NP Journal (Upto tw5) (outputs are not included)

Termwork 1

Problem Statement/Title:

Implementing IPC using Pipes and message queues.

Objectives of the Experiment:

- Understand the concept of IPC.
- Implementing IPC using pipe and message queue.

Theory:

Inter Process Communication:

- **Process**: Program in execution.
- A process can be of two types: **Independent process, Co-operating process.**
- An **independent process is not affected by the execution of other processes** while a **co-operating process can be affected by other executing processes.**
- Processes can communicate with each other through: **Shared Memory, Message passing**

IPC using Message Queues:

- **Message Queuing** – This allows **messages to be passed between processes using either a single queue or several message queue**. This is managed by **system kernel** these messages are coordinated using an **API.**
- This is managed by **system kernel.** These messages are coordinated using an **API.**

IPC using Pipes:

- **Pipes** (**Same Process**) – This allows flow of ***data in one direction only.*** Analogous to simplex systems (Keyboard). Data from the output is usually buffered until input process receives it which must have a common origin.
- Pipe uses two file descriptors: Writing end –**fd[1]**, Reading end- **fd[0]**

Programs:

Message Queues:

Writer

```
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

struct msg_buffer {
    long mesg_type;
    char mesg_text[100];

}message;

int main() {
    key_t key;
    int msgid;
    key = ftok("progfile", 65);
    msgid = msgget(key, 0666 | IPC_CREAT);
    msgrcv(msgid, &message, sizeof(message), 1, 0);
    printf("Data read is: %s\n", message.mesg_text);
    msgctl(msgid, IPC_RMID, NULL);
    return 0;
}
```

Reader

```
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

struct msg_buffer {
    long mesg_type;
    char mesg_text[100];

}message;

int main() {
    key_t key;
    int msgid;
    key = ftok("progfile", 65);
    msgid = msgget(key, 0666 | IPC_CREAT);
    msgrcv(msgid, &message, sizeof(message), 1, 0);
    printf("Data read is: %s\n", message.mesg_text);
    msgctl(msgid, IPC_RMID, NULL);
    return 0;
}
```

Pipes:

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

int main() {
    int fd[2], n;
    char buffer[100];
    pid_t p;
    pipe(fd);
    p = fork();
    if (p > 0) {
      printf("Parent process pid: %d\n", getppid());
      printf("Child process pid: %d\n", p);
      printf("Passing value child\n");
      write(fd[1], "Hello World!\n", 13);
    }
    else {
      printf("Child process pid: %d\n", getpid());
      printf("Parent process pid: %d\n", getppid());
      n = read(fd[0], buffer, 100);
      printf("Data received by child process: \n");
      write(1, buffer, n);
    }
    return 0;
}
```

Conclusion:

In conclusion, our experiment successfully demonstrated the implementation of Inter-Process Communication (IPC) using pipes and message queues, showcasing their effectiveness in facilitating data exchange between processes.

References:

Barry Nance: “Network Programming in C”, PHI 2002 3.Bob Quinn, Dave Shute: “Windows Socket Network Programming”, Pearson 2003.

Termwork 2

Problem statement/Title:

Implementing client server communication using socket programming that uses connection-oriented protocol at transport layer.

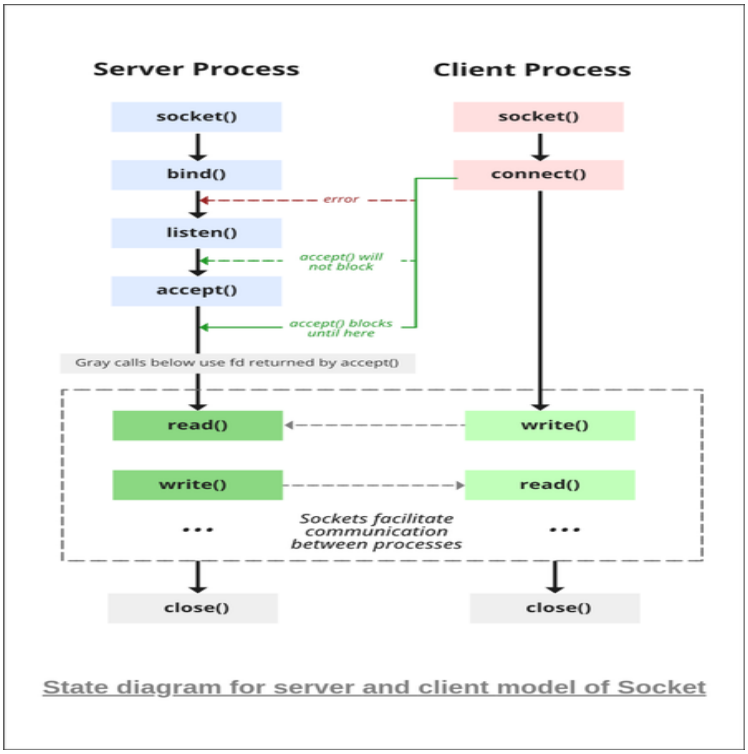
Objectives of the Experiment:

- Understand the concept of Client- Server Communication.
- Understand the concept of Socket.

Theory:

- Socket programming is a way of connecting two nodes on a network to communicate with each other.
- One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection.
- Server forms the listener socket while client reaches out to the server
- If we are creating a connection between client and server using TCP then it has few functionalities like, **TCP is suited for applications that require high reliability, and transmission time is relatively less critical.**
- TCP handles reliability **and congestion control.**
- It also does error **checking and error recovery**.

Diagram (on the blank side)



State diagram for server and client model of Socket

Program:

Client:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 4444

int main() {
    int sockfd;
    struct sockaddr_in servAddr;
    char buffer[1024];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    printf("[+] Client socket created successfully\n");

    bzero(&servAddr, sizeof(servAddr));
    servAddr.sin_family = AF_INET;
    servAddr.sin_port = htons(PORT);
    servAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    connect(sockfd, (struct sockaddr *) &servAddr,
sizeof(servAddr));
    printf("[+] Connected to server\n");

    recv(sockfd, buffer, 1024, 0);
    printf("[+] Data received from server: %s\n", buffer);
    printf("[+] Closing the connection\n");
    return 0;
}
```

Server

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#define PORT 4444

int main() {
    int listenfd, connfd;
    struct sockaddr_in servAddr, cliAddr;
    socklen_t clilen;
    char buffer[1024];

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    printf("[+] Server socket created successfully\n");

    bzero(&servAddr, sizeof(servAddr));
    servAddr.sin_family = AF_INET;
    servAddr.sin_port = htons(PORT);
    servAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    bind(listenfd, (struct sockaddr *) &servAddr, sizeof(servAddr));
    printf("[+] Bind to PORT %d successful\n", PORT);

    listen(listenfd, 5);
    printf("[+] Listening...\n");

    connfd = accept(listenfd, (struct sockaddr *) &cliAddr,
&clilen);

    strcpy(buffer, "Hello World!");
    send(connfd, buffer, strlen(buffer), 0);
    printf("[+] Data sent to client: %s\n", buffer);

