

**Department of Information Technology**

**LAB MANUAL**

**IT 1662 – COMPUTER NETWORKS LAB**

**(B. Tech VI Semester)**



**SMU** SIKKIM  
MANIPAL  
UNIVERSITY

Established under Govt. of Sikkim, Act 9 of 1995, recognised under 2(f) of the UGC Act, 1956

**Sikkim Manipal Institute of Technology**  
**Majitar, Rangpo, East Sikkim**

## **Vision**

To become a front-runner in preparing students equipped with attributes like efficient problem solvers, innovators, researchers and entrepreneurs, and in making them academia and industry ready.

## **Mission**

- ❖ To offer high-quality undergraduate program namely Bachelor of Technology in Information Technology incorporating latest developments in the area of Information and Communication Technology into the curriculum, along with doctoral program.
- ❖ To produce competent technocrats and researchers in order to meet the human resource needs of industries, government sectors, and the society as a whole.
- ❖ To offer a conducive academic environment to the students where creativity flourishes.
- ❖ To teach students how to think in order to become an efficient problem solver.

## **Program Specific Outcomes (PSO)**

**PSO1:** Students will be equipped with skills and competency to work and deliver quality results in industries, research organizations, and in the professional world as a whole.

**PSO2:** Students will be competent to develop software solutions for different relevant problems in the world of Information and Communication, by applying knowledge of various domains such as Database Technologies, Information Systems, Network Technologies, Cloud Technologies, Algorithms, Artificial Intelligence, Machine Learning, Cyber Security, Image Processing, and associated interdisciplinary subjects.

## **Program Educational Objectives (PEO)**

1. To improve the employability of students and prepare them for better employment.
2. To motivate students for higher education and research.
3. To encourage students for becoming future entrepreneur and provide them an ecosystem for the same.
4. To inculcate professional ethics in students and prepare them to become professionals driven by value System.

**Sikkim Manipal Institute of Technology**  
**Sikkim Manipal University-737136**  
**Department of Information Technology**

**CERTIFICATE**

This is to certify that Ms./Mr.....  
Reg. No.: ..... Section: ..... Roll No.: .....has satisfactorily  
completed the lab exercises prescribed for **COMPUTER NETWORKS LAB (IT 1662)**  
of Third Year B.Tech. Degree in Information Technology at SMIT, in the  
Academic Year.....

Date: .....

Signature:

Faculty in Charge

Signature:

Head of the Department

## LIST OF PROGRAMS

Sr. No.	Program Description	Date	Page No.	Remarks	Signature
1.	a) Write a program to create a child process and print the process ids of the parent and child processes. b) Write a program to create a child process and send a message from parent to child using PIPE.				
2	a) Write a program, to demonstrate two way communications between parent and child using PIPE. b) Write a program, to create an orphan process and print the process ids.				
3.	Write a reader and a writer program to send the message from writer to reader using message queue .				
4.	Write a reader cum writer programs to demonstrate two way communications between them. Write a reader cum writer programs to send a message from writer to reader and print the following: i) Number of bytes in the queue. ii) Number of messages in the queue. iii) PID of the last message sent. iv) PID of the last message received.				
5.	Write a reader and a writer program to send the message from writer to reader using shared memory.				
6.	Write a client program and server program using TCP socket API where client machine will send message and server machine will receive it. [One-way communication]				
7.	Write a client and server program using TCP socket API to implement an echo server.				
8.	Write a client program and a server program using TCP socket API where client machine will send message and the server machine will receive it and vice versa. [Bidirectional communication]				
9.	Write a client and server program using UDP socket API to implement an echo server.				
10.	Write a client program and a server program using				

	UDP socket where client machine will send a message and server machine will receive it. [One way communication]				
11.	Write a client program and a server program using UDP socket where client machine will send a message and the server machine will receive it and vice versa. [Bidirectional communication]				
12.	Write a client program and a server program using TCP/UDP socket API where client machine will send two numbers to server machine and then server machine will perform the addition of two numbers. The result will be send back to client machine by the server machine. [Bidirectional communication]				
13.	Write a client program and a server program using TCP/UDP socket API where client machine will send any string to server machine and then server machine will send back the reverse of the string to client machine. [Bidirectional communication]				
14.	Write a client program and a server program using TCP/UDP socket API where client machine will ask for any mathematical or string related operation to server machine and the server machine will reply to client machine with the solution. [Bidirectional communication]				

## PROGRAM OUTCOMES (PO)

<b>PO 1</b>	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
<b>PO 2</b>	Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
<b>PO 3</b>	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
<b>PO 4</b>	Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
<b>PO 5</b>	Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
<b>PO 6</b>	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
<b>PO 7</b>	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
<b>PO 8</b>	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice
<b>PO 9</b>	Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
<b>PO 10</b>	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
<b>PO 11</b>	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
<b>PO 12</b>	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

## Course Articulation Matrix (CAM)

SUB. CODE	STATEMENT	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO 10	PO 11	PO 12	PSO1	PSO2
CO1	To get acquainted with Linux /Unix system internals for Inter Process Communication.	1	-	2	1	1	-	-	-	1	1	1	1	2	2
CO2	To understand the concept of Inter Process Communication and client/server architecture in application development.	1	-	3	2	3	-	-	-	2	1	1	1	2	3
CO3	To understand how to use TCP and UDP based socket APIs and their differences.	1	2	3	2	3	-	-	-	2	1	1	1	2	3
CO4	To design reliable servers applications using both TCP and UDP sockets	1	2	2	2	1	-	-	-	1	1	-	1	-	-
	<b>IT1662</b>	<b>1.00</b>	<b>2.00</b>	<b>2.50</b>	<b>1.75</b>					<b>1.50</b>	<b>1.00</b>	<b>1.00</b>	<b>1.00</b>	<b>2.00</b>	<b>2.67</b>

## Internal Rubrics for Laboratory

	CO			
LAB NUMBERS	CO1	CO2	CO3	CO4
Lab 1+2	1	1	1	1
Lab 3+4	1	1	1	1
Lab 5+6	1	1	1	1
Lab 7+8	1	1	1	1
Lab 9+10	1	1	1	1
Lab 11+12	1	1	1	1

## Course Objectives

1. To get acquainted with Linux /Unix system internals for Inter Process Communication
2. To write, execute and debug C programs based on IPC and Socket API.
3. To understand the concept of Inter Process Communication and client/server architecture in application development.
4. To understand how to use TCP and UDP based socket APIs and their differences.
5. To design reliable server applications using both TCP and UDP sockets

## Course Outcomes (CO):

On completion of this course students will be able -

**CO1:** To get acquainted with Linux /Unix system internals for Inter Process Communication.

**CO2:** To understand the concept of Inter Process Communication and client/server architecture in application development.

**CO3:** To understand how to use TCP and UDP based socket APIs and their differences.

**CO4:** To design reliable server applications using both TCP and UDP sockets

## Evaluation Plan

- ✓ Internal Assessment Marks: 60 Marks
- ✓ Continuous evaluation component (Program Execution+ Lab File+Viva) for each experiment:10 marks
- ✓ The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce.
- ✓ Total marks of 12 experiments ( $12 \times 10$ ) scaled down to 60.
- ✓ End semester assessment of 3-hour duration: 40 Marks
- ✓ At least 12 laboratory classes to be conducted.

