mr_ifindex` contains the interface index for the interface whose status should be changed. The `mr_type` parameter specifies which action to perform. PACKET_MR_PROMISC enables receiving all packets (promiscuous mode), PACKET_MR_MULTICAST binds the socket to the physical layer multicast group specified in `mr_address` and `mr_alen`, and PACKET_MR_ALLMULTI sets the socket up to receive all multicast packets arriving at the interface.

Following lines can set an interface in promiscuous mode. To use `struct packet_mreq`, the header file `<netpacket/packet.h>` or `<linux/if_packet.h>` is to be included.

```
struct packet_mreq mr;

mr.mr_ifindex=req.ifr_ifindex;
mr.mr_type=PACKET_MR_PROMISC;
s = setsockopt(sockfd,SOL_PACKET,PACKET_ADD_MEMBERSHIP,(void *)&mr,sizeof(mr
if(s<0)
        perror("Promiscuous Mode");
```

[5] The `bind()`, widely used in TCP socket, is also discussed in Chapter 4.

While the interface is set in promiscuous mode, all the packets reached somehow at the interface can be captured. Various information of those packets, like source and destination hardware addresses, upper layer protocols etc. of the packets can be received.

2.4 Exercises

1. Write a program to capture all the packets in promiscuous mode.
2. Modify above program to report the upper layer protocol of each captured packet. (*Hints: Use* `sll_protocol` *field of* `struct sockaddr_ll` *to report the protocols. The protocol numbers are found in the* linux/if_ether.h *library file.*)
3. Prepare and inject junk Ethernet packets into the network. (*)
4. Write a program like *ifconfig* to get informations about the interfaces in a computer.

Chapter 3

Accessing Network layer

Application layer protocols generally interact with transport layer protocols, like TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). However, there are some powerful applications (for example, ping program) which directly interact with IP (Internet Protocol), the primary protocol of network layer. The sockets that provide the access to the network layer are called as *raw socket*. In this chapter, we discuss about the raw sockets, and how to insert any IP protocol based datagram into the network traffic.

3.1 The protocols IP, ICMP, TCP and UDP

Before proceeding further, we now introduce 4 important protocols of TCP/IP protocol stack – IP, ICMP (Internet Control Message Protocol), TCP and UDP. The detail discussion on these protocols can be found in any standard text book on Internet or Computer Networks. Here we introduce in short the protocol headers and there standard structures in Unix.

To inject our own packets, we need to know the structures of the protocols to be included. It is recommended to build the packet by using a structure, so that one can comfortably fill in the packet headers. Unix systems provide standard structures in the header files (eg. `struct ip`). One can always create his/her own structures, as long as the length of each option is correct. Here we shall use the little endian notation. On big endian machines (some other processor architectures than intel x86), the 4 bit-size variables exchange places. However, one can always use the structures in the same ways in this program. Here we have used BSD structure names. Linux systems support all these structures, but they also have their own structures. In a program, either BSD style or Linux style is to be followed; mixing up of them would create problems. Below the members of each header structure (in BSD style) are briefly explained, so that one can know what values should be filled in and which meaning they have.

IP

The Internet Protocol (IP) is the Internet/Network layer protocol, used for routing the data from the source to its destination. Every datagram contains an IP header followed by a transport layer protocol such as TCP. To form a packet with IP header in raw socket, a structure (`struct ip`) is utilized which is defined in $< netinet/ip.h >$ header file:

```
struct ip {
```

```
            u_int   ip_hl:4, ip_v:4; /* this means, each member is 4 bit long */
            u_char  ip_tos;
            u_short ip_len;
            u_short ip_id;
            u_short ip_off;
            u_char  ip_ttl;
            u_char  ip_p;
            u_short ip_sum;
            struct  in_addr ip_src, ip_dst;
};      /* total ip header length: 20 bytes (=160 bits) */
```

The data types/sizes are used: unsigned char (`u_int`, `u_char`), unsigned short int (`u_short`) and unsigned int (`struct in_addr`). The fields in the structure are explained below.

1. `ip_hl`: IP header length in 32-bit octets (i.e. (value set in ip_hl) * 4 = header length in bytes). Common Defaults: 5; sets header length to 20 bytes (header length without any routing options)

2. `ip_v`: Internet Protocol version. Default is always 4

3. `ip_tos`: *Type of Service* controls the priority of the packet. The first 3 bits stand for routing priority, the next 4 bits for the type of service (delay, throughput, reliability and cost). Common Defaults: 0x00 (normal)

4. `ip_len`: Length contains the total length of the IP datagram. This includes IP header, ICMP or TCP or UDP header and payload size in bytes

5. `ip_id`: The id sequence number is mainly used for reassembly of fragmented IP datagrams. When sending single datagrams, each can have an arbitrary ID

6. `ip_off`: The fragment $offset$ is used for reassembly of fragmented datagrams. The first 3 bits are the fragment flags, the first one always 0, the second the do-not-fragment bit (set by $ip\_off \mid = 0x4000$) and the third the more-flag or more-fragments-following bit ($ip\_off \mid = 0x2000$). The following 13 bits is the fragment offset, containing the number of 8-byte big packets already sent.

7. `ip_ttl`: The *Time To Live* is the amount of hops (routers to pass) before the packet is discarded, and an ICMP error message is returned. The maximum is 255.

8. `ip_p`: This field specifies the *protocol* to which the IP datagram is to be delivered for necessary action. The protocols can be TCP (6), UDP (17), ICMP (1), or whatever protocol follows the ip header. Look in /etc/protocols for more

9. `ip_sum`: The datagram checksum for the whole ip datagram. Every time anything in the datagram changes, it needs to be recalculated, or the packet will be discarded by the next router.

10. `ip_src` and `ip_dst`: The source and destination IP addresses, converted to long format, e.g. by inet_addr().

ICMP

Internet Control Messaging Protocol (ICMP) is an Internet/Network layer protocol which is merely an addition to IP to carry error, routing and control messages and data. IP itself has no mechanism for establishing and maintaining a connection, or even containing data as direct payload. The header structure of ICMP, defined in < *netinet/ip_icmp.h* >, is shown below.

```
struct icmp {
           unsigned char icmp_type;
           unsigned char icmp_code;
           unsigned short int icmp_cksum;
           /* The following data structures are ICMP type specific */
           unsigned short int icmp_id;
           unsigned short int icmp_seq;
}; /* total icmp header length: 8 bytes (=64 bits) */
```

Each field of the above structure is explained below.

1. `icmp_type`: The message type, for example 0 - echo reply, 8 - echo request, 3 - destination unreachable. See `netinet/ip_icmp.h` to get all message types.
2. `icmp_code`: This is significant when sending an error message (unreach), and specifies the kind of error. Again, consult the include file (`netinet/ip_icmp.h`) for more.
3. `icmp_cksum`: The checksum for the icmp header + data; same as the IP checksum. Note: The next 32 bits in an icmp packet can be used in many different ways. This depends on the icmp type and code. The most commonly seen structure, an ID and sequence number, is used in echo requests and replies, hence we only use this one, but the header is actually more complex.
4. `icmp_id`: used in echo request/reply messages, to identify the request.
5. `icmp_seq`: identifies the sequence of echo messages, if more than one is sent.

UDP

The User Datagram Protocol (UDP) is a transport protocol for sessions that need to exchange data. Both transport protocols, UDP and TCP provide 65535 different source and destination ports. The destination port is used to connect to a specific service on that port. Unlike TCP, UDP is not reliable, since it doesn't use sequence numbers and stateful connections. This means UDP datagrams can be spoofed, and might not be reliable (e.g. they can be lost unnoticed), since they are not acknowledged using replies and sequence numbers. Following is the structure, defined in < *netinet/udp.h* >.

```
struct udphdr {  /* For BSD Unix */
       u_short uh_sport;              /* source port */
       u_short uh_dport;              /* destination port */
       u_short uh_ulen;               /* udp length */
       u_short uh_sum;               /* udp checksum */
};
```

```
struct udphdr  {     /* For Linux */
       u_short source;
       u_short dest;
       u_short len;
       u_short check;
};
```

Here `u_short` is the typedef of unsigned short int.

TCP

The Transmission Control Protocol (TCP) is the mostly used transport protocol that provides mechanisms to establish a reliable connection with some basic authentication, using connection states and sequence numbers. Like UDP, a structure (`struct tcphdr`) is defined to form one's own TCP header in raw socket. There are two variant of the structure - one for Linux, another for BSD Unix. Following is the BSD version, defined in $< netinet/tcp.h >$ of the structure.

```
struct tcphdr {
        u_short th_sport;
        u_short th_dport;
        tcp_seq th_seq;
        tcp_seq th_ack;
        u_int   th_x2:4, th_off:4; /* each member is 4-bit long */
        u_char  th_flags;
        u_short th_win;
        u_short th_sum;
        u_short th_urp;
};   /* total tcp header length: 20 bytes (=160 bits) */
```

The data types/sizes are used: unsigned char (`u_int`, `u_char`), unsigned short int (`u_short`) and unsigned long int (`tcp_seq`). The fields in the structure are explained next.

