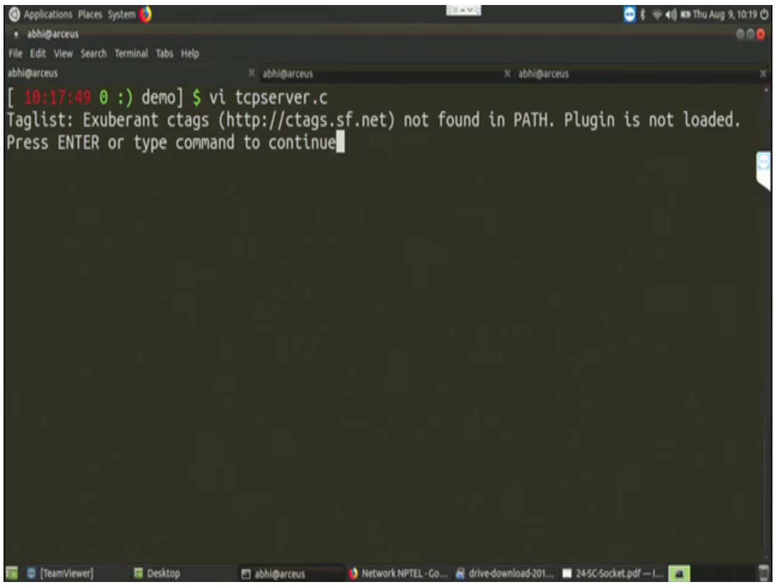
**Computer Networks and Internet Protocol**
**Prof. SandIP Chakraborthy**
**Department of Computer Science and Engineering**
**Indian Institute of Technology, Kharagpur**

# Lecture – 25
# Socket Programming – II

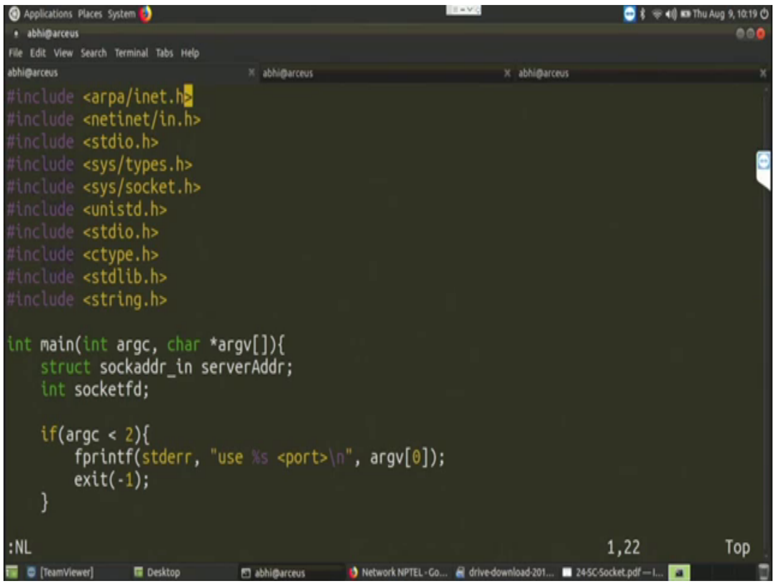
So, welcome back to the course on Computer Network and Internet Protocols and in the last class, we are discussing about socket programming in today's lecture, we will again continue with the socket programming in details. So, in the last class we have looked into the UDP socket, we have looked into the basic of socket programming along with the UDP server and the UDP client in details that how you can use this concept of soc datagram under datagram socket to transfer data receive data using UDP protocol. So, today we look into how you can do the same thing using tcp protocol and we look into different variants of tcp server.

(Refer Slide Time: 00:57)



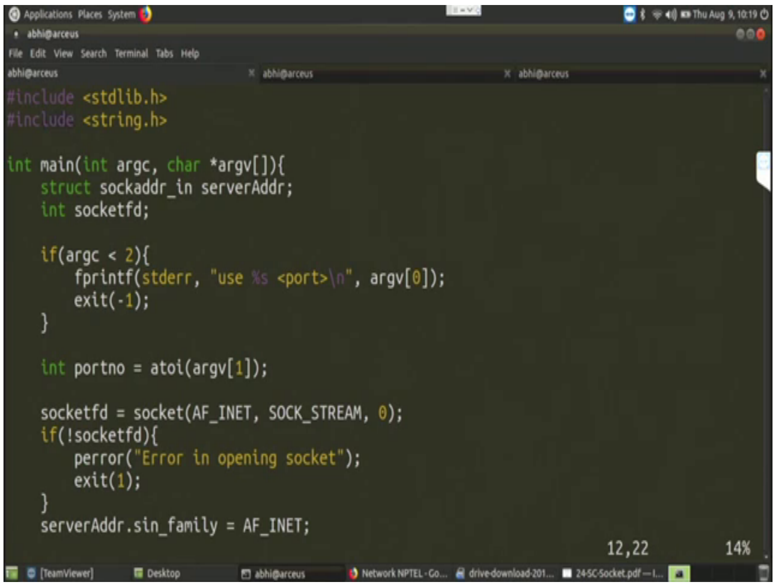
```
[ 10:17:49 0 :) demo] $ vi tcpserver.c
Taglist: Exuberant ctags (http://ctags.sf.net) not found in PATH. Plugin is not loaded.
Press ENTER or type command to continue
```

(Refer Slide Time: 01:11)


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

int main(int argc, char *argv[]){
    struct sockaddr_in serverAddr;
    int socketfd;

    if(argc < 2){
        fprintf(stderr, "use %s <port>\n", argv[0]);
        exit(-1);
    }

:NL                                         1,22          Top
```

So, first let us look into the demo of demo about the tcp server and a tcp client. So, first we look into this tcp server the source of the tcp server in details well, do not get confused to it end arrow. So, these are c program do not get with this end arrow and this dot just a formatting we have used for our clarity of writing the code. So, this is your c code well.

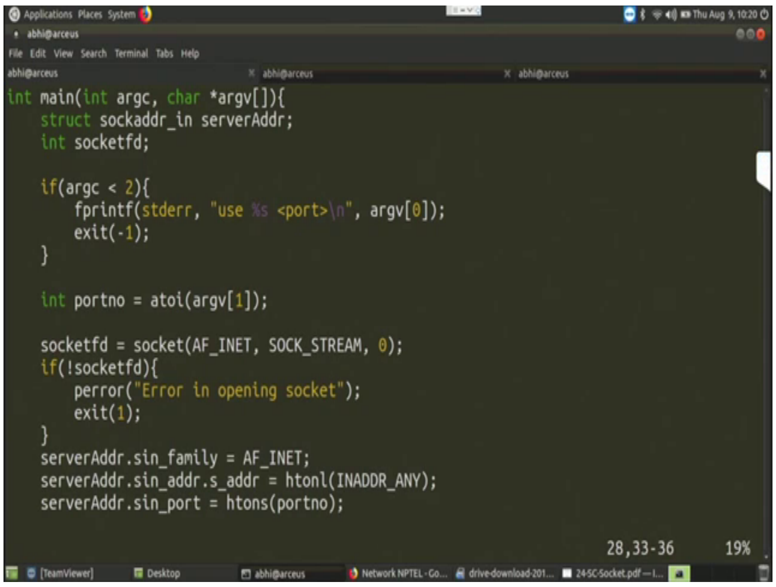
(Refer Slide Time: 01:35)


```
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]){
    struct sockaddr_in serverAddr;
    int socketfd;

    if(argc < 2){
        fprintf(stderr, "use %s <port>\n", argv[0]);
        exit(-1);
    }

    int portno = atoi(argv[1]);

    socketfd = socket(AF_INET, SOCK_STREAM, 0);
    if(!socketfd){
        perror("Error in opening socket");
        exit(1);
    }
    serverAddr.sin_family = AF_INET;
                                            12,22         14%
```

So, we have this network related header, which are being included followed by the main function so, in the main function similar to the UDP 1 UDP server that we have discussed, we are declaring a the server address first server address variable of type struck sockaddr in and file descriptor to store the socket id that will be created and this is the syntax that how you should use the code, we are taking the code number from the command line argument and then we are taking the port number from common line argument and we are creating the socket by this socket system call.

So, this socket system call similar to the earlier case it will take a finite as, the protocol family, but the socket type is now SOCK STREAM in state of sock degram. So, and finally, it is a protocol filed is 0 and as I have asked you earlier that try to explore that why we mostly use protocol family protocol type as 0.

(Refer Slide Time: 02:41)

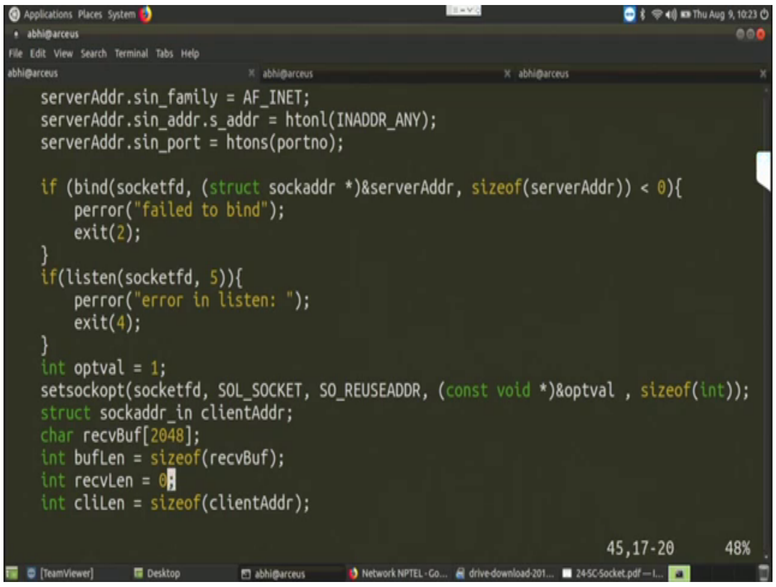


```
int main(int argc, char *argv[]){
    struct sockaddr_in serverAddr;
    int socketfd;

    if(argc < 2){
        fprintf(stderr, "use %s <port>\n", argv[0]);
        exit(-1);
    }

    int portno = atoi(argv[1]);

    socketfd = socket(AF_INET, SOCK_STREAM, 0);
    if(!socketfd){
        perror("Error in opening socket");
        exit(1);
    }
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddr.sin_port = htons(portno);

                                                  28,33-36      19%
```

Now if the socket is not created successfully some error message that is getting printed and after that we are initializing the server address field the ss in server address family protocol family we are setting it as AF INET, because we are going to use the IPv four address followed by the server address or server address we are taking it as INADDR_ANY. So, this INADDR ANY will help us to take the address of the local host then, we are providing the port number which are have taken from the concept.

So, we have initialized the sever address. So, once you have initialized the server address, you have to bind it with the socket. So, we are making a bind system call. So, the bind system call may fail sometime because, if you are going to use the same port number which is being used by another application, there are other reasons where a bind call may fail. So if the bind fails, we are having a error message that it is fail to bind that otherwise, the bind was successful. And the bind is taking the format as we have discussed earlier the socket file descriptor the address of the server and the size of the server address variable.

(Refer Slide Time: 03:49)



```
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddr.sin_port = htons(portno);

    if (bind(socketfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0){
        perror("failed to bind");
        exit(2);
    }
    if(listen(socketfd, 5)){
        perror("error in listen: ");
        exit(4);
    }
    int optval = 1;
    setsockopt(socketfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval , sizeof(int));
    struct sockaddr_in clientAddr;
    char recvBuf[2048];
    int bufLen = sizeof(recvBuf);
    int recvLen = 0;
    int cliLen = sizeof(clientAddr);

                                                              45,17-20      48%
```

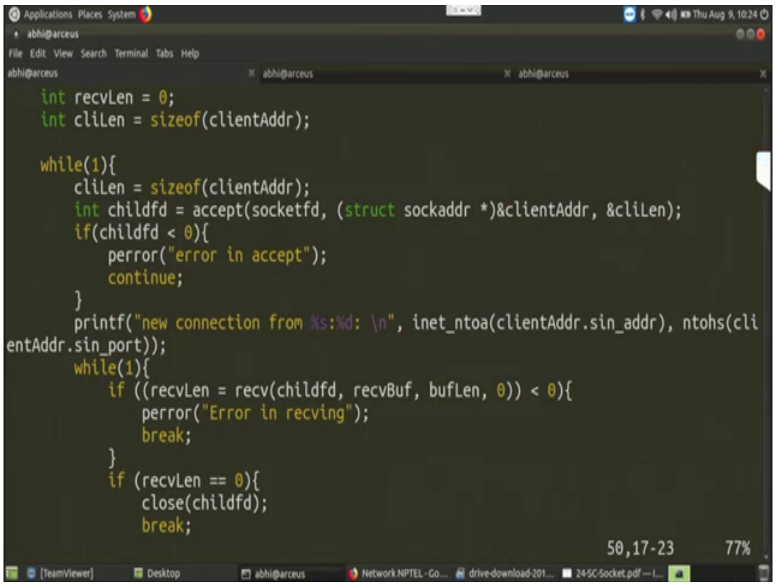
Now we are making a listen call, the listen call here we are providing the socket and socket file descriptor and we have specified 5 as the maximum number of connection that can be backlogged in the listened call. So this concept of backlog connection is that, the server can handle one client at a time. So, whenever it is taking the connection one client, during that time multiple client can again try simultaneously, just think of the scenario of a wave server, where thousands of request are coming to the web server per second.

So, during that time when the server is busy in executing this say the listen call for one connection during that time the other connection we get backlog. So, this parameter 5 which specifies how much such connection can get backlogged.

So, after that we similar to the earlier we are making a call to this setsockopt function to provide this facility of REUSEADDR. So, that we can use the port number for multiple codes together and after that, we are declaring a received buffer to store the incoming data in a stream format the buffer length and received length that would indicate how much data we have received.

And we are declaring this client addr variable that ah stuck sockaddr type of variable to store the client address and this client length this variable store the address of the client.

(Refer Slide Time: 05:23)