Internal Assessment(Continuous Lab Evaluation)						Total Marks
Algorithm/flow chart/logic diagram	Implementation	Output	Viva	File	Total marks/ Lab	
3 marks	3 marks	1 marks	2 marks	1 marks	10	60
120 marks is scaled down to 60						
End Semester Assessment						40



## Rubrics

Objective	High Proficiency (Marks)	Mid Proficiency (Marks)	Low Proficiency (Marks)	No Proficiency (Marks)
Algorithm/flow chart/logic diagram	3	$2 \leq x < 3$	$1 \leq x < 2$	0
Implementation	3	$2 \leq x < 3$	$1 \leq x < 2$	0
Output	1	$1 \leq x < 0.5$	$0.5 \leq x < 0$	0
Viva	2	$2 \leq x < 1$	$1 \leq x < 0$	0
File	1	$1 \leq x < 0.5$	$0.5 \leq x < 0$	0

## **INSTRUCTIONS TO THE STUDENTS**

### **Pre-Lab Session Instructions**

1. Students should carry the Lab Manual and the required stationery to every lab session.
2. Be in time and follow the institution dress code.
3. Must Sign in the log register.
4. Make sure to occupy the allotted seat and mark the attendance.
5. Adhere to the rules and maintain the decorum.

### **In-Lab Session Instructions**

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record.
- Prescribed textbooks and class notes can be kept ready for reference if required.

### **General Instructions for the exercises in Lab**

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Programs should perform input validation (Data type, range error, etc.) and five appropriate error messages and suggest corrective actions.
  - Comments should be used to give the statement of the problem and every member function should indicate the purpose of the member function, inputs and outputs.
  - Statements within the program should be properly indented.
  - Use meaningful names for variables, classes, interfaces, packages and methods.
  - Make use of constant and static members wherever needed.
- Plagiarism (copy from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
  - Solved exercise
  - Lab exercises- to be completed during lab hours
  - Additional Exercises – to be completed outside the lab or in the lab to enhance the skill.
- In case a student misses a lab class, he/she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and/or combinations of the questions.

### **The students should not**

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

### **Text Book:**

1. Unix Network Programming, W. Richard Stevens, PHI

### **Reference Books:**

1. Unix Network Programming, Volume 1: The Sockets Networking API, W. Stevens, Bill Fenner, Andrew Rudoff
2. UNIX Network Programming, Volume2: The Inter Process Communication, W. Richard Stevens.

# Sample Example Programs:

## Inter Process Communication:

**Prog #1. Write a program to create a parent and child process using the system call fork().**

### The fork () System Call

System call `fork()` is used to create processes. It takes no arguments and returns a process ID. The purpose of `fork()` is to create a *new* process, which becomes the *child* process of the caller. After a new child process is created, *both* processes will execute the next instruction following the `fork()` system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of `fork()`: If `fork()` returns a negative value, the creation of a child process was unsuccessful. `fork()` returns a zero to the newly created child process. `fork()` returns a positive value, the *process ID* of the child process, to the parent. The returned process ID is of type `pid_t` defined in `sys/types.h`. Normally, the process ID is an integer. Moreover, a process can use function `getpid()` to retrieve the process ID assigned to this process. Therefore, after the system call to `fork()`, a simple test can tell which process is the child. Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

Let us take an example to make the above points clear. This example does not distinguish parent and the child processes.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#define MAX_COUNT 200
#define BUF_SIZE 100

void main(void)
{
    pid_t pid;
    int i;
    char buf[BUF_SIZE];

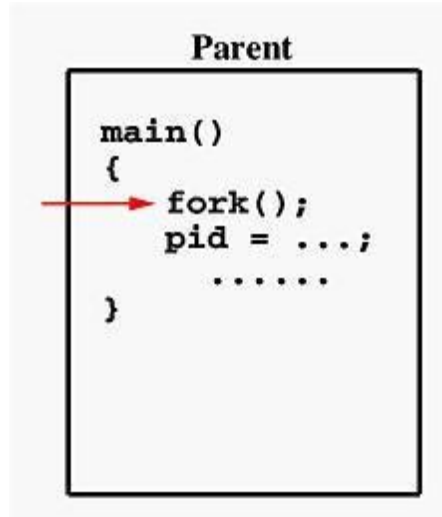
    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
```

```

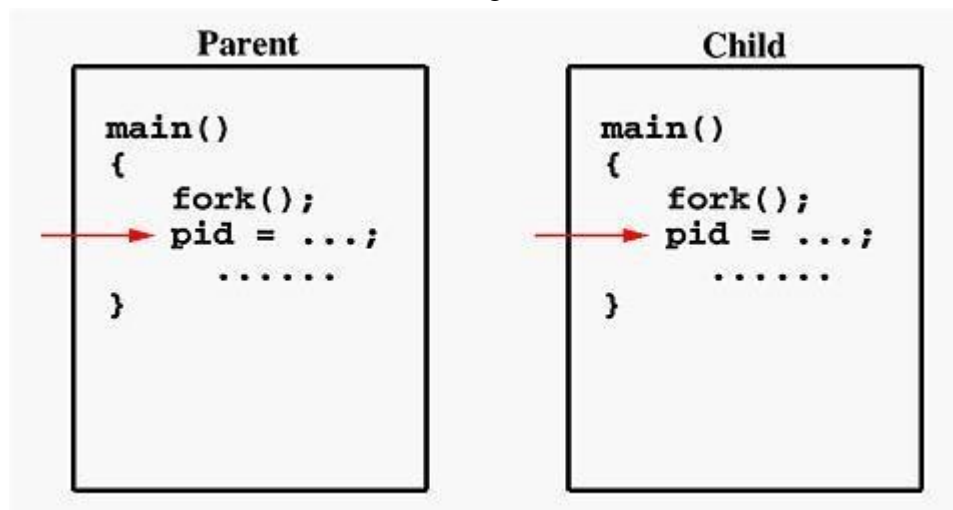
        write(1, buf, strlen(buf));
    }
}

```

Suppose the above program executes up to the point of the call to `fork()` (marked in red color):



If the call to `fork()` is executed successfully, Unix will make two identical copies of address spaces, one for the parent and the other for the child. Both processes will start their execution at the next statement following the `fork()` call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call `fork()`. Since both processes have identical but separate address spaces, those variables initialized before the `fork()` call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by `fork()` calls will not be affected even though they have identical variable names. What is the reason of using `write` rather than `printf`? It is because `printf()` is "buffered," meaning `printf()` will group the output of a process together. While buffering the output for the parent process, the child may also use `printf` to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the

right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" write.

If you run this program, you might see the following on the screen:

```
.....
This line is from pid 3456, value 13
This line is from pid 3456, value 14
```

```
.....
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
```

```
.....
This line is from pid 3456, value 21
This line is from pid 3456, value 22
```

.....

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

**Prog #2. Write a program, which distinguishes the parent process from the child process.**

```
#include <stdio.h>
#include <sys/types.h>

#define MAX_COUNT 200

void ChildProcess(void);          /* child process prototype */
void ParentProcess(void);        /* parent process prototype */

void main(void)
{
    pid_t pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void ChildProcess(void)
{
    int i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf(" This line is from child, value = %d\n", i);
    printf(" *** Child process is done ***\n");
}
```

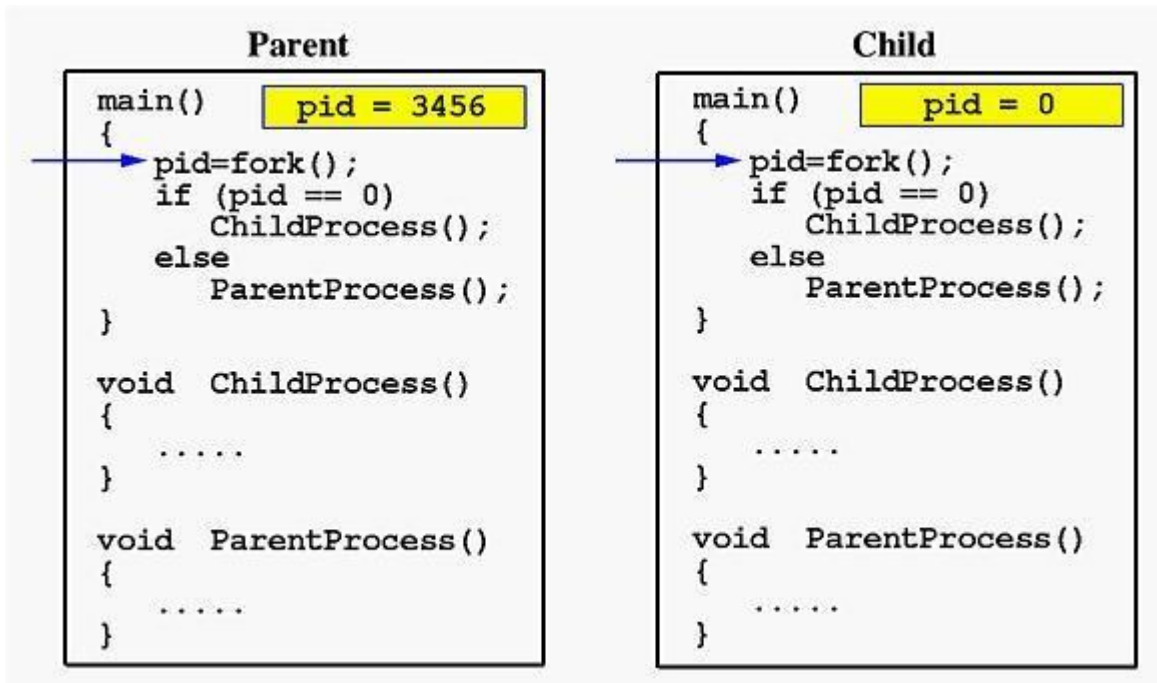
```