1. `th_sport`: The source port, which has the same function as in UDP.
2. `th_dport`: The destination port, which has the same function as in UDP.
3. `th_seq`: The sequence number is used to enumerate the TCP segments. The data in a TCP connection can be contained in any amount of segments (=single TCP datagrams), which will be put in order and acknowledged. For example, if you send 3 segments, each containing 32 bytes of data, the first sequence would be (N+)1, the second one (N+)33 and the third one (N+)65. "N+" because the initial sequence is random.
4. `th_ack`: Every packet that is sent is acknowledged with the ACK flag set (see below). The `th_ack` field contains the previous `th_seq` number.
5. `th_x2`: This is unused and contains binary zeroes.
6. `th_off`: The segment offset specifies the length of the TCP header in 32bit/4byte blocks. Without TCP header options, the value is 5.
7. `th_flags`: This field consists of six binary flags. Using BSD headers, they can be combined like this: `th_flags` = $FLAG1 \mid FLAG2 \mid FLAG3 \cdots$

TH_URG: Urgent. Segment will be routed faster, used for termination of a connection or to stop processes (using telnet protocol).

TH_ACK: Acknowledgement. Used to acknowledge data and in the second and third stage of a TCP connection initiation (see IV.).

TH_PSH: Push. The systems IP stack will not buffer the segment and forward it to the application immediately (mostly used with telnet).

TH_RST: Reset. Tells the peer that the connection has been terminated.

TH_SYN: Synchronization. A segment with the SYN flag set indicates that client wants to initiate a new connection to the destination port.

TH_FIN: Final. The connection should be closed, the peer is supposed to answer with one last segment with the FIN flag set as well.

8. `th_win`: Window. The amount of bytes that can be sent before the data should be acknowledged with an ACK before sending more segments.

9. `th_sum`: The checksum of pseudo header, TCP header and payload. The pseudo header is a structure containing IP source and destination address, 1 byte set to zero, the protocol (1 byte with a decimal value of 6), and 2 bytes (unsigned short) containing the total length of the TCP segment.

10. `th_urp`: Urgent pointer. Only used if the urgent flag is set, else zero. It points to the end of the payload data that should be sent with priority.

Following is the Linux format of the `struct tcphdr`.

```
struct tcphdr {
unsigned short source;
unsigned short dest;
unsigned long seq;
unsigned long ack_seq;
        #  if __BYTE_ORDER == __LITTLE_ENDIAN
        unsigned short res1:4;
        unsigned short doff:4;
        unsigned short fin:1;
        unsigned short syn:1;
        unsigned short rst:1;
        unsigned short psh:1;
        unsigned short ack:1;
        unsigned short urg:1;
        unsigned short res2:2;
        #  elif __BYTE_ORDER == __BIG_ENDIAN
        unsigned short doff:4;
        unsigned short res1:4;
        unsigned short res2:2;
        unsigned short urg:1;
        unsigned short ack:1;
        unsigned short psh:1;
        unsigned short rst:1;
        unsigned short syn:1;
        unsigned short fin:1;
        #  endif
```

```
unsigned short window;
unsigned short check;
unsigned short urg_ptr;
};
```

If you have a hybrid Linux/BSD header file, to use the BSD format, add `#define __USE_BSD` and `#define __FAVOR_BSD` at the very top of your definitions or as a compiler flag (e.g. *gcc -D__USE_BSD -D__FAVOR_BSD*), to ensure the flags are set before any use of `#include <netinet/tcp.h>`. Mac OS X systems include only BSD style, whereas Linux installations typically include both forms, making the BSD format more portable.

3.2 Raw sockets

The basic concept of low level sockets is to send a single packet at one time, with all the protocol headers filled in by the program (instead of the kernel). We create a raw socket with the following parameters to `socket()`.

```
int s;
s = socket (PF_INET, SOCK_RAW, protocol);
```

where *protocol* is one of the constants `IPPROTO_xxx` defined by including the <netinet/in.h> header, such as IPPROTO_UDP. While the raw socket is created, one can send his/her own IP packets over it, and receive IP packets having a *protocol* field that match with the *protocol* specified during the creation of raw socket. That means, for listening to TCP, UDP and ICMP traffic, you have to create 3 separate raw sockets, using IPPROTO_TCP, IPPROTO_UDP and IPPROTO_ICMP (the protocol numbers are 6 for TCP, 17 for UDP and 1 for ICMP). If a raw socket is created with a *protocol* of 0, then that socket receives a copy of every raw datagram that the kernel passes to raw sockets. The first parameter of `socket()` is the *protocol family*, which is set to `PF_INET` (PF_INET stands for Protocol Family of InterNET), and the second parameter is the *socket type* which is `SOCK_RAW` for creating raw socket.

However, only superuser can create a raw socket. This prevents normal users from writing their own IP datagrams to the network. Following is the code of a small sniffer, that dumps out the contents of all TCP packets, received. The IP and TCP headers are excluded from the application data. It also prints the source IP adress of the received packet.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>

int main()
{
int fd = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);
struct ip *iph;
char buffer[8192]; /* single packets are usually not bigger than 8192 bytes */

while (recvfrom (fd, buffer, 8192, 0, NULL, NULL) > 0)
     {
```

```
        iph = (struct ip*) buffer;
        printf("Received packet from %s\n", inet_ntoa(iph->ip_src));
        printf ("TCP data: %s\n", buffer+sizeof(struct ip)+sizeof(struct tcphdr));
     }
}
```

To receive the packets, `recvfrom()` is used in the above program. However, we can use `read()`, `recv()` instead of `recvfrom()`. The IP header of the received packet is in the front of *buffer*. To read the header, we typecast the buffer. We then use `inet_ntoa()` to print the source IP address of received packet in dotted decimal format.

If one wants to inject IP packets into the network, the `IP_HDRINCL` socket option can be set.

```
const int on = 1;
if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0)
error;
```

The IP_HDRINCL can be set to inform the kernel that the header is included in the data, so do not insert its own header into the packet before the data. If the option is not set, then the kernel builds the IP header in front of data to be injected into the network.

3.3 Building and injecting datagrams

Now, by putting together the knowledge about the protocol header structures with some basic C functions, it is easy to construct and send any datagram(s). We will demonstrate this with a small sample program that constantly sends out SYN requests to one host (Syn flooder). The SYN request is used to establish TCP connection between client and server (see Chapter 4).

```
#define P 25 /* lets flood the sendmail port */

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#define __FAVOR_BSD /* use bsd'ish tcp header. if not defined,
                       system's default of tcp header will be taken.
                       For example, linux will take the linux default,
                       where struct tcphdr members are different.
                      */
#include <netinet/tcp.h>

unsigned short /* this function generates header checksums */
csum (unsigned short *buf, int nwords)
{
  unsigned long sum;
  for (sum = 0; nwords > 0; nwords--)
    sum += *buf++;
  sum = (sum >> 16) + (sum & 0xffff);
```

```
  sum += (sum >> 16);
  return ~sum;
}

int
main (void)
{
  int s = socket (PF_INET, SOCK_RAW, IPPROTO_TCP); /* open raw socket */
  char datagram[4096]; /* this buffer will contain ip header, tcp header,
   and payload. we'll point an ip header structure
   at its beginning, and a tcp header structure after
   that to write the header values into it */
  struct ip *iph = (struct ip *) datagram;
  struct tcphdr *tcph = (struct tcphdr *) datagram + sizeof (struct ip);
  struct sockaddr_in sin;
/* the sockaddr\_in containing the dest. address is used
   in sendto() to determine the datagrams path */

  sin.sin_family = AF_INET;
  sin.sin_port = htons (P);/* you byte-order >1byte header values to network
      byte order (not needed on big endian machines) */
  sin.sin_addr.s_addr = inet_addr ("127.0.0.1");

  memset (datagram, 0, 4096); /* zero out the buffer */

/* we'll now fill in the ip/tcp header values, see above for explanations */
  iph->ip_hl = 5;
  iph->ip_v = 4;
  iph->ip_tos = 0;
  iph->ip_len = sizeof (struct ip) + sizeof (struct tcphdr); /* no payload */
  iph->ip_id = htonl (54321); /* the value doesn't matter here */
  iph->ip_off = 0;
  iph->ip_ttl = 255;
  iph->ip_p = 6;
  iph->ip_sum = 0; /* set it to 0 before computing the actual checksum later */
  iph->ip_src.s_addr = inet_addr ("1.2.3.4");/* SYN's can be blindly spoofed */
  iph->ip_dst.s_addr = sin.sin_addr.s_addr;
  tcph->th_sport = htons (1234); /* arbitrary port */
  tcph->th_dport = htons (P);
  tcph->th_seq = random ();/* in a SYN packet, the sequence is a random */
  tcph->th_ack = 0;/* number, and the ack sequence is 0 in the 1st packet */
  tcph->th_x2 = 0;
  tcph->th_off = 0; /* first and only tcp segment */
  tcph->th_flags = TH_SYN; /* initial connection request */
  tcph->th_win = htonl (65535); /* maximum allowed window size */
  tcph->th_sum = 0;/* if you set a checksum to zero, your kernel's IP stack
      should fill in the correct checksum during transmission */
  tcph->th_urp = 0;

  iph->ip_sum = csum ((unsigned short *) datagram, iph->ip_len >> 1);
```

```
/* finally, it is very advisable to do a IP_HDRINCL call, to make sure
   that the kernel knows the header is included in the data, and doesn't
   insert its own header into the packet before our data */