```
    int recvLen = 0;
    int cliLen = sizeof(clientAddr);

    while(1){
        cliLen = sizeof(clientAddr);
        int childfd = accept(socketfd, (struct sockaddr *)&clientAddr, &cliLen);
        if(childfd < 0){
            perror("error in accept");
            continue;
        }
        printf("new connection from %s:%d: \n", inet_ntoa(clientAddr.sin_addr), ntohs(cli
entAddr.sin_port));
        while(1){
            if ((recvLen = recv(childfd, recvBuf, bufLen, 0)) < 0){
                perror("Error in recving");
                break;
            }
            if (recvLen == 0){
                close(childfd);
                break;
                                                        50,17-23      77%
```

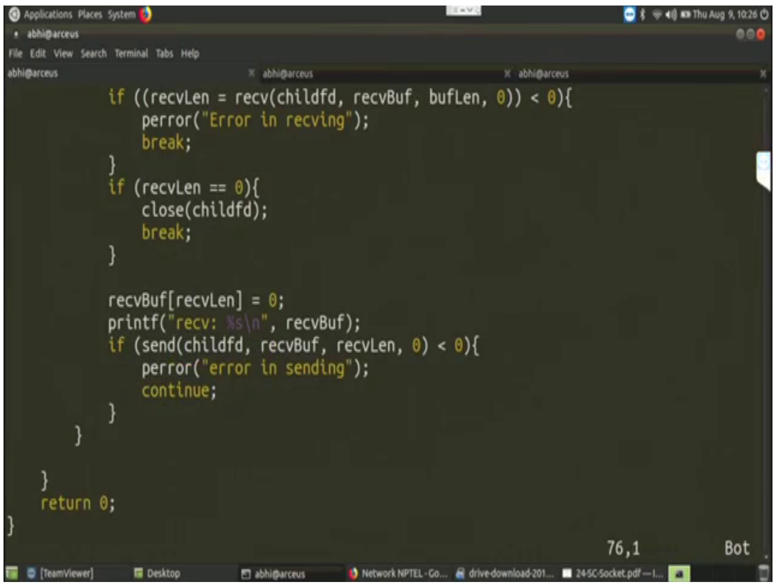
Now, in site alu what we are doing, we are making the accept called at the server site now tcp is a connection oriented service that is, why you have to make the connect call from the client side and accept call from the server site to initiate the connection, which will utilize the three way handshaking of the tcp protocol. So, after we are making the accept call the accept call, we are passing the parameter as the socket file descriptor, the address of the client and a length of the client address if there is an error. So, all this function if there is an error, they return some negative number. So, that is why you can check the return value if it is less than 0; that means certain error has occurred.

Ok. So, after that once it is accepted the connection, we are printing that we have received a new connection from the client address, we are specifying the client address and the client port. So, from that particular client address and the client port a new connection has been received and after that we are having a loop, where we are receiving the data from the new socket that has been created.

Now, here also while you are making the accept call in the accept call, it returns new file descriptor and this new file descriptor will initiate this end to end connection, between server and the client. So, it returns a new socket descriptor that we will used in the in sending or the receiving data, which is specific to a pipe or specific to a socket between a sever and the corresponding client.

So, we make this receive call to receive the data, we will provide this new socket file descriptor that, you have received after making the accept call and then the received buffer to store the data the buffer Len and this entire call will return the amount of bytes that have been received. So, that will returned by the received call and it will store inside the receiveLen function.

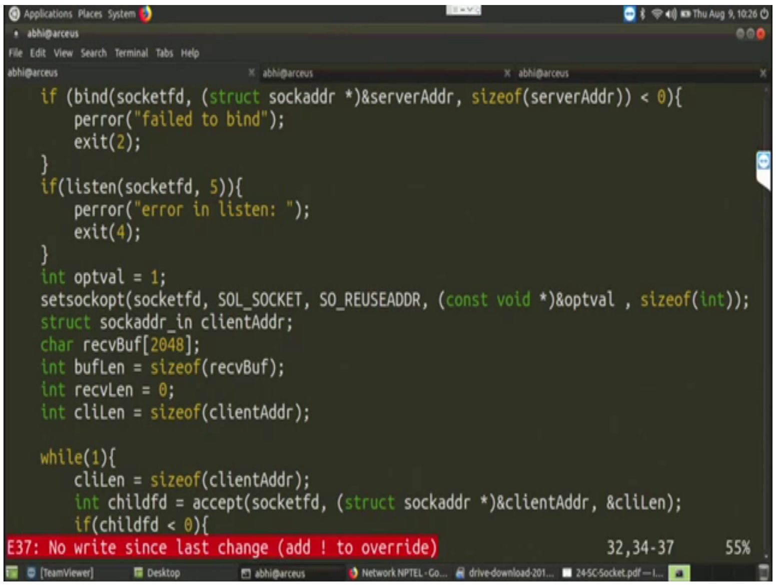
(Refer Slide Time: 07:42)



```
        if ((recvLen = recv(childfd, recvBuf, bufLen, 0)) < 0){
            perror("Error in recving");
            break;
        }
        if (recvLen == 0){
            close(childfd);
            break;
        }

        recvBuf[recvLen] = 0;
        printf("recv: %s\n", recvBuf);
        if (send(childfd, recvBuf, recvLen, 0) < 0){
            perror("error in sending");
            continue;
        }
    }

  }
  return 0;
}
                                                    76,1          Bot
```

Now, if receiveLen is equal to equal to 0; that means, you are not receiving any data from the client. So, you can close this particular connection. Otherwise you get the data and put in that data that you are receiving at the server side and after that we are actually whatever data we have received, we are requiring same data to the client by using this same function ok. So, you have received the data, which is stored in the received buffer same data is decode back to the to the client side.

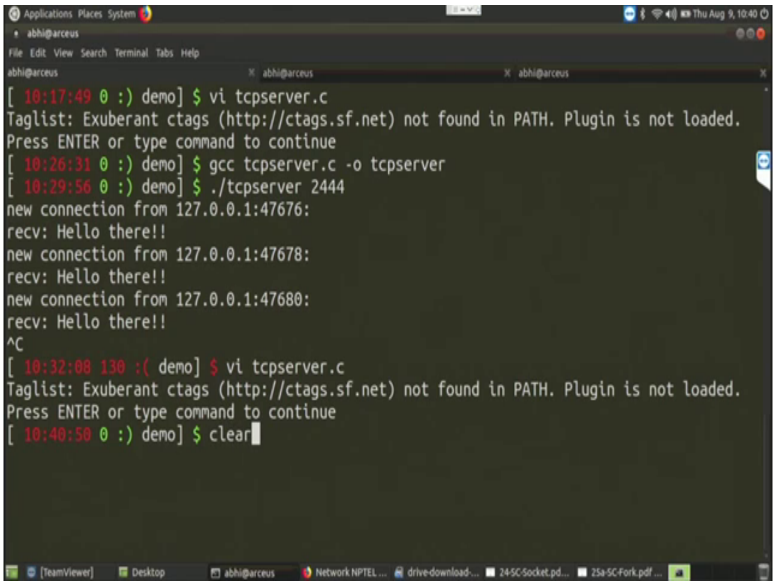
(Refer Slide Time: 08:15)



```
    if (bind(socketfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0){
        perror("failed to bind");
        exit(2);
    }
    if(listen(socketfd, 5)){
        perror("error in listen: ");
        exit(4);
    }
    int optval = 1;
    setsockopt(socketfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval , sizeof(int));
    struct sockaddr_in clientAddr;
    char recvBuf[2048];
    int bufLen = sizeof(recvBuf);
    int recvLen = 0;
    int cliLen = sizeof(clientAddr);

    while(1){
        cliLen = sizeof(clientAddr);
        int childfd = accept(socketfd, (struct sockaddr *)&clientAddr, &cliLen);
        if(childfd < 0){
E37: No write since last change (add ! to override)                32,34-37      55%
```

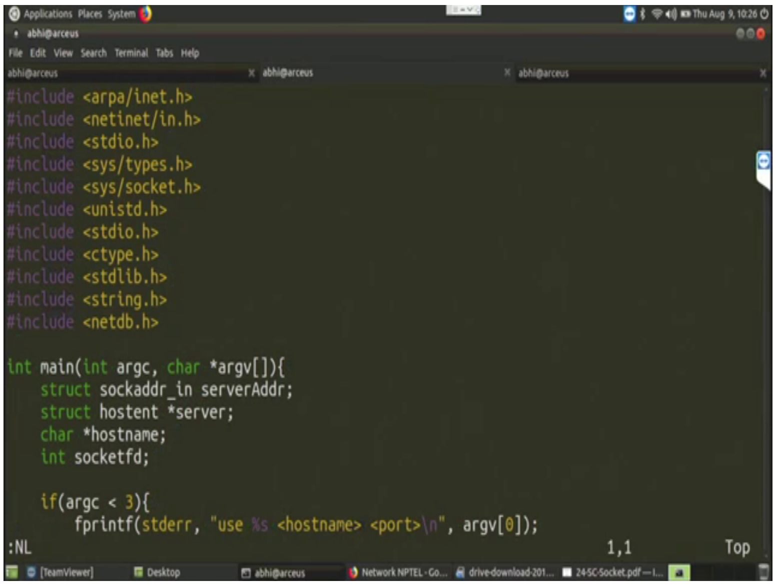
So, this is my code for the sever side now, let us look into the quote at the client side.

(Refer Slide Time: 08:22)



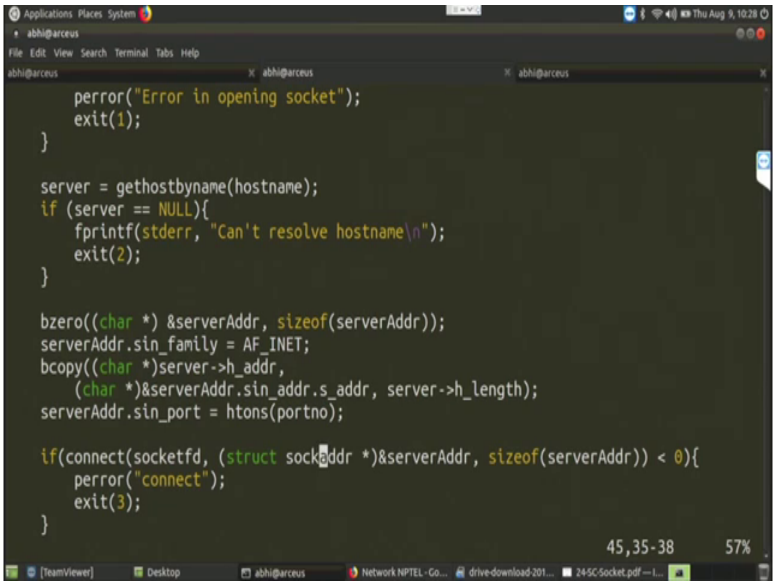
```
[ 10:17:49 0 :) demo] $ vi tcpserver.c
Taglist: Exuberant ctags (http://ctags.sf.net) not found in PATH. Plugin is not loaded.
Press ENTER or type command to continue
[ 10:26:31 0 :) demo] $ gcc tcpserver.c -o tcpserver
[ 10:29:56 0 :) demo] $ ./tcpserver 2444
new connection from 127.0.0.1:47676:
recv: Hello there!!
new connection from 127.0.0.1:47678:
recv: Hello there!!
new connection from 127.0.0.1:47680:
recv: Hello there!!
^C
[ 10:32:08 130 :( demo] $ vi tcpserver.c
Taglist: Exuberant ctags (http://ctags.sf.net) not found in PATH. Plugin is not loaded.
Press ENTER or type command to continue
[ 10:40:50 0 :) demo] $ clear
```

(Refer Slide Time: 08:32)



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

int main(int argc, char *argv[]){
    struct sockaddr_in serverAddr;
    struct hostent *server;
    char *hostname;
    int socketfd;

    if(argc < 3){
        fprintf(stderr, "use %s <hostname> <port>\n", argv[0]);
:NL                                                         1,1           Top
```

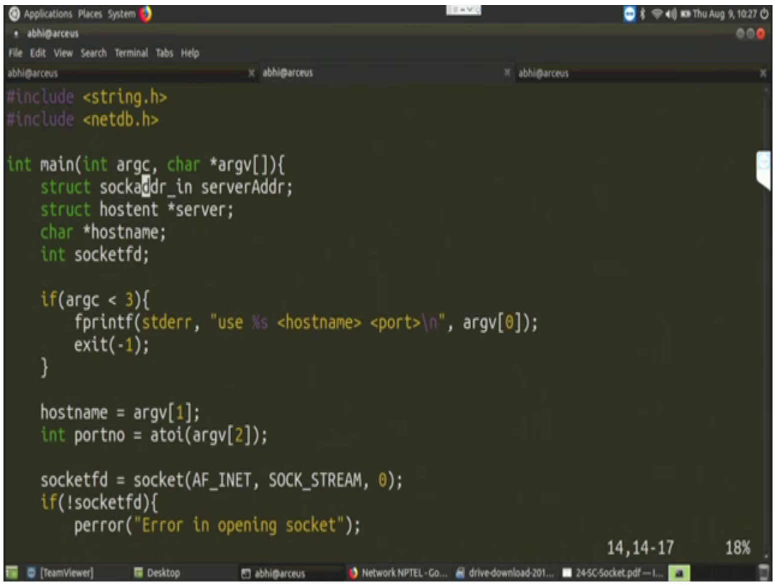
(Refer Slide Time: 08:35)



```
        perror("Error in opening socket");
        exit(1);
    }

    server = gethostbyname(hostname);
    if (server == NULL){
        fprintf(stderr, "Can't resolve hostname\n");
        exit(2);
    }

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

    if(connect(socketfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0){
        perror("connect");
        exit(3);
    }
                                                            45,35-38      57%
```

Ok, at the client side, things are pretty similar on we are declaring the, socket we are creating a new socket and after creating the new socket, we are getting the here in the client side, we are getting the server IP or the server host name and the port from the common line argument. So, we are using that common line argument to create the server address.

(Refer Slide Time: 09:03)

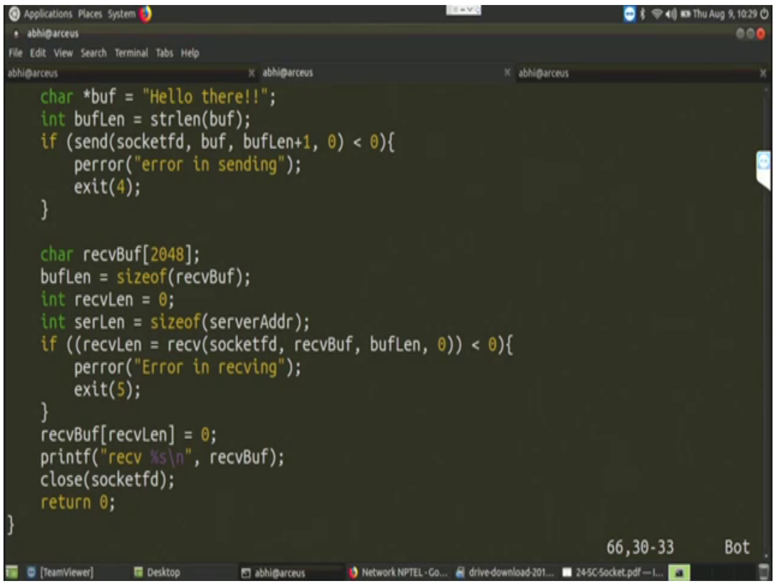


```
#include <string.h>
#include <netdb.h>

int main(int argc, char *argv[]){
    struct sockaddr_in serverAddr;
    struct hostent *server;
    char *hostname;
    int socketfd;

    if(argc < 3){
        fprintf(stderr, "use %s <hostname> <port>\n", argv[0]);
        exit(-1);
    }

    hostname = argv[1];
    int portno = atoi(argv[2]);

    socketfd = socket(AF_INET, SOCK_STREAM, 0);
    if(!socketfd){
        perror("Error in opening socket");
                                                  14,14-17       18%
```

So, we have declared the server address as a variable as a struct sockaddr in variable. So, inside that we are putting the server address. So, we are getting the server IP. So, get host by name this particular function returned a host IP from the host name. So, we are getting that and then we are putting that server address in this severAddr variable. So, we are initializing the serverAddr variable with all zero's that is to initialization that the sever address family that is AF INET to use the IPv 4 address then this h addr and this particular field we are using to provide the corresponding server address and the port number.