void ParentProcess(void)
{
    int i;

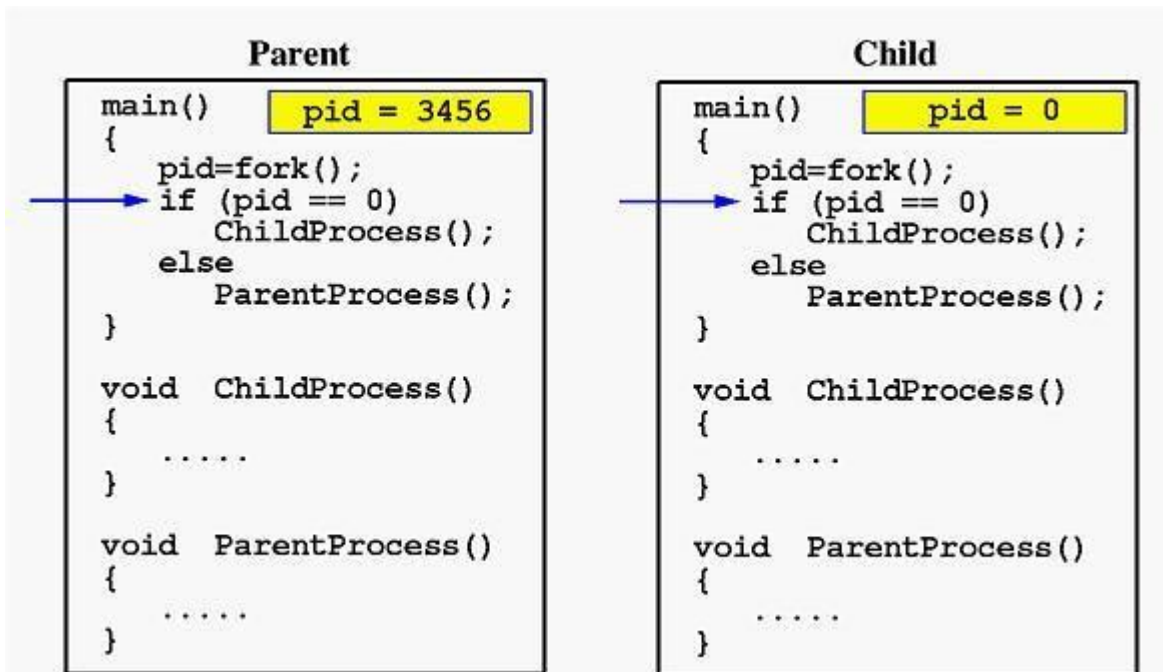
    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}

```

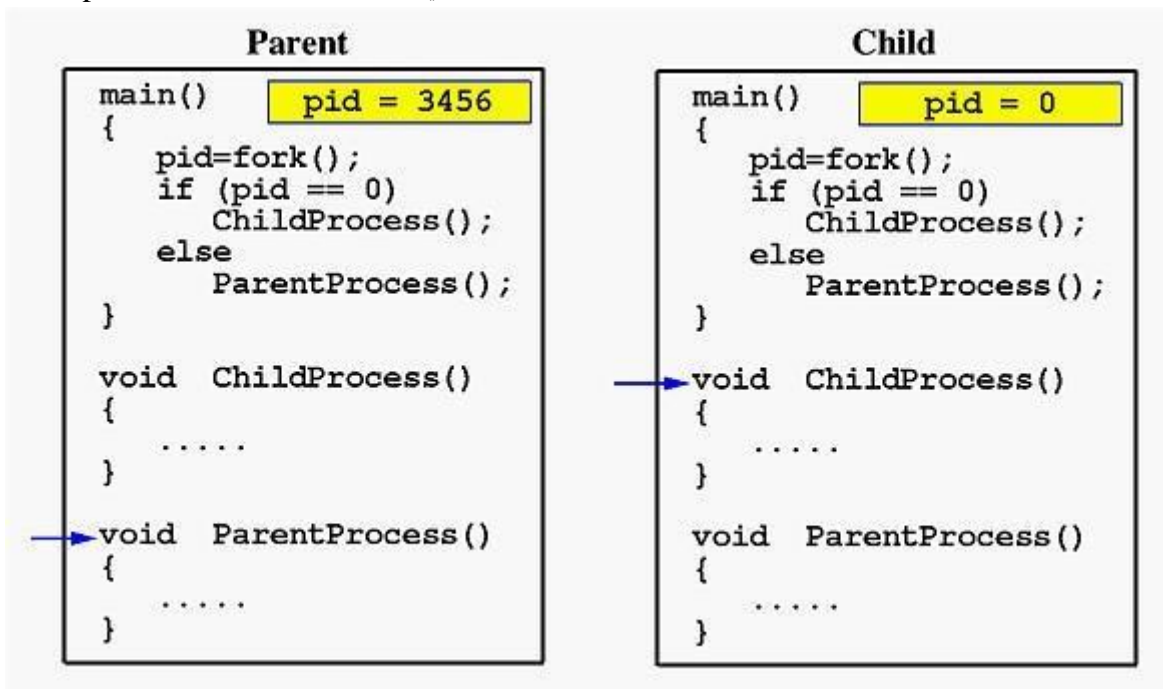
In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable *i*. For simplicity, `printf()` is used. When the main program executes `fork()`, an identical copy of its address space, including the program and all data, is created. System call `fork()` returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable `pid`. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



Now both programs (*i.e.*, the parent and child) will execute independent of each other starting at the next statement:



In the parent, since pid is non-zero, it calls function ParentProcess(). On the other hand, the child has a zero pid and calls ChildProcess() as shown below:



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, the value of MAX\_COUNT should be large enough so that both processes will run for at least two or more time quanta. If the value of MAX\_COUNT is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.



### **Prog #3. Write a program, to demonstrate how pipes are used in Linux Processes**

#### **Using Pipes in Linux Processes**

Pipes can be used in threads and processes. The program below demonstrates how pipes can be used in processes. A new process can be created using the system call `fork()`. It returns two different values to the child and parent. The value 0 is returned to the child (new) process and the PID (Process ID) of the child is returned to the parent process. This is used to distinguish between the two processes. In the program given below, the child process waits for the user input and once an input is entered, it writes into the pipe. And the parent process reads from the pipe.

A sample program to demonstrate how pipes are used in Linux Processes

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define MSGLEN 64

int main(){
    int  fd[2];
    pid_t pid;
    int  result;

    //Creating a pipe
    result = pipe (fd);
    if (result < 0) {
        //failure in creating a pipe
        perror("pipe");
        exit (1);
    }

    //Creating a child process
    pid = fork();
    if (pid < 0) {
        //failure in creating a child
        perror ("fork");
        exit(2);
    }

    if (pid == 0) {
        //Child process
        char message[MSGLEN];

        while(1) {
```

```

        //Clearing the message
        memset (message, 0, sizeof(message));
        printf ("Enter a message: ");
        scanf ("%s",message);

        //Writing message to the pipe
        write(fd[1], message, strlen(message));
    }
    exit (0);
}
else {
    //Parent Process
    char message[MSGLEN];

    while (1) {
        //Clearing the message buffer
        memset (message, 0, sizeof(message));

        //Reading message from the pipe
        read (fd[0], message, sizeof(message));
        printf("Message entered %s\n",message);
    }

    exit(0);
}
}
}

```

Pipes are one of the most commonly used mechanisms for IPC (inter process communication). IPC is the mechanism by which two or more processes communicate with each other. The commonly used IPC techniques include shared memory, message queues, pipes and FIFOs (and sometimes even files). Pipes are helpful in the unidirectional form of communication between processes. The `pipe()` creates a pipe.

