

Members: Eden Espinosa and Tra-Mi Cao
CSC 345 - 02
Dr. Yoon
Due: May 9, 2025

Project 4 Report

*** Check Point 1: Make sure your program smoothly runs without any hic-ups for the five requirements above before trying the harder requirement below. ***

Checkpoint completed

```
tcnj@tcnj-osc:~/work/project4/CSC345Project4$ ./main_server
Server started on port 3000
Connected: 127.0.0.1
Connected: 127.0.0.1
alice (127.0.0.1) joined the chat room!
Connected: 127.0.0.1
Connected: 127.0.0.1
bob (127.0.0.1) joined the chat room!
```

```
tcnj@tcnj-osc:~/work/project4/CSC345Project4$ ./main_client 127.0.0.1
Try connecting to 127.0.0.1...
Available chat rooms:
Room 1: 1 people
```

```
Choose the room number or type [new] to create a new room: 1
Type your user name: bob
Connected to 127.0.0.1 with room number 1
bob joined the chat room!
[alice] hi bob
what's up alice?
```

```
Choose the room number or type [new] to create a new room: new
Type your user name: alice
Connected to 127.0.0.1 with room number 1
alice joined the chat room!
hi bob
[alice] what's up bob?
```

*** Check Point 2: Make sure your program smoothly runs without any hic-ups for the requirement above before trying the harder requirement below. ***

Checkpoint completed

The screenshot below shows a correctly-implemented ‘new’ command and entering an occupied chat room.

tcnj@tcnj-osc: ~/work/project4/CSC345Project4

Server started on port 3000
Connected: 127.0.0.1
Connected: 127.0.0.1
alice (127.0.0.1) joined the chat room!
Connected: 127.0.0.1
Connected: 127.0.0.1
bob (127.0.0.1) joined the chat room!
alice (127.0.0.1) left the room!
bob (127.0.0.1) left the room!
Connected: 127.0.0.1
dog (127.0.0.1) joined the chat room!
Connected: 127.0.0.1
cat (127.0.0.1) joined the chat room!

tcnj@tcnj-osc: ~/work/project4/CSC345Project4

tcnj@tcnj-osc: ~/work/project4/CSC345Project4 64x24

tcnj@tcnj-osc:~/work/project4/CSC345Project4\$./main_client 127.0.0.1 2
Try connecting to 127.0.0.1...
Type your user name: cat
Connected to 127.0.0.1 with room number 2
cat joined the chat room!
[dog] it's raining cats and dogs!
hope you packed an umbrella!

tcnj@tcnj-osc: ~/work/project4/CSC345Project4

tcnj@tcnj-osc:~/work/project4/CSC345Project4\$./main_client 127.0.0.1 new
Try connecting to 127.0.0.1...
Type your user name: dog
Connected to 127.0.0.1 with room number 2
dog joined the chat room!
it's raining cats and dogs!
[cat] hope you packed an umbrella!

*** Check Point 3: Make sure your program smoothly runs without any hic-ups for the requirement above before trying the harder requirement below. ***

Checkpoint completed.

Below, the server lists rooms that are available.

tcnj@tcnj-osc: ~/work/project4/CSC345Project4

Connected: 127.0.0.1
Connected: 127.0.0.1
Connected: 127.0.0.1
george (127.0.0.1) joined the chat room!
Connected: 127.0.0.1
Connected: 127.0.0.1
Connected: 127.0.0.1
Connected: 127.0.0.1
john (127.0.0.1) joined the chat room!
john (127.0.0.1) left the room!
Connected: 127.0.0.1
Connected: 127.0.0.1
john (127.0.0.1) joined the chat room!

tcnj@tcnj-osc: ~/work/project4/CSC345Project4

tcnj@tcnj-osc: ~/work/project4/CSC345Project4 64x24

tcnj@tcnj-osc:~/work/project4/CSC345Project4\$./main_client 127.0.0.1
Try connecting to 127.0.0.1...
Available chat rooms:
Room 3: 1 people

Choose the room number or type [new] to create a new room: 3
Type your user name: john
Connected to 127.0.0.1 with room number 3
john joined the chat room!
[george] hi john
how are you george?

tcnj@tcnj-osc: ~/work/project4/CSC345Project4

tcnj@tcnj-osc:~/work/project4/CSC345Project4\$./main_client 127.0.0.1
Try connecting to 127.0.0.1...
No rooms available. Type 'new' to create one.