Now, you are making a connect call to initiate the connection at the client side. So, the connect call it sends this socket file descriptor along with the sever address that we have seen earlier.

(Refer Slide Time: 10:14)



```
    char *buf = "Hello there!!";
    int bufLen = strlen(buf);
    if (send(socketfd, buf, bufLen+1, 0) < 0){
        perror("error in sending");
        exit(4);
    }

    char recvBuf[2048];
    bufLen = sizeof(recvBuf);
    int recvLen = 0;
    int serLen = sizeof(serverAddr);
    if ((recvLen = recv(socketfd, recvBuf, bufLen, 0)) < 0){
        perror("Error in recving");
        exit(5);
    }
    recvBuf[recvLen] = 0;
    printf("recv %s\n", recvBuf);
    close(socketfd);
    return 0;
}
                                                    66,30-33      Bot
```

So, it initializes the connection to the connection to the server using the tcp three way handshaking procedure. Now, after that after the connect system call has made, you have initiated the connection to the server then, you make the call to the send function, the send function to send the data. So, we are creating a message called hello dear this particular message, we are sending to the server and as you have seen in the server code that the server will actually go back that particular thing to the client.

So, after you are sending that you are receiving the message. So, we are declaring a buffer as a character array, we will get the data in the form of a character array or a string. So, we are declaring that buffer for that and after that we are making a received call, this received call over that socket file descriptor that will get the data from the client side.

So from the server side so as you have seen in the server code, the server will actually go back the data that you are sending so, here you are sending this hello dear function, same hello dear function will be quote back. So, you will receive that data and after receiving that data you print that particular data at the concept. So, that is the code for the client side.

Now, let us compile and run this two things. So, first let us compile tcp server and compile tcp client. So, first we will run the server. So, we need to specify the port address. So, we are giving port address as 2 4 4 4, those sever will now the server is learning there. So, once the server is learning there from the client side, we can give so the server is running in my machine.

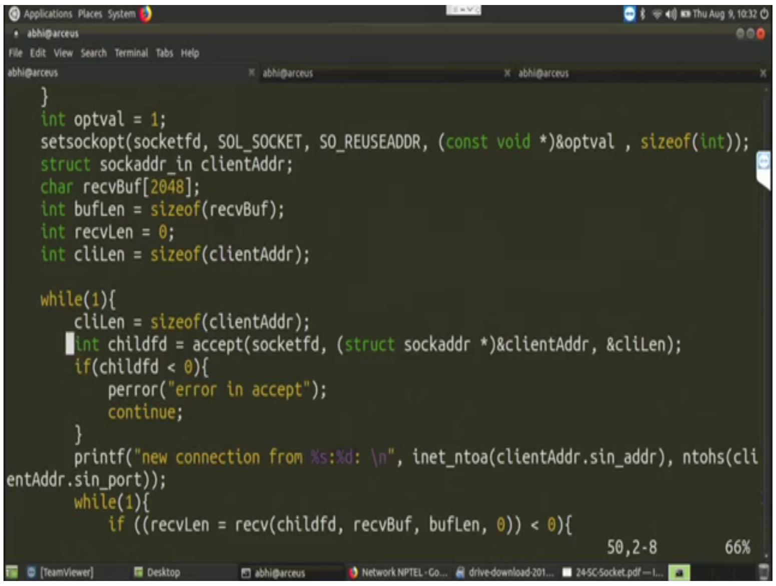
So, I am giving the server host name as local host and the corresponding server port that, we have used here 2 4 4 4. So, it has received the message and at the server side you can see that, it has received a new connection from the local machine the IP is 127 dot 0 dot 0 dot 1 and a port of 47676. So, it has received this hello dear message, it has ecode it back and the client has received that message.

So, if you again run the client, again it has received the hello message and you can see that it has received another connection at a different port 47678.

So, if you run it again, you see that it is again using a different port 47680. So, the same thing as we have seen for the UDP server case that, the server is announcing itself to a well known port, but the client need not to do it. So, the client is actually using some random port to initiate the connection from the client side, that particular code value we can see here.

Now, this is a kind of tcp server and a tcp client implementation, where in the tcp server implementation, we call it a kind of iterative server why we call it a iterative sever? Because if you look into the code, let me open the code again.

(Refer Slide Time: 14:06)


Applications Places System
abhi@arceus
File Edit View Search Terminal Tabs Help
abhi@arceus
abhi@arceus
abhi@arceus

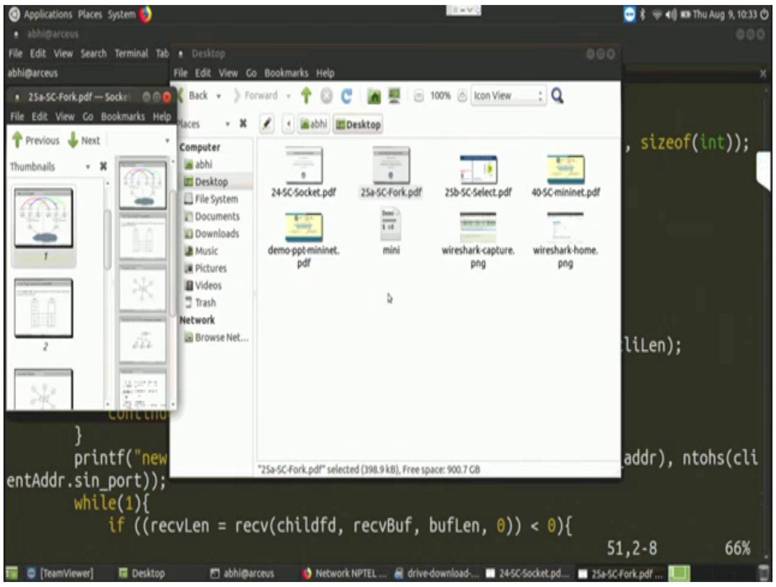
```
    }
    int optval = 1;
    setsockopt(socketfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval , sizeof(int));
    struct sockaddr_in clientAddr;
    char recvBuf[2048];
    int bufLen = sizeof(recvBuf);
    int recvLen = 0;
    int cliLen = sizeof(clientAddr);

    while(1){
        cliLen = sizeof(clientAddr);
        int childfd = accept(socketfd, (struct sockaddr *)&clientAddr, &cliLen);
        if(childfd < 0){
            perror("error in accept");
            continue;
        }
        printf("new connection from %s:%d: \n", inet_ntoa(clientAddr.sin_addr), ntohs(cli
entAddr.sin_port));
        while(1){
            if ((recvLen = recv(childfd, recvBuf, bufLen, 0)) < 0){
                                                            50,2-8        66%
```

So, here you will see that this connections. So, here we are making a while loop and inside this while loop, we are accepting the connection.

So, how this entire thing will work if multiple client equation are coming simultaneously then, one client will be taken that that connection will be taken that will get executed. Then the next client will be taken, that will get executed then, the third client will be taken that, will get executed and that way this entire thing will go on. So, that is why we call this kind of server implementation as an iterative server implementation, where the client requests are executed one by one.

So, now we will see that we. So, one thing is that this kind of iterative server implementation is not very useful, when you are trying to design a server like a web server while you can get thousands of request per second. Now if that is the case, then you have to handle the client request in parallel, otherwise the sever will not scale up. So, we will now see that how you can actually implement a parallel server using this help of socket programming as well as some concept of your operating system level programming and your op operating system concept of multiple processes.

(Refer Slide Time: 15:27)



```
, sizeof(int));

liLen);

        }
        printf("new                                    addr), ntohs(cli
entAddr.sin_port));
        while(1){
            if ((recvLen = recv(childfd, recvBuf, bufLen, 0)) < 0){
                                                          51,2-8         66%
```

So, if you do not have a understanding of this operating system concept of processes.

(Refer Slide Time: 15:44)

Socket Programming - Concurrent Servers

Sandip Chakraborty

Department of Computer Science and Engineering,

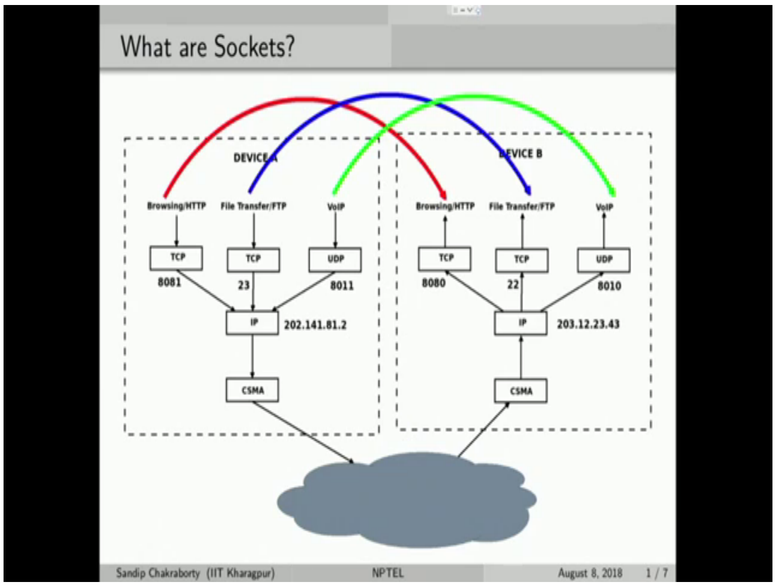
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

August 8, 2018

Sandip Chakraborty (IIT Kharagpur) NPTEL August 8, 2018 0 / 7

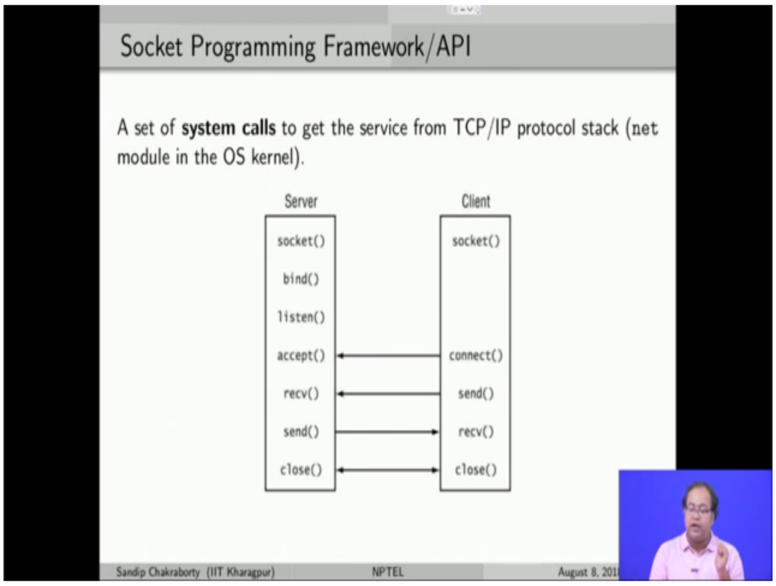
I will suggest you to look into this process concept in operating system and look into it in details. So, we will use this concept of process in the operating system to look into this implementation of concurrent servers at the parallel servers. So, as we have looked earlier that you are creating multiple such connections.

(Refer Slide Time: 15:50)


## What are Sockets?

Sandip Chakraborty (IIT Kharagpur) NPTEL August 8, 2018 1 / 7

(Refer Slide Time: 15:56)

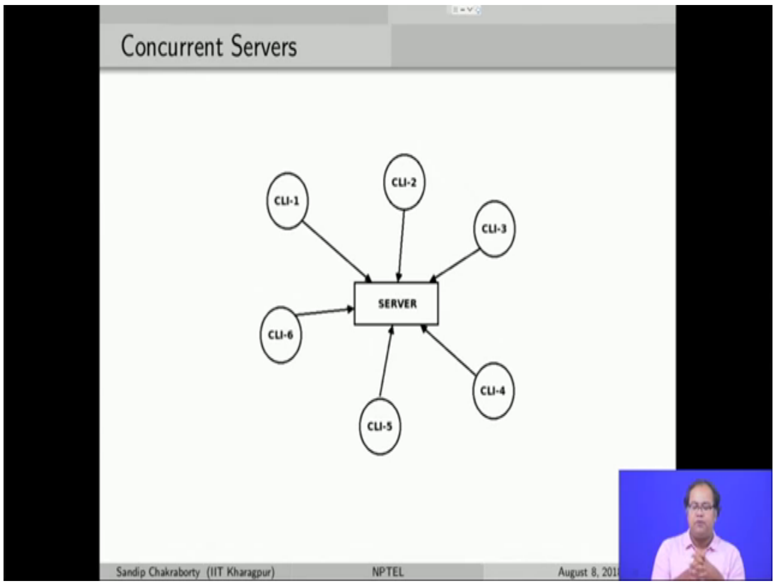


## Socket Programming Framework/API

A set of **system calls** to get the service from TCP/IP protocol stack (net module in the OS kernel).

Sandip Chakraborty (IIT Kharagpur) NPTEL August 8, 201

(Refer Slide Time: 15:59)

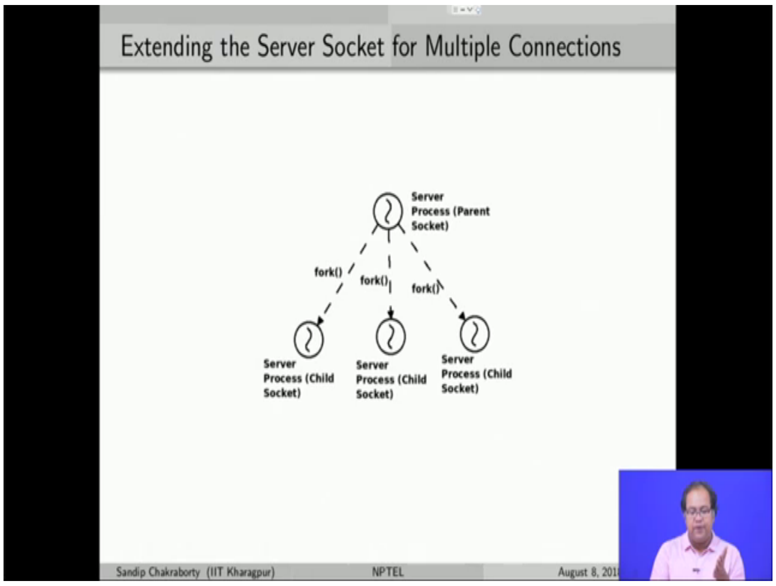


## Concurrent Servers

Sandip Chakraborty (IIT Kharagpur) NPTEL August 8, 201

We have already looked this socket programming framework or the api s. Now, the concept of concurrent sever is something like this where multiple clients are trying to connect to the server simultaneously.