### **Syntax:**

```
int pipe(fd[2])
```

`pipe()` creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by `fd`. `fd[0]` is for reading, `fd[1]` is for writing. Let's see a simple program how to use piping.

```

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

```

```
#define MSG_LEN 64
```

```

int main(){

    int    result;
    int    fd[2];
    char    *message="Linux World!!!";
    char    recvd_msg[MSG_LEN];

    result = pipe (fd); //Creating a pipe fd[0] is for reading and fd[1] is for writing

    if (result < 0) {
        perror("pipe ");
        exit(1);
    }

    //writing the message into the pipe

    result=write(fd[1],message,strlen(message));
    if (result < 0) {
        perror("write");
        exit(2);
    }

    //Reading the message from the pipe

    result=read (fd[0],recvd_msg,MSG_LEN);

    if (result < 0) {
        perror("read");
        exit(3);
    }

    printf("%s\n",recvd_msg);
    return 0;
}

```

As seen above , the pipe() system call creates a pipe. Now normal system calls like read() or write() can be used for reading and writing data to the pipe. As you have noticed the fd[0] can only be used for reading from the pipe and does not permit writing. Similarly fd[1] can only be used to perform writing to the pipe. You are writing into the pipe and reading from it. That one can do even from a file. Now let's see another scenario. If we add more writes in the above program before reading anything and then start reading, you will find that the first read call will fetch you the first written message and so on. So pipe() is helpful in Implementing a QUEUE strategy(First in First out) FIFO: the message which was written first will be available for the first read, then the message which came second, for the second read.

Note: The first read itself can get all of the messages written by different write, if it specifies a large size in the size field (a size which can span the size of one or more writes ) as demonstrated here

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

#define MSG_LEN 64

int main(){
    int    result;
    int    fd[2];
    char    message[MSG_LEN];
    char    recvd_msg[MSG_LEN];

    result = pipe (fd); //Creating a pipe//fd[0] is for reading and fd[1] is for writing

    if (result < 0) {
        perror("pipe ");
        exit(1);
    }

    strncpy(message,"Linux World!! ",MSG_LEN);

    result=write(fd[1],message,strlen(message));
    if (result < 0) {
        perror("write");
        exit(2);
    }
    strncpy(message,"Understanding ",MSG_LEN);

    result=write(fd[1],message,strlen(message));
    if (result < 0) {
        perror("write");
        exit(2);
    }

    strncpy(message,"Concepts of ",MSG_LEN);

    result=write(fd[1],message,strlen(message));
    if (result < 0) {
        perror("write");
        exit(2);
    }

    strncpy(message,"Piping ", MSG_LEN);
```

```

result=write(fd[1],message,strlen(message));

if (result < 0) {
    perror("write");
    exit(2);
}

result=read (fd[0],recvd_msg,MSG_LEN);
if (result < 0) {
    perror("read");
    exit(3);
}

printf("%s\n",recvd_msg);
return 0;

}

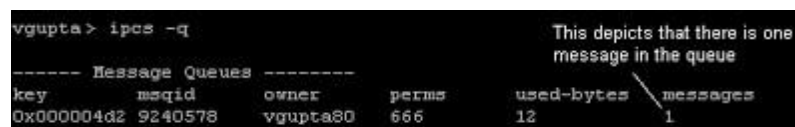
```

**Prog #4. Write a program, to implement message queue and demonstrate it.**

### IPC Message Queue Implementation in C

A simple implementation of IPC Message Queues  
 IPC\_msgq\_send.c adds the message on the message queue  
 IPC\_msgq\_rcv.c removes the message from the message queue.

To use this program first compile and run IPC\_msgq\_send.c to add a message to the message queue. To see the Message Queue type `ipcs -q` on your Unix/Linux Terminal.



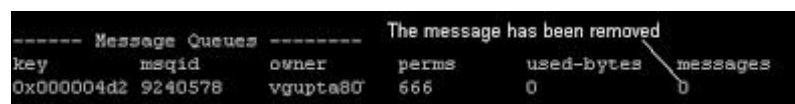
```

vgupta> ipcs -q
----- Message Queues -----
key      msqid    owner    perms    used-bytes  messages
0x000004d2 9240578  vgupta80  666      12           1

```

This depicts that there is one message in the queue

Now compile and run IPC\_msgq\_rcv.c to read the message from the Message Queue. To see that you have read the message again use `ipcs -q`



```

----- Message Queues -----
key      msqid    owner    perms    used-bytes  messages
0x000004d2 9240578  vgupta80  666      0           0

```

The message has been removed

//IPC\_msgq\_send.c

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

```

```

#include <string.h>
#include <stdlib.h>
#define MAXSIZE      128

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

struct msgbuf
{
    long    mtype;
    char    mtext[MAXSIZE];
};

main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    struct msgbuf sbuf;
    size_t buflen;

    key = 1234;

    if ((msqid = msgget(key, msgflg )) < 0)    //Get the message
queue ID for the given key
        die("msgget");

    //Message Type
    sbuf.mtype = 1;

    printf("Enter a message to add to message queue : ");
    scanf("%[^\n]", sbuf.mtext);
    getchar();

    buflen = strlen(sbuf.mtext) + 1 ;

    if (msgsnd(msqid, &sbuf, buflen, IPC_NOWAIT) < 0)
    {
        printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext,
buflen);
        die("msgsnd");
    }

    else

