TOPIC: Socket Programming using TCP Sockets

Department of Computer Science and Information Systems, BITS Pilani, Pilani

Objectives:

a) **To learn the concept of socket to create network applications**
b) **To learn Socket creation using LINUX system calls**
c) **Writing a simple TCP client and TCP server program to achieve communication between two processes running on same machine and on two different machines**

What is a Socket…?

A socket is an abstraction through which an application may send and receive data, in much the same way as an open file allows an application to read and write data to stable storage. A socket allows an application to "plug in" to the network and communicate with other applications that are also plugged in to the same network.

Sockets come in different flavors, corresponding to different underlying protocol families and different stacks of protocols within a family. In this course we deal only with the TCP/IP protocol family. The main flavors of sockets in the TCP/IP family are stream sockets and datagram sockets. Stream sockets use TCP as the end-to-end protocol (with IP underneath) and thus provide a reliable byte-stream service. Datagram sockets use UDP (again, with IP underneath) and thus provide a best-effort datagram service that applications can use.

A socket using the TCP/IP protocol family is uniquely identified by:

a) An Internet address,
b) An end-to-end protocol (TCP or UDP)
c) A port number.

Creating and Destroying Sockets

To communicate using TCP or UDP, a program begins by asking the operating system to create an instance of the socket abstraction. The function that accomplishes this is socket()

int socket(int protocolFamily, int type, int protocol)

The first parameter determines the protocol family of the socket. The constant **PF_INET** specifies a socket that uses protocols from the Internet protocol family. The second parameter specifies the type of the socket. The type determines the semantics of data transmission with the socket--for example, whether transmission is reliable, whether message boundaries are preserved, and so on. The constant **SOCK_STREAM** specifies a socket with reliable byte-stream semantics, whereas **SOCK_DGRAM** specifies a best-effort datagram socket. The third parameter specifies the particular end-to-end protocol to be used. For the **PF_INET** protocol family, we want TCP (identified by the constant **IPPROTO_TCP**) for a stream socket and UDP (identified by **IPPROTO_UDP**) for a datagram socket. Supplying the constant 0 as the third parameter requests the default end-to-end protocol for the specified protocol family and type.

The return value of **socket ()** is an integer: a nonnegative value for success and -1 for failure. A nonfailure value should be treated as an opaque handle, which we call a socket descriptor, and is passed to other API functions to identify the socket abstraction on which the operation is to be carried out.

When an application is finished with a socket, it calls **close()**, giving the descriptor for the socket that is no longer needed.

int close (int socket)

close() returns 0 on success or -1 on failure.

Specifying Addresses

Applications using sockets need to be able to specify Internet addresses and ports to the kernel. The sockets API defines a generic data type--the sockaddr structure--for specifying addresses associated with sockets:

```
struct sockaddr
{
  unsigned short sa_family; /* Address family (e. g. AF_INET) */
  char sa_data[14] ; /* Family-specific address information */
};
```

The first part of this address structure defines the address family--the space to which the address belongs. For our purposes, we will always use the constant **AF_INET**, which specifies the Internet address family. The second part is a blob of bits whose exact form depends on the address family.
The particular form of the **sockaddr** structure that is used for TCP/IP socket addresses is the **sockaddr_in** structure.

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

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

As you can see, the **sockaddr_in** structure has fields for the port number and Internet address in addition to the address family. It is important to understand that **sockaddr_in** is just another view of the data in a **sockaddr** structure, tailored to sockets using the Internet protocols. Thus, we can fill in the fields of a **sockaddr_in** and then cast it to a **sockaddr** to pass it to the socket functions, which look at the **sa_family** field to learn how the rest of the address is structured.

TCP CLIENT

Its job is to initiate communication with a server that is passively waiting to be contacted. The typical TCP client goes through four basic steps:

1. Create a TCP socket using socket().
2. Establish a connection to the server using connect ().
3. Communicate using send() and recv().
4. Close the connection with close().