  { /* lets do it the ugly way.. */
    int one = 1;
    const int *val = &one;
    if (setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof (one)) < 0)
      printf ("Warning: Cannot set HDRINCL!\n");
  }

  while (1)
    {
      if (sendto (s,/* our socket */
  datagram,/* the buffer containing headers and data */
  iph->ip_len,/* total length of our datagram */
  0,/* routing flags, normally always 0 */
  (struct sockaddr *) &sin,/* socket addr, just like in */
  sizeof (sin)) < 0) /* a normal send() */
printf ("error\n");
      else
printf (".");
    }

  return 0;
}
```

3.4 Ping program

The *ping* is a publicly available program that is used to check whether a host (or router/gateway) is reachable from the host. The operation of ping is very simple: ping uses the ICMP protocols mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams (pings) have an IP and ICMP header, followed by a struct timeval and then an arbitrary number of pad bytes used to fill out the packet. Here a different version of ping program is presented.

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

```
unsigned short checksum(unsigned short *addr, int len);


int main(int argc, char* argv[]) {
  int len, ip_len, n;
  char buff[1024];
  struct ip *iphdr;
  struct icmp *icmphdr;
  int sockfd;
  struct addrinfo hints, *res;
  struct sockaddr_in raddr;


  /* check args */
  if (argc < 2) {
    fprintf(stderr, "Usage: sping <ip_address>\n");
    exit(EXIT_SUCCESS);
  }

  /* get socket info */
  if ( (sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) == -1) {
    fprintf(stderr, "Cannot create raw socket. You must be root.\n");
    exit(EXIT_FAILURE);
  }

  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_flags = AI_CANONNAME;
  hints.ai_flags = 0;
  hints.ai_socktype = 0;

  if ( (n = getaddrinfo(argv[1], NULL, &hints, &res)) != 0) {
    fprintf(stderr, "Error in getaddrinfo()\n");
    exit(EXIT_FAILURE);
  }

  /* fill in icmp header info */
  icmphdr = (struct icmp *) buff;
  icmphdr->icmp_type = ICMP_ECHO;
  icmphdr->icmp_code = 0;
  icmphdr->icmp_id = 0;
  icmphdr->icmp_seq = 0;

  len = 8;
  icmphdr->icmp_cksum = 0;
  icmphdr->icmp_cksum = checksum((u_short*) icmphdr, len);

  /* send the packet! */
  sendto(sockfd, buff, len, 0, res->ai_addr, res->ai_addrlen);

  /* wait for reply packet and set timeout */
```

```
  memset(buff, 0, 1024);
   n = sizeof(struct sockaddr_in);
  recvfrom(sockfd, buff, 1024, 0, (struct sockaddr *) &raddr, &n);

  /* print results to screen */
  iphdr = (struct ip *)buff;
  ip_len = iphdr->ip_hl << 2;
  icmphdr = (struct icmp *) (ip_len + buff);
  if (icmphdr->icmp_type == ICMP_ECHOREPLY)
    printf("Alive.\n", argv[1]);
  else
    printf("%s is not responding.\n", argv[1]);

  printf("message received from %s\n",inet_ntoa(raddr.sin_addr));

  close(sockfd);

  return 0;
}


/*-------------------------------------------------------------- CHECKSUM() --
 *  Internet checksum function. Taken from Steven's UNIX Network
 *  Programming book.
 */
unsigned short checksum(unsigned short *addr, int len) {
  int nleft = len;
  int sum = 0;
  unsigned short *w = addr;
  unsigned short answer = 0;

  /*
   * Our algorithm is simple, using a 32 bit accumulator (sum), we add
   * seqential 16 bit words to it, and at the end, fold back all the
   * carry bits from the top 16 bits into the lower 16 bits.
   */
  while (nleft > 1) {
    sum += *w++;
    nleft -= 2;
  }

  /* mop up an odd byte, if necessary */
  if (nleft == 1) {
    *(unsigned char *) (&answer) = *(unsigned char *) w;
    sum += answer;
  }

  /* add back carry outs from top 16 bits to low 16 bits */
  sum = (sum >> 16) + (sum & 0xffff);  /* add hi 16 to low 16 */
  sum += (sum >> 16);                  /* add carry */
```

```
  answer = ~sum;                          /* truncate to 16 bits */

  return answer;
}
```

3.5 Exercise

1. Write a program that reads all TCP, UDP and ICMP packets arriving in the host and outputs the packet type (TCP, UDP or ICMP) and sender's IP address.

2. Write a program to capture all IP packets. Now run this program and run another program that capture all DLL packets. Identify which packets are absent here.

3. Run the *ping* program described above. Point out the limitations (comparing with available *ping* program) and modify the program to overcome those limitations.

4. Write a program to trace route between two machines. The output should be a list of all routers along a path to a given destination.
   Hints: The Time-To-Live field (`th_ttl`) in an IP header is used to recover from routing errors. When the datagram passes a router, the Time-To-Live field is decremented by 1. If it reaches to 0, the router discards the datagram and sends ICMP *time exceeded* (ICMP_TIMXCEED) message back to the sender. The sender now can get the router's IP address.

Chapter 4

TCP Sockets

Two or more networked computers can interact among themselves while some network application programs are executed on those computers. The programs written using packet socket (Chapter 2) and raw socket (Chapter 3) are generally used to monitor and administer the networks. Almost all application programs, however, are written by using TCP sockets (also known as stream socket) and UDP socket (datagram socket).

The network application processes, developed using TCP and UDP sockets, can be divided into two classes – *clients* and *servers*. A number of architecture, like peer-to-peer architecture, single tier client-server architecture, two tier client-server architecture etc., are proposed to develop network applications. However, basic components of these architecture are client processes and server processes. Client initiates communication by requesting the server for some service. The server then replies the client. This scenario is depicted in Figure 4.1.

Since TCP (Transmission Control Protocol) is connection oriented protocol, communication between client and server using TCP is reliable. Unreliable communication channel and lower layer protocols do not affect the reliability of TCP. The protocol virtually forms a tube through which a stream of data flow between client and server. So, the TCP socket is also called as stream socket.

A TCP socket is defined as an endpoint for communication. A connection can be viewed as the pair <IP Address, Port>. An IP address locates the computer in the network, whereas a port number, a 16-bit integer, identifies a particular application layer protocol running on that machine.[1] On Unix machines, the file /etc/services contains a list of services provided by that machine, along with the ports, used by the services.

4.1 TCP Clients and Servers

The basic building blocks of network programming using TCP are TCP clients and servers. The client initiates communication by requesting the server for some service. The server then replies the client (see Figure 4.1). A TCP connection can be thought as a pair of connections – one from client to server and other from server to client. A server listens on a port, waiting for incoming requests from clients.

For example, a web server listens at port 80 for incoming request from clients (web browsers). When a client wishes to make a connection with a server, the client is assigned a port by the

[1] See Section 1.2 for the discussions on IP addresses and port numbers.

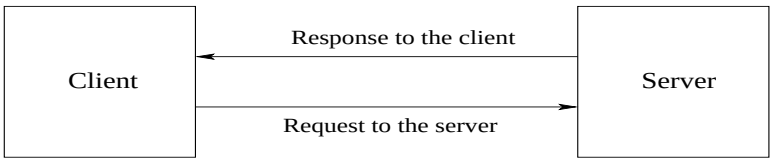

Figure 4.1: Communication between client and server

local host. Suppose that a client C (at IP address 146.86.3.15) wishes to browse a web page on the server 146.86.5.20. If the port 12345 is assigned to C by the local host of C, the connection between the client and the server is uniquely identified by the following:

$< 146.86.3.15 : 12345, \quad 146.86.5.20 : 80 >$

To establish connection between client and server, the server is to be ready first to accept the connection request from client. A number of socket APIs are to be called to prepare the server. While server is ready and client makes request for service, a three way handshaking[2] is performed to establish the connection between client and server. The client always initiates the communication.

4.2 Socket address

The sockets, created by some application program, are uniquely identified by the socket addresses. All the communication between two processes in the network (client and server) are carried out by the socket address.

A socket address is the combination of an IP address and a port (which is mapped to the application process) into a single identity. The socket address structures are defined in Unix/Linux. Most of the socket APIs require a pointer to a socket address structure as an argument. An IPv4 socket address structure, commonly called an "Internet socket address structure" is named `sockaddr_in` and defined in `<netinet/in.h>` header.

