Multiclient-Server Chat application

- **Describe the project**
    - I worked on a multiclient-server chat application built in C using socket programming. The main goal was to create a command-line chat application that allowed users on the same network to communicate
    - The application was designed to handle multiple clients concurrently, enabling seamless communication between users. Users could engage in peer-to-peer messaging, allowing direct communication between two individuals. The application supported broadcast functionality, enabling a single user's message to be sent to all connected clients.
    - Users could create and participate in group chats, The application supported both permission-based and permissionless group creation, giving users the flexibility to organize chats as per their preferences.
    - I implemented features that allowed administrators to manage the group effectively. This included actions like adding and removing members, changing the admin role, and enabling admin-only chat.
    - One of the innovative aspects of the project was the integration of an abusive text classification model. If a user attempted to send abusive text more than 5 times, they were automatically blocked and removed from the group. To perform abusive text classification, I utilized the `execlp` system call to execute the Python model on the server side. This allowed the server to classify incoming messages in real-time and take appropriate actions.
    - In summary, my work involved designing and developing this chat application, incorporating advanced features like group formation, admin privileges, and abusive text classification. The integration of the abusive text classification model enhanced user experience.
- **How you managed running the Python model in C?**
    - I used execlp system call to run the Python script with required cmd args and stored the result (prediction) in a separate file. read the

    content of the file and if it is found abusive then the server would not proceed further to send the message.

- **Other ways to handle this multilanguage information exchange?**
    - Pipes: The C program can spawn the Python script as a child process and exchange data through pipes.
    - Shared memory: You can use shared memory to allow communication between the C and Python processes. Both processes can read and write to the shared memory region.
        - cons: platform dependent, needs sync mechanism
    - **Interprocess Communication Libraries:** Libraries like ZeroMQ, RabbitMQ, or gRPC can be used to establish communication between the C and Python processes.

Client Flow:

- The socket in the client should be of same type and protocol as server. We can use getaddrinfo to get these details of the server.
- **Connect()**
    - We pass the client sockefd and destination server address and port to connect. In client we don't call bind because we do not care about which port client is using for communication. we just want it to send data to correct port of server.
- I have designed the client code in such a way that it can **simultaneously able to read data coming from the server as well as send requests to the server**. There are several ways to handle this. I implemented this feature using the Parent-child process. I created a child process and saved its process id. Now, my child process is responsible only for sending data to the server and the parent is for reading data from the server response. And when a termination request is fired, the parent process will kill the child process and terminate the program.
    - Another way to implement this feature is using threads. We can create two separate threads for read and write functionality. To implement the termination functionality, we can cancel the sender thread from the receiver thread based on some conditions.
    - Why I didn’t use threads?

        - In general thread cancellation is not a really good idea. It is better, whenever possible, to have a shared flag, that is used by the threads to break out of the loop. That way, you will let the threads perform any cleanup they might need to do before actually exiting.
        - If I terminate the thread from another thread then there are chances of not cleaning the memory used by that thread. We also have to clear up the memory used threads first. Also, ***The pthreads manual page says that when the mode is set to asynchronous, the cancellation can be immediate, but the system is not required to do so. Killing a thread abruptly might leave a critical resource that must be closed properly, open.***
- **How did you handle longer input remembering that send function has a limit on bytes?**
    - First, let me describe how I handled the out-of-limit user input. I used dynamic memory allocation to store infinitely long inputs. Whenever the memory gets filled and there is still some input coming I reallocate the buffer with an extra memory block. While sending the data, I first send the length of the whole string separated by the string. On the receiving side, I will store the length in a variable and iterate accordingly to get the complete message from the sender.
- A **server** socket listens on a single port. All established client connections on that server are associated with that same listening port **on the server side** of the connection. An established connection is uniquely identified by the combination of client-side and server-side IP/Port pairs. Multiple connections on the same server can share the same **server-side** IP/Port pair as long as they are associated with different **client-side** IP/Port pairs, and the **server** would be able to handle as many clients as available system resources allow it to.

Feature Flows:

1. /active:
    - iterate over clients array and print all the clients in the array
2. /send
    - extract clientid and message from request
    - find socket of client from records

- send message to that socketfd.