Choose the room number or type [new] to create a new room: new
Type your user name: george
Connected to 127.0.0.1 with room number 3
george joined the chat room!
hi john
[john] how are you george?

Check Point 4 in the report.pdf. Do not attempt the extra item below unless you reached this point without any hic-ups.

Checkpoint completed.

Sending the file on Alice's end, and receiving the file on Bob's end:

The image shows four terminal windows arranged in a 2x2 grid, illustrating the file transfer process between Alice and Bob.

- Top Left Terminal:** Shows the server side (main_server) running on port 3000. It logs connections from Alice (127.0.0.1) and Bob (127.0.0.1). Alice sends a file transfer request to Bob, and Bob accepts it.
- Top Right Terminal:** Shows the client side (main_client) connecting to Alice's server at 127.0.0.1. Bob receives a file transfer request from Alice, accepts it, and saves the file as hello_world_1.bin.
- Bottom Left Terminal:** Shows Alice connecting to Bob's server at 127.0.0.1. Alice sends a file transfer request to Bob, and Bob accepts it.
- Bottom Right Terminal:** Shows Bob accepting the file transfer from Alice, saving the file as hello_world_1.bin, and responding with "enjoy!"

Below, Alice is in Room 1, and Doug is in Room 2. Doug cannot send Alice a file because they are in different rooms.

The image shows four terminal windows arranged in a 2x2 grid, illustrating a failed file transfer attempt between Alice and Doug.

- Top Left Terminal:** Shows Alice connected to the server at 127.0.0.1. Alice sends a file transfer request to Doug, and Doug accepts it.
- Top Right Terminal:** Shows Doug connecting to Alice's server at 127.0.0.1. Doug receives a file transfer request from Alice, but the transfer fails because Doug is in a different room (Room 2).
- Bottom Left Terminal:** Shows Alice connecting to Doug's server at 127.0.0.1. Alice sends a file transfer request to Doug, and Doug accepts it.
- Bottom Right Terminal:** Shows Doug accepting the file transfer from Alice, saving the file as hello_world_1.bin, and responding with "enjoy!"

AI Usage

With the short deadline of the project as well as balancing other projects due so late, and with the suggestion from a friend, we tried using deepseek to help me with the completion of some parts of the code. One usage we used for it was to cut time on certain aspects of the project, such as doing more research:

I want my messages to have multiple of different colors in the console. Can you make me a list of colors i can use in the terminal?



Terminal Text Colors for Your C Chat Server

Here's a comprehensive list of ANSI color codes you can use to add colorful output to your chat server's console:

Basic 8/16 Colors

```
c
Copy Download

// Text colors
#define RED      "\x1B[31m"
#define GREEN    "\x1B[32m"
#define YELLOW   "\x1B[33m"
#define BLUE     "\x1B[34m"
#define MAGENTA  "\x1B[35m"
#define CYAN     "\x1B[36m"
#define WHITE    "\x1B[37m"

// Background colors
#define BG_RED    "\x1B[41m"
#define BG_GREEN  "\x1B[42m"
#define BG_YELLOW "\x1B[43m"
#define BG_BLUE   "\x1B[44m"
#define BG_MAGENTA "\x1B[45m"
#define BG_CYAN   "\x1B[46m"
#define BG_WHITE  "\x1B[47m"

// Bright variants
#define BRIGHT_RED    "\x1B[91m"
#define BRIGHT_GREEN  "\x1B[92m"
#define BRIGHT_YELLOW "\x1B[93m"
#define BRIGHT_BLUE   "\x1B[94m"
#define BRIGHT_MAGENTA "\x1B[95m"
#define BRIGHT_CYAN   "\x1B[96m"
#define BRIGHT_WHITE  "\x1B[97m"

// Reset to default
#define RESET     "\x1B[0m"
```

