EXPERIMENT-3

TRANSMISSION CONTROL PROTOCOL

AIM : To implement transmission control protocol.

THEORY :

A TCP (transmission control protocol) is a connection-oriented communication. TCP is designed to send the data packets over the network. It ensures that data is delivered to the correct destination.TCP creates a connection between the source and destination node before transmitting the data and keeps the connection alive until the communication is active.

Server-client model is a communication model for sharing the resource and provides the service to different machines. Server is the main system which provides the resources and different kind of services when client requests to use it

ALGORITHM :

1. Start
2. Declare variables and structures for socket communication, including sd (socket descriptor), servAddr (server address structure).
3. Create a socket using the socket() function. Handle errors gracefully if socket creation fails.
4. Set up the server address structure (servAddr) with the server's IP address and port number.
5. Establish a connection to the server using the connect() function. Handle errors gracefully if connection fails.
6. Prepare the data to be sent to the server.
7. Transmit the data to the server using the send() function. Handle errors gracefully if data transmission fails.
8. Wait for the server's response.
9. Receive data from the server using the recv() function. Handle errors gracefully if data reception fails.
10. Process the received data as required (e.g., print it to the console).
11. Perform any additional processing steps based on the received data.
12. Close the socket using the close() function to release system resources. Handle any errors that may occur during the closing process.
13. Stop

EXPERIMENT - 4

USER DATAGRAM PROTOCOL

AIM : To implement user datagram protocol.

THEORY :

UDP is a connection-less protocol that, unlike TCP, does not require any handshaking prior to sending or receiving data, which simplifies its implementation. In UDP, the client does not form a connection with the server like in TCP and instead just sends a datagram. Similarly, the server need not accept a connection and just waits for datagrams to arrive.

Server-client model is a communication model for sharing the resource and provides the service to different machines. Server is the main system which provides the resources and different kind of services when client requests to use it

ALGORITHM :

Client Program:

1. Start
2. Declare necessary variables and structures.
3. Create a UDP socket using **socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)**.
4. Check if the socket creation was successful, if not, print an error message and exit.
5. Print a message indicating successful socket creation.
6. Set up the server address structure (**servAddr**) with appropriate values for family, port, and IP address.
7. Prompt the user to enter a message for the server and store it in **cliMsg**.
8. Send the message to the server using **sendto()**.
9. Check if the message sending was successful, if not, print an error message and exit.
10. Receive a message from the server using **recvfrom()** and store it in **servMsg**.
11. Check if the message reception was successful, if not, print an error message and exit.
12. Close the client socket
13. Stop

Server Program:

1. Start
2. Declare necessary variables and structures.
3. Create a UDP socket using **socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)**.
4. Check if the socket creation was successful, if not, print an error message and exit.
5. Print a message indicating successful socket creation.
6. Set up the server address structure (**servAddr**) with appropriate values for family, port, and IP address.
7. Bind the socket to the server address using **bind()**.
8. Check if the binding was successful, if not, print an error message and exit.
9. Print messages indicating successful binding and listening.
10. Receive a message from the client using **recvfrom()** and store it in **cliMsg**.
11. Check if the message reception was successful, if not, print an error message and exit.
12. Print the received message from the client.
13. Copy the received message to **servMsg**.
14. Send a message to the client using **sendto()**.
15. Check if the message sending was successful, if not, print an error message and exit.
16. Close the server socket
17. Stop

EXPERIMENT - 5 a

STOP AND WAIT

AIM : To implement stop and wait.

THEORY :

Sliding window protocols are data link layer protocols for reliable and sequential delivery of data frames. In this protocol, multiple frames can be sent by a sender at a time before receiving an acknowledgment from the receiver. The term sliding window refers to the imaginary boxes to hold frames.