3. /broadcast
    - iterate over all client in records, get their socketfd and send message to them
4. /makegroup
    - extract all clientids from request
    - generate a unique id for group
    - add members in members array of group
    - initialize default variables of group
    - make group as active group
    - send message to all group members
5. /makegroupreq
    - extract all clientids from request
    - generate a unique id for group
    - send request to clients to join group
    - and store request count and clientids to whom request has been sent
    - mark group as inactive for now
6. /joingroup
    - check whether request is sent to this client or not
    - if yes then add him to group by checking group limit also
    - send a confirmation message
    - decrease the request count and remove clientid from pending request array
7. /declinergroup
    - check whether request is sent to this client or not
    - decrease the request count and remove clientid from pending request array
    - send a confirmation message to admin about decline request

8. /makegroupbroadcast
    - enable the toggle variable of group structure for broadcast
9. /sendgroup
    - extract message from request
    - iterate over clientid and send message to all using their sockfd
10. /addadmin
    - check whether groupid and clientid exist or not
    - check whether he is already admin or not
    - check whether caller client is admin or not
    - else add the client to admin array and increment admin count
11. /addtogroup
    - check whether groupid and clientid exist or not
    - check whether caller client is admin or not
    - else add the client to members array and increment groupsize
12. /removefromgroup
    - check whether groupid and clientid exist or not
    - check whether caller client is admin or not
    - else remove the client to members array and decrement groupsize
    - check whether group is empty or not, if yes then deactivate the group
13. \activegroups
    - iterate over all groups and check whether this client is member in it or not
    - if yes then print details of group
14. \makeadminreq
    - created a separate structure to store admin requests
    - added grpid, requester id, admin count of that group
    - send approval request to each admin of group
15. \approveadminreq

- extract grpid and clientid from request and check whether the request for admin exist or not
- if yes then increase the poscount by 1
- check whether response from all admins have been received or not, if yes then change the status of request according to majority and send message to client about status

16. \declineadminreq
    - extract grpid and clientid from request and check whether the request for admin exist or not
    - if yes then increase the negcount by 1
    - check whether response from all admins have been received or not, if yes then change the status of request according to majority and send message to client about status
17. \quit
    - remove client from clients array
    - remove client from all groups he is part of
    - check whether group is empty or not, if yes then deactivate the group
    - remove it from master fd_set
18. **Ctrl+C & Ctrl+Z handler in client: ****
    - redirected the handler function to user-defined function which sends \quit command to server instead of killing process

Socket Programming

- **What are sockets and where are they stored?**
    - Sockets are basically a way to communicate between programs. They are Unix file descriptors. As we know that everything in Unix is a file. So when you want to communicate with another program over the Internet you're gonna do it through a file descriptor called Socket.
    - When you open a file there's a table in the kernel where descriptors of files are stored. So, when you opened your file, you created an entry in that table. If you don't close the file (with its descriptor) the entry is never deleted

- **Types of Internet Sockets and How to create a socket?**
    - Sockets are classified according to communication properties. Processes usually communicate between sockets of the same type. However, if the underlying communication protocols support the communication, sockets of different types can communicate.
    - Stream sockets: Stream sockets are reliable two-way connected communication streams. If you output two items into the socket in the order "1, 2", they will arrive in the order "1, 2" at the opposite end. They will also be error-free. HTTP, TelNet and SSH applications use stream socket
    - Datagram sockets: if you send a datagram, it may arrive. It may arrive out of order. If it arrives, the data within the packet will be error-free. They are generally used when a few dropped packets here and there doesn't mean the end of the Universe. Apps for video streaming, online gaming, audio streaming etc. use this type of socket. These sockets are useful when your goal is speed
- **Describe briefly the working of TCP protocol**
    - connection-oriented protocol that provides a stream of data communication between two devices. Before data transfer begins, a connection is established between the sender (client) and the receiver (server) through a process called the three-way handshake. This involves three steps: SYN, SYN-ACK, and ACK, ensuring both parties are ready for communication.
    - Once the connection is established, data is sent in the form of a stream of bytes. TCP ensures reliable delivery by dividing the data into segments, assigning sequence numbers, and maintaining a sliding window mechanism for flow control.
    - Upon receiving data segments, the receiver sends acknowledgments (ACKs) back to the sender. If a sender doesn't receive an ACK within a certain time, it assumes the segment was lost and retransmits it.
    - TCP uses a sliding window mechanism to manage the rate of data transmission based on the receiver's capacity. This prevents data overload and ensures efficient communication.
    - TCP guarantees that data segments are delivered in the same order they were sent. Duplicate segments are detected and discarded.