    printf("[+] Closing the connection\n");
    return 0;
}
```

Conclusion:

In conclusion, our experiment successfully implemented client-server communication using socket programming with a connection-oriented transport layer protocol. This approach ensures reliable data exchange between systems.

References:

Barry Nance: “Network Programming in C”, PHI 2002 3.Bob Quinn, Dave Shute: “Windows Socket Network Programming”, Pearson 2003.

Termwork 3

Problem Statement/Title:

Implement the distance vector routing algorithm

Objectives of the Experiment:

- To implement Distance Vector Routing algorithm to find suitable path for transmission
- Understand the concept of Distance Vector Routing.
- Implement 'C' program to find the shortest path between the two nodes.

Theory:

- **Routing Protocols:** Distance vector routing is a class of routing protocols used in computer networks to determine the best path for data packets to travel from the source to the destination.
- An internet needs dynamic routing table in order to incorporate changes into the table whenever there is a change in internet. They need to be updated when the ***route is down***, or when a ***better route is created***.
- **Distance Vector routing:**
    - **Dynamic Routing**: Routers quickly adapt routing decisions in milliseconds, using Vector tables for distance information, updated through neighbour exchanges.
    - **Vector Table Entries**: Entries for each destination router contain preferred routes and estimated hop distances, enabling rapid routing adjustments.

Algorithm:

1. Send my routing table to all my neighbours whenever my link table changes.

2. When I get a routing table from a neighbour on port **P** with link metric **M:**

    ❖ add **L** to each of the neighbour's metrics.

    ❖ for each entry **(D, P', M')** in the updated neighbour's table

    i) if I do not have an entry for **D**, add **(D, P, M')** to my routing table

    ii) if I have an entry for **D** with metric **M"**, add **(D, P, M')** to my routing table if **M' < M"**

3. If my routing table has changed, send all the new entries to all my neighbours.

[if sir/ma'am asks to draw the routing table with the nodes and all of that, well, figure it out by yourself 😊]

Program:

```
#include <stdio.h>
#define NODES 10
#define NO_ROUTE 999
#define NO_HOP 1000
int no;
struct node {
    int a[NODES][4];
}router[NODES];
void init(int r) {
    int i;
    for (i = 1; i <= no; i++) {
        router[r].a[i][1] = i;
        router[r].a[i][2] = NO_ROUTE;
        router[r].a[i][3] = NO_HOP;
    }
    router[r].a[r][2] = 0;
    router[r].a[r][3] = r;
}
void inp(int r) {
    int i;
    printf("\nEnter distance from node %d to other nodes\n", r);
    printf("Enter 999 if there is no direct route\n");
    for (i = 1; i <= no; i++) {
        if (i != r) {
                printf("Enter distance to node %d: ", i);
                scanf("%d", &router[r].a[i][2]);
                router[r].a[i][3] = i;
        }
    }
}
void display(int r) {
    int i;
    printf("\nThe routing table for node %d is as follows", r);
    for (i = 1; i <= no; i++) {
        if (router[r].a[i][2] == 999)
                printf("\n%d \t no link \t no hop", router[r].a[i][1]);
        else
```

```
                printf("\n%d \t %d \t %d", router[r].a[i][1], router[r].a[i][2],
router[r].a[i][3]);

    }

}

void dv_algo(int r) {

    int i, j, z;

    for (i = 1; i <= no; i++) {

    // r → source router

    // i → step taken (via which router to reach the dest router)

    // j → destination router

    // cannot jump from the source router or to a router which is not reachable or from the
source router

         if (router[r].a[i][2] != 999 && router[r].a[i][2] != 0) {

                for (j = 1; j <= no; j++) {

                        z = router[r].a[i][2] + router[i].a[j][2];

                        if (z < router[r].a[j][2]) {

                                router[r].a[j][2] = z;

                                router[r].a[j][3] = i;

                        }

                }

         }

    }

}

int main() {

    int i, j, x, y;

    char choice = 'y';

    printf("Enter the number of nodes: ");

    scanf("%d", &no);

    for (i = 1; i <= no; i++) {

         init(i);

         inp(i);