```

```

        printf("Message Sent\n");

    exit(0);
}

//IPC_msgq_rcv.c

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE      128

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

typedef struct msgbuf
{
    long      mtype;
    char      mtext[MAXSIZE];
} ;

main()
{
    int msqid;
    key_t key;
    struct msgbuf rcvbuffer;

    key = 1234;

    if ((msqid = msgget(key, 0666)) < 0)
        die("msgget()");

    //Receive an answer of message type 1.
    if (msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0) < 0)
        die("msgrcv");

    printf("%s\n", rcvbuffer.mtext);
    exit(0)
}

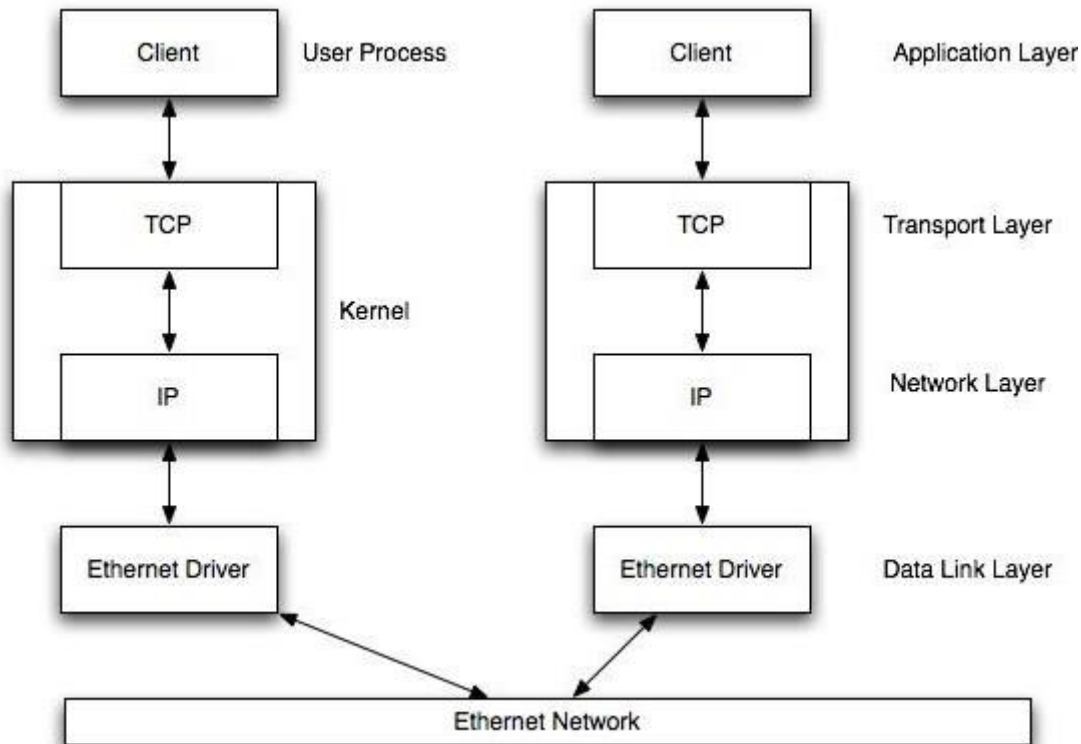
```

## Socket Programming:

How do we build Internet applications? In this lecture, we will discuss the socket API and support for TCP and UDP communications between end hosts. Socket programming is the key API for programming distributed applications on the Internet.

BTW, Kurose/Ross only cover Java socket programming and not C socket programming discussed below.

The scenario of the client and the server on the same local network (usually called LAN, Local Area Network) is shown in Figure



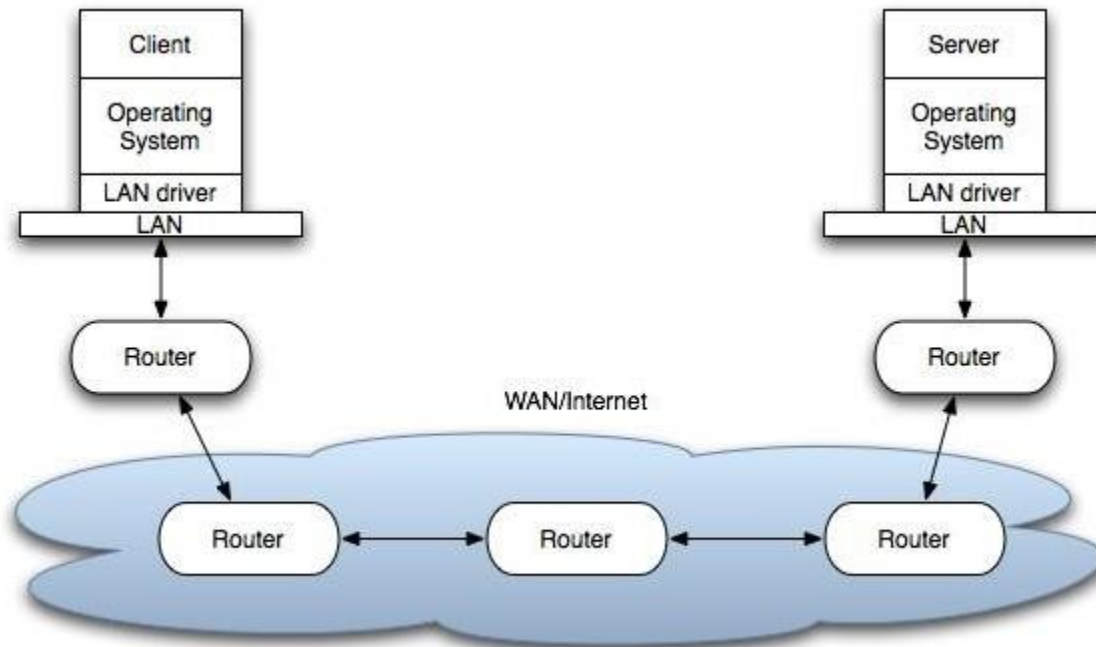
Client and server on the same Ethernet communicating using TCP/IP.



---

The client and the server may be in different LANs, with both LANs connected to a Wide Area Network (WAN) by means of *routers*. The largest WAN is the Internet, but companies may have their own WANs. This scenario is depicted in Figure 2.

---



Client and server on different LANs connected through WAN/Internet.

---

The flow of information between the client and the server goes down the protocol stack on one side, then across the network and then up the protocol stack on the other side.

### User Datagram Protocol (UDP)

UDP is a simple transport-layer protocol. The application writes a message to a UDP socket, which is then encapsulated in a UDP datagram, which is further encapsulated in an IP datagram, which is sent to the destination.

There is no guarantee that a UDP will reach the destination, that the order of the datagrams will be preserved across the network or that datagrams arrive only once.

The problem of UDP is its lack of reliability: if a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not automatically retransmitted.

Each UDP datagram is characterized by a length. The length of a datagram is passed to the receiving application along with the data.

No connection is established between the client and the server and, for this reason, we say that UDP provides a *connection-less service*.

It is described in RFC 768.

## **Transmission Control Protocol (TCP)**

TCP provides a *connection oriented service*, since it is based on connections between clients and servers.

TCP provides reliability. When a TCP client send data to the server, it requires an acknowledgement in return. If an acknowledgement is not received, TCP automatically retransmit the data and waits for a longer period of time.

We have mentioned that UDP datagrams are characterized by a length. TCP is instead a byte-stream protocol, without any boundaries at all.

TCP is described in RFC 793, RFC 1323, RFC 2581 and RFC 3390.

## **Socket addresses**

IPv4 socket address structure is named `sockaddr_in` and is defined by including the `<netinet/in.h>` header.

The POSIX definition is the following:

```
struct in_addr{
in_addr_t s_addr;          /*32 bit IPv4 network byte ordered address*/
};

struct sockaddr_in {
    uint8_t sin_len; /* length of structure (16)*/
    sa_family_t sin_family; /* AF_INET*/
    in_port_t sin_port; /* 16 bit TCP or UDP port number */
    struct in_addr sin_addr; /* 32 bit IPv4 address*/
    char sin_zero[8]; /* not used but always set to zero */
};
```

The `uint8_t` datatype is unsigned 8-bit integer.

## Generic Socket Address Structure

A socket address structure is always passed by reference as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.

A problem arises in declaring the type of pointer that is passed. With ANSI C, the solution is to use `void *` (the generic pointer type). But the socket functions predate the definition of ANSI C and the solution chosen was to define a generic socket address as follows:

```
struct sockaddr {
    uint8_t sa_len;
    sa_family_t sa_family; /* address family: AD_XXX value */
    char sa_data[14];
};
```

## Host Byte Order and Network Byte Order Conversion

There are two ways to store two bytes in memory: with the lower-order byte at the starting address (*little-endian* byte order) or with the high-order byte at the starting address (*big-endian* byte order). We call them collectively *host byte order*. For example, an Intel processor stores the 32-bit integer as four consecutive bytes in memory in the order 1-2-3-4, where 1 is the most significant byte. IBM PowerPC processors would store the integer in the byte order 4-3-2-1.

Networking protocols such as TCP are based on a specific *network byte order*. The Internet protocols use big-endian byte ordering.