A TCP socket must be connected to another socket before any data can be sent through it. The connection establishment process is the biggest difference between clients and servers: The client initiates the connection while the server waits passively for clients to connect to it. To establish a connection with the server, we call connect () on the socket.

int connect (int socket, struct sockaddr *foreignAddress, unsigned int addressLength)

socket is the descriptor created by socket (). **foreignAddress** is declared to be a pointer to a **sockaddr** because the sockets API is generic; for our purposes, it will always be a pointer to a **sockaddr_in** containing the Internet address and port of the server, **addressLength** specifies the length of the address structure and is invariably given as **sizeof(struct sockaddr_in).**

When **connect ()** returns successfully, the socket is connected and communication can proceed with calls to **send()** and **recv()**.

int send (int socket, const void *msg, unsigned int msgLength, int flags)

int recv (int socket, void *rcvBuffer, unsigned int bufferLength, int flags)

send () and **recv()** have very similar arguments, **socket** is the descriptor for the connected socket through which data is to be sent or received. For **send()**, **msg** points to the message to send, and **msgLength** is the length (in bytes) of the message. For **recv()**, **rcvBuffer** points to the buffer--that is, an area in memory such as a character array--where received data will be placed, and **bufferLength** gives the length of the buffer, which is the maximum number of bytes that can be received at once. The flags parameter in both **send()** and **recv()** provides a way to change the default behavior of the socket call. Setting flags to 0 specifies the default behavior. **send()** and **recv()** return the number of bytes sent or received or -1 for failure.

TCP Server

The server's job is to set up a communication endpoint and passively wait for a connection from the client. As with clients, the setup for a TCP and UDP server is similar. For now, let’s focus on a TCP server. There are four steps for TCP server communication:

1. Create a TCP socket using **socket()**.
2. Assign a port number to the socket with **bind()**.
3. Tell the system to allow connections to be made to that port, using **listen()** .
4. Repeatedly do the following:

- Call **accept ()** to get a new socket for each client connection.

- Communicate with the client via that new socket using **send()** and **recv()**.
- Close the client connection using **close()**.

Creating the socket, sending, receiving, and closing are the same as in the client. The differences in the server have to do with binding an address to the socket and then using the socket as a channel to "receive" other sockets that are connected to clients. For the client to contact the server, the server's socket must have an assigned local address and port; the function that accomplishes this is **bind()**. Notice that while the client has to supply the server's address to **connect()**, the server has to specify its own address to **bind()**. It is this piece of information (i.e., the server's address and port) that they have to agree on to communicate; neither one really needs to know the client's address.

int bind (int socket, struct sockaddr *localAddress, unsigned int addressLength)

The first parameter is the descriptor returned by an earlier call to **socket()**. As with **connect()**, the address parameter is declared as a pointer to a **sockaddr**, but for TCP/IP applications, it will actually point to a **sockaddr_in** containing the Internet address of the local interface and the port to listen on. **addressLength** is the length of the address structure, invariably passed as **sizeof(struct sockaddr_in). bind()** returns 0 on success and - 1 on failure. If successful, the socket identified by the given descriptor (and no other) is associated with the given Internet address and port. The Internet address can be set to the special wildcard value **INADDR_ANY**, which means that connections to the specified port will be directed to this socket, regardless of which Internet address they are sent to; this practice can be useful if the host happens to have multiple Internet addresses.

Now that the socket has an address (or at least a port), we need to instruct the underlying TCP protocol implementation to listen for connections from clients by calling **listen()** on the socket.

int listen(int socket, int queueLimit)

listen() causes internal state changes to the given socket, so that incoming TCP connection requests will be handled and then queued for acceptance by the program. The **queueLimit** parameter specifies an upper bound on the number of incoming connections that can be waiting at any time. **listen()** returns 0 on success and - 1 on failure.