    }

    printf("\nThe routing tables of nodes after initialization is as follows");

    for (i = 1; i <= no; i++)

         display(i);

    printf("\n\nComputing shortest paths...\n");

    for (i = 1; i <= no; i++)

         dv_algo(i);

    printf("\nThe routing tables of nodes after computation of shortest paths is as follows");

    for (i = 1; i <= no; i++)
```

```
        display(i);
    printf("\n");
    while (choice != 'n'){
        printf("\nEnter the nodes between which shortest distance is to be found: ");
        scanf("%d %d", &x, &y);
        getchar();
        printf("The length of the shortest path between nodes %d and %d is %d\n", x, y,
router[x].a[y][2]);
        printf("Continue? (y/n): ");
        scanf("%c", &choice);
    }
    return 0;
}
```

Conclusion:

In conclusion, the implementation of the distance vector routing algorithm demonstrated its effectiveness in finding optimal paths in a network, offering a valuable solution for routing and communication efficiency.

References:

Barry Nance: “Network Programming in C”, PHI 2002 3. Bob Quinn, Dave Shute: “Windows Socket Network Programming”, Pearson 2003.

Termwork 4:

Problem statement/Title:

Using WIRESHARK observe the data transferred in client server communication using UDP

and identify the UDP datagram.

Objective of the experiment:

- Understand the significance of Wireshark tool.
- Understand how to analyse the UDP Datagram using Wireshark.

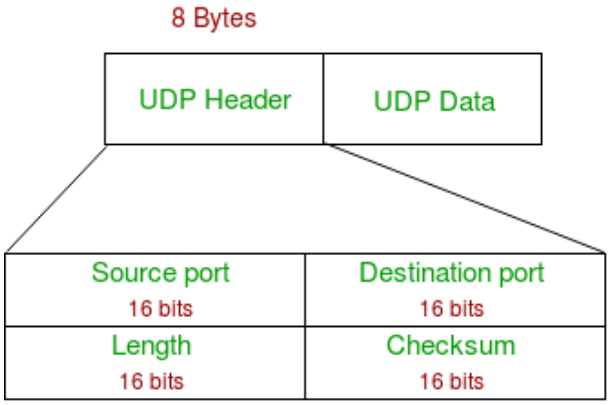
Theory:

- Wireshark is a software tool used to monitor the network traffic through a network interface. Most widely used network monitoring tool.
- Wireshark is a network packet analyzer that captures and inspects data packets.
- It involves capturing, examining, and interpreting UDP packets to understand their structure and content.
- UDP (User Datagram Protocol) is a connectionless transport layer protocol used for data transmission.
- It's faster but less reliable than TCP due to its lack of error-checking and retransmission.
- Findings can help diagnose network issues and optimize UDP-based applications.

Program Execution Steps:

Capturing UDP packets with browser:

1. Open Wireshark and double-click on any-interface to start the packet capture process.
2. Open the browser and enter any website's fully qualified domain name in the browser address bar and hit enter.
3. After the site is fully loaded, stop the capturing process in Wireshark go to edit in the menu bar and select find packet option or just press **CTRL+F.**
4. In Find Packet menu bar, select the String option in the display filter drop-down menu and enter the name of the website in the next box and click on find.
5. The arrow indicating towards the packet is the **request packet**, and the arrow coming out from the packet is the **response packet**.
6. Click on any request or response DNS packet and examine UDP packet.
7. Go to statistics: Generate I/O Graph, Flow Graph and study and analyze both the graphs