```
struct in_addr {
       in_addr_t    s_addr;    /*  32-bit IP address; network byte ordered */
};

struct sockaddr_in {
       uint8_t        sin_len;     /* length of structure */
       sa_family_t    sin_family;  /* PF_INET */
       in_port_t      sin_port;    /* TCP/UDP port number; network byte ordered */
       struct in_addr sin_addr;    /* IP address */
       char           sin_zero[8]; /* unused */
};
```

This definition is noted in Posix unix standard. Description of the datatype is noted in the following table:

[2]See any text book on computer networks for *three way handshaking*

Datatype	Description	Defined in
`init8_t`	signed 8-bit integer	< *sys/types.h* >
`uint8_t`	unsigned 8-bit integer	< *sys/types.h* >
`init16_t`	signed 16-bit integer	< *sys/types.h* >
`int32_t`	signed 32-bit integer	< *sys/types.h* >
`unit32_t`	unsigned 32-bit integer	< *sys/types.h* >
`sa_family_t`	address family of socket address structure	< *sys/socket.h* >
`socklen_t`	length of socket address structure, normally `uint32_t`	< *sys/socket.h* >
`in_addr_t`	IPv4 address, normally `uint32_t`	< *sys/types.h* >
`in_port_t`	TCP or UDP port, normally `uint16_t`	< *sys/types.h* >

Before using this structure in socket APIs, the fields of the structure are to filled up with the appropriate values and those values are to be in proper format. For example, the *s_addr* field can not be set with the dotted decimal format of an IP address, rather it is to be filled with *network byte ordered* format (that is, *big-endian format*, see Section 4.4).

`socket(), bind(), listen(), accept()` and `socket(), connect()` are the elementary socket APIs required for server and client respectively. The APIs are defined in <sys/socket.h>. The APIs are to be called in a particular sequence. The sequence is shown in Figure 4.2. It is clear from the figure that after execution of `socket(), bind(), listen(), accept()` APIs in server side, a client can request for connection. The socket APIs with its argument are discussed in the next section. However, `read()` and `write()` are used here for data transfer. There are other functions for the same – `recv(), recvfrom(), send(), sendto()`.

4.3 TCP Socket APIs

The APIs, shown in Figure 4.2, are discussed below.

socket:

```
# include <sys/socket.h>

int socket(int family, int type, int protocol);
```

Returns: non-negative integer if successful, -1 on error.

In short, this function creates an end point for communication. The return value from this function is a handle (an integer value) to a socket. This integer is passed as an argument to all of the other socket APIs. The *family* parameter should be set to AF_INET (Address Family for InterNET) or PF_INET (Protocol Family for InterNET). Practically both, AF_INET and PF_INET are same, and any one can be used. The *type* parameter can be either SOCK_STREAM (for TCP), or SOCK_DGRAM (for UDP). The *protocol* field is intended for specifying a specific protocol in case the network model support different types of stream and datagram models. However, TCP/IP only has one protocol for each, so this field should always be set to 0. Following is an example to create a TCP socket:

```
int s;
s = socket(AF_INET, SOCK_STREAM, 0);
```

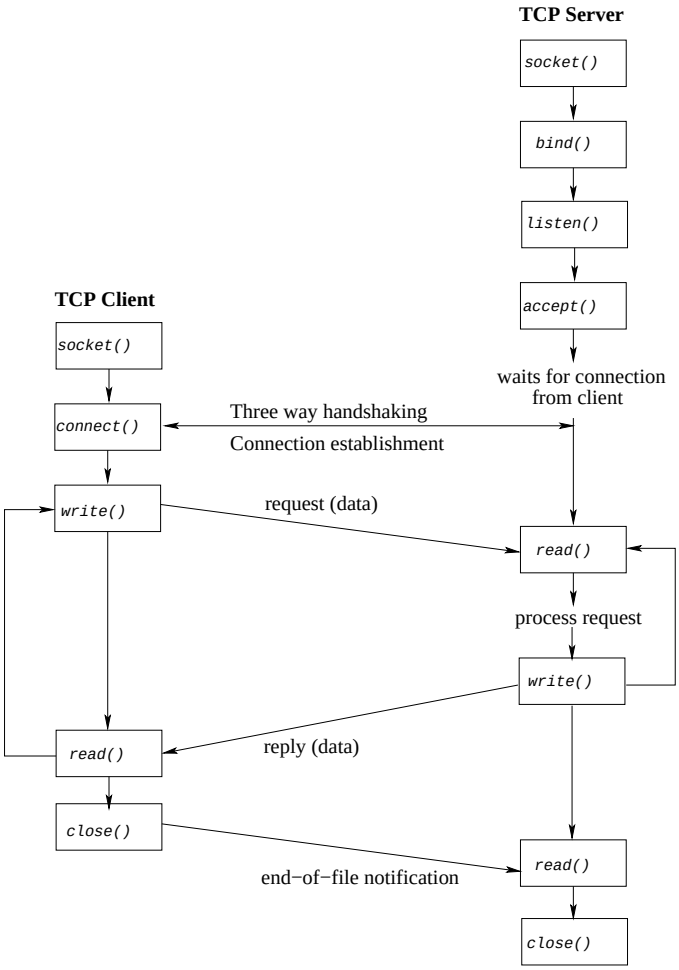

Figure 4.2: Socket APIs for elementary TCP client-server

connect:

```
# include <sys/socket.h>

int connect(int s, const struct sockaddr *addr, int addr_len);
```
Returns: 0 if successful, -1 on error.

This API, issued by a client, establishes a connection with the server. The first argument is a handle, created by *socket*(). The address of the server is specified in a structure, pointed by *addr*, and its size is noted in *addr_len*. The `connect()` blocks until either the connection is successfully established, or an error occurs. If successful, the socket descriptor gets ready for reading and writing. The *connect*() in client side and the *accept*() in server side are directly involved in connection establishment through *three-way handshaking*. Example of a connect call:

```
int c;
struct sockaddr_in server_addr;

s = socket(AF_INET, SOCK_STREAM, 0);
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(9999);
inet_pton(AF_INET,"146.86.5.20",&server_addr.sin_addr);

c = connect(s, (struct sockaddr*)&server_addr, sizeof(sockaddr_in));
if(c<0)
{
printf("connect error");
exit(0);
}
```

The socket address structure is filled properly before calling *connect*(). Port number and IP address are considered arbitrarily. It can be noted that `server_address` is type casted into `struct sockaddr`. The reason is, `sockaddr` is a *generic* socket address structure, whereas `sockaddr_in` is IPv4 Internet socket address structure. The function `inet_pton()` is discussed in next section.

bind:

The `bind()` is commonly called by TCP server. Before sending and receiving data using a socket, it must first be associated with a local source port and a network interface address. The mapping of a socket to a TCP/UDP source port and IP address is called a "binding". The prototype for bind is as follows:

```
# include <sys/socket.h>

int bind(int s, const struct sockaddr *myaddr, int addr_len);
```
Returns: 0 if successful, -1 on error.

The first argument is a socket handle (the number returned from `socket()`). The second argument specifies the server's address, and its size is the third argument. The address includes a port number and an IP address. If specifying an exact source port is not required, setting this value to INADDR_ANY(0) allows the operating system to pick any available port number. The `sin_addr` field specifies which network interface device to use. Since most hosts only have one network interface and only one IP address, this field should be set with the host's own IP address. However, the socket library provides no immediate way for a host to determine its own IP address! However, specifying the value of INADDR_ANY(0) in this field tells the operating system to pick any available interface and address.

Example:

```
struct sockaddr_in sin;
int s, b;


s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_port = htons(9999);
sin.sin_addr.s_addr = INADDR_ANY;
b = bind(s, (struct sockaddr *)&sin, sizeof(sin));
if(b<0)
{
printf("bind error");
exit(0);
}

/* s is now a usable TCP socket. Server port is 9999. */
```

It is recommended that the return from bind be checked; *bind*() will fail by returning -1 if the port that is being requested for use is already taken. If *bind*() is unsuccessful, connection can not be established.

listen:

This API is called only by a TCP server. The *listen*() converts an unconnected socket (prepared by *socket*() and *bind*()) into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. The prototype for `listen()` is as follows:

```
# include <sys/socket.h>


int listen(int s, int backlog);
```

Returns: 0 if successful, -1 on error.

The first argument is a socket handle. The second argument specifies the maximum number of connections that can be pending on the specified socket (typically set to a large value, such as 1024).

accept:

This is a blocking operation that does not return until a remote client has established a connection; when it completes, it returns a new socket that corresponds to this just-established connection. The socket is sometime referred as *accepted socket*. To distinguish with the previous, the socket prepared after `listen()` called *listening socket*. All communication with client is carried out by the connected socket. The prototype for `accept()` is as follows:

```
# include <sys/socket.h>

int accept(int s, struct sockaddr *Caddr, int *Caddr_len);
```

Returns: non-negative descriptor if successful, -1 on error.

After successful execution of *accept*(), the socket address *Caddr* (and its size *Caddr_len*) is filled up with client's information. The *Caddr_len* is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by *Caddr*; on return it will contain the actual size of the client's address. If the server does not want to know the client's information, second and third arguments can be set as NULL.

The *listening socket* can be viewed as a pair server's IP address and server's port number. Since the pair of client's IP address and client's port number is not associated with listening socket, no data exchange between client and server is possible. On the other hand, an *accepted socket* associates both the pairs – a pair of server's IP address and server's port number, and another pair of client's IP address and client's port number. When we get an *accepted socket*, we declare that connection is established. Client and server exchange their data through *accepted socket*.

write, read, send and recv:

Once connection is established, the application processes invoke the following functions to send and receive data:

```
int write(int confd, char *buffer, int msg_len);
int read(int confd, char *buffer, int msg_len);
```

OR

```
int send(int confd, char *buffer, int msg_len, int flags);
int recv(int confd, char *buffer, int buf_len, int flags);
```

The first argument of above functions is the connected socket descriptor, which is returned by `connect()`. The `read()`, `write()` operations are also used in file handling. `send()`, `recv()` take a set of flags that control certain details of the operation.

4.4 Few essential functions

Apart for the socket APIs, a few more functions are almost essential for socket programming. Some functions are discussed here.

IP address conversion

The users deal with the *presentation* format (that is, dotted decimal format) of an IP address. On the other hand, the computers understand only the *network byte order* (that is, big-endian) format. Following functions play the role of conversion between two formats.