Stop-and-Wait is the simplest flow control method. In this, the sender will transmit one frame at a time to the receiver. The sender will stop and wait for the acknowledgement from the receiver. When the sender gets the acknowledgement (ACK), it will send the next data packet to the receiver and wait for the disclosure again, and this process will continue as long as the sender has the data to send. The sender and receiver window size is 1

ALGORITHM :

Server Program:

```
1. Start
2. Declare necessary variables and structures, including server, newSock, k, m, p, and buffer.
3. Create a socket for TCP communication using socket(PF_INET, SOCK_STREAM, 0).
4. Set up the server address structure (servAddr) with appropriate values for family, port, and IP address.
5. Bind the socket to the server address using bind().
6. Listen for incoming connections with a backlog of 5 using listen().
7. Accept a connection from a client using accept(), storing the new socket descriptor in newSock.
8. Loop while k is not equal to 0:
    1. Receive data from the client into buffer using recv().
    2. Check if the received data starts with "frame":
        If yes, print a success message.
        If no, print an error message.
9. Determine the acknowledgment to send based on the value of m.
10. Send the acknowledgment to the client using send().
11. Decrement k and increment m.
12. Close the client socket and server socket using close().
13. Stop
```

Client Program:

```
1. Start
2. Declare necessary variables and structures, including client, y, x, k, m, p, and buffer.
3. Create a socket for TCP communication using socket(PF_INET, SOCK_STREAM, 0).
4. Set up the server address structure (servAddr) with appropriate values for family, port, and IP address.
```

5. Connect to the server using connect().
6. Loop while k is not equal to 0:
    1. Check if m is less than or equal to 5 and print the message to send.
    2. Determine the data to send based on the value of m.
    3. Send the data to the server using send().
    4. Receive acknowledgment from the server into buffer using recv().
    5. Print whether the acknowledgment was received or not.
    6. Decrement k and increment m.
7. Close the client socket using close().
8. Stop

EXPERIMENT - 5 b

SELECTIVE REPEAT

AIM : To implement selective repeat .

THEORY :

The selective repeat ARQ is one of the Sliding Window Protocol strategies that is used where reliable in-order delivery of the data packets is required. The selective repeat ARQ is used for noisy channels or links and it manages the flow and error control between the sender and the receiver. In the selective repeat ARQ, we only resend the data frames that are damaged or lost. On the other hand, the correct frames are received at the receiver's end and are buffered for future usage.

ALGORITHM :

Server

1. Start
2. Create a server socket using socket() with parameters AF_INET, SOCK_STREAM, and 0.
3. Initialize server and client address structures with zeros using memset().
4. Set the server address family (AF_INET), port number (htons(7004)), and IP address (INADDR_ANY) in the server structure.
5. If binding the server socket fails, print "Binding failed" and exit the program.
6. Print a message indicating the server is ready.
7. Listen for incoming connections on the server socket with a backlog of 10 using listen().
8. Accept incoming connections from clients using accept(), storing the client address in other.
9. Initialize variables for message buffers, flags, timeout and fd_sets.
10. Enter a loop until a total of 9 messages are received:
    1. Send 3 messages to the client.
    2. Receive acknowledgments for each message:
        1. Initialize a loop for each message.
        2. Set up the file descriptor set set1 with the client socket.
        3. Set a timeout of 2 seconds.
        4. Perform select() on the client socket with set1 and timeout1.
        5. If select() times out, print a timeout message, resend the message, and go to step 9.2.
        6. If select() succeeds, read the acknowledgment from the client.
        7. If the acknowledgment indicates a corrupted message, resend the message and go to step 9.2.
        8. If the acknowledgment is valid, increment tot.

1. Close the client and server sockets.
2. Stop

Client

1. Start
2. Generate a random number generator with the current time using srand().
3. Create a client socket using socket() with parameters AF_INET, SOCK_STREAM, and 0.
4. Initialize the client address structure with zeros using memset().
5. Set the client address family (AF_INET), port number (htons(7004)), and IP address (127.0.0.1) in the client structure.
6. If connecting to the server fails, print "Connection failed" and exit the program.
7. Print a message indicating that the client is ready.
8. Initialize message buffers: msg1 for positive acknowledgments, msg3 for negative acknowledgments, msg2 for storing acknowledgment messages, and buff for receiving data from the server.
9. Initialize variables: count to -1 to track the message count, and flag to handle the special case when count is 7.
10. Enter a loop until a total of 8 messages are received:
    1. Clear the buff and msg2 buffers.
    2. If count is 7 and flag is 1, read a message from the server without storing the data and continue to the next iteration.
    3. Read a message from the server into the buff buffer.
    4. Extract the message index i from the received message.
    5. Generate a random value to simulate message corruption using the isfaulty() function.
    6. Print the received message and the corruption status.
11. Construct the acknowledgment message in msg2 based on the corruption status:
    1. If the message is corrupted, use msg3.
    2. If the message is not corrupted, use msg1 and increment count.
        1. Append the message index i to the acknowledgment message.
        2. Send the acknowledgment message to the server using write().
12. Close the client socket.
13. Stop