256-Color Mode (Extended Palette)

```
c
Copy Download

// Format: \x1B[38;5;<color_code>m (text) or \x1B[48;5;<color_code>m (background)

// Some popular 256-color codes:
#define ORANGE_256    "\x1B[38;5;208m"
#define PURPLE_256    "\x1B[38;5;129m"
#define PINK_256       "\x1B[38;5;205m"
#define LIME_256       "\x1B[38;5;118m"
#define TEAL_256       "\x1B[38;5;30m"
#define GOLD_256       "\x1B[38;5;220m"
```

RGB Colors (True Color)

RGB Colors (True Color)

```
c                                     ⌂ Copy ⌂ Download

// Format: \x1B[38;2;<r>;<g>;<b>m (text) or \x1B[48;2;<r>;<g>;<b>m (background)

// Examples:
#define CUSTOM_RGB(r,g,b)      "\x1B[38;2;" #r ";" #g ";" #b "m"
#define CUSTOM_BG_RGB(r,g,b)   "\x1B[48;2;" #r ";" #g ";" #b "m"

// Predefined RGB colors
#define DEEP_BLUE    "\x1B[38;2;0;0;128m"
#define NEON_GREEN   "\x1B[38;2;57;255;20m"
#define HOT_PINK     "\x1B[38;2;255;105;180m"
```

Text Attributes

```
c                                     ⌂ Copy ⌂ Download

#define BOLD      "\x1B[1m"
#define DIM       "\x1B[2m"
#define UNDERLINE "\x1B[4m"
#define BLINK     "\x1B[5m"
#define REVERSE   "\x1B[7m" // Invert foreground/background
#define HIDDEN    "\x1B[8m"
```

Usage Examples

```
c                                     ⌂ Copy ⌂ Download

printf(RED "Error message" RESET "\n");
printf(GREEN BG_YELLOW BOLD "Warning!" RESET "\n");
printf(BRIGHT_CYAN "Client connected" RESET "\n");
printf(ORANGE_256 "Custom color message" RESET "\n");
printf("\x1B[38;5;214mHello in orange\x1B[0m\n");
printf("\x1B[38;2;255;165;0mTrue color orange\x1B[0m\n");
```

Important Notes

1. **Reset after each colored string** with `RESET` to prevent color bleeding
2. **Terminal compatibility:**
 - Basic 16 colors work in nearly all terminals
 - 256-color mode works in most modern terminals
 - True color (RGB) requires newer terminals (iTerm2, GNOME Terminal, etc.)
3. **Windows** requires enabling ANSI support or using alternative methods
4. **Combine attributes:**

```
c                                     ⌂ Copy ⌂ Download

printf(BOLD BLUE UNDERLINE "Fancy text" RESET "\n");
```

Would you like me to show you how to integrate these colors with your chat server's message handling system? For example, you could color-code different types of messages (errors, status updates, chat messages from different clients, etc.).

We asked for it to give preset colors we can just add to the code. So we were able to strip it for some needed colors.

```
#define RED    "\x1B[31m"
#define GREEN  "\x1B[32m"
#define YELLOW "\x1B[33m"
#define BLUE   "\x1B[34m"
#define MAGENTA "\x1B[35m"
#define CYAN   "\x1B[36m"
#define BRIGHT_RED  "\x1B[91m"
#define BRIGHT_GREEN "\x1B[92m"
#define BRIGHT_YELLOW "\x1B[93m"
#define BRIGHT_BLUE  "\x1B[94m"
#define BRIGHT_MAGENTA "\x1B[95m"
#define BRIGHT_CYAN  "\x1B[96m"
```

This worked great in the code, and saved me a lot of time getting these manually.