### The `htons()`, `htonl()`, `ntohs()`, and `ntohl()` Functions

The following functions are used for the conversion:

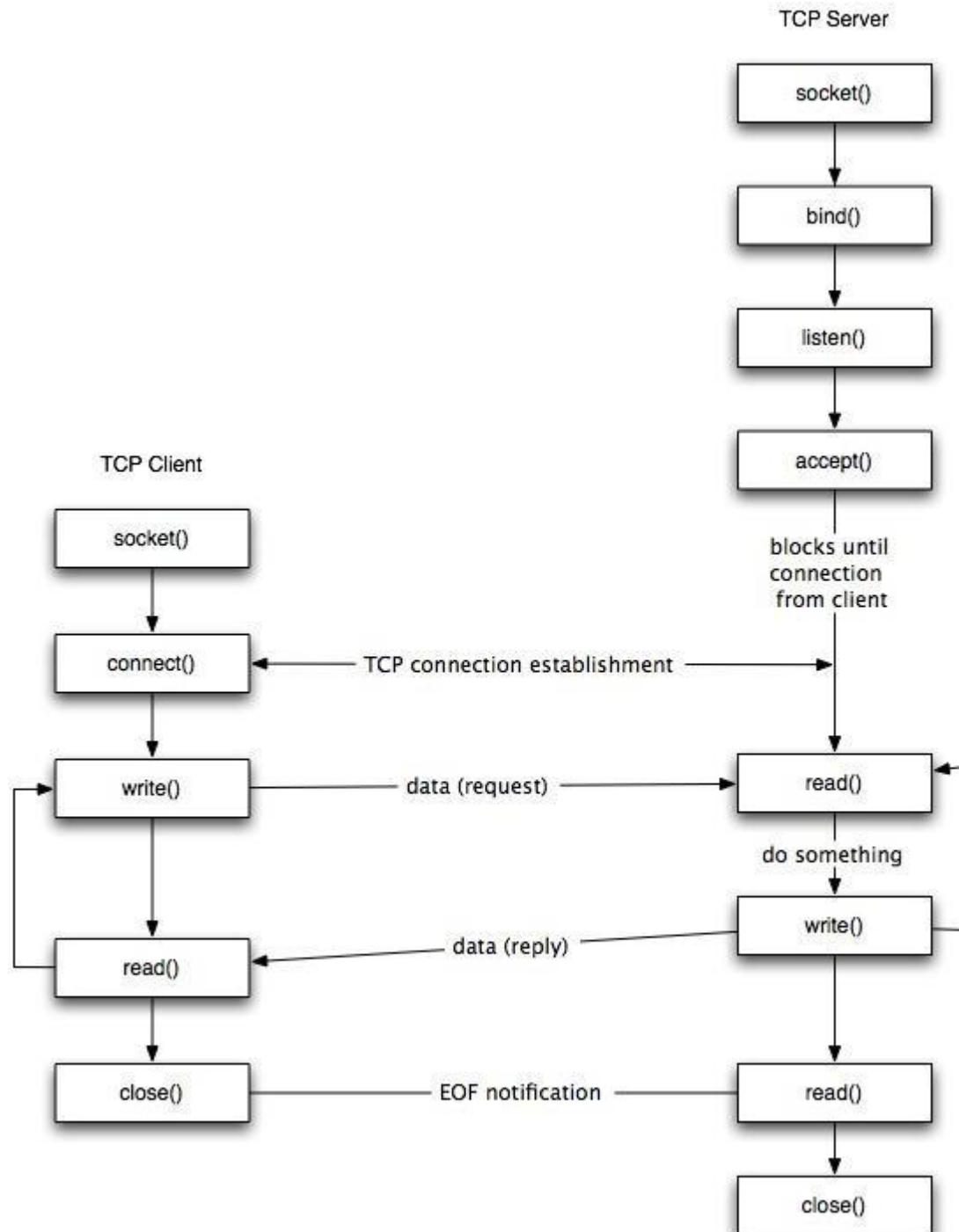
```
#include <netinet/in.h>

uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
```

The first two return the value in network byte order (16 and 32 bit, respectively). The latter return the value in host byte order (16 and 32 bit, respectively).

## TCP Socket API

The sequence of function calls for the client and a server participating in a TCP connection is presented in Figure



TCP client-server.

As shown in the figure, the steps for establishing a TCP socket on the client side are the following:

- Create a socket using the `socket()` function;
- Connect the socket to the address of the server using the `connect()` function;
- Send and receive data by means of the `read()` and `write()` functions.
- Close the connection by means of the `close()` function.

The steps involved in establishing a TCP socket on the server side are as follows:

- Create a socket with the `socket()` function;
- Bind the socket to an address using the `bind()` function;
- Listen for connections with the `listen()` function;
- Accept a connection with the `accept()` function system call. This call typically blocks until a client connects with the server.
- Send and receive data by means of `send()` and `receive()`.
- Close the connection by means of the `close()` function.

### **The socket() Function**

The first step is to call the `socket` function, specifying the type of communication protocol (TCP based on IPv4, TCP based on IPv6, UDP).

The function is defined as follows:

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

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

where `family` specifies the protocol family (`AF_INET` for the IPv4 protocols), `type` is a constant described the type of socket (`SOCK_STREAM` for stream sockets and `SOCK_DGRAM` for datagram sockets).

The function returns a non-negative integer number, similar to a file descriptor, that we define *socket descriptor* or -1 on error.

### **The connect() Function**

The `connect()` function is used by a TCP client to establish a connection with a TCP server/

The function is defined as follows:

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

```
int connect (int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

where `sockfd` is the socket descriptor returned by the `socket` function.

The function returns 0 if it succeeds in establishing a connection (i.e., successful TCP three-way handshake, -1 otherwise.

The client does not have to call `bind()` in Section before calling this function: the kernel will choose both an ephemeral port and the source IP if necessary.

### **The `bind()` Function**

The `bind()` assigns a local protocol address to a socket. With the Internet protocols, the address is the combination of an IPv4 or IPv6 address (32-bit or 128-bit) address along with a 16 bit TCP port number.

The function is defined as follows:

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

```
int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

where `sockfd` is the socket descriptor, `servaddr` is a pointer to a protocol-specific address and `addrlen` is the size of the address structure.

`bind()` returns 0 if it succeeds, -1 on error.

This use of the generic socket address `sockaddr` requires that any calls to these functions must cast the pointer to the protocol-specific address structure. For example for an IPv4 socket structure:

```
struct sockaddr_in serv; /* IPv4 socket address structure */
```

```
bind(sockfd, (struct sockaddr*) &serv, sizeof(serv))
```

A process can bind a specific IP address to its socket: for a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the sockets. For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address.

Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP socket is connected, based on the outgoing interface that is used. If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the incoming packets as the server's source address.

`bind()` allows to specify the IP address, the port, both or neither.

The table below summarizes the combinations for IPv4.

IP Address	IP Port	Result
INADDR_ANY	0	Kernel chooses IP address and port
INADDR_ANY	non zero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	non zero	Process specifies IP address and port

Note, the local host address is 127.0.0.1; for example, if you wanted to run your echoServer (see later) on your local machine the your client would connect to 127.0.0.1 with the suitable port.

### The listen() Function

The `listen()` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. It is defined as follows:

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

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

where `sockfd` is the socket descriptor and `backlog` is the maximum number of connections the kernel should queue for this socket. The `backlog` argument provides an hint to the system of the number of outstanding connect requests that it should enqueue on behalf of the process. Once the queue is full, the system will reject additional connection requests. The `backlog` value must be chosen based on the expected load of the server.

The function `listen()` return 0 if it succeeds, -1 on error.