    - When the data transfer is complete, a connection termination process occurs using a four-way handshake (FIN, ACK-FIN, ACK, FIN), ensuring a graceful closure of the connection.
- **Describe briefly the working of the UDP protocol**
    - a connectionless, lightweight protocol that provides fast and low-overhead data communication between devices. Data is divided into small units called datagrams and sent as independent packets. Each datagram includes source and destination port numbers, as well as a checksum for error detection. UDP does not guarantee delivery, ordering, or duplication handling. Due to its minimal error checking and lack of acknowledgment mechanism, UDP has lower overhead compared to TCP. UDP supports broadcasting and multicasting.
- **ByteOrder of storing data of different processor manufacturers is different. How do you handle this issue while communicating between devices?**
    - You just get to assume the Host Byte Order isn't right, and you always run the value through a function to set it to Network Byte Order.
    - Basically, you'll want to convert the numbers to Network Byte Order before they go out on the wire and convert them to Host Byte Order as they come in off the wire.
    - All network byte order is in big-endian format. If your host's memory computer architecture is in little-endian format, the `htons()` function becomes necessary but in the case of big-endian format memory architecture, it is not necessary.
- **How do you check the endian system of your computer?**

```
int x = 1;
cout<<*(char*)&x<<endl;
if (*(char*)&x) {
    cout << "Little Endian\n";
} else {
    cout << "Big Endian\n";
}
```

  - Printing the least significant byte (8 bits) of 1 by type-casting it as a character. if it’s 00000001 then it’s a little-endian system.

- **Use of addrinfo struct and getaddrinfo()**
  - `struct addrinfo` is a C structure that **holds information about a particular network address.** It contains various fields that describe the address, such as the address family (IPv4 or IPv6), socket type, and protocol.
  - useful when you need to create or bind a socket to a specific address.
  - `getaddrinfo()` is a function that **retrieves a list of** `struct addrinfo` **structures based on the provided host and service names.** It is used to convert human-readable host and service information into **a set of socket addresses** that can be used in socket functions like `socket()`, `bind()`, `connect()`, etc.
  - We pass **hints** to this function like Address family, socket type, flags, etc. to search according to our need
  - `getaddrinfo()` returns 0 on success and sets up the `res` parameter with a linked list of suitable address structures.
  - Once we get a list of suitable service provider addresses, we check for a free socket in it. As one port can have multiple sockets so we look for a free socket where we can put our server to start communication with clients.

- **How to get local socket to bind my server with it?**
  - Pass NULL in getaddrinfo. It returns address information for the local host's network interfaces. The resulting `struct addrinfo` will contain a list of IP addresses associated with the local host, which you can then use for setting up your server's socket.

- **What is address family?**
  - The address family is the type of IP address like IPv4 or IPv6

- **Explain ai_flags a bit.**
  - it is used to modify the behavior of getaddrinfo function. It provides additional instructions to the function regarding how to handle the provided input.
  - When flag=AI_PASSIVE, it indicates that the returned socket addresses will be suitable for binding a server socket for accepting incoming

connections.

- **AI_CANONNAME**: this flag indicates that you want the `ai_canonname` field of the `struct addrinfo` to be filled with the canonical name of the host. It is **the complete address of an internet host or computer**.