```
# include <arpa/inet.h>

int  inet_pton(int family, const char *presentation, void *numeric);
```

Returns: 1 if OK, 0 if input is not a valid presentation format, -1 on error

Above function converts an IP address from the presentation format to network byte order format. Here, *family* is AF_INET, *presentation* is a string points to an IP address (dotted decimal format), and *numeric* is the pointer where the converted IP address is stored. The same task is also performed by `inet_aton(const char *presentation, struct in_addr *numeric)`. However, the following function does the reverse operation.

```
# include <arpa/inet.h>

const char  *inet_ntop(int family, const void *numeric, char *presentation,
socklen_t plen);
```

Returns: pointer to result if OK, NULL on error

This function converts the IP address *numeric* (which is in network-byte order format) into a character string *presentation* of *plen* byte long. The function `char *inet_ntoa(struct in_addr numeric)`, which takes IP address in network-byte order format and returns a string containing IP address in dotted decimal format performs similar task like `inet_ntop()`.

Byte ordering functions

There are two ways to store the bytes of a word in memory – *little-endian* byte order and *big-endian* byte order. In little-endian byte order, the first byte is the low-order byte that holds the LSB. On the other hand, the first byte of big-endian byte order is the high-order byte that holds the MSB. The byte ordering of a computer (host) may be little-endian or big-endian. We refer the byte ordering of a host as *host byte order*. However, the network protocols have to follow a single byte order, so that the sending protocol stack and the receiving protocol stack understand the data sent uniquely. We call this byte order used by the protocols as *network byte order*. The Internet protocols follow big-endian byte ordering.

To fill up the `sockaddr_in` structure, IP address and port number are to be set in proper format. We have already discussed the function that is used to convert an IP address into its proper format. To fill up the *sin_port* field with a 16-bit port number, the port number (supplied by the user in *host byte order* format) is to be converted into network byte order. Following functions perform such conversion.

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

uint32_t  htonl(uint32_t hostlong);

uint16_t  htons(uint16_t hostshort);
```

Return: value in network byte order

In the names of these functions, *h* stands for *host*, *n* stands for *network*, *l* stands for *long*, and *s* stands for *short*. *hostlong* and *hostshort* denote 32-bit (long) integer and 16-bit (short) integer respectively, supplied by the host. Since port number is 16- bit long, *htons*() function is used. However, for the reverse conversion, that is, from network byte order to host byte order, following functions can be used.

```
uint32_t  ntohl(uint32_t netlong);

uint16_t  ntohs(uint16_t netshort);
```

Return: value in host byte order

Apart from the above, some more functions for string operations are used in socket programming. Few of such functions are – *memset*(), *bzero*() (to set the supplied bytes to 0), *memcpy*(), *bcopy*() (to copy the content from source byte to destination byte), etc.

4.5 A simple ftp client-server program

We present here a TCP client and a TCP server program for file transfer. The client uploads a file to the server.

4.5.1 Client code

The client program takes the server's IP address and the file to be uploaded as input. A chunk of data read from the file, and the data is sent to the server. This process continues until the whole is transferred.

```
 1 #include<stdio.h>
 2 #include<sys/socket.h>
 3 #include<arpa/inet.h>
 4
 5 #define MAXLEN  100
 6 #define PORT    12345
 7
 8 int main(int argc, char **argv)
 9 {
10         int s;
11         char buff[MAXLEN], ACK[3];
12         struct sockaddr_in saddr;
13         FILE *fp;
14
15         if(argc!=3)
16         {
17                 printf("Usage:%s <Server's address> <file name>\n",argv[0]);
18                 exit(0);
19         }
```

```
20          s = socket(AF_INET, SOCK_STREAM, 0);
21          if(s<0)
22          {
23                  perror("socket");
24                  exit(0);
25          }
26          saddr.sin_family=AF_INET;
27          saddr.sin_port=htons(PORT);
28          if(inet_pton(AF_INET,argv[1],&saddr.sin_addr)<=0)
29          {
30                  printf("Error. Invalid IP address\n");
31                  exit(0);
32          }
33
34          if(connect(s,(struct sockaddr *)&saddr,sizeof(saddr))<0)
35          {
36                  perror("connet");
37                  exit(0);
38          }
39
40          write(s,argv[2],strlen(argv[2]));
41          if((fp=fopen(argv[2],"r"))==NULL)
42          {
43                  perror(argv[2]);
44                  exit(0);
45          }
46          read(s,ACK,2);
47          while(1)
48          {
49                  fgets(buff,MAXLEN-1,fp);
50                  if(feof(fp))
51                          break;
52                  write(s,buff,sizeof(buff));
53                  read(s,ACK,2);
54          }
55          fclose(fp);
56          close(s);
57  }
```

4.5.2 Server code

The server program does not take any input. It waits for the connection from some client. When connection is established, file is uploaded.

```
1  #include<stdio.h>
2  #include<sys/socket.h>
3  #include<netinet/in.h>
4
5  #define MAXLEN  100
6  #define PORT    12345
```

```
 7
 8 int main(int argc,char **argv)
 9 {
10         int s, connfd, len, n;
11         char recv[MAXLEN], recvaddr[16];
12         struct sockaddr_in saddr, caddr;
13         FILE *fp;
14
15         s=socket(AF_INET, SOCK_STREAM, 0);
16         if(s<0)
17         {
18                 perror("socket");
19                 exit(0);
20         }
21         saddr.sin_family=AF_INET;
22         saddr.sin_addr.s_addr=htonl(INADDR_ANY);
23         saddr.sin_port=htons(PORT);
24
25         if(bind(s,(struct sockaddr *)&saddr,sizeof(saddr))<0)
26         {
27                 perror("bind");
28                 exit(0);
29         }
30         listen(s,5);
31
32         while(1)
33         {
34                 printf("Waiting for request...\n");
35                 len=sizeof(struct sockaddr_in);
36                 connfd=accept(s,(struct sockaddr*)&caddr,&len);
37                 n=read(connfd,recv,MAXLEN-1);
38                 recv[n]='\0';
39                 fp=fopen(recv,"w");
40                 write(connfd,"OK",2);
41                 while((n=read(connfd,recv,MAXLEN-1))>0)
42                 {
43                         recv[n]='\0';
44                         fwrite(recv,1,strlen(recv),fp);
45                         write(connfd,"OK",2);
46                 }
47                 fclose(fp);
48                 inet_ntop(AF_INET,&caddr.sin_addr,recvaddr,sizeof(recvaddr));
49                 printf("File uploaded from %s, port=%d\n\n",recvaddr,ntohs(caddr.
50                 close(connfd);
51         }
52 }
```

In both the programs, we have used `htons()`, `inet_pton()`, `inet_ntop()` function. The first function *htons* (host to network short) is used to order the byte in a specific order (*network byte order*). Other two functions convert the IP address from dotted decimal format to network format (pton – presentation to network) and network format to presentation format

(ntop – network to presentation).

4.5.3 Execution of the programs

Suppose, file names for client and server programs are client.c and server.c respectively. To compile both the programs type

cc client.c -lsocket -lnsl -o client
cc server.c -lsocket -lnsl -o server

OR simply

cc client.c -o client
cc server.c -o server

If cc is unavailable, try the same with gcc. While compilation is complete, type ./server to run the server program.

```
$ ./server
Waiting for request...
```

Next run the client program.

```
$ ./client 10.24.95.238 reco.txt
$
```

When the client program is finished, the output of the server is:

```
$ ./server
Waiting for request...
File uploaded from 10.24.105.3, port=32808

Waiting for request...
```

The server is again waiting for some other request. However, after serving a request, the server prints the client's IP address (10.24.105.3) and port number (32808). Both are not supplied explicitly in the client program. The IP address is the client's IP address and the port number is chosen arbitrarily by the kernel. One may explicitly bind the client with some port number, but this is generally avoided. It is suggested to provide this task to the kernel.

4.6 Error handling

Two problems are commonly encountered while the client and server are executed – connection error and bind error.

connection error

While `connect()` of client side returns -1, then connection error occurs. This means no connection is established between client and server. Following are the probable reasons.