EXPERIMENT - 5 b

GO BACK N

AIM : To implement go back n algorithm
.

THEORY :

In Go Back N protocol, multiple frames can be sent by a sender at a time before receiving an acknowledgement from the receiver. It has a transmit window size of N and a receive window size of 1. It can transmit N frames to the peer before requiring an ACK. If any frame is corrupted or lost, all subsequent frames have to be sent again. If the receiver receives a corrupted frame, it cancels it. The receiver does not accept a corrupted frame. When the timer expires, the sender sends the correct frame again. If the acknowledgement is not shared with the sender side within a certain time frame, all the frames after the non-acknowledged frame are to be retransmitted to the receiver side.

ALGORITHM :

Server

1. Start
2. Create a server socket using socket() with parameters AF_INET, SOCK_STREAM, and 0.
3. Initialize server address structure serv_addr with zeros using memset().
4. Set the server address family (AF_INET), port number (7004), and IP address (INADDR_ANY) in the serv_addr structure.
5. Bind the server socket to the server address using `bind()`. If the binding fails, print "Binding failed" and exit the program.
6. Print a message indicating the server is ready.
7. Listen for incoming connections on the server socket with a backlog of 5 using `listen()`.
8. Accept incoming connections from clients using `accept()`, storing the client address in `other_addr`.
9. Initialize all variables.
10. Label zero:
    1. Increment `i` by 1.
    2. Clear the `write_buff`.
    3. Construct the message in `write_buff`.
    4. Send the message to the receiver.
    5. Increment `i` by 1.
    6. Sleep for 1 second.
11. Label one:
    1. Clear the ‘write buff’.
    2. Construct the message in `write_buff`.
    3. Send the message to the receiver.
    4. Set up a file descriptor set `set` and add `acpt_sock` to it.
    5. Set a timeout of 2 seconds.
    6. Perform `select()` on `acpt_sock` with the `set` and timeout.
    7. If `select()` returns -1, print "Error in select()".
    8. If `select()` times out (returns 0):

        1. Print a timeout message.
        2. Go back to label zero.
    9. If `select()` succeeds:
        1. Read the acknowledgement from the receiver.
        2. Print the acknowledgement message based on the value of `i`.
        3. Increment `i`.
        4. If `i` exceeds 7, exit the program.
        5. If `i` is 5 or 3, go back to label zero.
        6. If `i` is less than 8 or not equal to 5, go back to label one.
12. Close the client and server sockets.
13. Stop.

Client

1. Start
2. Create a client socket using socket() with parameters AF_INET, SOCK_STREAM, and 0.
3. Initialize the client address structure client with zeros using memset().
4. Set the client address family (AF_INET), port number (htons(7004)), and IP address (inet_addr("127.0.0.1")) in the client structure.
5. If connecting to the server fails, print "Connection failed" and exit the program.
6. Print a message indicating that the client is ready.
7. Initialize message buffers:
    1. Initialize msg1 for acknowledgment message.
    2. Initialize write_buff and read_buff for reading and writing messages.
8. Initialize variables:
    1. Initialize i as a loop counter.
    2. Initialize j for ACK numbering.
    3. Initialize flag to handle the special case when i is 5.
9. 8. Start a loop from 0 to 7 to receive messages and send acknowledgments:
    1. Clear the read_buff and write_buff.
    2. If i is 5 and flag is 1, simulate a loss by reading from the server without storing the data.
    3. Read the message from the server into the read_buff.
    4. Check if the received message is in order by comparing the last character of the message with the expected value (i + '0').
    5. If the message is out of order:
        1. Print a discard message.
        2. Decrement i by 1.
    6. If the message is in order:
        1. Print the received message.
        2. If i is 0, read an additional message from the server and increment i.
        3. If i is 3 or 5, read an additional message from the server and increment i.
        4. Determine the ACK number j.
        5. Print the acknowledgment message based on the value of i.
        6. Construct the acknowledgment message in write_buff.
        7. Send the acknowledgment message to the server using write().
10. Close the client socket.
11. Stop

EXPERIMENT-6

DISTANCE VECTOR ROUTING

AIM : To implement distance vector routing
.