(Refer Slide Time: 16:14)



## Extending the Server Socket for Multiple Connections

Server Process (Parent Socket)
fork()
fork()
fork()
Server Process (Child Socket)
Server Process (Child Socket)
Server Process (Child Socket)
Sandip Chakraborty (IIT Kharagpur)
NPTEL
August 8, 201

Now, if multiple clients are trying to connect to the server simultaneously, then how will you handle that thing. So, one ways to handle it is using the iterative server, but as we have seen that the iterative server may not be very useful. So, we implement the parallel server using this concept of multiple processes in operating system.

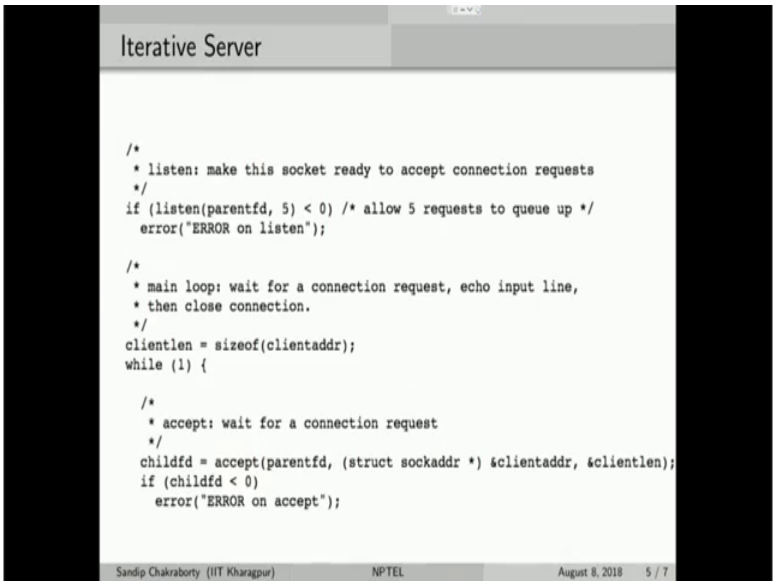
So the idea is something like this you have a server process, which is having the parent socket then, you can make a fork system call. So, in operative system, this fork system call create such i process. So, what you can do that once, you are accepting a connection then you can make this fork system call to create a child process, which will actually handle the data transmission and data reception from that particular child connection.

If you are not doing that, if you are doing it in the earlier way that we are doing inside the while loop in the form of iterative server, then the time until a particular server is keep on sending and receiving data, it is not closing the connection up to that point the connection will remain blocked. So, the server will not be able to handle the second connection until the first connection is complete.

So, that is why the idea is that, whenever a new connection is coming, you make a fork system call and after making a fork system call have a child process, that child process that is the part of the server process, but that will create a child socket it will. So, we have seen that after you are making an accept call at the server site, it returns a new socket identify, which is used to send or receive data to the client.

So, you pass that particular new socket file descriptor to the child socket, which will handle data transmission and reception in parallel. So, the broad idea here is that you get all the connections one after another, but do not wait for the send and a receive functionality of our individual connections and keep other connections in the waiting queue. So, you create a child process and that child process will or we call it as a child socket, that will handle the send and a receive data functionalities of individual client request that you are getting.

(Refer Slide Time: 18:32)



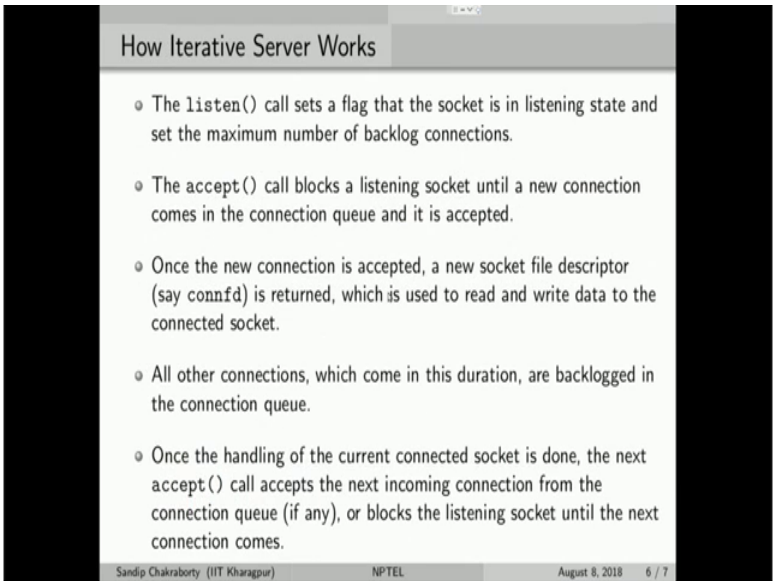
## Iterative Server

```
/*
 * listen: make this socket ready to accept connection requests
 */
if (listen(parentfd, 5) < 0) /* allow 5 requests to queue up */
  error("ERROR on listen");

/*
 * main loop: wait for a connection request, echo input line,
 * then close connection.
 */
clientlen = sizeof(clientaddr);
while (1) {

  /*
   * accept: wait for a connection request
   */
  childfd = accept(parentfd, (struct sockaddr *) &clientaddr, &clientlen);
  if (childfd < 0)
    error("ERROR on accept");
```

Sandip Chakraborty (IIT Kharagpur) NPTEL August 8, 2018 5 / 7

So, that was our implementation of the iterative server that we have done, we have a while loop, inside the while loop you are making a accept call then, you have the send and receive and it is inefficient because until you are complete that send and a receive functionalities, you will not be able to come out of that and get or make the accept call again to accept the new incoming connection.

(Refer Slide Time: 18:56)



## How Iterative Server Works

- The `listen()` call sets a flag that the socket is in listening state and set the maximum number of backlog connections.
- The `accept()` call blocks a listening socket until a new connection comes in the connection queue and it is accepted.
- Once the new connection is accepted, a new socket file descriptor (say `connfd`) is returned, which is used to read and write data to the connected socket.
- All other connections, which come in this duration, are backlogged in the connection queue.
- Once the handling of the current connected socket is done, the next `accept()` call accepts the next incoming connection from the connection queue (if any), or blocks the listening socket until the next connection comes.

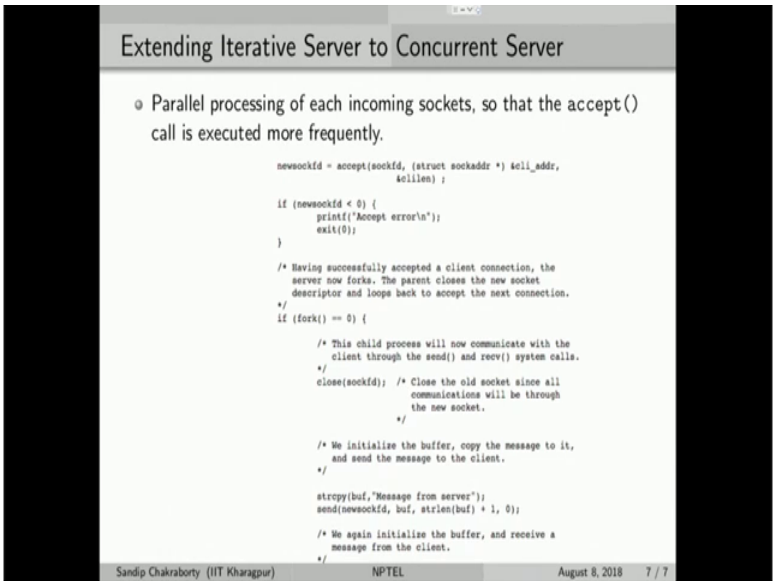
Sandip Chakraborty (IIT Kharagpur) NPTEL August 8, 2018 6 / 7

So, if you look into that how iterative server works that the listen call it sets a flag that, the socket is in the listening state and set the maximum number of backlog connections that you have seen earlier then, the accept call it blocks a receiving socket, until a new connection comes in the connection tube and it is accepted. So, this accept call is a blocking call. So, the system will keep on waiting here until, you are making a connect call from the client side. So it is a blocking call.

Now, once this new connection is accepted, then the new socket file descriptor say, the connection ft it is written, which is used to read and write data or to send and receive data to the connective socket. Now all other connection, which comes in this duration are backlogged in the connection queue, because the process is busy inside this while loop to send and receive data. So, once the handling of this current connection socket is done current connected socket is done, then the next accept call they accept the next incoming connections from the connection tube if there is any or blocks the receiving socket until the next connection comes.

So, once the execution of this while loop complete, then only you are connect coming in here and accepting the next connection. So, everything is actually blocked on the send and a received call that you have there at the this particular while block.

(Refer Slide Time: 20:30)



## Extending Iterative Server to Concurrent Server

- Parallel processing of each incoming sockets, so that the accept() call is executed more frequently.

```
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                    &clilen) ;

if (newsockfd < 0) {
        printf("Accept error\n");
        exit(0);
}

/* Having successfully accepted a client connection, the
   server now forks. The parent closes the new socket
   descriptor and loops back to accept the next connection.
*/
if (fork() == 0) {

        /* This child process will now communicate with the
           client through the send() and recv() system calls.
        */
        close(sockfd);  /* Close the old socket since all
                           communications will be through
                           the new socket.
                        */

        /* We initialize the buffer, copy the message to it,
           and send the message to the client.
        */

        strcpy(buf,"Message from server");
        send(newsockfd, buf, strlen(buf) + 1, 0);

        /* We again initialize the buffer, and receive a
           message from the client.
        */
```

Sandip Chakraborty (IIT Kharagpur) NPTEL August 8, 2018 7 / 7

Now, we extend this iterative sever to a concurrent server. So, our idea is that the parallel processing of each incoming socket so, that the accept call is executed more frequently.

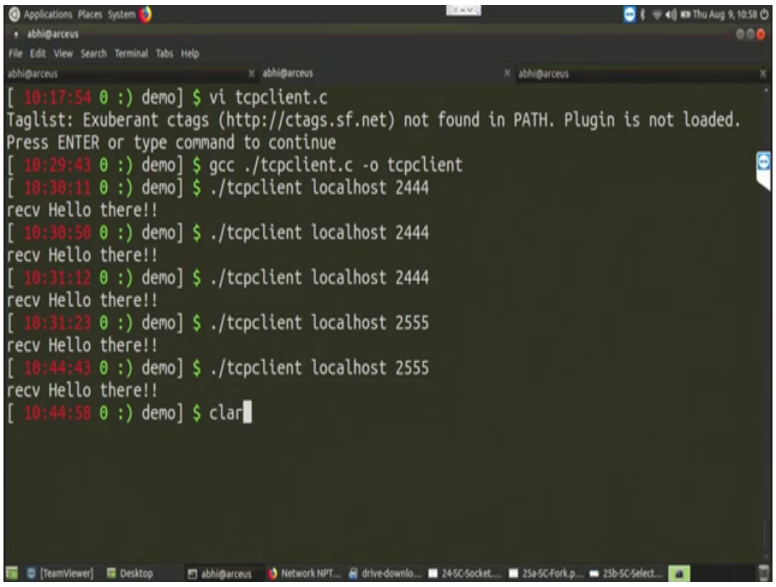
So, what we do here? Here you see that after we are making this accept call, which is returning the new socket file descriptor then, we have successfully accepted a client connection then we making a fork system call. So, this fork system call at the parent process, it returns the idea of the child process and at the child process it returns 0.

So, whenever you are making this fork system call and if the fork system call is returning 0; that means, you are inside the child process. Now this child process, what it does? It closes the original socket the server socket and it will use this new socket d that you got to send and receive data. So, you close the old socket since all communication will be through the new socket and then you initialize the buffer copy the message to it send or do the receipt whatever you want to do.

Now, what will happen here that whatever is there inside the fork system call, inside this fork block in this e block that will executed get executed in parallel. So, in the parent process the parent process will not so for the parent process, this will receive return falls, because parent in the n plus is fork returns the idea of the child process and in the child process fork returns 0.

So, in the parent process you will return back and the parent process will again come here and make the next accept call. So, the parent process is now, do not need to wait for this send and receive functionalities, which will be handled by a child process.

(Refer Slide Time: 22:38)



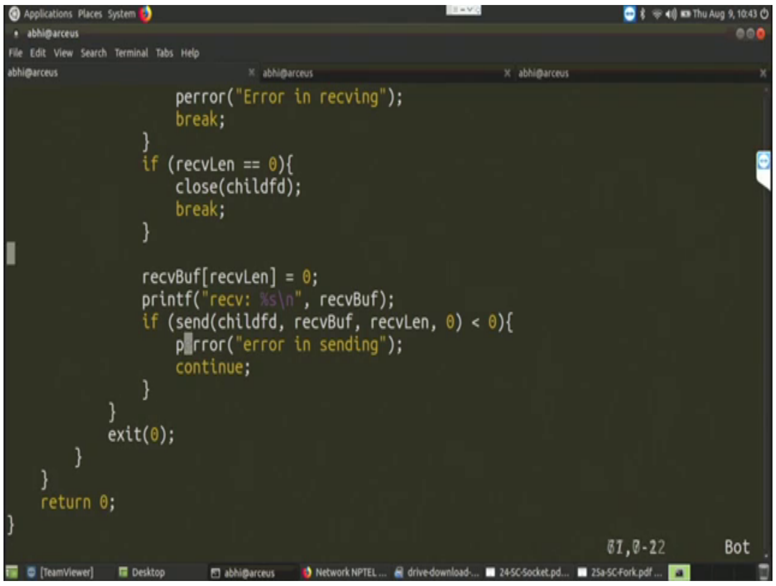
```
[ 10:17:54 0 :) demo] $ vi tcpclient.c
Taglist: Exuberant ctags (http://ctags.sf.net) not found in PATH. Plugin is not loaded.
Press ENTER or type command to continue
[ 10:29:43 0 :) demo] $ gcc ./tcpclient.c -o tcpclient
[ 10:30:11 0 :) demo] $ ./tcpclient localhost 2444
recv Hello there!!
[ 10:30:50 0 :) demo] $ ./tcpclient localhost 2444
recv Hello there!!
[ 10:31:12 0 :) demo] $ ./tcpclient localhost 2444
recv Hello there!!
[ 10:31:23 0 :) demo] $ ./tcpclient localhost 2555
recv Hello there!!
[ 10:44:43 0 :) demo] $ ./tcpclient localhost 2555
recv Hello there!!
[ 10:44:58 0 :) demo] $ clar
```

Now, let us look into the demo of this one. So, here we do not make any change in the client implementation. The client implementation remains as it was earlier, we will only make change at the server implementation. So, at the server implementation, so the change that, we have made the entire code is similar if you look into that we are declaring this server address, followed by a socket call then initializing the addresses at the server address field, making a bind call, making a listen call to broadcast the or not to actually broadcast this may not be a correct term to announce the quote, where the sever is actually listening and after that making that sets of opt call as earlier and then inside this while loop, where you are accepting accepting the connection we are accepting accepting a new connection and it is going to this child fd we are creating a new file descriptor.