- `socket()` function with proper argument is not called before calling of `connect()`. Successfully called `socket(...)` function returns a socket handler, an non-negative integer, which is passed to `connect()`. If a negative handler is passed to `connect()`, then connection error appears.
- If wrong IP address and/or port number is mentioned, then connection can not be established. The port number in a computer must be unique. The client has to mention the same port during connection, on which the server is running.
- The server is not in the network or the server machine is not reachable from the client machine.
- The server is not started before calling `connect()` in client side. That is, the server is not ready to accept the connection request of client.
- The `sockaddr_in` structure that holds the server address is not set properly. The IP address and port number in this structure must be in specific format: we call the library function `htons` ("host to network short") to convert the binary port number, and we call the library function `inet_pton` ("presentation to numeric") to convert the 4-byte IP address (dotted decimal format) into proper format (See Section 1.2 and Section 4.4 for more details).
- The server machine filters the client machine's IP address. In this situation, some configuration of server machine is to be modified.

bind error

The bind error in server side is a most frequently faced problem. The `bind()` function returns -1 on error. Following are the probable reasons.

- `socket()` function with proper argument is not called before calling of `bind()`. Successfully called `socket(...)` function returns a socket handler, a non-negative integer, which is passed to `bind()`. If a negative handler is passed to `bind()`, then also bind error occures.
- The `sockaddr_in` structure for the server is not set properly. That is, one or more members of the structure are not filled up, or filled with some wrong values (Section 4.4).
- The port number is already in use. The port number is to be unique. The numbers from 1 to 1023 are the well known ports, so these port numbers should not be used in user program. For example, if the web server is installed in the server machine, and the server program is trying with port number 80, then bind error will occur because port 80 is predefined to web server. Few ports are registered port – from 1024 to 49151. The rest ports (49152 to 65535) are the dynamic or private ports. These ports can safely be used in program.

If the the problem is with the port number, then the following socket option can be set before calling `bind()`. If the port number is unique in a program, then also bind error appear for several reasons. The option SO_REUSEADDR allows to reuse the port number.

```
int on = 1;
if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) <
0) error
```

If the problem still remains, you would have to wait for four minutes (in the TIME_WAIT state of TCP) before using the address again (see RFC 1337 for details). However, Linux uses 60 seconds, BSD implementations normally use around 30 seconds. It can be noted that multiple TCP servers can never run on same IP address and port number. At least one of them is to be different.

4.7 Exercise

The following are the exercise for this chapter.

1. Write a EchoClient and EchoServer program using TCP socket, where the server echoes the message which is sent by the client in reverse case (that is, if client says $Hello$, then server replies with $hELLO$), and also print the client's IP address on server's console. Use separate machines on same network for client and server, if possible.
2. Write a simple ftp client and server program, where the client can upload as well as download a file.
3. Modify the above ftp server code by setting proper options, so that port number can be reused and bind error does not appear by saying "address already in use".

Chapter 5

UDP socket

User Datagram Protocol or UDP is a transport layer protocol, which is much simpler than TCP. The protocol is connectionless, unreliable, datagram protocol, quite unlike the connection oriented, reliable byte stream provided by TCP. There is no connection establishment phase between client and server. Unlike TCP, UDP supports broadcasting and multicasting facilities. The application designers consider UDP as an alternative of TCP, where the application is a simple request-reply application. In this case, error detection is to be built into the application to provide reliability. If the application requires broadcasting facility, then UDP is the only option. However, for bulk data transfer, UDP is not a preferred transport protocol, because then congestion control and error control are to be built into application. DNS (Domain Name System), NFS (Network File System), SNMP (Simple Network Management Protocol) are the well-known applications developed using UDP.

5.1 Creating a UDP Socket Application

To design applications using UDP, UDP sockets in both ends, client and server, are to be created. UDP sockets are also referred as datagram socket. There are many similarities of creating a UDP socket with TCP socket. The sequence of function calling to prepare UDP server and client are shown in Figure 5.1. Here, the `socket()` and `bind()` functions or APIs are same with TCP. However the second argument of `socket()` is different with TCP.

```
int s;
s = socket(AF_INET, SOCK_DGRAM, 0);
```

One can compare UDP functions and client-server data exchange (Figure 5.1) with the same of TCP (Figure 4.2). The client does not establish connection with server (Figure 5.1). A UDP server does not have to listen for and accept client connections, and a UDP client does not have to connect to a server. Instead, the client just sends a datagram to the server using `sendto()` function, and server just calls the `recvfrom()` function to prepare to receive data from a client and waits until data arrives from some client[1]. These two functions, defined in `<sys/socket.h>` are described bellow.

[1]The `connect()` may also be used for UDP socket. But this is not similar with TCP connection and there is no three-way handshaking. Instead, the kernel just records the IP address and port number, which are included in the socket address structure posed to `connect()`, and returns immediately to the calling process. However, overloading `connect()` function with capabilities of UDP socket is confusing.

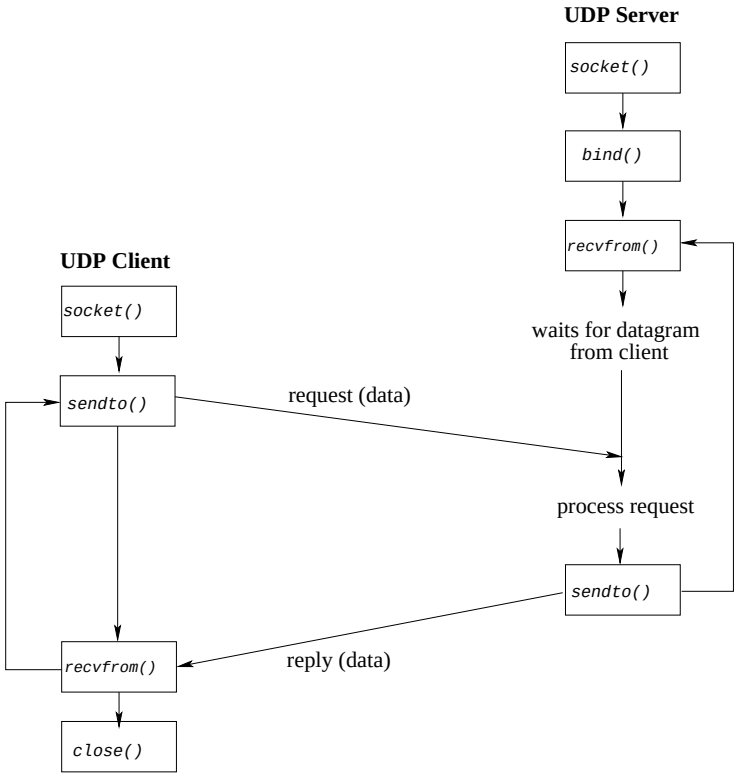

Figure 5.1: Socket functions and their sequence for UDP client and server

```
int  recvfrom(int s, void *buf, size_t len, int flags,
          struct sockaddr *from, socklen_t *fromlen);

int  sendto(int  s,  const  void *buf, size_t len, int flags,
        const struct sockaddr *to, socklen_t tolen);
```

Both return: number of bytes read or written if successful, -1 or error.

The first three arguments, s, $buff$, and $nbytes$ are identical to the first three arguments for `read()` and `write()`: socket descriptor, pointer to buffer to read into or write from, and number of bytes to read or write. The $flags$ argument is normally set to 0. The remaining two arguments of `recvfrom()` are same with the last two arguments of `accept()`, and of `sendto()` are same with the last two arguments of `connect()`.

5.2 Day time UDP client-server

A day time client asks for the current time to the server, and server accordingly replies with its current time. The application is developed using UDP socket. The codes for the client and server are presented next. Note the difference of the following client and server from the TCP client and TCP server of Section 4.5.