- **sockaddr in addrinfo.**
    - it holds the socket address information like port number, address (32-bit), and address family
    - sockaddr_in is for IPv4 only.
    - sockaddr_storage is suitable for both ipv4 and ipv6.
    - **we pass this structure during the connection acceptance function call in the server. The accept function fills up the details of the client in this structure.**
- **Is there any default encryption mechanism in C for socket programming?**
    - No, but if want to implement encryption then we can use the OpenSSL library. It provides a set of APIs for encrypted communication.
    - First, we have to create a context for OpenSSL, then configure it for desired encryption settings.
    - Then we can use SSL-specific functions to encrypt and decrypt data.
    - Close the SSL connection at the end also.
- **What is the context?**
    - the environment, settings, or conditions in which a particular piece of code operates. When dealing with network communication, the context could include information about the network configuration, such as IP addresses, ports, protocols, and security settings.
- **socket() system call**
    - It returns a file descriptor pointing to the selected socket based on args we passed. The socketFD is put into the global FD list to keep that FD allocated to current socket address until we terminate the connection and free up the socket.
- **bind system call**
    - You need to bind the socketfd with a port if you want to listen to incoming connection requests on that port.

    - If you are connecting to a remote machine and you don’t care what your local port is. So, no need to call bind() when you are trying to connect as a client.
    - **setsockopt:** Sometimes, you might notice, you try to rerun a server and bind() fails, claiming “Address already in use.” What does that mean? Well, a little bit of a socket that was connected is still hanging around in the kernel, and it’s hogging the port. You can either wait for it to clear (a minute or so), or use this function. **It Allows other sockets to bind() to this port, unless there is an active listening socket bound to the port already.**
- **Listen()**
    - These calls are useful only for creating a server. Incoming connections are going to wait in this queue until you accept() them and Listen sets up a limit for this queue.
    - The basic flow of creating a server and starting getting connection request

      1. getaddrinfo();
      2. socket();
      3. bind();
      4. listen();
- **Accept():**
    - **It’ll return to you a brand new socket file descriptor to use for this single connection**
    - The original one is still listening for more new connections, and the newly created one is finally ready to send() and recv().
    - You can **get the ip address of client** and the port of connection by using the sockaddr_in structure passed to accept function. It is very simple :

  ```
  char *client_ip = inet_ntoa(client.sin_addr);
  int client_port = ntohs(client.sin_port);
  ```
- **What if multiple clients are waiting simultaneously to connect? How does accept() function handle this?**

    - The `accept()` function is a blocking call. When there are no incoming connections to accept, it blocks and waits until at least one client attempts to connect.

        > *So we call accept() only when the listener socket is SET in read_fds*

    - We have to call accept multiple times to create connection with new sockets in the wait queue.

- **How client information is connected to socket fd?**
    - The operating system maintains data associated with each socket FD, which includes client-specific information. This information is stored in a data structure called the "socket control block" or "socket structure." This data structure contains various details about the connection, including the client's IP address, port number, connection state, communication parameters, and more.

- **how do you handle multiple clients simultaneously on the server?**
    - Multiple clients are handled concurrently with the help of multithreading. Each client operates independently, making it easier to manage client-specific logic. As soon as you accept a request, create a separate thread containing the request handler function and the newly generated socket fd in it. **Thread creation and management can be resource-intensive. Possibility of thread synchronization issues (race conditions) if not handled properly.**
    - Another approach is select and poll. The server maintains a list of active client sockets and monitors them for incoming data using `select()` or `poll()`. This approach is a bit complex to implement but efficient in terms of resource utilization.

- **Which approach is good when I have number of clients <50 , >50 and <=100000 and >100000 trying to connect concurrently?**
    - **<50:** With a relatively small number of clients (less than 50), you have more flexibility in choosing an approach. Both multithreading and select/poll can work effectively in this range.
    - **>50 and <=100000:** This range could still be managed using both multithreading and select/poll, but you might want to lean towards the

select/poll approach for better resource management. If you're open to using event-driven libraries like libevent or libuv, this could be a good time to consider them because select() can support up to 1024 socket FDs.