THEORY :

Distance Vector Routing (DVR) is a routing algorithm used in computer networks to determine the optimal path for sending data packets between nodes. In DVR, each node maintains a routing table that contains information about the distances to reach other nodes in the network. Nodes exchange this information with their neighbouring nodes, allowing each node to update its routing table based on the received information. This process continues until all nodes have converged on the shortest paths to all destinations. By implementing and simulating the DVR algorithm, researchers and network administrators can gain practical insights into how routers exchange routing information, collaborate to find the best paths, and adapt to changes in network topology.

The DVR experiment enables the study of various aspects of routing, such as efficiency, scalability, and stability. It helps evaluate how well the algorithm performs under different network conditions, including link failures and topology changes. By observing the evolution of routing tables over time, researchers can analyse the effectiveness of the DVR algorithm in finding optimal paths and maintaining network connectivity. Additionally, the experiment provides valuable insights into the dynamics of information propagation and routing decisions in a decentralized network. This understanding contributes to the development of more robust and efficient routing protocols for real-world networks.

ALGORITHM :

1. Start
2. Declare structures **Link** and **Network**.
3. Define the **main()** function.
4. Declare variables **H**, **L**, **S**, **i**, and **j**.
5. Prompt the user to enter the number of hops (**H**), number of links (**L**), and the source node (**S**).
6. Allocate memory for the **Network** structure and initialize its members.
7. Allocate memory for the **dist** array to store distances from the source node.
8. Initialize all distances in **dist** array to **INT_MAX**.
9. Set the distance from source node **S** to 0.
10. Prompt the user to enter details for each link: source, destination, and weight.
11. For each link, update distances in **dist** array using Bellman-Ford algorithm.
12. Check for negative weight cycles:
    12.1. If there's a shorter path, update **dist**.
    12.2. If any distance changes in the last iteration, there's a negative weight cycle.

13 Print distances from the source node.
14 Free allocated memory.
15 Stop

EXPERIMENT - 7

FTP : File Transfer Protocol

AIM : To implement and simulate algorithms for Distance Vector Routing protocol.

THEORY :

FTP, which stands for File Transfer Protocol, was developed in the 1970s to allow files to be transferred between a client and a server on a computer network. The FTP protocol uses two separate channels: the command (or control) channel and the data channel to exchange files. The command channel is responsible for accepting client connections and executing other simple commands. It typically uses server port 21.

FTP clients will connect to this port to initiate a conversation for file transfer and authenticate themselves by sending a username and password. After authentication, the client and server will then negotiate a new common server port for the data channel, over which the file will be transferred. Once the file transfer is complete, the data channel is closed. If multiple files are to be sent concurrently, a range of data channel ports must be used. The control channel remains idle until the file transfer is complete. It then reports that the file transfer was either successful or failed.

ALGORITHM:

Server :

```
1. Start
2. Initialize variables : sd, port, servaddr, name: fname, rcvg, fp
3. Prompt the user to enter the port number and read it into 'port'.
4. Create a socket with socket() system call with address family AF_INET, type,
   SOCK_STREAM, default protocol and assign the descriptor to 'sd'.
5. Initialize the server address structure 'servaddr' with appropriate values (IP
   address and port).
6. Bind the socket to the server address using bind() system call.
7. Listen for incoming connections with listen().
8. Accept a client connection with accept() and obtain a new socket descriptor.
9. Prompt user to enter the existing file name and read it into 'name'.
10. Prompt user to enter the new file name and read it into 'fname'.
11. Open the new file for writing with fopen().
12. Send the existing file name to the client using send().
13. Loop:
        1. Receive data from the client using recv().
        2. If the received data size 's' is less than or equal to 0:
                1. Handle the case where there is an error in receiving
                   data or the connection is terminated.
                2. Break out of the loop.
            3. Null-terminate the received data.
            4. If the received data 'rcvg' is "error":
                1. Print "File is not available.."
            5. If the received data 'rcvg' is "completed":
            6. Print "File Transfer Complete"
            1. Close the file, close the socket, and exit the program.
    13.6. Otherwise (normal data transfer):
            Terminate the Loop.
14. Close the file, close the socket.
```