5.2.1 The client code

```
 1 #include<sys/socket.h>
 2 #include<arpa/inet.h>
 3 #include<time.h>
 4
 5 #define PORT 1819
 6
 7 int main(int argc,char *argv[])
 8 {
 9         int s,n;
10         char buffer[100];
11         struct sockaddr_in saddr;
12
13         if(argc!=2)
14         {
15                 printf("Usage::<%s> <Serv_IP>\n",argv[0]);
16                 exit(0);
17         }
18         s=socket(AF_INET,SOCK_DGRAM,0);
19         if(s<0)
20         {
21                 perror("socket");
22                 exit(0);
23         }
24         saddr.sin_family=AF_INET;
25         saddr.sin_port=htons(PORT);
26         if(inet_pton(AF_INET,argv[1],&saddr.sin_addr)<=0)
27         {
```

```
28                  printf("Error. Invalid IP address\n");
29                  exit(0);
30          }
31
32          strcpy(buffer,"time");
33          printf("Sending request to server %s\n",argv[1]);
34          if(sendto(s,buffer,strlen(buffer),0,(struct sockaddr *)&saddr,sizeof(sadd
35          {
36                  perror("Send");
37                  exit(0);
38          }
39          printf("Waiting for response...\n");
40          n=recvfrom(s,buffer,sizeof(buffer)-1,0,NULL,NULL);
41          if(n<0)
42          {
43                  perror("Receive");
44                  exit(0);
45          }
46          buffer[n]=0;
47          printf("Reply from server %s is %s\n",argv[1],buffer);
48
49          close(s);
50          return(0);
51  }
```

5.2.2 The server code

Following is the corresponding server code.

```
 1  #include<sys/socket.h>
 2  #include<arpa/inet.h>
 3  #include<time.h>
 4
 5  #define PORT 1819
 6
 7  int main(void)
 8  {
 9          int s,len,n;
10          char buffer[100],recvaddr[16];
11          struct sockaddr_in saddr,caddr;
12          time_t ticks;
13
14          s=socket(AF_INET,SOCK_DGRAM,0);
15          if(s<0)
16          {
17                  perror("socket");
18                  exit(0);
19          }
20
21          saddr.sin_family=AF_INET;
```

```
22          saddr.sin_port=htons(PORT);
23          saddr.sin_addr.s_addr=htonl(INADDR_ANY);
24          if(bind(s,(struct sockaddr *)&saddr,sizeof(saddr))<0)
25          {
26                  perror("bind");
27                  exit(0);
28          }
29          while(1)
30          {
31                  printf("Waiting for request...\n");
32                  len=sizeof(struct sockaddr_in);
33                  n=recvfrom(s,buffer,sizeof(buffer)-1,0,(struct sockaddr *)&caddr,
34                  if(n<0)
35                  {
36                          perror("recv");
37                          break;
38                  }
39                  buffer[n]=0;
40                  inet_ntop(AF_INET,&caddr.sin_addr,recvaddr,sizeof(recvaddr));
41                  printf("\nReceived request from %s\n",recvaddr);
42                  if(strcasecmp(buffer,"time"))
43                  {
44                          strcpy(buffer,"Unknown request.");
45                          printf("%s\n",buffer);
46                          sendto(s,buffer,strlen(buffer),0,(struct sockaddr *)&cadd
47                          continue;
48                  }
49                  ticks=time(NULL);
50                  snprintf(buffer,sizeof(buffer),"%24s",ctime(&ticks));
51                  printf("Serving the request..\n");
52                  if(sendto(s,buffer,strlen(buffer),0,(struct sockaddr *)&caddr,len
53                  {
54                          perror("send");
55                          break;
56                  }
57                  else
58                          printf("Request served..\n\n");
59          }
60          close(s);
61          return 0;
62  }
```

5.2.3 Execution of the codes

The programs can be compiled like before (by typing, cc < *filename* > -o < *output file* >). Suppose, `server` and `client` are the executable files for server and client respectively. Consider, server is running on a machine with IP address < 10.24.95.238 > and client is running in any machine on the same network. Following is an example run of the application.

Run the server first.

```
$ ./server
Waiting for request....
```

Now run the client.

```
$ ./client 10.24.95.238
Sending request to server 10.24.95.238
Waiting for response...
Reply from server 10.24.95.238 is Fri Jul  2 10:07:36 2010
```

5.3 Broadcasting in a network

Broadcasting is sometime required to locate some server in the network, when the specific IP address of the server is unknown. Since point-to-point connection is established before data transmission in TCP, broadcasting with TCP is not possible. But it works with UDP. Suppose, a client wants to get the IP addresses of the time servers, if there is any, running on the network. Then the client broadcasts a UDP request in the network. The request reaches to all the computers, and the computers running the UDP time server respond. Eventually, the client receives the response, and the servers' IP addresses are also received. However, the time servers are to run on the same port. The default port for the UDP time server is 13.

To broadcast a request in a specific network, broadcast address is required. The broadcast address identifies the network with the $net-id$ and $subnet-id$[2]. The host bits of the address are all-1, which indicate the broadcasting. The broadcast address can easily be found by the `ifconfig` utility in Unix/Linux.

The server program is the same as before. Few modifications are required in the client program. Before broadcasting the request, the client has to tell the kernel explicitly regarding broadcasting. This can be done by setting the SO_BROADCAST socket option in the following way:

```
int on = 1;
if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) <
0) error
```

After the broadcast, the client has to receive all the replies. So, a loop is required. The client code to ask the day time is given next.

```
 1 #include<sys/socket.h>
 2 #include<arpa/inet.h>
 3 #include<time.h>
 4
 5 #define PORT 1819
 6
 7 int main(int argc,char *argv[])
 8 {
 9         int s,n,on=1,len;
10         char buffer[100],recvaddr[16];
11         struct sockaddr_in saddr;
```

[2]Consult any text book on *Computer Networks* to understand the net_id and subnet_id.

```
12          if(argc!=2)
13          {
14                  printf("Usage::<%s> <Broadcast address>\n",argv[0]);
15                  exit(0);
16          }
17          s=socket(AF_INET,SOCK_DGRAM,0);
18
19          if(setsockopt(s,SOL_SOCKET,SO_BROADCAST,&on,sizeof(on))<0)
20          {
21                  perror("setsockopt");
22                  exit(0);
23          }
24          saddr.sin_family=AF_INET;
25          saddr.sin_port=htons(PORT);
26          if(inet_pton(AF_INET,argv[1],&saddr.sin_addr)<=0)
27          {
28                  printf("Error. Invalid IP address\n");
29                  exit(0);
30          }
31
32          strcpy(buffer,"time");
33          if(sendto(s,buffer,strlen(buffer),0,(struct sockaddr *)&saddr,
                                                          sizeof(saddr))<0)
34          {
35                  perror("Send");
36                  exit(0);
37          }
38          while(1)
39          {
40                  len=sizeof(struct sockaddr_in);
41                  n=recvfrom(s,buffer,sizeof(buffer)-1,0,
                                          (struct sockadd *)&saddr,&len);
42                  if(n<0)
43                  {
44                          perror("Receive");
45                          exit(0);
46                  }
47                  buffer[n]=0;
48                  inet_ntop(AF_INET,&saddr.sin_addr,recvaddr,sizeof(recvaddr));
49                  printf("Reply from server at:%s is %s\n",recvaddr,buffer);
50          }
51          close(s);
52          return(0);
53  }
```

Consider the broadcast address for a network is 10.24.255.255. Three time servers are running in the network on port 1819. The IP addresses of those servers are 10.24.1.1, 10.24.105.5 and 10.24.107.3. Following is the result of the run of client.

```
$ ./client 10.24.255.255
Reply from server 10.24.1.1 is Fri Jul  2 10:07:36 2010
```

```
Reply from server 10.24.105.5 is Fri Jul  2 10:07:38 2010
Reply from server 10.24.107.3 is Fri Jul  2 10:08:06 2010
```

5.4 Excercise

1. Modify the day time client program, so that it can send the same request to multiple servers. The client should not wait for the response of the a server, rather it should continue to request to anothe server.

2. Write a EchoClient and EchoServer program using UDP socket, where the server echos the message which is sent by the client in reverse case (that is, if client says $Hello$, then server replies with $hELLO$), and also print the client's IP address on server's console. Use separate machines on same network for client and server, if possible.

Chapter 6

Concurrent Server

Chapter 4 and Chapter 5 have described the creation of two types of servers – one is based on TCP socket and another is UDP socket. Both types of servers process the clients' request one by one. If the number of client per server is 1, then these types of servers perform nicely. However, in most of the cases the number clients is more than 1. In this situation also the server may perform well if the clients make request rarely. But if the number of clients is very high and/or the clients make request frequently, the clients can not access the server parallely, and as a result response time of the clients is increased.

The first class of server is called iterative server, i.e. it iterates through each client and serves one request at a time. Alternatively, a server can handle multiple clients at the same time in parallel, and this type of server is called a *concurrent server*. By *concurrent server*, we normally mean TCP concurrent server. However, UDP concurrent server can also be designed.

6.1 Writing Concurrent Server

The simplest way to write concurrent server in Unix is to `fork()` a child process to handle each client. The `fork()` is a system call or function that creates a copy of the process that has issued the `fork()`. The function is called once, but returns twice since it results in two processes from one. If return value is 0, the process is child; if some positive integer is returned, the process is parent.

We next present the outline of the concurrent server. When a connection is established, `accept()` returns a connected socket (say, `connfd`), the server calls `fork()`, and then child process services the client using `connfd` and the parent process waits for another connection (on `listenfd`, the listening socket). The parent closes the connected socket (`connfd`) since the child handles the new client. Outline of the concurrent server is given below.

```
int         main(int argc, char ** argv)
{
        pid_t pid;
        int listenfd, connfd;

        listenfd = socket(...);
        .
        .
```

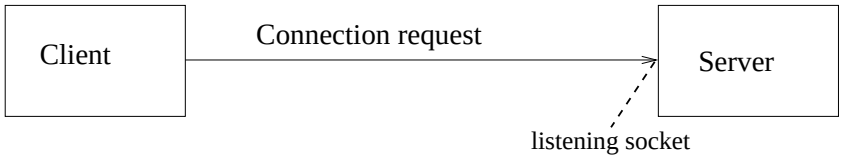

Figure 6.1: A client just requests the server for connection.

```
        .
        bind(listenfd,...);
        listen(listenfd,...);

        while(1)
        {
                connfd = accept(listenfd,...);

                pid = fork();

                if(pid == 0) /* child code */
                {
                        close(listenfd);

                        ***process the request***

                        close(connfd);
                        exit(0);
                }
                else        /* parent code */
                        close(connfd);
        }
}
```

The scenario of handling multiple clients is shown in Figure 6.1, Figure 6.2 and Figure 6.3. The server is always ready to accept new connection request. While a connection request is accepted, a child is forked to service the request. The parent server then again becomes ready to accept the new connection request by closing the connected socket, as the connected socket is now dealt by the child.