    - **>100000:** The event-driven approach is often favored in this scenario, as it's designed to handle a massive number of clients without consuming excessive resources. **The event-driven approach in socket programming involves using event loops and callbacks to efficiently manage multiple clients. The server waits for events (such as incoming data or new connections) and invokes predefined callbacks to handle those events.**

- **How to handle the Limit of serving the clients simultaneously?**
    - **To control the maximum number of clients that the server can simultaneously handle, you should manage this in your application's logic. The `listen()` function itself doesn't impose a direct limit on the number of clients; it's more about the server's ability to handle incoming connections.**
    - Implement mechanisms for gracefully rejecting new connections when the server is at its capacity, like sending a "server busy" message to clients or delaying new connections.
    - For example, I used the modulo operator on socket fds. I was storing currently active clients in an array of structures. If an fd is already reserved by some other client then I deny the connection at that time. Now, you might have a question what if any other place is empty in array??? But this scenario will not occur because FDs are allocated in the least as the first manner. That means if 4,5,6 FDs are empty then 4 will be allocated first. And when a socket gets freed up, then a new socket will get the FD which is the least of all the free FDs.

Multiple ways to handle clients concurrently:

- **Making sockets Non-Blocking and then polling:**
    - accept(), recv, recvfrom, etc. functions block the OS till some data arrives. Because the kernel sets the socket to blocking by default while creating. **In all (good) production applications non-blocking sockets are used.**

    - So, we can set the socket to non-blocking while creating it and then poll the socket for information. If no data is found then it won't block but return -1. This is called asynchronous programming
    - This type of polling is a bad idea because the program is in a **busy-wait** state. It will take up heavy CPU time. One better approach can be using poll() or select()
- **Poll(): IMP: It is used for Synchronous I/O multiplexing**
    - We want to somehow monitor all the sockets at once and then handle the ones which have data ready. One way to do this use poll() function. The operating system does all the dirty work for us, and just let us know when some data is ready to read on which sockets. In the meantime, our process can go to sleep.
    - We keep track of active sockets in an array of structs that contain socketfd, events to look for (ready to read/write), and events that occurred.
    - **The OS will block on the poll() call until one of those events occurs (e.g. "socket ready to read!") or until a user-specified timeout occurs.**
    - If we want our server to keep running until a single ready socket is encountered, we pass the timeout as -1.
    - While creating a server, we put its socket in the poll with POLLIN status (i.e. ready to read state) so that it can accept the new connection. For the other sockets, we will handle them as a normal client. Whenever we get a new client in the listener socket, we add it to our polling fdlist.
- **Select(): IMP: It is used for *Synchronous I/O* multiplexing**
    - this function is useful when you want to listen to new connections as well as keep reading from existing connections also. It allows a program to monitor multiple file descriptors (including sockets) for activity, such as data to read or write, within a specified time interval. This enables a single thread of execution to manage multiple I/O operations without blocking on each individual operation.
    - So, select blocks until at least one socket is ready to use but it does not block on any specific socket for some operation or event.
    - Select() maintains sets of read and write socketfds. On calling the select(), it will modify the socket status of sockets present as "Ready to

read" in readfd set. We can check it using FD_ISSET()

> ***sometimes, in rare circumstances, Linux's select() can return "ready-to-read" and then not actually be ready to read! This means it will block on the read() after the select() says it won't! Why you little—! Anyway, the workaround solution is to set the O_NONBLOCK flag on the receiving socket so it errors with EWOULDBLOCK***

- master fds: the master list to store all FDs. Whenever a new socketfd is generated, add it to masterfd. On closing the connection, clear this socket from the master list
    - *The reason I have the master set is that select() actually changes the set you pass into it to reflect which sockets are ready to read. Since I have to keep track of the connections from one call of select() to the next, I must store these safely away somewhere.*
- fd_set read_fds: set to store socketfd which are ready to read after select() call
- FD_SET: set the socket status to ready to do ops
- fd_max: maximum socketfd number seen till now
- FD_ISSET(): check if an fd is set or not
- **Send() and Recv() - For Stream sockets**
    - send() return the actual size of the sent information and if it mismatches with len then we need to send the remaining data again
    - recv() can **return 0**. This can mean only one thing: the remote side has **closed the connection** on you.
- Close and shutdown
    - Close cuts off communication in both ways (i.e. send and receive)
    - in case you want a little more control over how the socket closes, you can use the shutdown() function. It allows you to cut off communication in a certain direction (either send or received), or both ways
    - **shutdown() doesn't actually close the file descriptor—it just changes its usability. To free a socket descriptor, you need to use close().**

- **How do you handle partially sent messages?**
    1. Keep track of returned byte count from send() function and check whether it is equal to actually sent bytes.
- **If the packets are variable length, how does the receiver know when one packet ends and another begins?**
    - For variable-length packets, determining where one packet ends and another begins requires additional mechanisms like,
        - Delimiter-based framing: using a special delimiter sequence to mark the end of each packet.
        - Length-prefix: sender adds a prefix to each packet that indicates the length of the packet data.