15. End.

Client :

1. Start.
2. Initialize variables : sd, port, servaddr, name: fname, rcvg, fp
3. Prompt the user to enter the port number and read it into 'port'.
4. Create a socket with socket() system call with address family AF_INET, type, SOCK_STREAM, default protocol and assign the descriptor to 'sd'.
5. Initialize the server address structure 'servaddr' with appropriate values (IP address and port).
6. Connect to the server using connect() system call.
7. Prompt user to enter the existing file name and read it into 'name'.
8. Prompt user to enter the new file name and read it into 'fname'.
9. Open the new file for writing with fopen().
10. Send the existing file name to the server using send().
11. Loop:
    1. Receive data from the server using recv().
    2. If the received data size 's' is less than or equal to 0:
        1. Handle the case where there is an error in receiving data or the connection is terminated.
        2. Break out of the loop.
    3. Null-terminate the received data.
    4. If the received data 'rcvg' is "error":
        1. Print "File is not available.." indicating the requested file is not found on the server.
    5. If the received data 'rcvg' is "completed":
        1. Print "File is transferred..." indicating the file transfer is complete.
        2. Close the file, close the socket, and exit the program.
    6. Otherwise (normal data transfer):
        1. Write the received data to the file using fputs().
        2. Print the received data to stdout.
12. Terminate the Loop.
13. Close the file, close the socket.
14. Stop

EXPERIMENT-8

LEAKY BUCKET

AIM : To implement congestion control using a leaky bucket algorithm.

THEORY :

The Leaky Bucket Algorithm used to control rate in a network. It regulates the data flow by allowing only a certain amount of data to be transmitted within a specified time period, preventing bursty traffic and ensuring a controlled flow of data.

It is implemented as a single-server queue with constant service time. If the bucket (buffer) overflows then packets are discarded. It enforces a constant output rate (average rate) regardless of the burstiness of the input. It does nothing when input is idle. The host injects one packet per clock tick onto the network. This results in a uniform flow of packets, smoothing out bursts and reducing congestion.When packets are the same size (as in ATM cells), the one packet per tick is okay. For variable length packets though, it is better to allow a fixed number of bytes per tick.

ALGORITHM :

```
    1. Start
    2. Initialize variables:  bucket_size, outgoing, n, incoming, store = 0 as integers.
    3. Read and store the bucket size (bucket_size) from the user.
    4. Read and store the number of inputs (n) from the user.
    5. Read and store the packet outgoing rate (outgoing) from the user.
    6. While n is not equal to 0:
        1. Read, store and display the incoming packet size (incoming) from the user.
        2. If incoming is less than or equal to the available space in the bucket
           (bucket_size - store), then:
            1. Add the incoming packet size (incoming) to the bucket (store +=
               incoming).
            2. Display "Bucket Buffer Size store out of bucket_size"
           If in is greater than the available space in the bucket (bucket_size - store),
           then:
3.      Calculate the number of dropped packets (incoming - (bucket_size - bucket)).
4.          - Display "Dropped x no: of packets."
5.          - Display "Bucket Buffer Size x out of y"
6.      Set the bucket size (bucket) to its maximum capacity (bucket_size).
7.      Print the number of dropped packets.
        6.3.  Print the bucket status (bucket packets out of bucket_size).
        6.4.  Subtract the packet outgoing rate (outgoing) from the bucket size (store).
        6.5.  Print the bucket status after outgoing packets (bucket packets out of
        bucket_size).
        6.6.  Decrement the value of n by 1.
7.      End While Loop.
8.      End.
```