After that you see we are making a fork call. So, you look into this statement in for return equal to fork. So, we are making a fork call which will create a child process, now I will suggest you to look into this operating system concept in details, we do not have scope to discuss that, the idea here is that whenever you are making a fork call this entire code will be copied to the child process as well. So, the child process what it will do? In the child process, the fork call will written a 0 and in the parent process the fork call will return the idea of the child process.

So, if you make a if loop with this return value if the return value is 0; that means, you are inside the child process.

(Refer Slide Time: 24:32)


Applications Places System
abhi@arceus
File Edit View Search Terminal Tabs Help
abhi@arceus
abhi@arceus
abhi@arceus
Thu Aug 9, 10:43

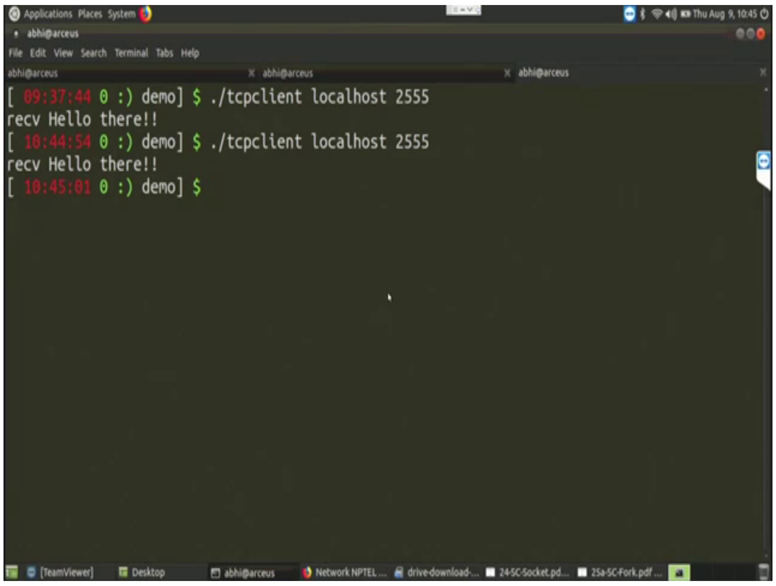
```
                    perror("Error in recving");
                    break;
                }
                if (recvLen == 0){
                    close(childfd);
                    break;
                }

                recvBuf[recvLen] = 0;
                printf("recv: %s\n", recvBuf);
                if (send(childfd, recvBuf, recvLen, 0) < 0){
                    perror("error in sending");
                    continue;
                }
            }
            exit(0);
        }
    }
    return 0;
}
                                                           61,0-22       Bot
```

So, inside the child process this part of the code will get executed. Now you see this the second while loop that we had which actually deals with sending and a receiving data, it is inside only the child process. So, it is inside this e block here we are making this received call and a send call.

And we are having that hand blink in parallel at the sever side. So, in the server side this particular e block will not get executed, because in the server in the parent side, parent socket this will not get executed, because the parent socket will return the ID of the child. So, it will be not equal to 0.

So, this part will only execute at the child process. So the child process is now dealing with sending and receiving of the data, but the parent process can come out of that the parent process is not getting blocked in this second while loop, it can directly come out and accept the next connection.

(Refer Slide Time: 25:48)

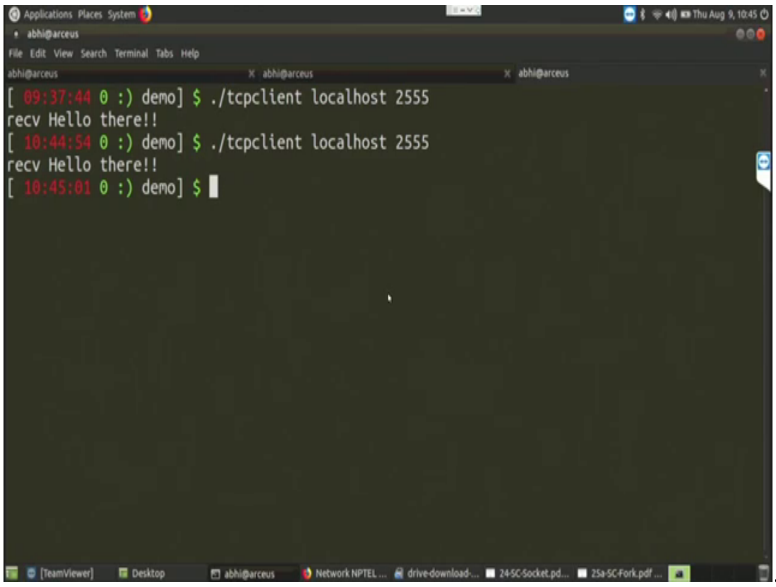


```
[ 09:37:44 0 :) demo] $ ./tcpclient localhost 2555
recv Hello there!!
[ 10:44:54 0 :) demo] $ ./tcpclient localhost 2555
recv Hello there!!
[ 10:45:01 0 :) demo] $
```

Now net let us run it. So let us first compile it minus o we name it as forkserver and running the forkserver here we have to give the fork.

Now this distance say, if the port has 2 5 5 5. So, the my server is running and the client is the same tcp client that we used earlier. So, let us initiate the connection from here another connection say from another task.

(Refer Slide Time: 26:33)

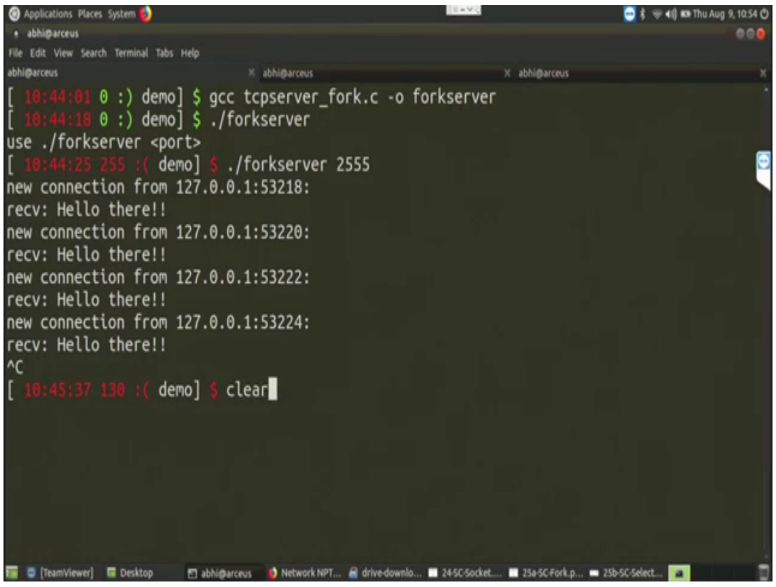


```
[ 09:37:44 0 :) demo] $ ./tcpclient localhost 2555
recv Hello there!!
[ 10:44:54 0 :) demo] $ ./tcpclient localhost 2555
recv Hello there!!
[ 10:45:01 0 :) demo] $
```

.

So, I am not doing much tasks.

(Refer Slide Time: 26:51)

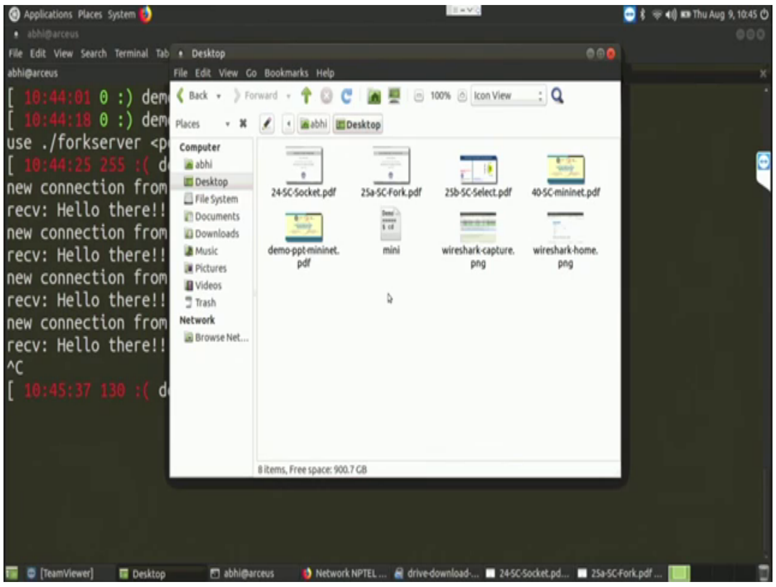


```
[ 10:44:01 0 :) demo] $ gcc tcpserver_fork.c -o forkserver
[ 10:44:18 0 :) demo] $ ./forkserver
use ./forkserver <port>
[ 10:44:25 255 :( demo] $ ./forkserver 2555
new connection from 127.0.0.1:53218:
recv: Hello there!!
new connection from 127.0.0.1:53220:
recv: Hello there!!
new connection from 127.0.0.1:53222:
recv: Hello there!!
new connection from 127.0.0.1:53224:
recv: Hello there!!
^C
[ 10:45:37 130 :( demo] $ clear
```

So, we will possibly not get a feel of this parallel execution, but what is happening here. A new child process gets created and it handles this individual request that we are sending here.

So, you see that we have made some client execution here in this tab and as well as in the another tab and correspond to that, you are getting the connections at different port and they are getting executed in parallel. Now this handling is done by a child process that are then the original parent process.

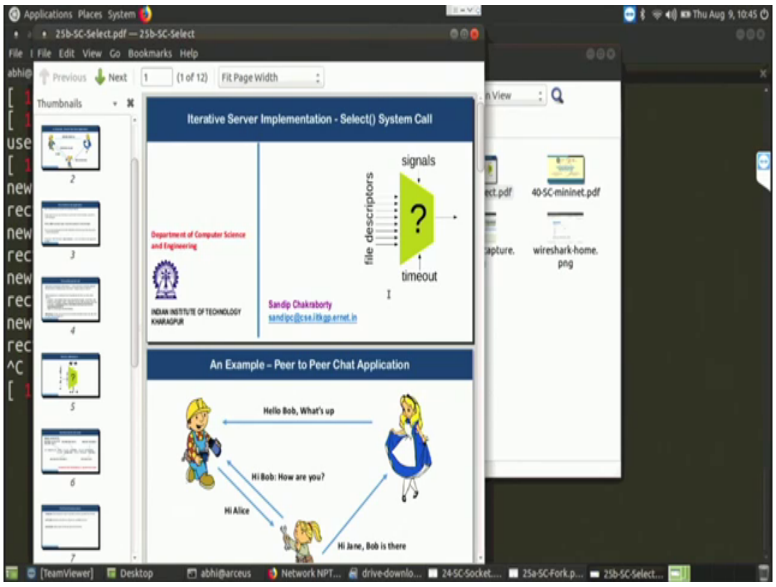
(Refer Slide Time: 27:28)



```
[ 10:44:01 0 :) dem
[ 10:44:18 0 :) dem
use ./forkserver <p
[ 10:44:25 255 :( d
new connection from
recv: Hello there!!
new connection from
recv: Hello there!!
new connection from
recv: Hello there!!
new connection from
recv: Hello there!!
^C
[ 10:45:37 130 :( d
```

Ok now, we have another interesting thing to discuss. So, ok.

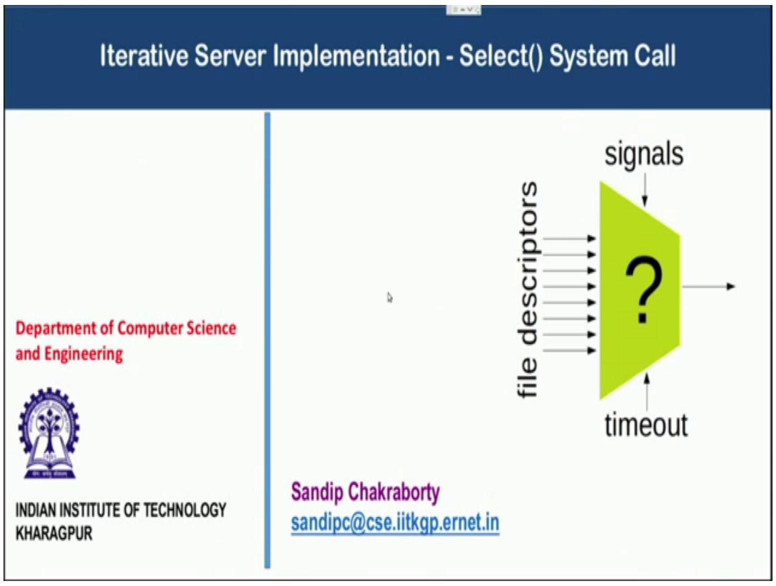
(Refer Slide Time: 27:33)



Iterative Server Implementation - Select() System Call

An Example – Peer to Peer Chat Application

(Refer Slide Time: 27:38)



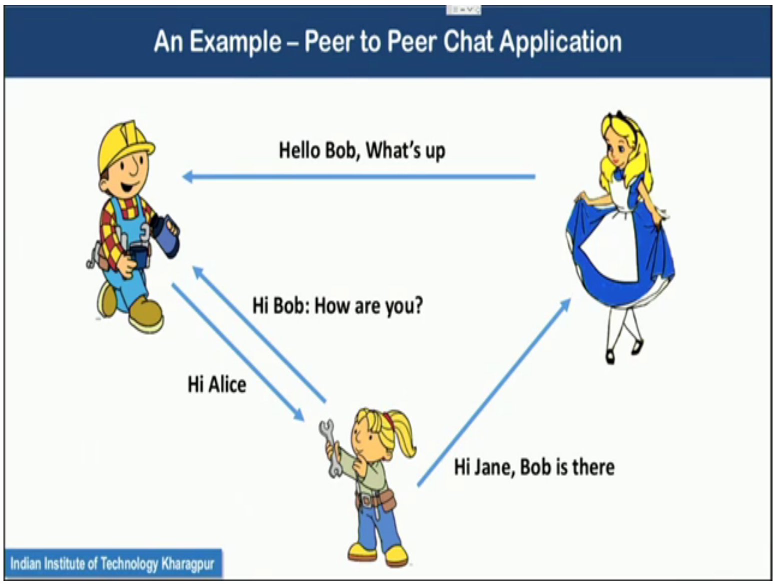
## Iterative Server Implementation - Select() System Call

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

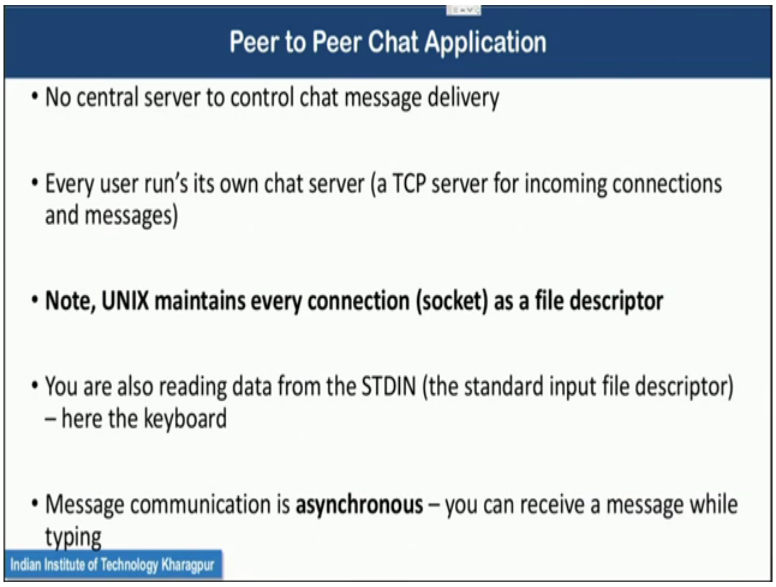
Sandip Chakraborty
sandipc@cse.iitkgp.ernet.in

(Refer Slide Time: 27:44)


## An Example – Peer to Peer Chat Application

So, what we have seen till now, that we can do a kind of server implementation. Concurrent server implementation, now think of a application something like that a kind of peer to peer chat application. So, multiple people they want to send or receive messages to each other.

