MULTI USER CHAT USiNG TCP

Client

1. Include Necessary Libraries

   - Include standard C libraries for input/output, socket programming, and network-related functionalities.

2. Define Helper Functions

   a. send_msg(client_socket)

      - Read a message from stdin (user input) into the 'message' buffer.

      - If the message is "quit\n", print an exit message and terminate the client program.

      - Otherwise, send the message to the server using the client socket.

   b. receive_msg(client_socket)

      - Receive a message from the server through the client socket into the 'message' buffer.

      - Print the received message to stdout.

      - Handle errors if receiving data fails or the server goes offline.

3. Main Function Execution

   - Create a client socket using socket() with AF_INET and SOCK_STREAM parameters.

   - Handle errors if socket creation fails.

4. Configure Server Address

   - Define the server address (server_addr) with the server's IP address (INADDR_ANY) and port number (4951).

5. Connect to Server

   - Establish a connection to the server using connect() with the client socket.

   - Handle errors if the connection fails.

6. Initialize File Descriptors

   - Set up file descriptors for monitoring using fd_set and initialize them with FD_ZERO.

7. Main Communication Loop

   - Use select() to monitor multiple file descriptors for activity (stdin for user input and client socket for server messages).

8. Handle User Input

   - If there is input available on stdin (FD_ISSET(0, &selected_sockets)), call send_msg() to send the message to the server.

9. Handle Server Messages

- If there is incoming data on the client socket (FD_ISSET(client_socket, &selected_sockets)), call receive_msg() to receive and display the server's message.

10. Cleanup and Exit

- Close the client socket when the communication loop ends or when the user chooses to quit ("quit\n").

Server

1. Include Necessary Libraries

- Include standard C libraries for input/output, socket programming, and network-related functionalities.

2. Define Helper Functions

a. accept_new_connection(server_socket, all_sockets, num_sockets)

- Accept a new client connection on the server socket (server_socket).

- Add the new client socket to the set of all sockets (all_sockets) and update the maximum socket number (num_sockets).

b. receive_and_broadcast(client_socket, server_socket, all_sockets, num_sockets)

- Receive a message from a client on the specified client socket (client_socket).

- Broadcast the received message to all connected clients except the sender.

3. Main Function Execution

- Create a server socket using socket() with AF_INET and SOCK_STREAM parameters.

- Handle errors if socket creation fails.

4. Configure Server Address and Port

- Define the server address (server_addr) with the server's IP address (INADDR_ANY) and port number (4951).

5. Set Socket Options

- Set socket options to allow reusing the same address and port (SO_REUSEADDR) to avoid "Address already in use" errors.

6. Bind and Listen

- Bind the server socket to the specified address and port using bind().

- Start listening for incoming connections using listen() with a backlog queue size of 10.

7. Initialize File Descriptors

- Set up file descriptors for monitoring using fd_set and initialize them with FD_ZERO.

8. Main Communication Loop

- Use select() to monitor multiple file descriptors for activity (server socket for new connections and client sockets for incoming messages).

9. Handle New Connections

- If the server socket has activity (FD_ISSET(server_socket, &selected_sockets)), call accept_new_connection() to accept and add a new client connection.

10. Handle Client Messages

- If a client socket has activity (FD_ISSET(client_socket, &selected_sockets)), call receive_and_broadcast() to receive and broadcast messages to other clients.

11. Cleanup and Exit

- Close the server socket and exit the program gracefully when done.

CONCURRENT TIME USING UDP

Client

Step 1: Include necessary libraries and define constants (PORT, MAXLINE).

Step 2: Define main function:

a. Declare variables: sockfd (socket file descriptor), buffer (for data), t (time), hello (current time string).

b. Get current time and format it as a string using ctime().

Step 3: Create a UDP socket:

a. Check for socket creation failure.

Step 4: Initialize server address structure (servaddr):

a. Set sin_family to AF_INET, sin_port to specified PORT, and sin_addr.s_addr to INADDR_ANY.

Step 5: Send data to the server:

a. Use sendto() to send the hello message (current time) to the server.

Step 6: Receive data from the server:

a. Use recvfrom() to receive data (time from server) into the buffer.

Step 7: Display server response (time received from the server).

Step 8: Close the socket.