### The accept() Function

The `accept()` is used to retrieve a connect request and convert that into a request. It is defined as follows:

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

int accept(int sockfd, struct sockaddr *cliaddr,
socklen_t *addrlen);
```

where `sockfd` is a new file descriptor that is connected to the client that called the `connect()`. The `cliaddr` and `addrlen` arguments are used to return the protocol address of the client. The new socket descriptor has the same socket type and address family of the original socket. The original socket passed to `accept()` is not associated with the connection, but instead remains available to receive additional connect requests. The kernel creates one connected socket for each client connection that is accepted.

If we don't care about the client's identity, we can set the `cliaddr` and `addrlen` to `NULL`. Otherwise, before calling the `accept` function, the `cliaddr` parameter has to be set to a buffer large enough to hold the address and set the integer pointed by `addrlen` to the size of the buffer.

### The `send()` Function

Since a socket endpoint is represented as a file descriptor, we can use `read` and `write` to communicate with a socket as long as it is connected. However, if we want to specify options we need another set of functions.

For example, `send()` is similar to `write()` but allows to specify some options. `send()` is defined as follows:

```
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags
);
```

where `buf` and `nbytes` have the same meaning as they have with `write`. The additional argument `flags` is used to specify how we want the data to be transmitted. We will not consider the possible options in this course. We will assume it equal to 0.

The function returns the number of bytes if it succeeds, -1 on error.

### The `recv()` Function

The `recv()` function is similar to `read()`, but allows to specify some options to control how the data are received. We will not consider the possible options in this course. We will assume it equal to 0.

`recv` is defined as follows:

```
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

The function returns the length of the message in bytes, 0 if no messages are available and peer had done an orderly shutdown, or -1 on error.



## The close() Function

The normal `close()` function is used to close a socket and terminate a TCP socket. It returns 0 if it succeeds, -1 on error. It is defined as follows:

```
#include <unistd.h>

int close(int sockfd);
```

## UDP Socket API

There are some fundamental differences between TCP and UDP sockets. UDP is a connection-less, unreliable, datagram protocol (TCP is instead connection-oriented, reliable and stream based). There are some instances when it makes to use UDP instead of TCP. Some popular applications built around UDP are DNS, NFS, SNMP and for example, some Skype services and streaming media.

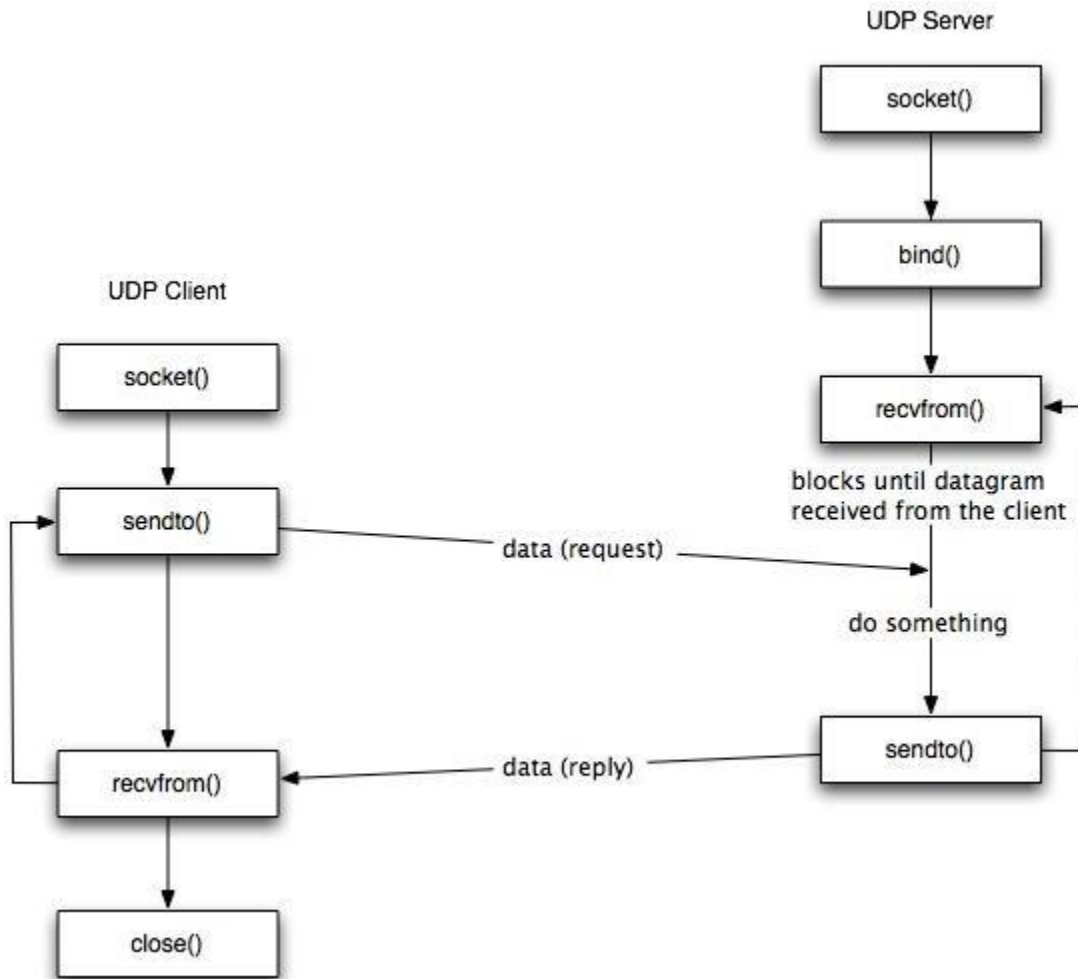
Figure 4 shows the the interaction between a UDP client and server. First of all, the client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the `sendto` function which requires the address of the destination as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the `recvfrom` function, which waits until data arrives from some client. `recvfrom` returns the IP address of the client, along with the datagram, so the server can send a response to the client.

As shown in the Figure, the steps of establishing a UDP socket communication on the client side are as follows:

- Create a socket using the `socket()` function;
- Send and receive data by means of the `recvfrom()` and `sendto()` functions.

The steps of establishing a UDP socket communication on the server side are as follows:

- Create a socket with the `socket()` function;
- Bind the socket to an address using the `bind()` function;
- Send and receive data by means of `recvfrom()` and `sendto()`.



UDP client-server.

In this section, we will describe the two new functions `recvfrom()` and `sendto()`.

### The `recvfrom()` Function

This function is similar to the `read()` function, but three additional arguments are required. The `recvfrom()` function is defined as follows:

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

```
ssize_t recvfrom(int sockfd, void* buff, size_t nbytes,
                 int flags, struct sockaddr* from,
                 socklen_t *addrlen);
```

The first three arguments `sockfd`, `buff`, and `nbytes`, are identical to the first three arguments of `read` and `write`. `sockfd` is the socket descriptor, `buff` is the pointer to read into, and `nbytes` is number of bytes to read. In our examples we will set all the values of the `flags` argument to 0. The `recvfrom` function fills in the socket address structure pointed to by `from` with the protocol address of who sent the datagram. The number of bytes stored in the socket address structure is returned in the integer pointed by `addrlen`.

The function returns the number of bytes read if it succeeds, -1 on error.