▼ CODE EXAMPLE

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

int main() {
    char buffer[1024];  // Buffer to hold incomin
    int bytes_received = 0;
    int expected_packet_length = -1;

    while (1) {
        // Receive data and append to buffer
        // bytes_received = receive_data(buffer +

        // Process incoming data
        while (bytes_received > 0) {
            // If we don't know the expected pack
            if (expected_packet_length == -1) {
                if (bytes_received >= sizeof(int)
                    memcpy(&expected_packet_lengt
                    memmove(buffer, buffer + size
                    bytes_received -= sizeof(int)
                } else {
                    break;  // Not enough data fo
                }
            }

            // Check if we have received the comp
            if (bytes_received >= expected_packet
                // Process the packet (buffer con
                // ...

                // Remove processed packet from b
                memmove(buffer, buffer + expected
                bytes_received -= expected_packet
                expected_packet_length = -1;  //
            } else {
                break;  // Packet is not yet comp
            }
        }
    }

    return 0;
}
```

- **Ways to send int, float, and binary data?**
    1. Convert them to string and send ⇒ slow to convert
    2. Send raw data, directly passing the pointer to send() ⇒ easy but dangerous *(not all architectures represent a double (or int for that matter) with the same bit representation or even the same byte ordering)*
    3. **Encode the number into a portable binary form. The receiver will decode it. *(Good choice for a portable program)***
        a. See page no. 53 on been guide for coding example
- **How does the client know when one message starts and another stops?**
    - The problem is that the messages can be of varying lengths.
    - One non-efficient solution is making all messages of the same byte length regardless of actual length. But it’s a waste of bandwidth.

- So we encapsulate the data in a tiny header and packet structure. Both the client and server know how to pack and unpack.
- When you're sending this data, you should be safe and use a command similar to sendall(), above, so you know all the data is sent, even if it takes multiple calls to send() to get it all out.
- when you're receiving this data, you need to do a bit of extra work. To be safe, you should assume that you might receive a partial packet. We need to call recv() over and over again until the packet is completely received.
    - ***But how?*** Well, we know the number of bytes we need to receive in total for the packet to be complete since that number is tacked on the front of the packet. We also know the maximum packet size. you know every packet starts off with a length, you can call recv() just to get the packet length. Then once you have that, you can call it again specifying exactly the remaining length of the packet possibly repeatedly to get all the data) until you have the complete packet.

Communication through datagram sockets

- The flow of communication using Datagram sockets

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

int main() {
    // Create a socket
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) {
        perror("socket");
        return 1;
    }

    // Specify server address
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(12345);  // Specify the d
    server_addr.sin_addr.s_addr = INADDR_ANY;  // Listen o

    // Bind the socket
    if (bind(sockfd, (struct sockaddr *)&server_addr, size
        perror("bind");
        close(sockfd);
        return 1;
    }

    // Receive and process data
    char buffer[1024];
    struct sockaddr_in client_addr;
    socklen_t client_addr_len = sizeof(client_addr);

    while (1) {
        ssize_t bytes_received = recvfrom(sockfd, buffer, 
                                          (struct sockaddr

        if (bytes_received == -1) {
            perror("recvfrom");
            continue;
        }

        // Process the received data and prepare a respons
        // ...

        // Send a response
        // sendto(sockfd, response_message, response_lengt
    }

    // Close the socket
    close(sockfd);

    return 0;
}
```

- Setting up queue limit:

> *if you connect() a datagram socket, you can then simply use send() and recv() for all your transactions. The socket itself is still a datagram socket and the packets still use UDP, but the socket interface will automatically add the destination and source information for you.*

- **sendto() and recvfrom() - For Datagram sockets**
    - Two extra params: Destination sockaddr struct and size of this struct