(Refer Slide Time: 27:59)


## Peer to Peer Chat Application

- No central server to control chat message delivery
- Every user run's its own chat server (a TCP server for incoming connections and messages)
- **Note, UNIX maintains every connection (socket) as a file descriptor**
- You are also reading data from the STDIN (the standard input file descriptor) – here the keyboard
- Message communication is **asynchronous** – you can receive a message while typing

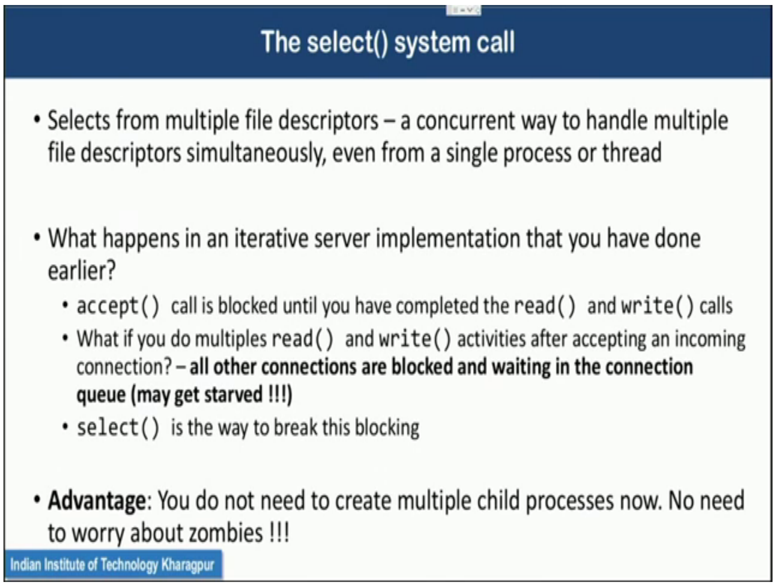
Indian Institute of Technology Kharagpur

Now in this particular case, what may happen that we do not have any central server to control the chat message delivery, every user runs it is own chat server; that means, it runs the TCP server for incoming come incoming connections and messages, now in UNIX, it maintains every connection as a file descriptor that we have already seen.

Now, at that time in a chat server you have a typical requirement that you need to also read data from standard input. So, you need to type something. So, when you are receiving a message during that time say, you are typing something. Now whenever you are typing something that standard input it is again a file descriptor. So in UNIX, everything is a file a socket is represented by a file descriptor and at the same time the input from the keyboard that is also represented as an file descriptor by this STDIN file descriptor.

Now in that case, the question come how will you handle it? Because your data is coming from multiple places, you are getting data from the socket as well as you are entering data from the keyboard, how you actually switch between this multiple thing, where this message communication is asynchronous. So, you can receive a message while you are doing the typing.

(Refer Slide Time: 29:16)



## The select() system call

- Selects from multiple file descriptors – a concurrent way to handle multiple file descriptors simultaneously, even from a single process or thread
- What happens in an iterative server implementation that you have done earlier?
  - `accept()` call is blocked until you have completed the `read()` and `write()` calls
  - What if you do multiples `read()` and `write()` activities after accepting an incoming connection? – **all other connections are blocked and waiting in the connection queue (may get starved !!!)**
  - `select()` is the way to break this blocking
- **Advantage**: You do not need to create multiple child processes now. No need to worry about zombies !!!

Indian Institute of Technology Kharagpur

So for that we use this concept at the select system call, which is again an operating system level system call.

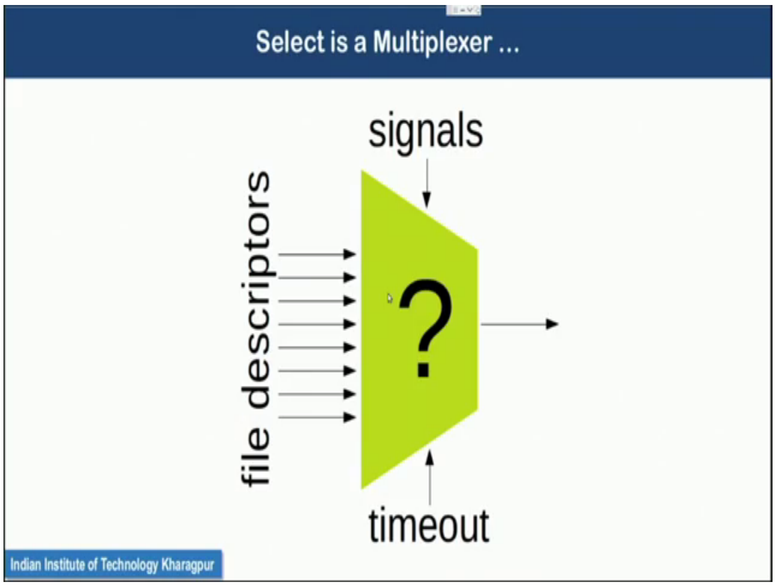
So the select system call, it selects from multiple file descriptor, which is a concurrent way to handle multiple file descriptor simultaneously even from a single process are failed. So, you can get the data from the socket, which is one of the file descriptor as well as the keyboard file descriptor simultaneously.

Now, what we have seen that, what happens in an iterative server implementation that we have done earlier that the accept call is blocked until you have completed the read and the write calls. Now what if you do multiple read and write activates after accepting an incoming connection, that the other connections are blocked and waiting for the connection queue, they may get starved. Now select is the way to break this blocking.

So, one way to break the blocking is using this parallel implementation another way to do that thing is to use the select system call. The advantage with this select system call is that, you do not need to create multiple child processes, now no you do not need to worry about the zombies. So, child processes has always a problem that if you are if sometime the parent process get killed or the parent process stops, then the child process become zombie. So, with the select system call you do not need to worry about this zombie. So, you can possibly manage resources more efficiently.

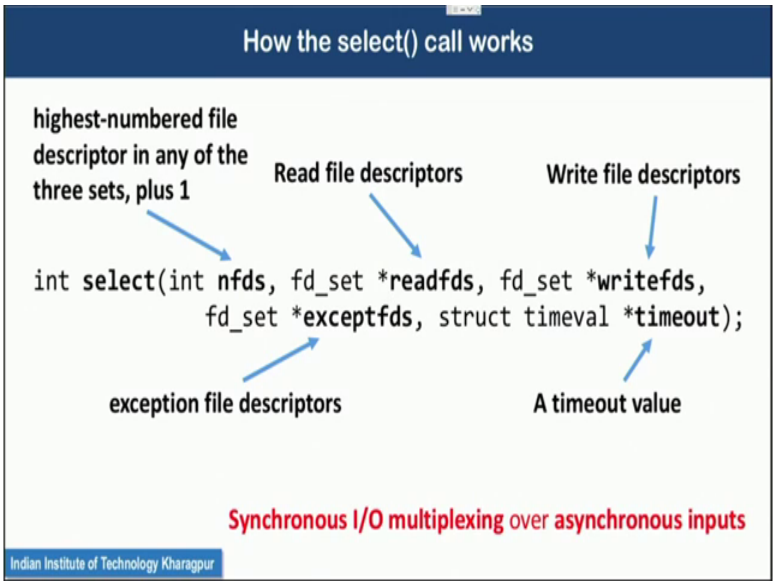
So, this selects system call it is nothing, but a multiplexors.

(Refer Slide Time: 30:45)


Select is a Multiplexer ...
signals
file descriptors
?
timeout
Indian Institute of Technology Kharagpur

So, what happens that you have multiple file descriptors, you have certain signals and the time out and out of this multiple file descriptor, it selects one of the file descriptor. So, it finds out that among this file descriptor which one is currently active and it select that particular file descriptor.

(Refer Slide Time: 31:02)



## How the select() call works

highest-numbered file descriptor in any of the three sets, plus 1
Read file descriptors
Write file descriptors
exception file descriptors
A timeout value

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
               fd_set *exceptfds, struct timeval *timeout);
```

Synchronous I/O multiplexing over asynchronous inputs

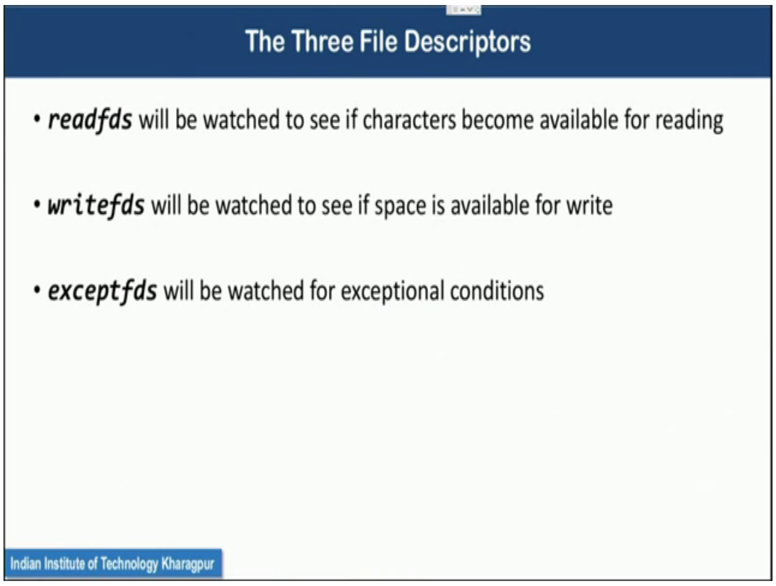
Indian Institute of Technology Kharagpur

So, here is the format of the select system call, in the select system call you are providing the number of file descriptor, it is the highest number of file descriptor it is the highest number of file descriptor in any of the three sets plus 1, we have three different sets of file descriptor, the read file descriptor to read something as an input, the right file descriptor to write something at the output and except file descriptor to handle the exceptions.

So, this file descriptors are a kind of structure called fd set and we have a timeout value. So, the timeout value is that if you are not getting anything from this file descriptor for this timeout value. So, it will come out of the things.

So, it is actually providing you a synchronous I O multiplexing over asynchronous input. So, as we have mentioned earlier that your input can be asynchronous, you can get a message over the socket while you are doing the typing, but it provides you a multiplexing to select either the keyboard or the socket at one time distance.

(Refer Slide Time: 32:09)



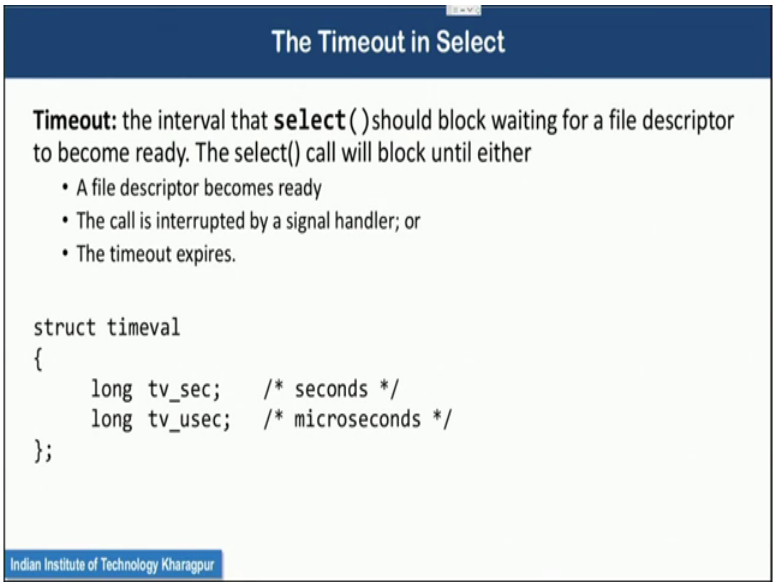
## The Three File Descriptors

- ***readfds*** will be watched to see if characters become available for reading
- ***writefds*** will be watched to see if space is available for write
- ***exceptfds*** will be watched for exceptional conditions

Indian Institute of Technology Kharagpur

So, you have three different file descriptor that each file descriptor will be watch to see if characters become available for reading, the write file descriptor will be watched to see if the spaces available for write and the exceptional file descriptor will be watched for exceptional condition.

(Refer Slide Time: 32:23)



## The Timeout in Select

**Timeout:** the interval that **select()** should block waiting for a file descriptor to become ready. The select() call will block until either

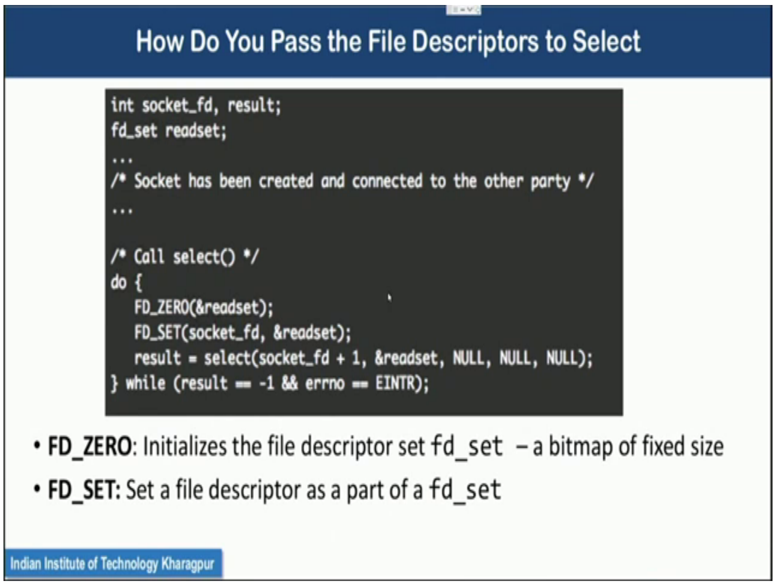
- A file descriptor becomes ready
- The call is interrupted by a signal handler; or
- The timeout expires.