We've also ended up using it for debugging purposes

when I test the code. The client will connect to the server, but it will send a message. It will not send the first message, but any message following send/work fine. Why is this?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <pthread.h>

#define PORT_NUM 1004

#define RESET "\x1B[0m"
#define RED "\x1B[31m"
#define GREEN "\x1B[32m"
#define YELLOW "\x1B[33m"
#define BLUE "\x1B[34m"
#define MAGENTA "\x1B[35m"
#define CYAN "\x1B[36m"
```

<Message includes the rest of the code for this file>

This issue was particularly having trouble with as a client would connect to a room, then it automatically boots them on their first attempt.

The issue where the first message isn't being sent but subsequent messages work fine appears to be related to how the server handles the initial connection setup and message flushing. Here's the problem and solution:

The Problem

1. In the server's `thread_main` function, after accepting a new connection and setting up the room, there's this line:

```
c Copy Download
char flush_buf[256];
recv(clisockfd, flush_buf, sizeof(flush_buf), MSG_DONTWAIT);
```
2. This non-blocking receive is meant to flush any leftover data, but it might actually be consuming the client's first message before it can be properly processed.
3. The client sends its first message normally, but the server discards it with this flush operation.

The Solution

Remove the flush operation entirely. It's not necessary in this case and is causing the first message to be lost. Here's how to fix it:

In the server code (`thread_main` function), remove or comment out these lines:

```
c Copy Download
char flush_buf[256];
recv(clisockfd, flush_buf, sizeof(flush_buf), MSG_DONTWAIT);
```

Additional Improvements

1. In the client's receive thread (`thread_main_recv`), you're calling `recv()` twice - once before the loop and once inside. This could also cause message loss. Change it to:

```
c Copy Download
void *thread_main_recv(void *args)
{
    pthread_detach(pthread_self());
    int sockfd = ((ThreadArgs *)args)->clisockfd;
    free(args);

    char buffer[512];
    int n;

    while ((n = recv(sockfd, buffer, 512, 0)) > 0)
    {
        buffer[n] = '\0'; // Null-terminate the received data
        print_colored_message(buffer);
        memset(buffer, 0, 512);
    }

    if (n < 0)
        error("ERROR recv() failed");

    return NULL;
}
```
2. In the server's broadcast function, consider adding error checking for the `send()` calls:

```
c Copy Download
if (send(cur->clisockfd, buffer, strlen(buffer), 0) < 0) {
    perror("send failed");
    // Handle error (maybe remove the client)
}
```

These changes should ensure that:

1. The first message from the client isn't accidentally discarded
2. Message receiving is more reliable
3. Error conditions are better handled

The core issue was that unnecessary "flushing" of the socket was actually consuming the first real message from the client. Removing that operation allows all messages to be processed properly.

However the solutions it provided weren't very helpful.

i removed these two lines

```
char flush_buf[256];
recv(clisockfd, flush_buf, sizeof(flush_buf), MSG_DONTWAIT);
```

It seems to work fine, but the first message does not work



If removing the flush didn't fully fix the issue and the **first message still isn't working**, the problem likely lies in **how the server and client synchronize their initial communication**. Here's why this might happen and how to fix it:

Root Cause

1. Race Condition in Client Setup

- The client sends:
 1. Room number (first `send`)
 2. Username (second `send`)
- The server expects:
 1. Room number (first `recv`)
 2. Username (second `recv`)
- If the server isn't immediately ready to `recv` the username, the client might send the first message before the server is fully set up.

2. Buffering in TCP

- TCP may combine small packets (`room_number`, `username`, and first message) into one, causing the server to misread them.

Solution: Proper Synchronization

1. Server-Side Fix (Best Approach)

After receiving the username, **send an acknowledgment (ACK) to the client before proceeding**:

```
c
// After receiving username in server's thread_main:
send(clisockfd, "ACK", 3, 0); // Tell client it's ready
```

Then, modify the client to **wait for this ACK** before sending messages:

```
c
// In client's main(), after sending username:
char ack[4] = {0};
recv(sockfd, ack, 3, 0); // Wait for server's "ACK"
if (strcmp(ack, "ACK") != 0) {
    error("Server did not acknowledge");
}
printf("> "); // Now the client can start chatting
```

2. Alternative: Small Delay in Client (Quick Fix)

If you want a quick (but less reliable) fix, add a small delay after sending the username:

```
c
// In client's main(), after sending username:
usleep(100000); // 100ms delay (not ideal, but works for testing)
printf("> ");
```

Here we tried again after the initial suggestion failed. we tried different methods however all failed. we believe that some of it is still left in with usleep(100000); However, this attempt was unhelpful for the overall process of the code. It was later solved because our socket was prematurely closing with close(clisockfd); the position of it was closing it prematurely so it was moved.

I would like the client to connect to the server without a room number, be presented with a list of open rooms and let the user choose one or type 'new' to create a new one. How would I go about accomplishing that cleanly in the client code without getting in the way of the initial handshake with the server?

Do not give me code, but provide pointers where I should go.



Here's a structured approach to achieve this cleanly while maintaining the handshake flow:

Key Pointers for the Client-Side Logic:

1. Special Initial Request Signal:

- Use a reserved value (e.g., `-2` or `"LIST"`) as the initial "room number" to signal the server that the client wants to fetch available rooms. This keeps the handshake intact while branching server behavior early.

2. Phased Connection Workflow:

◦ Phase 1 (Fetch Rooms):

- Connect to the server, send the special signal, and wait for the server's response (list of rooms).
- Server should close the connection after sending the list (design assumption).

◦ Phase 2 (User Choice):

- Display the room list to the user, prompt for input (existing room number or `"new"`).
- Validate the input (numeric for existing rooms, or `"new"` for a new room).

3. Graceful Reconnection:

- After the user selects a room (or `"new"`), reconnect to the server and follow the standard handshake:
 - Send the chosen room number (or a "new room" signal).
 - Send the username.
- Reuse the same connection logic for both initial fetch and final join to avoid duplication.

4. Error Handling & Re-prompting:

- If the user enters an invalid room number, re-prompt them without re-fetching the list (store it temporarily).
- Handle edge cases like empty lists (e.g., prompt to create a new room automatically).

5. Server Coordination:

- Ensure the server treats the special signal (`-2`) as a distinct case:
 - Respond *only* with the room list, then close the connection.
 - Avoid mixing this logic with the standard room-join workflow.

6. User Experience (UX) Clarity:

- Clearly label prompts (e.g., `"Available rooms: [1, 3, 5] – Enter a number or 'new':"`).
- Add slight delays or visual separators between phases to avoid confusion.

Flow Summary:

1. Client → Server: `-2` (request rooms).
2. Server → Client: Sends room list, disconnects.
3. Client: Shows list, prompts user.
4. Client → Server: Reconnects, sends user's choice + username.

This keeps the handshake protocol clean while adding flexibility for room selection. No research needed—just align client/server behavior as above.



Additionally it was used as a guide for certain aspects of the code. We were stuck for a while trying to set up the different views. The goal was to allow clients to join without having to specify a room in advance and instead choose from an active list of rooms or create one dynamically.

This code was directly influenced by the guidance of AI:

```
if (argc == 2) {
    // We did not specify room; ask server for room list
    int input = -2; // tell server we want room list
    send(sockfd, &input, sizeof(int), 0);

    char list[1024];
    int n = recv(sockfd, list, sizeof(list) - 1, 0);
    if (n <= 0)
        error("ERROR receiving room list");
    list[n] = '\0';

    printf("%s\n", list);
    printf("Choose the room number or type [new] to create a new room: ");

    char choice[20];
    fgets(choice, sizeof(choice), stdin);
    choice[strcspn(choice, "\n")] = 0;

    close(sockfd);

    if (strcmp(choice, "new") == 0) {
        room_number = -1;
    } else {
        room_number = atoi(choice);
    }