The listening socket must be distinguished from the connected socket on the server host. Although both sockets use the same local port on the server machine, they are indicated by distinct socket descriptors, returned server's call of functions `socket()` and `accept()` respectively for listening and connected sockets.

6.2 Concurrent server with thread

Traditionally, a concurrent TCP server calls fork to spawn a child to handle each client. This allows the server to handle numerous clients at the same time, one client per process. The only limit on the number of clients is the OS limit on the number of child processes for the user ID under which the server is running.

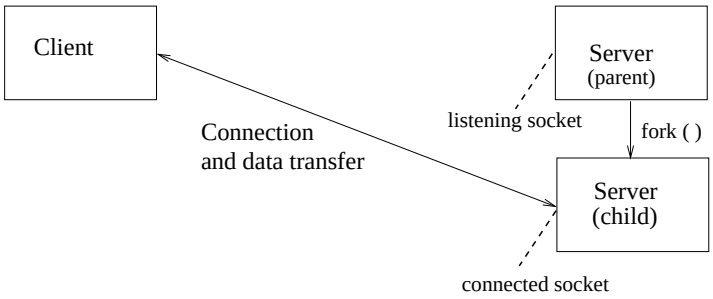

Figure 6.2: The client is connected with newly created server. The parent server is ready to accept new request.

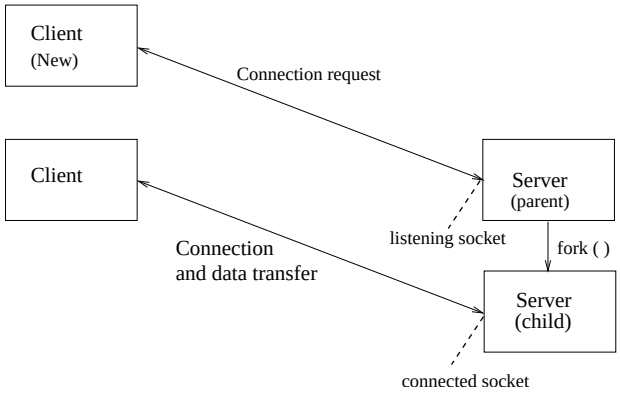

Figure 6.3: A new client requests the server (parent) for connection.

The problem with these concurrent servers is the amount of CPU time it takes to fork a child for each client. Years ago (the late 1980s), when a busy server handled hundreds or perhaps even a few thousand clients per day, this was acceptable. But the explosion of the Web has changed this attitude. Busy Web servers measure the number of TCP connections per day in the millions. This is for an individual host, and the busiest sites run multiple hosts, distributing the load among the hosts.

One way to speed up the server is to use thread, if the server machine supports, instead of using child process while creating concurrent server. The one-thread-per-client version is many times faster than the one-child-per-client version. Following two functions are needed in the server of thread version.

```
#include<pthread.h>

int    pthread_create(pthread_t  *thread, pthread_attr_t *attr,
                      void * (*start_routine)(void *), void *arg);
```

Returns: 0 on success, non-zero error code on error.

The `pthread_create` creates a new thread of control that executes concurrently with the calling thread. Each thread within a process is identified by a *thread ID*, whose datatype is `pthread_t` (often an `unsigned int`). On successful creation of a new thread, its ID is returned through the pointer *thread*. The new thread applies the function *start_routine* passing it *arg* as first argument. The new thread terminates either explicitly, by calling `pthread_exit()` (a function that is used to terminate the calling thread), or implicitly, by finishing the execution of *start_routine* function. The *attr* argument specifies thread attributes to be applied to the new thread. We set it as NULL, in which case default attributes are used.

```
#include <pthread.h>

int        pthread_detach(pthread_t th);
```

Returns: 0 on success, non-zero error code on error.

A thread can be *detached* from the main thread. A detached thread is like a daemon process: when it terminates all its resources are released an we can not wait for it to terminate. This function is commonly called by the thread that wants to detach itself, as in

```
pthread_detach(pthread_self());
```

Here, `pthread_self()` returns the ID of the thread itself.[1]

The basic functions (APIs) of designing TCP concurrent server using thread and using fork are the same. Only difference, from programming point of view, is, instead of creating one-child per client, a new thread is created to serve the child. The outline of such server is noted next.

```
int     main(int argc, char **argv)
{
```

[1] `pthread_join(pthread_t th, void **thread_return)` is another basic thread function that suspends the execution of the calling thread until the thread identified by *th* terminates, either by calling `pthread_exit` or by being cancelled. This function, if used, can forcefully make a server as a sequential or iterative server.

```
        pthread_t th;
        int     listenfd, connfd;

        listenfd = socket(...);
        .
        .
        .
        bind(listenfd,...);
        listen(listenfd,...);

        for( ; ; )
        {
                connfd = accept(listenfd,...);

                pthread_create(&tid, NULL, &func, (void *) &connfd);
                /* create thread instead of forking */
        }
}



void *func(void *connfd)
{
        pthread_detach(pthread_self()); /* equivalent to closing listening socket */

        ***process the request***

        close(*((int*)connfd));
}
```

The program can be compiled in the following way.

cc server.c -lthread -o server

OR

cc server.c -o server

6.3 Concurrent UDP server

Most UDP servers are iterative: the server waits for a client request, reads the request, processes the request, sends back the reply, and then waits for the next client request. But when the processing of the client takes a *long time*, some form of concurrency is desired.

With TCP it is simple to just create a new child/thread and let the child/thread handle the new client. What simplifies this server concurrency when TCP is being is that every client connection is unique: the TCP socket pair (listening socket, connected socket) is unique for every connection. But in UDP, simple fork is not sufficient; we have to deal with two types of servers.

First one is a simple UDP server that reads a client request, sends a reply, and is then finished with the client. In this scenario the server that reads the client request can fork a child and let it handle the request. The child then sends its reply directly to the client.

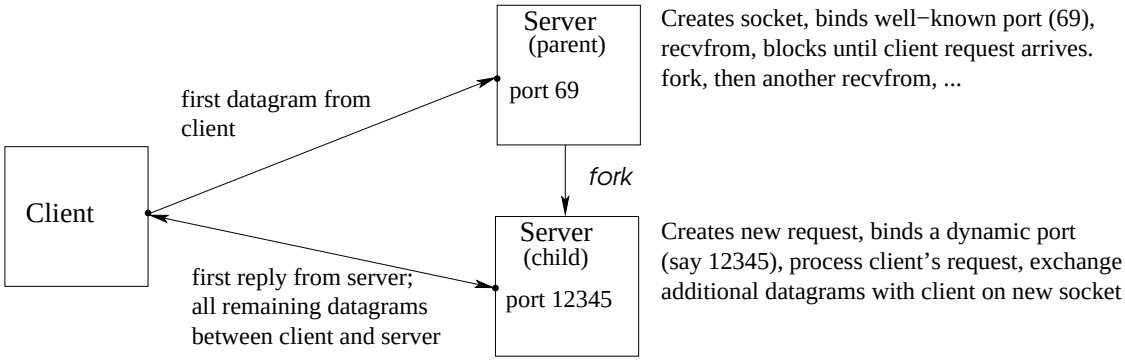

Figure 6.4: Processes involved in stand-alone concurrent UDP server

Second one is a UDP server that exchanges multiple datagrams with the client. While the server receives the first datagram (request), a child is forked to service the client. The problem is that the only port number is known to client, on which the server is running. The client sends the first datagram of its request to that port, but how does the server distinguish between subsequent datagrams from that client, and new request? The typical solution is for the server to create a new socket for each client, bind a dynamic port to that socket and use that socket for all its replies. This requires that the client looks at the port number of the server's first reply and send subsequent datagrams for this request to that port.

Second type of UDP server can handle concurrency in a better way. The scenario is shown in Figure 6.4. An example of such server is TFTP (Trivial File Transfer Protocol). To transfer a file using TFTP normally requires many datagrams, because the protocol sends only 512 bytes per datagram. The client sends a datagram to the server's well known port (69) specifying the file to send or receive. The server reads the request but sends its reply from another socket that it creates and binds to a dynamic port. All subsequent datagrams between the client and server for this file use the new socket. This allows the main TFTP server to continue to handle other client requests that arrive at port 69.

6.4 Exercise

1. Write a concurrent version of TCP echo server by forking a child per client, where the server echos the message which is sent by the client in reverse case (that is, if client says $Hello$, then server replies with $hELLO$).
2. Write the above program using thread.
3. Write the udp concurrent server - TFTP server and corresponding client.