To be drawn on the blank page

Conclusion:

In this experiment, we employed Wireshark to analyze client-server communication over UDP. By inspecting captured data, we successfully identified and examined UDP datagrams, enhancing our understanding of this protocol.

References:

Barry Nance: “Network Programming in C”, PHI 2002 3. Bob Quinn, Dave Shute: “Windows Socket Network Programming”, Pearson 2003.

Termwork 5:

Problem Statement/Title:

Using WIRESHARK analyse three-way handshaking connection establishment, data transfer and connection termination in client server communication using TCP.

Objectives of the Experiment:

- Understand the significance of Wireshark tool.
- Understand how to analyse the TCP packets using Wireshark.

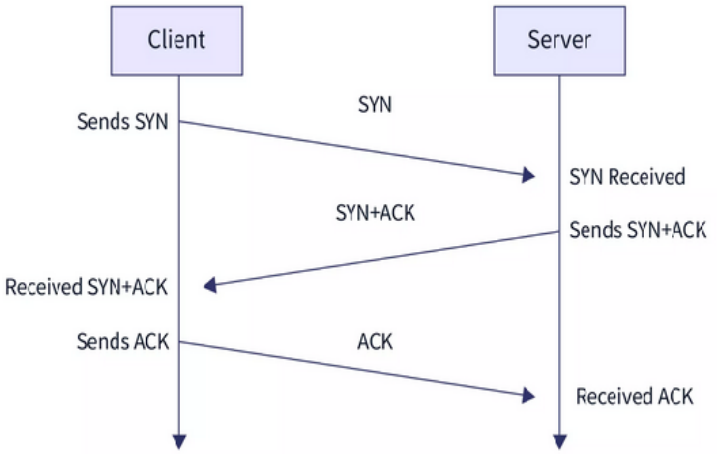
Theory:

- Wireshark is a software tool used to monitor the network traffic through a network interface. Most widely used network monitoring tool.
- Wireshark is a network packet analyser that captures and inspects data packets.
- It involves capturing, examining, and interpreting UDP packets to understand their structure and content.
- Transmission Control Protocol (TCP) is a fundamental communication protocol for reliable data transfer in client-server applications.
- TCP uses a three-step process to establish a connection between a client and a server, ensuring reliability and synchronization.
- During the handshake, the client sends a SYN packet, the server responds with a SYN-ACK, and the client acknowledges with an ACK, establishing a connection.
- After the connection is established, data is exchanged in segments. TCP ensures data integrity, sequencing, and flow control.

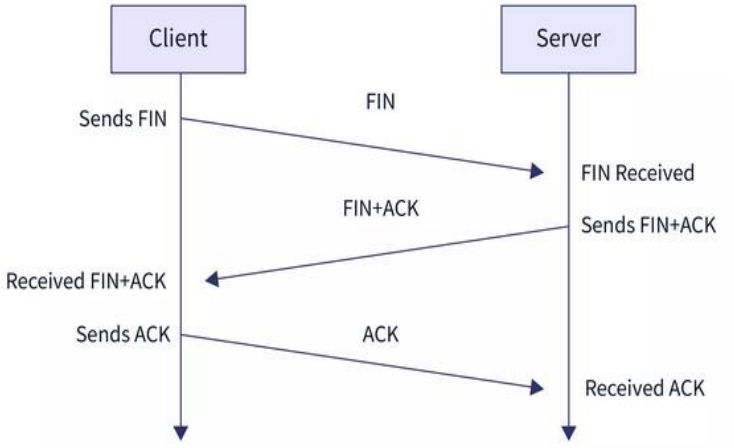
Program Execution Steps:

1. Open Wireshark and double-click on any-interface to start the packet capture process.
2. Open the browser and enter any website's fully qualified domain name in the browser address bar and hit enter.
3. After the site is fully loaded, stop the capturing process, in Wireshark.
4. Type the following in, apply a filter column and hit-enter: **tcp.flags.fin==1 and tcp.flags.ack ==1**
5. Select any one of these listed packets, right-click and hover on conversation filter and select TCP.
6. Once done analyze the TCP Packets.
7. Go to statistics: Generate I/O Graph, Flow Graph and study and analyze both the graphs
8. Observe TCP 3-way Handshake mechanism, data transfer and connection termination through TCP

Diagram (on the blank page):



TCP 3-way handshake process



TCP 3-Way Handshake for Terminating the Connection

Conclusion:

In this experiment, we employed Wireshark to examine TCP client-server communication. We observed the three-way handshake, data transfer, and connection termination, gaining insights into network protocol behaviour.

References:

Barry Nance: “Network Programming in C”, PHI 2002 3. Bob Quinn, Dave Shute: “Windows Socket Network Programming”, Pearson 2003.