```
struct timeval
{
     long  tv_sec;    /* seconds */
     long  tv_usec;   /* microseconds */
};
```

Indian Institute of Technology Kharagpur

Now we have a timeout value, the interval that select should block waiting for a file descriptor to become ready, now the select call remain block either the file descriptor becomes ready or the call is interrupted by a signal handler or a timeout expires. So, in any of the cases it comes out of the select. So, whenever one of the file descriptor becomes ready it comes out of the select call, if that particular call is interrupted by some other signals or the timeout happens. So, this is the procedure to set the timeout you provide the value in the form of second and microseconds.

(Refer Slide Time: 32:59)



## How Do You Pass the File Descriptors to Select

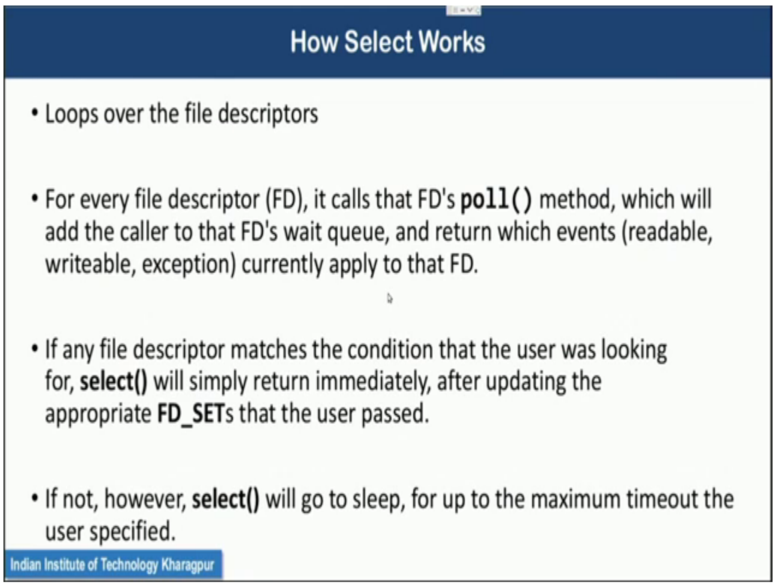
```
int socket_fd, result;
fd_set readset;
...
/* Socket has been created and connected to the other party */
...

/* Call select() */
do {
   FD_ZERO(&readset);
   FD_SET(socket_fd, &readset);
   result = select(socket_fd + 1, &readset, NULL, NULL, NULL);
} while (result == -1 && errno == EINTR);
```

- **FD_ZERO**: Initializes the file descriptor set `fd_set` – a bitmap of fixed size
- **FD_SET:** Set a file descriptor as a part of a `fd_set`

Indian Institute of Technology Kharagpur

Now, how do you pass the file descriptor to select? So for that what we do, we first initialize the file descriptor set fd set, which is a bitmap of fix size with this fd 0, then we make the call to fd set. So, if this fd set selects a file descriptor say, my socket that I have defined and the corresponding the file descriptor, which where the socket is getting added. So, this particular bit corresponds to this socket file descriptor that will get set; that means, some data is available there.

(Refer Slide Time: 33:42)



## How Select Works

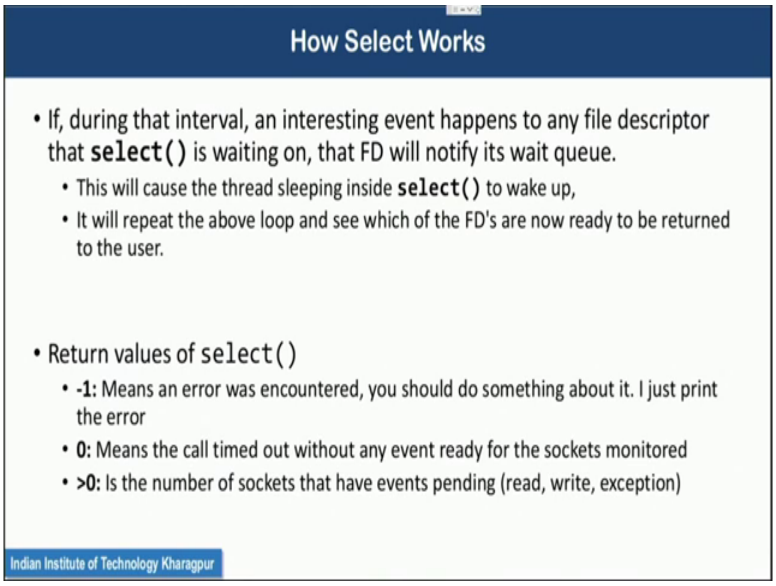
- Loops over the file descriptors
- For every file descriptor (FD), it calls that FD's `poll()` method, which will add the caller to that FD's wait queue, and return which events (readable, writeable, exception) currently apply to that FD.
- If any file descriptor matches the condition that the user was looking for, **select()** will simply return immediately, after updating the appropriate **FD_SET**s that the user passed.
- If not, however, **select()** will go to sleep, for up to the maximum timeout the user specified.

Indian Institute of Technology Kharagpur

Ok then you make the select call so, how select call? So, it looks over all the file descriptor for every file descriptor it calls the file descriptor poll method. So, this poll method is to check that whether something is available to or some event is waiting on that file descriptor. So, it will add the caller to that file descriptor wait queue and return, which events currently apply to that file descriptor whether it is file descriptor is readable.

If it is a read file descriptor whether it is writable, if it is a write file descriptor or some exception has happened. Now if any of the file descriptor matches the condition that if there was looking for read write or exception, then the select will simply written immediately, after updating the appropriate file descriptor set that the user passed and if not the select will go to sleep for the timeout value once, the timeout occurs it will come out of that select call.

(Refer Slide Time: 34:43)


## How Select Works

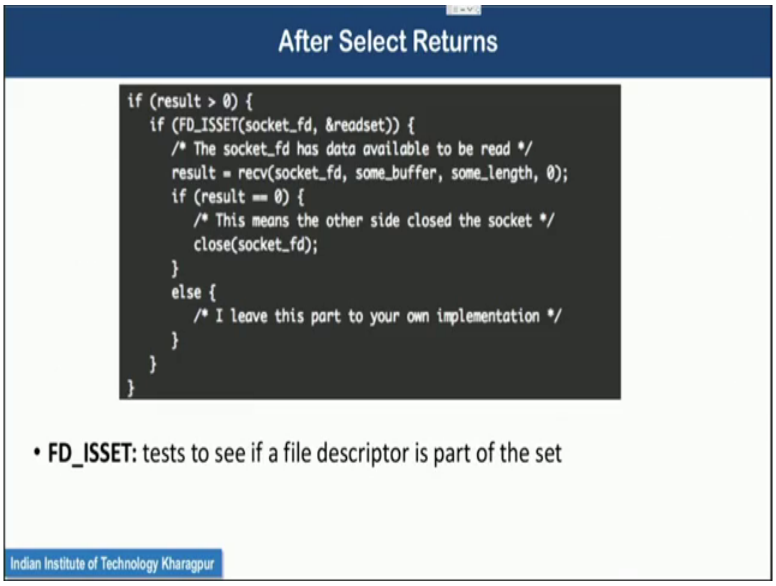
- If, during that interval, an interesting event happens to any file descriptor that **select()** is waiting on, that FD will notify its wait queue.
  - This will cause the thread sleeping inside **select()** to wake up,
  - It will repeat the above loop and see which of the FD's are now ready to be returned to the user.
- Return values of `select()`
  - **-1:** Means an error was encountered, you should do something about it. I just print the error
  - **0:** Means the call timed out without any event ready for the sockets monitored
  - **>0:** Is the number of sockets that have events pending (read, write, exception)

Indian Institute of Technology Kharagpur

And if some other events happen within that timeout event, it will made this FD set and come out of that. And during that interval if an interesting event happens to any of the file descriptor that select is waiting on that file descriptor will notify it is wait queue. So, this will cause that thread sleeping inside the select wakeup and it will repeat the above loop and see which of the file descriptor are now ready to be returned to the user.

Now, the return value of the select we have three values if this minus 1 means some error has encountered. 0 means that the timeout has happened and greater than 0 means, that that the number of sockets that has the event pending like read write or exception. So, for how many sockets that event is pending, whether you are going to read or write or having certain exception.

(Refer Slide Time: 35:23)



## After Select Returns

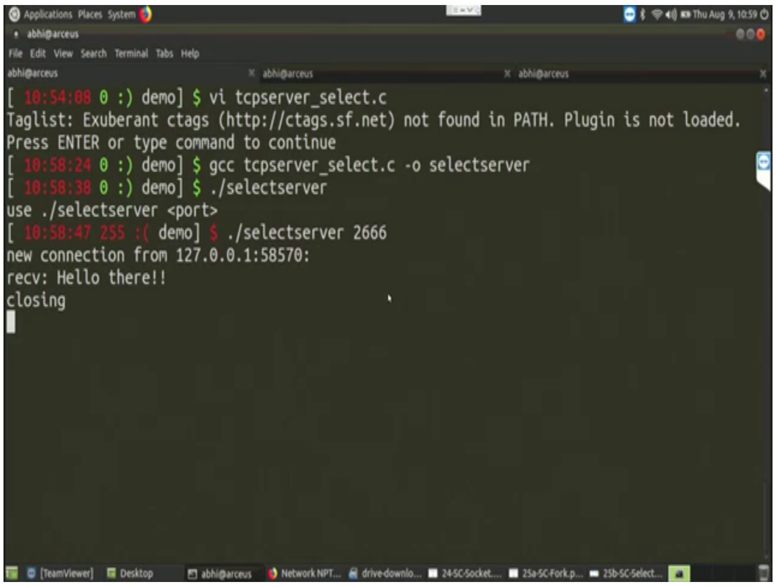
```
if (result > 0) {
   if (FD_ISSET(socket_fd, &readset)) {
      /* The socket_fd has data available to be read */
      result = recv(socket_fd, some_buffer, some_length, 0);
      if (result == 0) {
         /* This means the other side closed the socket */
         close(socket_fd);
      }
      else {
         /* I leave this part to your own implementation */
      }
   }
}
```

- **FD_ISSET:** tests to see if a file descriptor is part of the set

Indian Institute of Technology Kharagpur

So, after select returns you can use the function called FD set to test, whether a file descriptor is a part of that set. So, you can check whether a file descriptor has been set or not if the file descriptor has sat set; that means, you have something to read, if it is a read file descriptor or you have something to write if it is a write file descriptor.

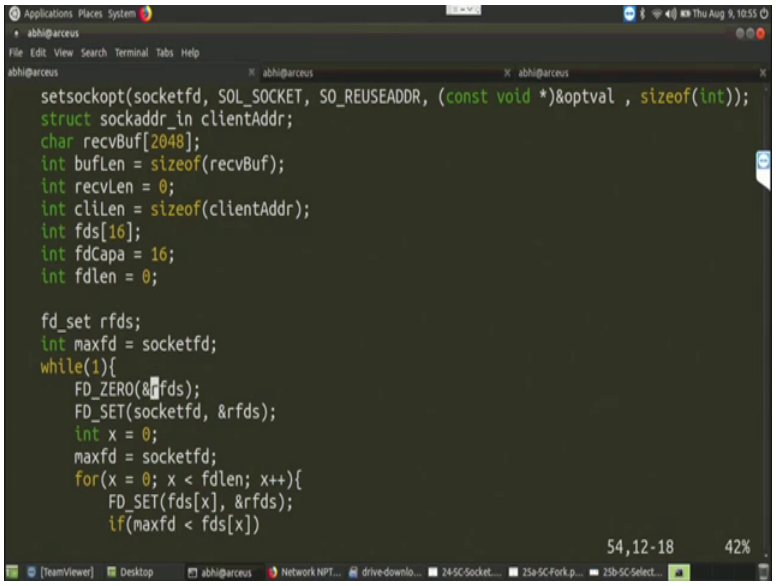
(Refer Slide Time: 35:55)


Applications Places System
Thu Aug 9, 10:59
abhi@arceus
File Edit View Search Terminal Tabs Help
abhi@arceus
abhi@arceus
abhi@arceus

```
[ 10:54:08 0 :) demo] $ vi tcpserver_select.c
Taglist: Exuberant ctags (http://ctags.sf.net) not found in PATH. Plugin is not loaded.
Press ENTER or type command to continue
[ 10:58:24 0 :) demo] $ gcc tcpserver_select.c -o selectserver
[ 10:58:38 0 :) demo] $ ./selectserver
use ./selectserver <port>
[ 10:58:47 255 :( demo] $ ./selectserver 2666
new connection from 127.0.0.1:58570:
recv: Hello there!!
closing
```

So, let us look into a quote which uses this select call. We will use the same tcp server implementation with this select.

(Refer Slide Time: 36:18)



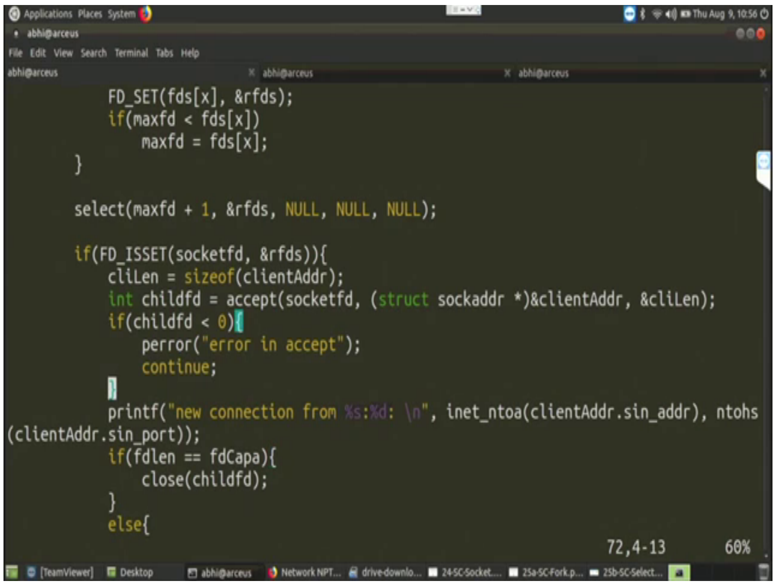
```
    setsockopt(socketfd, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval , sizeof(int));
    struct sockaddr_in clientAddr;
    char recvBuf[2048];
    int bufLen = sizeof(recvBuf);
    int recvLen = 0;
    int cliLen = sizeof(clientAddr);
    int fds[16];
    int fdCapa = 16;
    int fdlen = 0;