End Algorithm

Server

1. Initialize variables and socket structures.

2. Create a UDP socket (sockfd).

3. Set up server address (servaddr) with IP (INADDR_ANY) and port (PORT).

4. Bind the socket to the server address.

5. Loop:

   a. Receive a message from a client using recvfrom().

   b. Extract and process the received message (client time).

   c. Send a response message (server time) back to the client using sendto().

6. Close the socket and exit.

DISTANCE VECTOR ROUTING

1. Define node structure for distance and next hop information.

2. Initialize variables and data structures.

3. Input the number of nodes and the cost matrix.

Declare variables **`dmat`** for the cost matrix, and **`n`**, **`i`**, **`j`**, **`k`**, **`count`** for loop control and counting. Input the number of nodes **`n`** and the cost matrix **`dmat`** from the user.

4. Initialize routing tables with initial distances and next hops.

Initialize the routing tables **`rt[i].dist[j]`** and **`rt[i].from[j]`** with the corresponding values from the cost matrix.

5. Repeat until convergence:

   a. Update routing tables based on the distance vector algorithm.

   b. Count the number of updates.

Distance Vector Algorithm:
Implement the distance vector algorithm using a do-while loop: Iterate through all nodes **`i`** and for each destination node **`j`**. For each node pair **`(i, j)`**, check if there is a shorter path via node **`k`**. If a shorter path is found (**`rt[i].dist[j] > dmat[i][k] + rt[k].dist[j]`**), update the distance and next hop information in the routing table and increment **`count`**.

6. Print the final routing tables for each node showing destination, next hop, and distance.

7. Exit.

STOP AND WAIT PROTOCOL

Client

1. Include necessary libraries:

   - stdio.h, unistd.h, sys/types.h, sys/socket.h, time.h

   - string.h, stdlib.h, netinet/in.h, arpa/inet.h

2. Define constants and structures:

   - PORT, SERVER_IP, MAXSZ

   - FRAMEKIND enum { DATA, ACK }

   - struct MSG { char data[MAXSZ]; }

   - struct Frame { FRAMEKIND type; unsigned int len; int seq; char *msg; }

3. Initialize variables and structures:

   - int sockfd;

   - struct sockaddr_in serverAddress;

   - char msg1[MAXSZ], msg2[MAXSZ], msg3[MAXSZ], q, w;

   - char del = '$';

   - struct Frame *frame = malloc(30 * sizeof(struct Frame));

   - Initialize socket, server address, and connection.

4. Main function:

   - Loop until 30 frames are received:

     - Receive data into msg2 using recv().

     - Extract sequence number and message from msg2.

     - Check for duplicate frames based on the sequence number.

     - If not a duplicate:

       - Populate the frame structure with frame details.

       - Print received frame information.

       - Send acknowledgment (ACK) back to the server.

       - Update expected sequence number.

       - Increment count for received frames.

5. Acknowledgment (ACK) Transmission:

   - Generate ACK based on expected sequence number.

   - Send ACK to the server using send().

- Introduce random sleep delay for simulation.

6. Exit condition:

   - Check if no data is received (n == 0), indicating receiver exit.

7. Cleanup and return:

   - Free allocated memory (not done in the provided code).

   - Return 0 to indicate successful execution.

Server

1. Include necessary libraries:

   - stdio.h, unistd.h, sys/types.h, sys/socket.h, sys/time.h

   - string.h, stdlib.h, netinet/in.h, arpa/inet.h

2. Define constants and structures:

   - PORT, MAXSZ

   - FRAMEKIND enum { DATA, ACK }

   - struct timeval timeout

   - struct Frame { FRAMEKIND type; unsigned int len; int seq; char *msg; }

3. Define function makeframes():

   - Initialize variables i, seqno=1.

   - Allocate memory for 30 frames using malloc.

   - Loop to create 30 frames:

     - Set type as DATA, sequence number alternating between 0 and 1.

     - Assign a fixed message string to each frame.

     - Calculate and assign the length of the message.

   - Return the array of frames.

4. Main function:

   - Initialize variables sockfd, newsockfd, num, n, clientAdrLen, k.

   - Initialize structures serverAddress, clientAddress, f.

   - Set timeout for socket operations using timeval structure.

   - Create a socket sockfd using socket().

   - Bind the socket to serverAddress using bind().

   - Listen for incoming connections using listen().