At first it might seem that a server should now wait for a connection on the socket that it has set up, send and receive through that socket, close it, and then repeat the process. However, that is not the way it works. The socket that has been bound to a port and marked "listening" is never actually used for sending and receiving. Instead, it is used as a way of getting new sockets, one for each client connection; the server then sends and receives on the new sockets. The server gets a socket for an incoming client connection by calling accept ().

int accept(int socket, struct sockaddr *clientAddress, unsigned int *addressLength)

accept() de-queues the next connection on the queue for socket. If the queue is empty, **accept()** blocks until a connection request arrives. When successful, **accept()** fills in the **sockaddr** structure, pointed to by **clientAddress**, with the address of the client at the other end of the connection, **addressLength** specifies the maximum size of the **clientAddress** address structure and contains the number of bytes actually used for the address upon return. If successful, **accept()** returns a descriptor for a new socket that is connected to the client. The socket sent as the first parameter to **accept()** is unchanged (not connected to the client) and continues to listen for new connection requests. On failure, **accept()** returns -1.

The server communicates with the client using **send()** and **recv()**; when communication is complete, the connection is terminated with a call to **close()**.

Now, let's implement a simple client and server program which can send one message to each other.

Read, understand, and save the following file as client.c.

```
#include <stdio.h>
#include <sys/socket.h>   //for socket(), connect(), send(), recv()
functions
#include <arpa/inet.h>   // different address structures are declared
here
#include <stdlib.h>   // atoi() which convert  string to integer
#include <string.h>
#include <unistd.h>  // close() function
#define BUFSIZE 32
int main()
{
/* CREATE A TCP SOCKET*/
 int sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (sock < 0) { printf ("Error in opening a socket"); exit (0);}
 printf ("Client Socket Created\n");

/*CONSTRUCT SERVER ADDRESS STRUCTURE*/
 struct sockaddr_in serverAddr;
 memset (&serverAddr,0,sizeof(serverAddr));

/*memset() is used to fill a block of memory with a particular value*/
serverAddr.sin_family = AF_INET;
 serverAddr.sin_port = htons(12345); //You can change port number here
 serverAddr.sin_addr.s_addr = inet_addr("A.B.C.D"); //Specify server's
IP address here
 printf ("Address assigned\n");

/*ESTABLISH CONNECTION*/
int c = connect (sock, (struct sockaddr*) &serverAddr , sizeof
(serverAddr));
 printf ("%d\n",c);
 if (c < 0)
   { printf ("Error while establishing connection");
     exit (0);
   }
 printf ("Connection Established\n");
```

```
/*SEND DATA*/
 printf ("ENTER MESSAGE FOR SERVER with max 32 characters\n");
 char msg[BUFSIZE];
 gets(msg);
 int bytesSent = send (sock, msg, strlen(msg), 0);
 if (bytesSent != strlen(msg))
   { printf("Error while sending the message");
     exit(0);
   }
 printf ("Data Sent\n");

/*RECEIVE BYTES*/
char recvBuffer[BUFSIZE];
 int bytesRecvd = recv (sock, recvBuffer, BUFSIZE-1, 0);
 if (bytesRecvd < 0)
   { printf ("Error while receiving data from server");
     exit (0);
   }
 recvBuffer[bytesRecvd] = '\0';
 printf ("%s\n",recvBuffer);

close(sock);
}
```

Read, understand, and save the following file as server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define MAXPENDING 5
#define BUFFERSIZE 32
int main ()
{
/*CREATE A TCP SOCKET*/
 int serverSocket = socket (PF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (serverSocket < 0) { printf ("Error while server socket
creation"); exit (0); }
 printf ("Server Socket Created\n");

/*CONSTRUCT LOCAL ADDRESS STRUCTURE*/
struct sockaddr_in serverAddress, clientAddress;
 memset (&serverAddress, 0, sizeof(serverAddress));
 serverAddress.sin_family = AF_INET;
 serverAddress.sin_port = htons(12345);
 serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
 printf ("Server address assigned\n");
```