    // reconnect
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR reopening socket");

    if (connect(sockfd, (struct sockaddr *)&serv_addr, slen) < 0)
        error("ERROR reconnecting");

    send(sockfd, &room_number, sizeof(int), 0);
    ...
}
```

This block directly implements the AI-based suggestion by using -2 to get room info, then handling reconnection nicely based on user selection. By relying on AI not for code, but for design thinking/planning.

For the file transfer requirement, I needed to make sure the entire file would be transferred from the sender to the receiver. I was unsure about what to expect regarding certain messages or errors that occur during this process, so I had ChatGPT list me messages that I later hard-coded into the `main_server.c` code:

My prompt:

In C, list file transfer protocol messages to check for in a client-server program where a client can send a whole file or some of the file to another client in the same chat room.

Its answer:

In a C-based client-server program for transferring files between clients in a chat room, the file transfer protocol messages can be custom-defined for your application. Here's a list of common message types you should define and check for to manage full or partial file transfers:

1. FILE_TRANSFER_REQUEST

Sent by the sender client to the server to request file transfer to another client. Includes metadata:

- Target recipient ID
- Filename
- Filesize
- Transfer type (FULL or PARTIAL)

2. FILE_TRANSFER_ACCEPT

Sent by the recipient client (via the server) to accept the file transfer.

3. FILE_TRANSFER_REJECT

Sent by the recipient client to reject the transfer.

4. FILE_TRANSFER_START

Sent by the server to notify both parties that the transfer can begin.

5. FILE_DATA

Sent from the sender client containing a chunk of file data.

- Offset (if partial)
- Length of data
- Actual data (e.g., binary blob or encoded)



6. FILE_TRANSFER_PROGRESS (optional)

Used for progress updates or acknowledgments.

7. FILE_TRANSFER_COMPLETE

Indicates the sender has finished sending the file or portion.

8. FILE_TRANSFER_ACK

Sent by the recipient after receiving a chunk, acknowledging receipt.

9. FILE_TRANSFER_ERROR

Indicates an error occurred (e.g., timeout, corruption, permission denied).

10. FILE_TRANSFER_CANCEL

Sent by either party to cancel the ongoing transfer.

Each of these should be uniquely identifiable (e.g., via enums, opcode integers, or tagged strings like "MSG::FILE_TRANSFER_REQUEST") in your protocol. You can implement them with a struct-based message format and send them over sockets using read/write or send/recv.

Would you like a sample struct-based protocol definition in C to get started?



I took some of these examples and implemented them in both the server and client code. Reflecting on my use of AI, it would've taken me longer to search for each individual message; I did however manually search for other messages that the generated list did not cover, such as FILE_TRANSFER_CHUNK.

In transferring the actual size of the file, I used AI to bounce ideas off of what I would use as the appropriate data type to use for file sizes. I ended up using type `size_t` for my variable `filesize` in my `FileTransfer` structure.

regarding file transfer in C, what type should I make the filesize if I have to account for every byte in the file?

To account for every byte in a file when transferring or reading it in C, you should use a `size_t` or `off_t`, depending on the context:

Recommended types:

- `size_t`:
 - Use this when you're working with memory buffers, like reading a file into memory.
 - It is the return type of `fread()`, and the type expected for buffer sizes.
 - It is **unsigned**, and designed to represent the size of an object in memory.
- `off_t`:
 - Use this when dealing with file sizes and offsets (especially for larger files).
 - It is the return type of `stat()` (via `st_size`) and used in functions like `lseek()`.
 - It is **signed**, to allow for negative values (e.g., seek backwards).

For the sake of time, I also asked AI how to output the contents of a variable with type `size_t`:

how do i print a variable of type size_t? My filesize variable in my FileTransfer structure is of type size_t

To print a `size_t` variable in C, you should use the `%zu` format specifier with `printf`:

```
c

#include <stdio.h>
#include <stddef.h> // for size_t

int main() {
    size_t filesize = 123456;
    printf("Filesize: %zu bytes\n",
    return 0;
}
```

Notes:

- `z` is the correct length modifier for `size_t`.
 - `%lu` might seem to work on some platforms (like 64-bit Linux), but it's not portable or guaranteed.
 - Always prefer `%zu` for portability across platforms and compilers.

Let me know if you're printing it inside a `struct` or passing it around differently.



(I noted anywhere that uses %zu as AI assisted because I have never used this format specifier before.)

Other general questions for helping foundational knowledge when coding are added below. This of course taught me the behaviors of `recv()`:

What does `recv()` return when the client leaves, and how do I deal with it in C?

Understanding `recv()` Behavior and Handling Client Disconnections

When a client disconnects from your C socket server, the `recv()` function provides specific return values that you must handle properly to maintain server stability.

`recv()` Return Values Explained

Return Value	Meaning	Typical Scenario
> 0	Number of bytes received	Normal data reception
0	Connection closed	Client performed graceful shutdown
-1 (with <code>errno</code>)	Error occurred	Client crashed, network issue, etc.

Proper Handling in Code

Here's how to properly handle all cases:

```
c
ssize_t bytes_received = recv(client_socket, buffer, buffer_size, 0);

if (bytes_received > 0) {
    // Process received data normally
    buffer[bytes_received] = '\0'; // Null-terminate if treating as string
    printf("Received: %s\n", buffer);
}

else if (bytes_received == 0) {
    // Client performed orderly shutdown
    printf("Client %d disconnected gracefully\n", client_socket);
    remove_client(client_socket);
    break; // Exit receive loop
}

else {
    // Error occurred (bytes_received == -1)
    switch(errno) {
        case EAGAIN: // or EWOULDBLOCK on some systems
        case ENTR:
            // Temporary conditions - might want to try again
            continue;

        case ECONNRESET:
            printf("Client %d disconnected abruptly\n", client_socket);
            break;

        case ENOTCONN:
        case EBADF:
            printf("Socket %d is invalid\n", client_socket);
            break;

        default:
            perror("recv() error");
    }

    remove_client(client_socket);
    break; // Exit receive loop
}
```

Important Considerations

- Blocking vs Non-blocking Sockets:**
 - The above assumes blocking sockets (default)
 - For non-blocking sockets, `EAGAIN` / `EWOULDBLOCK` means no data available
- Partial Reads:**
 - `recv()` may return less than requested even when more data is coming
 - For protocols where message boundaries matter, implement proper framing
- Thread Safety:**
 - Always protect shared data (like client lists) with mutexes
 - `errno` is thread-specific in modern systems
- Resource Cleanup:**
 - Always close sockets when done
 - Free allocated memory
 - Update any tracking data structures
- Signal Interruptions:**
 - `ENTR` means the call was interrupted by a signal
 - You can either retry or handle it as a disconnection

By properly handling all `recv()` return cases, your server will be robust against:

- Graceful client disconnections (CTRL+C, proper close)
- Abrupt disconnections (network cable pulled)
- Network errors and timeouts
- Malformed client behavior

Would you like me to elaborate on any specific scenario or show how this integrates with the broader server architecture?

Overall these were the main contributions of AI. As used we didn't find the debugging tool very helpful when running into issues. If anything, it gave pointers so we knew what was working so we could focus on what parts are left to retest and work with. In addition, it is great at giving general knowledge for general questions and doing tedious things which would have been manually grabbing all the colors somewhere else to put in a list. It seems at this moment to be better at giving directions rather than direct code. Overall however, it was a valuable experience using it. It taught us where we can speed up our processes and help with research which cuts down on a lot of time.