    fd_set rfds;
    int maxfd = socketfd;
    while(1){
        FD_ZERO(&rfds);
        FD_SET(socketfd, &rfds);
        int x = 0;
        maxfd = socketfd;
        for(x = 0; x < fdlen; x++){
            FD_SET(fds[x], &rfds);
            if(maxfd < fds[x])
                                                        54,12-18      42%
```

So, the base part of the code is same as earlier we are declaring the server address, the bind call, the listen call at the server side the sets of opt call and after that we are declaring the file descriptor. So, here we are declaring that we can have a maximum of 16 different file descriptor and this fd len returns, that how many are how many such file descriptor are currently set or currently active.

So, we are declaring the set of file descriptor and the maximum file descriptor which is equal to the current socket file descriptor. Now inside this while loop, we will first initialize the file descriptor, we are here only going to use the read file descriptor because we are going to read data from the socket and we are setting a go through this fd set call, we are setting the read file descriptor corresponds to the socket that we have defined or the socket where the server is actually listening.

Now, we are looping over the available file descriptor. So, here the idea is that whenever we are getting a new connection, we are adding it a inside the file descriptor inside that file descriptor set.

(Refer Slide Time: 37:25)



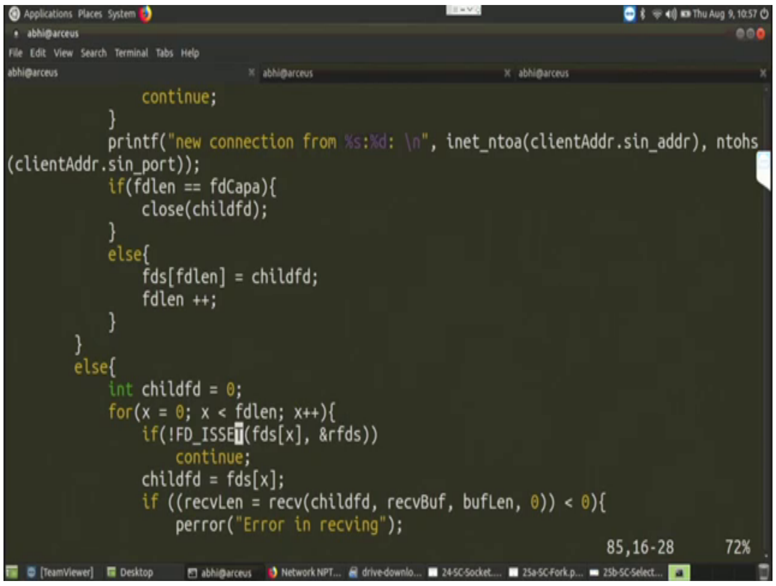
```
            FD_SET(fds[x], &rfds);
            if(maxfd < fds[x])
                maxfd = fds[x];
        }

        select(maxfd + 1, &rfds, NULL, NULL, NULL);

        if(FD_ISSET(socketfd, &rfds)){
            cliLen = sizeof(clientAddr);
            int childfd = accept(socketfd, (struct sockaddr *)&clientAddr, &cliLen);
            if(childfd < 0){
                perror("error in accept");
                continue;
            }
            printf("new connection from %s:%d: \n", inet_ntoa(clientAddr.sin_addr), ntohs
(clientAddr.sin_port));
            if(fdlen == fdCapa){
                close(childfd);
            }
            else{
                                                                      72,4-13       60%
```

After that we are making a select call. So, as we have seen earlier that there we are not giving any timeout value, because we are not giving any timeout value. So, it will keep on waiting for infinite duration, whenever some event will occur then only it will come out. So, we are initializing with this read file descriptors.

So whenever a new connection will come, it will come to this read file descriptor through this loop and then it will keep on waiting here, when some event will occur that old function will get triggered and that old function will return, whether certain event is there in the read file descriptors certain event means, whether that particular socket is ready to read the data.

And after that if select returns; that means, some socket is ready for reading the data then you check using fd set, whether that particular socket is ready to read the data if that socket is ready to read the data then, you make a accept call accept call to accept that particular connection ah; that means, some connection is waiting you will accept that connection and after doing that, the way we are closing the child file descriptor and then add that particular thing to that file descriptor loop.

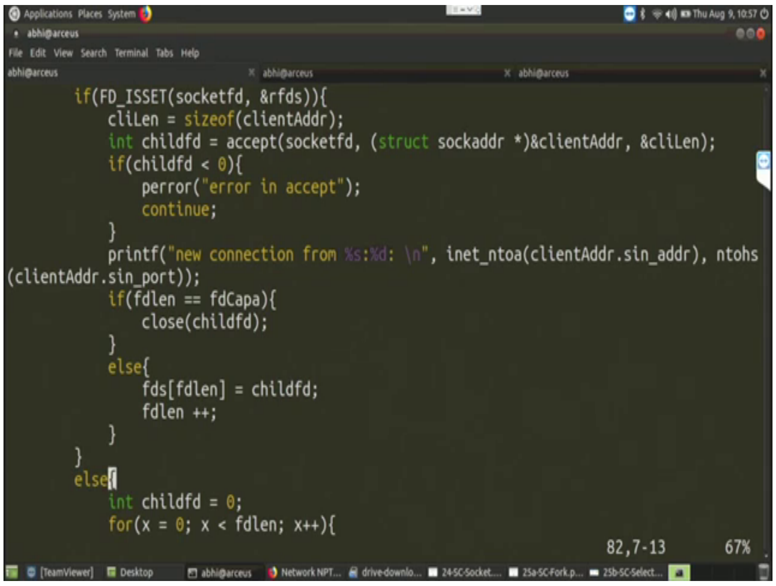
(Refer Slide Time: 38:52)



```
                continue;
            }
            printf("new connection from %s:%d: \n", inet_ntoa(clientAddr.sin_addr), ntohs
(clientAddr.sin_port));
            if(fdlen == fdCapa){
                close(childfd);
            }
            else{
                fds[fdlen] = childfd;
                fdlen ++;
            }
        }
        else{
            int childfd = 0;
            for(x = 0; x < fdlen; x++){
                if(!FD_ISSET(fds[x], &rfds))
                    continue;
                childfd = fds[x];
                if ((recvLen = recv(childfd, recvBuf, bufLen, 0)) < 0){
                    perror("Error in recving");
                                                                85,16-28      72%
```

Then in this fd set you loop over all the file descriptor that you have been added.

(Refer Slide Time: 38:59)

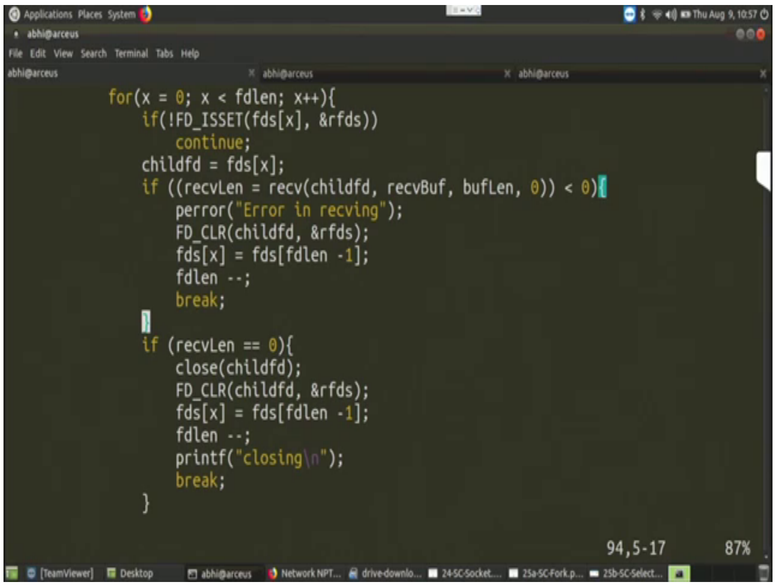


```
        if(FD_ISSET(socketfd, &rfds)){
            cliLen = sizeof(clientAddr);
            int childfd = accept(socketfd, (struct sockaddr *)&clientAddr, &cliLen);
            if(childfd < 0){
                perror("error in accept");
                continue;
            }
            printf("new connection from %s:%d: \n", inet_ntoa(clientAddr.sin_addr), ntohs
(clientAddr.sin_port));
            if(fdlen == fdCapa){
                close(childfd);
            }
            else{
                fds[fdlen] = childfd;
                fdlen ++;
            }
        }
        else{
            int childfd = 0;
            for(x = 0; x < fdlen; x++){
                                                                82,7-13       67%
```

So, here earlier what we are doing, that whenever some new connection is coming using that select call we are checking that some new connection has arrived. So, if that is a new connection, we add that new connection to the set of file descriptor that we have otherwise some file descriptor from the existing file descriptor is ready to use.

So, what we do that we check we look over the available file descriptors that we have and the file descriptor that is currently ready on top of that we make the receive call and get the data and then return back the data there.

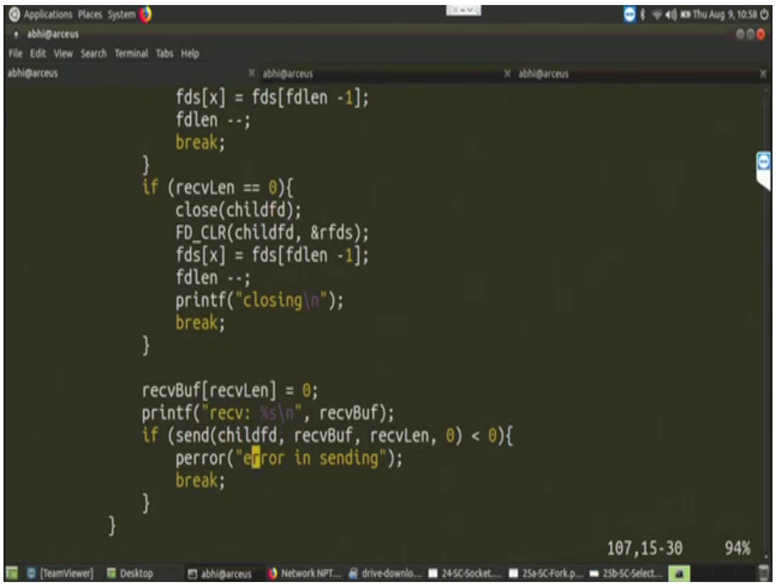
(Refer Slide Time: 39:36)



```
        for(x = 0; x < fdlen; x++){
            if(!FD_ISSET(fds[x], &rfds))
                continue;
            childfd = fds[x];
            if ((recvLen = recv(childfd, recvBuf, bufLen, 0)) < 0){
                perror("Error in recving");
                FD_CLR(childfd, &rfds);
                fds[x] = fds[fdlen -1];
                fdlen --;
                break;
            }
            if (recvLen == 0){
                close(childfd);
                FD_CLR(childfd, &rfds);
                fds[x] = fds[fdlen -1];
                fdlen --;
                printf("closing\n");
                break;
            }
                                                        94,5-17       87%
```

So, if that received len is equal to equal to 0; that means, you are not receiving any data. So, you close that file descriptor and clear it from your file descriptor set.

(Refer Slide Time: 39:48)

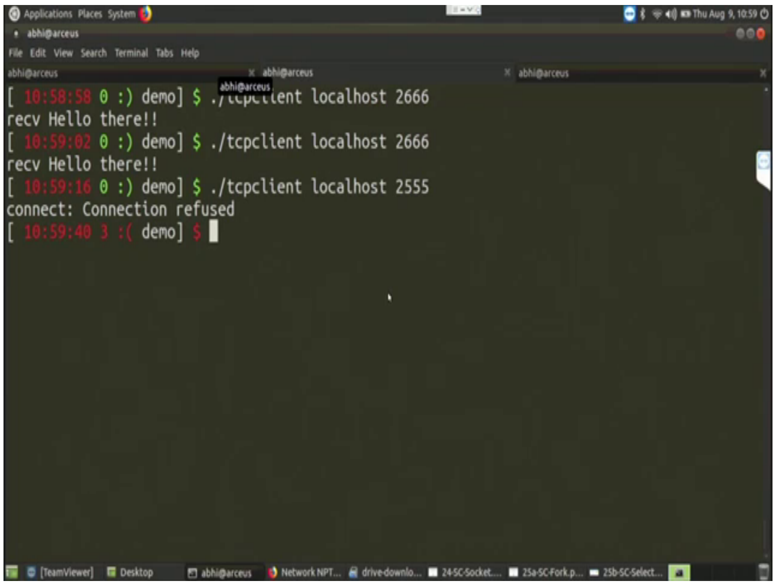


```
                fds[x] = fds[fdlen -1];
                fdlen --;
                break;
            }
            if (recvLen == 0){
                close(childfd);
                FD_CLR(childfd, &rfds);
                fds[x] = fds[fdlen -1];
                fdlen --;
                printf("closing\n");
                break;
            }

            recvBuf[recvLen] = 0;
            printf("recv: %s\n", recvBuf);
            if (send(childfd, recvBuf, recvLen, 0) < 0){
                perror("error in sending");
                break;
            }
        }
                                                        107,15-30     94%
```

So, that is this entire idea and after that you save the data in our normal procedure. So, the idea is that whenever you are getting a new connection, we are adding that new connection in the set of file descriptors and then we are looping over that file descriptor to find out, which one have certain data and if someone has certain data, we are sending back that data.

Now, let us run this code first compile it, again we are going to use the same client implementation that we have, the client implementation we are not making any change. So, now, my server runs over select and giving a code 2 6 6 6, it is running from the client side.

(Refer Slide Time: 40:51)



```
[ 10:58:58 0 :) demo] $ ./tcpclient localhost 2666
recv Hello there!!
[ 10:59:02 0 :) demo] $ ./tcpclient localhost 2666
recv Hello there!!
[ 10:59:16 0 :) demo] $ ./tcpclient localhost 2555
connect: Connection refused
[ 10:59:40 3 :( demo] $
```

You send the data, you send something it has received that, then once the task is done it has closed that particular connection.

Again you can execute it you see it is from a again a different port, the same thing we are keep on observing it has received the data decode it back to the client and closing that connection that particular connection. And well one thing just I wanted to show that if you try to connect to a port, where the server is not running. So, the server is now running at 4 2 6 6 6, now if you try to connect it, it will get a connection receives message from the connect call, because none of the server is currently running and the port 2 5 5 5.

So, this is all about our discussion on tcp server, I was quite fast in describing the things with the assumption that you have a basic knowledge of C programming and operating system, we will share all the codes with you and I will suggest you to browse the code and look into the tutorials that we have shared.

And if you have any doubt or anything feel free to post the questions in the forum. So, with this particular socket programming, you can develop your own applications you can even implement the chat server application that we are talking about. So, I will suggest you to look into that and implement multiple such applications on your own with the help of this different variance of socket programming.

So, thank you all for attending this class; from the next class onwards, we will go for again the theoretical aspects of the tcp IP protocol stack so.

Thank you all for attending this class.