```
 int temp = bind(serverSocket, (struct sockaddr*) &serverAddress,
sizeof(serverAddress));
 if (temp < 0)
  { printf ("Error while binding\n");
    exit (0);
  }
 printf ("Binding successful\n");
 int temp1 = listen(serverSocket, MAXPENDING);
 if (temp1 < 0)
  {  printf ("Error in listen");
     exit (0);
  }
 printf ("Now Listening\n");
 char msg[BUFFERSIZE];

  int clientLength = sizeof(clientAddress);
  int clientSocket = accept (serverSocket, (struct sockaddr*)
&clientAddress, &clientLength);
  if (clientLength < 0) {printf ("Error in client socket"); exit(0);}
  printf ("Handling Client %s\n", inet_ntoa(clientAddress.sin_addr));
  int temp2 = recv(clientSocket, msg, BUFFERSIZE, 0);
  if (temp2 < 0)
    { printf ("problem in temp 2");
      exit (0);
    }
  printf ("%s\n", msg);
  printf ("ENTER MESSAGE FOR CLIENT\n");
  gets (msg);
  int bytesSent = send (clientSocket,msg,strlen(msg),0);
  if (bytesSent != strlen(msg))
    { printf ("Error while sending message to client");
      exit(0);
    }
 close (serverSocket);
 close (clientSocket);
}
```

Once you have understood the above programs, execute them on the same machine on two different terminals. Probably, both the client and server should be able to communicate with each other.

Objectives:

a) **Client-server program writing using UDP sockets**
b) **File Transfer Program (with partial file sending option) using TCP sockets**
c) **Creating concurrent server (handling multiple clients) using fork() system call**

---

a) A simple ECHO server using UDP sockets

In the last lab, we created a client/server based simple application using the TCP socket (or byte stream socket). Today, as the first exercise, we will see how to create a simple ECHO server and the client using UDP (message stream) sockets. By ECHO server, we mean that the server will reply the same message back, whatever it receives from the client.

Read, understand, and save the following file as **client_udp.c.** You can also download this file from Nalanda. While reading, try to locate and understand the specific differences with the TCP based client program that we did in the previous lab.

```
/* Simple udp client */

#include<stdio.h> //printf
#include<string.h> //memset
#include<stdlib.h> //exit(0);
#include<arpa/inet.h>
#include<sys/socket.h>
#define BUFLEN 512  //Max length of buffer

#define PORT 8888   //The port on which to send data

void die(char *s)
{
    perror(s);
    exit(1);
}

int main(void)
{
    struct sockaddr_in si_other;
    int s, i, slen=sizeof(si_other);
    char buf[BUFLEN];
    char message[BUFLEN];

    if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
    {
        die("socket");
    }

    memset((char *) &si_other, 0, sizeof(si_other));
    si_other.sin_family = AF_INET;
    si_other.sin_port = htons(PORT);
    si_other.sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
    while(1)
    {
        printf("Enter message : ");
        gets(message);

        //send the message
        if (sendto(s, message, strlen(message) , 0 , (struct sockaddr *)
&si_other, slen)==-1)
        {
            die("sendto()");
        }
        //receive a reply and print it
        //clear the buffer by filling null, it might have previously received
data
        memset(buf,'\0', BUFLEN);
        //try to receive some data, this is a blocking call
        if (recvfrom(s, buf, BUFLEN, 0, (struct sockaddr *) &si_other, &slen)
== -1)
        {
            die("recvfrom()");
        }

        puts(buf);
    }

    close(s);
    return 0;
}
```

Read, understand, and save the following file as **server_udp.c.** You can also download this file from Nalanda. While reading, try to locate and understand the specific differences with the TCP based server program that we did in the previous lab.

```
/*  Simple udp server */

#include<stdio.h> //printf
#include<string.h> //memset
#include<stdlib.h> //exit(0);
#include<arpa/inet.h>
#include<sys/socket.h>

#define BUFLEN 512  //Max length of buffer
#define PORT 8888   //The port on which to listen for incoming data

void die(char *s)
{
    perror(s);
    exit(1);
}

int main(void)
{
    struct sockaddr_in si_me, si_other;
    int s, i, slen = sizeof(si_other) , recv_len;
    char buf[BUFLEN];
```

```
    //create a UDP socket
    if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
    {
        die("socket");
    }

    // zero out the structure
    memset((char *) &si_me, 0, sizeof(si_me));

    si_me.sin_family = AF_INET;
    si_me.sin_port = htons(PORT);
    si_me.sin_addr.s_addr = htonl(INADDR_ANY);

    //bind socket to port
    if( bind(s , (struct sockaddr*)&si_me, sizeof(si_me) ) == -1)
    {
        die("bind");
    }