5. Frame Transmission Loop:

- Loop until 30 frames are sent or the user-defined num frames are sent:
  - Accept a new connection and get client details.
  - Create an array of frames using makeframes().
  - Input the number of frames to send (num).
  - Initialize count to track sent frames and expAck for expected acknowledgments.
  - Loop until count reaches num:
    - Set receive timeout for newsockfd using setsockopt().
    - Get the next frame f from the array of frames.
    - Format the frame data into msg (sequence number, length, message).
    - Send the frame msg to the client using send().
    - Print sent frame information.
    - Receive acknowledgment msg1 from the client using recv().
    - Check for acknowledgment and update count if ACK is received correctly.
  - Close the connection after sending all frames.
  - Print transfer success message and break if all frames sent.

6. Cleanup and Return:

- Close sockets and free allocated memory.
- Return 0 to indicate successful execution.

GO BACK N ARQ PROTOCOL

Client

1. Include necessary libraries:

- stdio.h, stdlib.h, sys/socket.h, sys/types.h
- netinet/in.h, sys/time.h, sys/wait.h, string.h
- unistd.h, arpa/inet.h

2. Main Function:

  a. Create a socket `c_sock` using socket() for TCP communication.

  b. Initialize a sockaddr_in structure `client` for client address.

    - Set address family (AF_INET), port (9009), and IP address ("127.0.0.1").

  c. Connect to the server using connect() with the specified client address.

    - Print "Connection failed" if connection is unsuccessful.

3. Communication Loop:

  a. Print a message indicating the client has started with an individual acknowledgment scheme.

  b. Initialize message strings `msg1` and `msg2`, and a buffer `buff`.

  c. Set flags `flag` and `flg` to 1 for loop control.

  d. Loop from 0 to 9 (inclusive) for 10 frames:

    i. Reset buffer and msg2 using bzero() to clear previous data.

    ii. Read data from the server into buffer `buff`.

    iii. Check if the received frame matches the expected frame (i).

      - If not, discard the frame, print a discard message, and decrement loop index (i) to recheck.

    iv. If the frame matches:

      - Print the received message from the server.

      - Construct an acknowledgment message (`msg2`) based on the received frame number.

      - Send the acknowledgment message back to the server using write().

4. Close Connection and Return:

  a. Close the socket `c_sock` after the loop completes.

  b. Return 0 to indicate successful execution.

Server

1. Include necessary libraries:

  - stdio.h, stdlib.h, sys/socket.h, sys/types.h, sys/time.h

  - netinet/in.h, string.h, unistd.h, arpa/inet.h, fcntl.h

2. Main Function:

a. Create a TCP socket `s_sock` for server communication using `socket(AF_INET, SOCK_STREAM, 0)`.

b. Set up server address (`server`) details: port (9009) and IP address (INADDR_ANY).

c. Bind the socket to the server address using `bind()`. Print "Binding failed" if unsuccessful.

d. Start listening for incoming connections using `listen(s_sock, 10)` with a backlog of 10 connections.

e. Accept an incoming connection and create a new socket `c_sock` for communication with the client.

3. Communication Loop:

a. Initialize variables and structures for managing timeouts (`timeout1`, `timeout2`, `set1`, `set2`).

b. Prepare messages (`msg`) to send to the client with a frame number appended.

c. Implement a reliable transmission protocol using go-back-N:

i. Send multiple messages in a sequence to the client using `write()` with appropriate delays (`usleep()`).

ii. Use `select()` to monitor the client socket for incoming data or timeout events.

iii. Handle timeouts by retransmitting appropriate frames based on the protocol's requirements.

iv. Continue sending messages until all are successfully transmitted and acknowledged.

4. Timeout Handling:

a. Use `select()` with appropriate timeout values (`timeout1`, `timeout2`) to monitor socket activity and manage timeouts.

b. If a timeout occurs, retransmit the appropriate frames based on the protocol's requirements.

c. Continue the communication loop until all messages are successfully transmitted and acknowledged.

5. Closure:

a. Close the client and server sockets (`c_sock`, `s_sock`) after communication completion using `close()`.

b. Return 0 to indicate successful execution and termination of the program.

CONCURRENT FILE SERVER

Client

1. Initialize Constants and Libraries

   - Include necessary header files for socket programming.

   - Define constants PORT for the server port number and SIZE for buffer size.

2. Define writeFile Function

   - Open a file in write mode to store received data (output.txt).