### The `sendto()` Function

This function is similar to the `send()` function, but three additional arguments are required. The `sendto()` function is defined as follows:

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buff, size_t nbytes,
               int flags, const struct sockaddr *to,
               socklen_t addrlen);
```

The first three arguments `sockfd`, `buff`, and `nbytes`, are identical to the first three arguments of `recv`. `sockfd` is the socket descriptor, `buff` is the pointer to write from, and `nbytes` is number of bytes to write. In our examples we will set all the values of the `flags` argument to 0. The `to` argument is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is sent. `addrlen` specified the size of this socket.

The function returns the number of bytes written if it succeeds, -1 on error.

### Concurrent Servers

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). We do not consider this kind of servers in this course.

### The `fork()` function

The `fork()` function is the only way in Unix to create a new process. It is defined as follows:

```
#include <unistd.h>

pid_t fork(void);
```

The function returns 0 if in child and the process ID of the child in parent; otherwise, -1 on error.

In fact, the function `fork()` is called once but returns *twice*. It returns once in the calling process (called the parent) with the process ID of the newly created process (its child). It also returns in the child, with a return value of 0. The return value tells whether the current process is the parent or the child.

## Example

A typical concurrent server has the following structure:

```
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 ) {

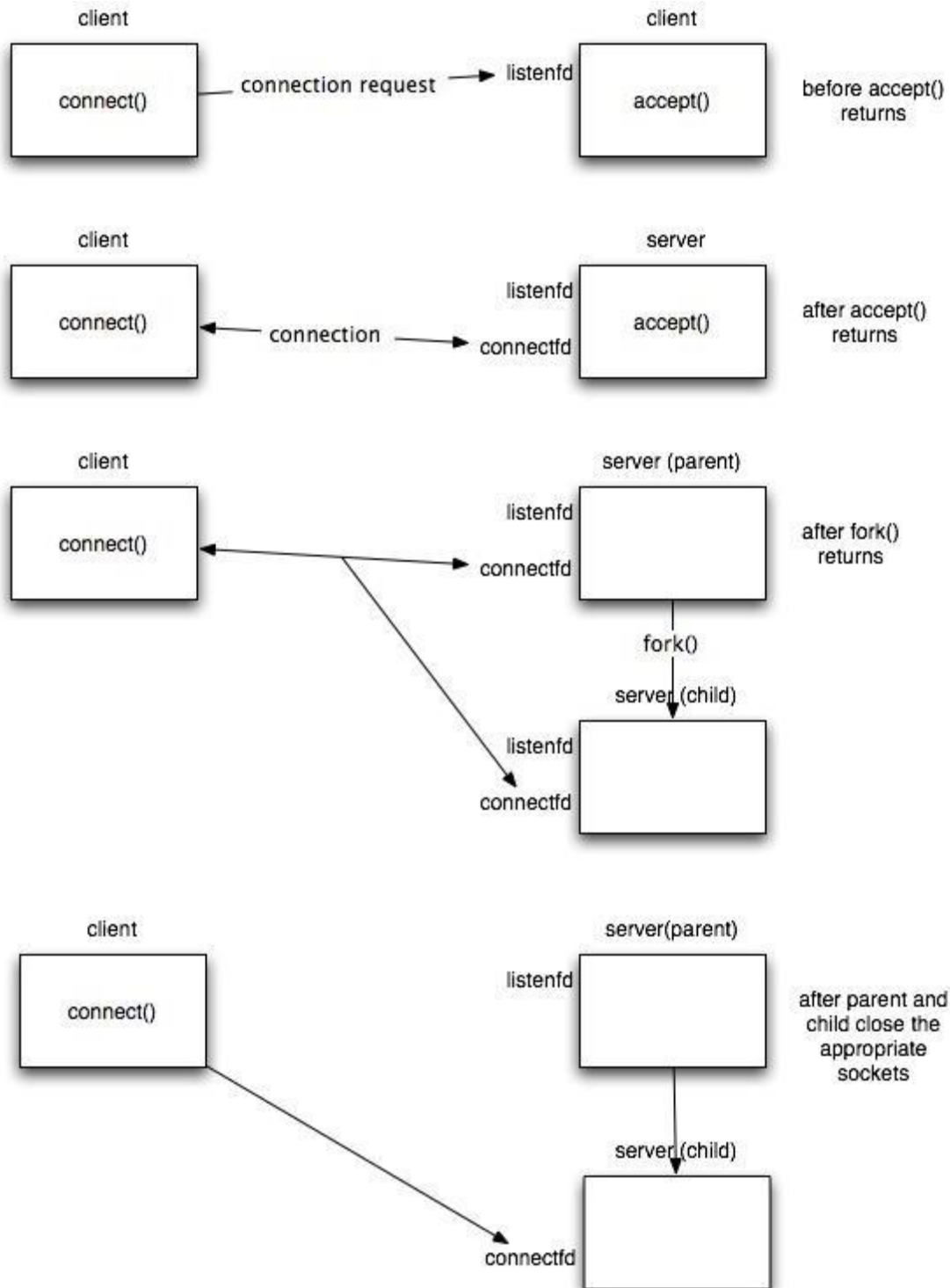
        close(listenfd); /* child closes listening socket */

        /**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 in Figure .

---



Example of interaction among a client and a concurrent server.

---

## TCP Client/Server Examples

We now present a complete example of the implementation of a TCP based echo server to summarize the concepts presented above. We present an iterative and a concurrent implementation of the server.

**echoClient.c source:**

### TCP Echo Client

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

#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/

int
main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr;
    char sendline[MAXLINE], recvline[MAXLINE];

    //basic check of the arguments
    //additional checks can be inserted
    if (argc !=2) {
        perror("Usage: TCPClient <IP address of the server>");
        exit(1);
    }

    //Create a socket for the client
    //If sockfd<0 there was an error in the creation of the socket
    if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) <0) {
        perror("Problem in creating the socket");
        exit(2);
    }

    //Creation of the socket
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr= inet_addr(argv[1]);
```

```

servaddr.sin_port = htons(SERV_PORT); //convert to big-
endian order

//Connection of the client to the socket
if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servadd
r))<0) {
    perror("Problem in connecting to the server");
    exit(3);
}

while (fgets(sendline, MAXLINE, stdin) != NULL) {

    send(sockfd, sendline, strlen(sendline), 0);

    if (recv(sockfd, recvline, MAXLINE, 0) == 0){
        //error: server terminated prematurely
        perror("The server terminated prematurely");
        exit(4);
    }
    printf("%s", "String received from the server: ");
    fputs(recvline, stdout);
}

exit(0);
}

```

**echoServer.c source:**

### **TCP Iterative Server**

```

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

#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/
#define LISTENQ 8 /*maximum number of client connections */

int main (int argc, char **argv)
{
    int listenfd, connfd, n;
    socklen_t clilen;

```

```

char buf[MAXLINE];
struct sockaddr_in cliaddr, servaddr;

//creation of the socket
listenfd = socket (AF_INET, SOCK_STREAM, 0);

//preparation of the socket address
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

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

listen (listenfd, LISTENQ);

printf("%s\n", "Server running...waiting for connections.");

for ( ; ; ) {

    clilen = sizeof(cliaddr);
    connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &clilen
);
    printf("%s\n", "Received request...");

    while ( (n = recv(connfd, buf, MAXLINE, 0)) > 0) {
        printf("%s", "String received from and resent to the client:");
        puts(buf);
        send(connfd, buf, n, 0);
    }

    if (n < 0) {
        perror("Read error");
        exit(1);
    }
    close(connfd);

}
//close listening socket
close (listenfd);
}

```



## conEchoServer.c source:

### TCP Concurrent Echo Server

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

#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/
#define LISTENQ 8 /*maximum number of client connections*/

int main (int argc, char **argv)
{
    int listenfd, connfd, n;
    pid_t childpid;
    socklen_t clilen;
    char buf[MAXLINE];
    struct sockaddr_in cliaddr, servaddr;

    //Create a socket for the socket
    //If sockfd<0 there was an error in the creation of the socket
    if ((listenfd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Problem in creating the socket");
        exit(2);
    }

    //preparation of the socket address
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    //bind the socket
    bind (listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

    //listen to the socket by creating a connection queue, then wait
    //for clients
    listen (listenfd, LISTENQ);

    printf("%s\n", "Server running...waiting for connections.");
```

```

for ( ; ; ) {

    clilen = sizeof(cliaddr);
    //accept a connection
    connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &clilen
);

    printf("%s\n","Received request...");

    if ((childpid = fork ()) == 0) { //if it's 0, it's child process
        printf ("%s\n","Child created for dealing with client requests")
;

        //close listening socket
        close (listenfd);

        while ( (n = recv(connfd, buf, MAXLINE,0)) > 0) {
            printf("%s","String received from and resent to the client:");
            puts(buf);
            send(connfd, buf, n, 0);
        }

        if (n < 0)
            printf("%s\n", "Read error");
        exit(0);
    }
    //close socket of the server
    close(connfd);
}
}

```