    //keep listening for data
    while(1)
    {
        printf("Waiting for data...");
        fflush(stdout);

        //try to receive some data, this is a blocking call
        if ((recv_len = recvfrom(s, buf, BUFLEN, 0, (struct sockaddr *)
&si_other, &slen)) == -1)
        {
            die("recvfrom()");
        }

        //print details of the client/peer and the data received
        printf("Received packet from %s:%d\n", inet_ntoa(si_other.sin_addr),
ntohs(si_other.sin_port));
        printf("Data: %s\n" , buf);

        //now reply the client with the same data
        if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &si_other, slen)
== -1)
        {
            die("sendto()");
        }
    }
    close(s);
    return 0;
}
```

The following diagram showing the sequence of function calls for the client and a server participating in a TCP and UDP would help you understand the differences between TCP and UDP socket programming better.

Exercise #1

Modify the ECHO server and client programs to a guessing game, where the server will generate a number (say between 1 to 6 or the name of a famous personality with some hint) and ask the client to guess it. The user will enter the guessed number (or name) through the terminal. If

the guess is correct the client will win, otherwise it will lose. An appropriate message about the outcome can be printed at the client side.

b) Designing and implementing simple FTP client and server with broken download handling capability using TCP sockets.

In the second lab, we performed few experiments with Wireshark to understand functioning of standard FTP protocol. Here, we will develop our own simple client/server based application to get a file from the server. Currently, the program given here, simply downloads a predefined file from the server. It can be extended to include many other functionalities such as directory listing, “get” and “put” commands, as present in the standard FTP program. However, our program has the broken download capability which allows the client to complete a file transfer by downloading the remaining portion of a file only, if the file to be download is already present with the client partially. For example, if the client is already having initial 100 bytes of a files then next time, instead of downloading the complete file from the server, the client can request the server to send the file starting from byte number 101.

Read, understand, and save the following file as **client_broken_ftp.c.** You can also download this file from Nalanda.

```
/* Client program Broken FTP */

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

int main(void)
{
    int sockfd = 0;
    int bytesReceived = 0;
    char recvBuff[256];
    unsigned char buff_offset[10];    // buffer to send the File offset
value
    unsigned char buff_command[2];    // buffer to send the Complete
File (0) or Partial File Command (1).
    int offset;                       // required to get the user input
for offset in case of partial file command
    int command;               // required to get the user input for
command
    memset(recvBuff, '0', sizeof(recvBuff));
    struct sockaddr_in serv_addr;
```

```
    /* Create a socket first */
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0))< 0)
    {
        printf("\n Error : Could not create socket \n");
        return 1;
    }

    /* Initialize sockaddr_in data structure */
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(5001); // port
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    /* Attempt a connection */
    if(connect(sockfd, (struct sockaddr *)&serv_addr,
sizeof(serv_addr))<0)
    {
        printf("\n Error : Connect Failed \n");
        return 1;
    }

   /* Create file where data will be stored */
        FILE *fp;
        fp = fopen("destination_file.txt", "ab");
        if(NULL == fp)
        {
          printf("Error opening file");
            return 1;
        }
        fseek(fp, 0, SEEK_END);
        offset = ftell(fp);
        fclose(fp);
        fp = fopen("destination_file.txt", "ab");
        if(NULL == fp)
        {
          printf("Error opening file");
            return 1;
        }

    printf("Enter (0) to get complete file, (1) to specify offset, (2)
calculate the offset value from local file\n");
    scanf("%d", &command);
    sprintf(buff_command, "%d", command);
    write(sockfd, buff_command, 2);



    if(command == 1 || command == 2)   // We need to specify the
offset
    {

        if(command == 1)  // get the offset from the user
        {
        printf("Enter the value of File offset\n");
        scanf("%d", &offset);
```

```
        }
        // otherwise offset = size of local partial file, that we have
already calculated
        sprintf(buff_offset, "%d", offset);
        /* sending the value of file offset */
        write(sockfd, buff_offset, 10);
    }