   - Continuously receive data from the server until no more data is available.

   - Write received data to the file.

   - Close the file once file transfer is complete.

3. Define main Function

   - Create a client socket using socket() function.

   - Initialize server address structure (serverAddr) with server IP, port, and address family.

   - Connect to the server using connect() function.

   - Prompt the user to input the name of the file to request from the server.

   - Send the file name to the server using send() function.

   - Call the writeFile function to handle file data transfer.

   - Close the client socket after file transfer completion.

4. Client Workflow

   - Create a socket and connect to the server.

   - Send the desired file name to the server.

   - Receive file data from the server and write it to a local file.

5. Server Side

   - Implement a corresponding server program to listen for client connections.

   - Receive the file name from the client.

   - Open the requested file and send its contents back to the client.

   - Handle multiple client connections concurrently if necessary.

6. Error Handling

   - Check for errors during socket creation and connection.

   - Handle errors gracefully using perror() and exit() functions.

7. User Interaction

   - Prompt the user to input the file name they want to request from the server.

- Ensure proper input validation and handling to avoid buffer overflow or incorrect file names.

8. Network Communication

- Use TCP sockets (SOCK_STREAM) for reliable data transfer.

- Utilize socket functions (send, recv) for sending and receiving data between client and server.

Server

1. Include Necessary Libraries

- Include standard C libraries for I/O operations, socket programming, process management, and network-related functionalities.

2. Define Constants and Variables

- Define constants such as PORT for the server port number and SIZE for buffer size.

- Declare variables for socket descriptors, process IDs, file names, buffers, file pointers, and client connection count.

3. Define sendFile Function

- Accepts a file pointer (fp) and a socket file descriptor (sockfd).

- Reads data from the file in chunks and sends it to the connected client using the socket.

- Sends a message containing the server process ID to the client after file transfer completion.

4. Main Function Execution

- Create a server socket using socket() with AF_INET and SOCK_STREAM parameters.

- Bind the server socket to a specific port (PORT) and IP address (127.0.0.1 in this case) using bind().

- Listen for incoming client connections using listen() with a backlog queue size of 10.

5. Accept Client Connections in a Loop

- Continuously accept client connections using accept() within a loop.

- Fork a child process to handle each client connection independently.

6. Child Process Handling

- Close the parent socket in the child process as it is not needed.

- Receive the file name sent by the client using recv().

- Check if the requested file exists using access().

  - If the file exists, open it in read mode and call the sendFile function to send its contents to the client.

  - If the file doesn’t exist, inform the client and send a message with the server process ID.

7. Client Interaction

   - Communicate with the client using socket functions (send(), recv()).

   - Send file data or error messages as appropriate.

8. Error Handling

   - Handle errors related to socket creation, binding, accepting connections, and file access gracefully.

   - Use perror() to print error messages for debugging and troubleshooting.

9. Cleanup and Exit

   - Close open file pointers and sockets before exiting the program.

LEAKY BUCKET ALGORITHM

1. Initialize variables:

   - outrate: Constant output rate (Bytes/sec)

   - drop: Number of dropped packets

   - bsize: Bucket size (maximum amount of data the bucket can hold)

   - rem: Remaining bytes in the bucket

   - nsec: Number of seconds

   - input[20]: Array to store input packet sizes

2. Input Parameters:

- Get user input for bsize (Bucket Size) and outrate (Output Rate).

- Prompt the user to enter the size of packets arriving at each second and store them in the input array.

3. Bucket Operations Loop:

- Loop through each second (i) and process incoming packets.

4. Packet Handling:

- Calculate the amount of data to be sent (sent) based on the current bucket state (rem) and input packet size (input[i]).

5. Packet Arrival and Output:

- Check if adding the incoming packet size to rem exceeds bsize.

  - If yes, calculate sent based on outrate and update rem and drop accordingly.

  - If no, update rem based on input packet size and outrate.

6. Output Display:

- Display time received, packet sent, dropped packets, and remaining bytes for each second.

- Continue processing until all input packets are processed and rem becomes zero.

7. Output Rate Regulation:

- The algorithm regulates the output rate (outrate) to control packet flow from the bucket.

- Dropped packets occur when the incoming packet size plus current rem exceeds bsize.

8. Algorithm Completeness:

- Ensures the bucket does not overflow by limiting the output rate and dropping packets when necessary.