    // Else { command = 0 then no need to send the value of offset }


    /* Receive data in chunks of 256 bytes */
    while((bytesReceived = read(sockfd, recvBuff, 256)) > 0)
    {
        printf("Bytes received %d\n",bytesReceived);
        // recvBuff[n] = 0;
        fwrite(recvBuff, 1,bytesReceived,fp);
        // printf("%s \n", recvBuff);
    }

    if(bytesReceived < 0)
    {
        printf("\n Read Error \n");
    }

    return 0;
}
```

Read, understand, and save the following file as **server_broken_ftp.c.** You can also download this file from Nalanda.

/* Server program for broken ftp */

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

int main(void)
{
    int listenfd = 0;
    int connfd = 0;
    struct sockaddr_in serv_addr;
    char sendBuff[1025];
    int numrv;
```

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

    printf("Socket retrieve success\n");

    memset(&serv_addr, '0', sizeof(serv_addr));
    memset(sendBuff, '0', sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5001);

    bind(listenfd, (struct sockaddr*)&serv_addr,sizeof(serv_addr));

    if(listen(listenfd, 10) == -1)
    {
        printf("Failed to listen\n");
        return -1;
    }


    while(1)
    {
        unsigned char offset_buffer[10] = {'\0'};
        unsigned char command_buffer[2] = {'\0'};
        int offset;
        int command;
        connfd = accept(listenfd, (struct sockaddr*)NULL ,NULL);


        printf("Waiting for client to send the command (Full File (0)
Partial File (1)\n");

        while(read(connfd, command_buffer, 2) == 0);
                sscanf(command_buffer, "%d", &command);

        if(command == 0)
                offset = 0;
        else
        {
                printf("Waiting for client to send the offset\n");
                while(read(connfd, offset_buffer, 10) == 0);
                sscanf(offset_buffer, "%d", &offset);

        }


        /* Open the file that we wish to transfer */
        FILE *fp = fopen("source_file.txt","rb");
        if(fp==NULL)
        {
            printf("File opern error");
            return 1;
        }
```

```
        /* Read data from file and send it */
                 fseek(fp, offset, SEEK_SET);
        while(1)
        {
            /* First read file in chunks of 256 bytes */
            unsigned char buff[256]={0};
            int nread = fread(buff,1,256,fp);
            printf("Bytes read %d \n", nread);

            /* If read was success, send data. */
            if(nread > 0)
            {
                printf("Sending \n");
                write(connfd, buff, nread);
            }

            /*
             * There is something tricky going on with read ..
             * Either there was error, or we reached end of file.
             */
            if (nread < 256)
            {
                if (feof(fp))
                    printf("End of file\n");
                if (ferror(fp))
                    printf("Error reading\n");
                break;
            }
        }
        close(connfd);
        sleep(1);
    }
    return 0;
}
```

Exercise #2

So, now as you understand the programming with UDP sockets and the working of our simple broken FTP application running over TCP socket, modify the above broken FTP client/server programs to make it run using UDP sockets.

c) Handling multiple clients at the same time

There are two main classes of servers, iterative and concurrent. An iterative server iterates through each client, handling it one at a time. A concurrent server handles multiple clients at the same time. The simplest technique for a concurrent server is to call the **fork** function, creating one child process for each client. An alternative technique is to use **threads** instead (i.e., light-weight processes). In this lab, we consider fork based technique only.

A typical concurrent server has the following structure. The code in **blue color** is something which is different for iterative server and concurrent server.

```
pid_t pid;
int listenfd, connfd;
listenfd = socket(...);

/***fill the socket address with server's well known port***/

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

for ( ; ; ) {

  connfd = accept(listenfd, ...); /* blocking call */

  if ( (pid = fork()) == 0 ) {

    close(listenfd); /* child closes listening socket */

    /***process the request doing something using connfd ***/
    /* ................ */

    close(connfd);
    exit(0);  /* child terminates
   }
   close(connfd);  /*parent closes connected socket*/
}
}
```

When a connection is established, accept returns, the **server calls fork**, and the child process services the client (on the connected socket connfd). The parent process waits for another connection (on the listening socket listenfd. The parent closes the connected socket since the child handles the new client. The interactions among client and server are presented.

****