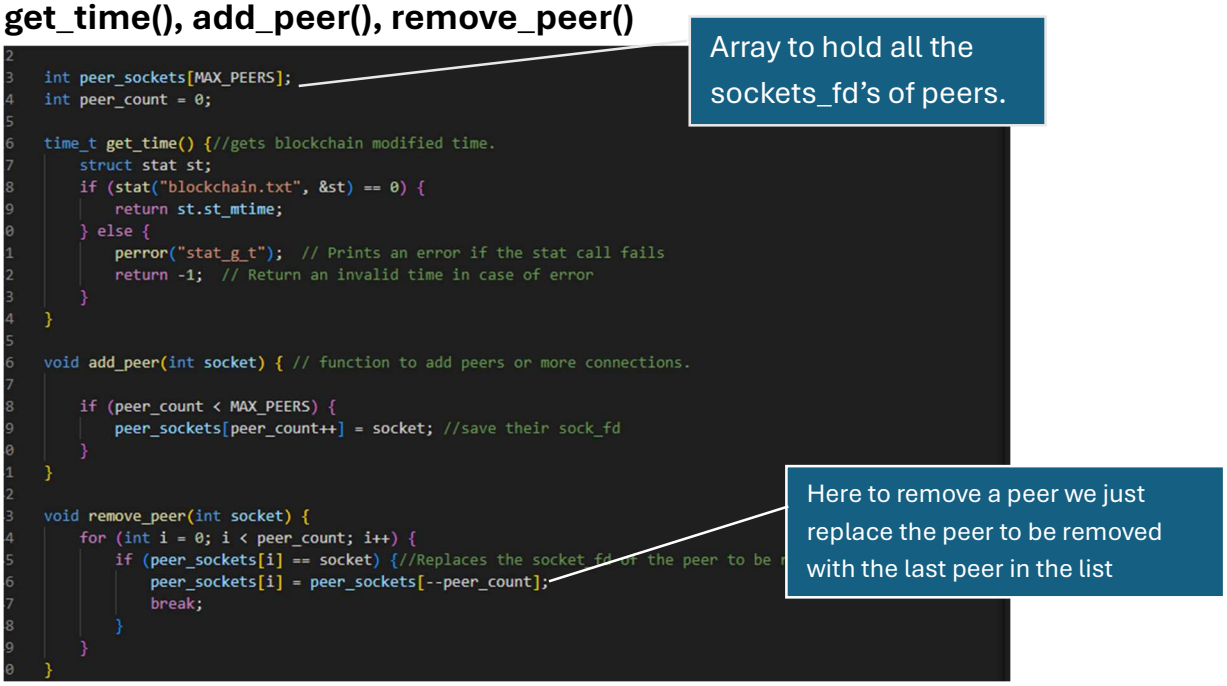
Approach:

To tackle this challenging project, we spent a large amount time understanding the provided flow charts. This helped us come up with 9 functions that helped us create a demo peer to peer network: get_time(), add_peer(), remove_peer(), broadcast_blockchain(), update_blockchain(), monitor_blockchain(), handle_peer(), listen_for_peers(), and connect_to_peer()

get_time(), add_peer(), remove_peer()




```c
int peer_sockets[MAX_PEERS];
int peer_count = 0;

time_t get_time() {//gets blockchain modified time.
    struct stat st;
    if (stat("blockchain.txt", &st) == 0) {
        return st.st_mtime;
    } else {
        perror("stat_g_t");  // Prints an error if the stat call fails
        return -1;  // Return an invalid time in case of error
    }
}

void add_peer(int socket) { // function to add peers or more connections.

    if (peer_count < MAX_PEERS) {
        peer_sockets[peer_count++] = socket; //save their sock_fd
    }
}

void remove_peer(int socket) {
    for (int i = 0; i < peer_count; i++) {
        if (peer_sockets[i] == socket) {//Replaces the socket_fd of the peer to be r
            peer_sockets[i] = peer_sockets[--peer_count];
            break;
        }
    }
}
```

get_time() function used stat function to get meta data about the file. We specifically used the st.mtime to get the last modified time of the file which helped figure out if the file need changes or not. Add_peer() and remove_peer() is used to add and remove peers from the list of connected socket_file descriptors.

Broadcast_Blockchain()

broadcast_blockchain function took in two parameters: sender socket and a bool. The main purpose of broadcast_blockchain is to broadcast the blockchain.txt file to all peers. The first parameter took in sock_fd of a peer in the case where a peer sent us the file. Bool was used to decide if we want to send the file to a specific peer(host mode has to send the file to incoming peers when they connect). The function basically reads in the blockchain.txt file and saves it into a buffer. It then sends it out to all or specific peers depending on how we call the function.

```c
void broadcast_blockchain(int sender_socket, bool single) {/**********
    //broadcast should only broadcast to everyone except the sender. it
    //if the bool single is true then we are sending the buffer to one

    FILE *file = fopen("blockchain.txt", "rb");
    if (file == NULL) {
        perror("Error opening blockchain.txt");
        return;
    }
    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);
    fseek(file, 0, SEEK_SET);

    char *buffer = malloc(file_size + 1);
    if (buffer == NULL) {
        perror("Memory allocation failed");
        fclose(file);
        return;
    }
    fread(buffer, 1, file_size, file);
    buffer[file_size] = '\0';
    fclose(file);

    if(single){ //if its true then I send the file to a single person.
        send(sender_socket, buffer, file_size + 1, 0);
    } else {//otherwise broadcast to everyone except sender.
    for (int i = 0; i < peer_count; i++) {
        if (peer_sockets[i] != sender_socket) {//if its -1, it'll send
            send(peer_sockets[i], buffer, file_size + 1, 0);
        }
    }
    }
    free(buffer);
}
```

This section deals with reading in the data in blockchain.txt file and saves it into buffer.

Here, if single = true then we send the buffer to a specific sender

This part sends the buffer to everyone except the sender of the file

Update_blockchain()

```c
void update_blockchain(const char *data, int sock_fd) {
    pthread_mutex_lock(&blockchain_mutex);
    update_in_progress = true;

    FILE *file = fopen("blockchain.txt", "w");
    if (file != NULL) {
        time_t now = time(NULL);
        //fprintf(file, "%s", ctime(&now));
        fprintf(file, "%s\n", data);
        fclose(file);
    }

    current_modified_time = get_time();
    saved_modified_time = current_modified_time;
    update_in_progress = false;

    pthread_cond_signal(&update_cond);
    pthread_mutex_unlock(&blockchain_mutex);

    if(sock_fd != -1){
        broadcast_blockchain(sock_fd, false);
    }
}
```

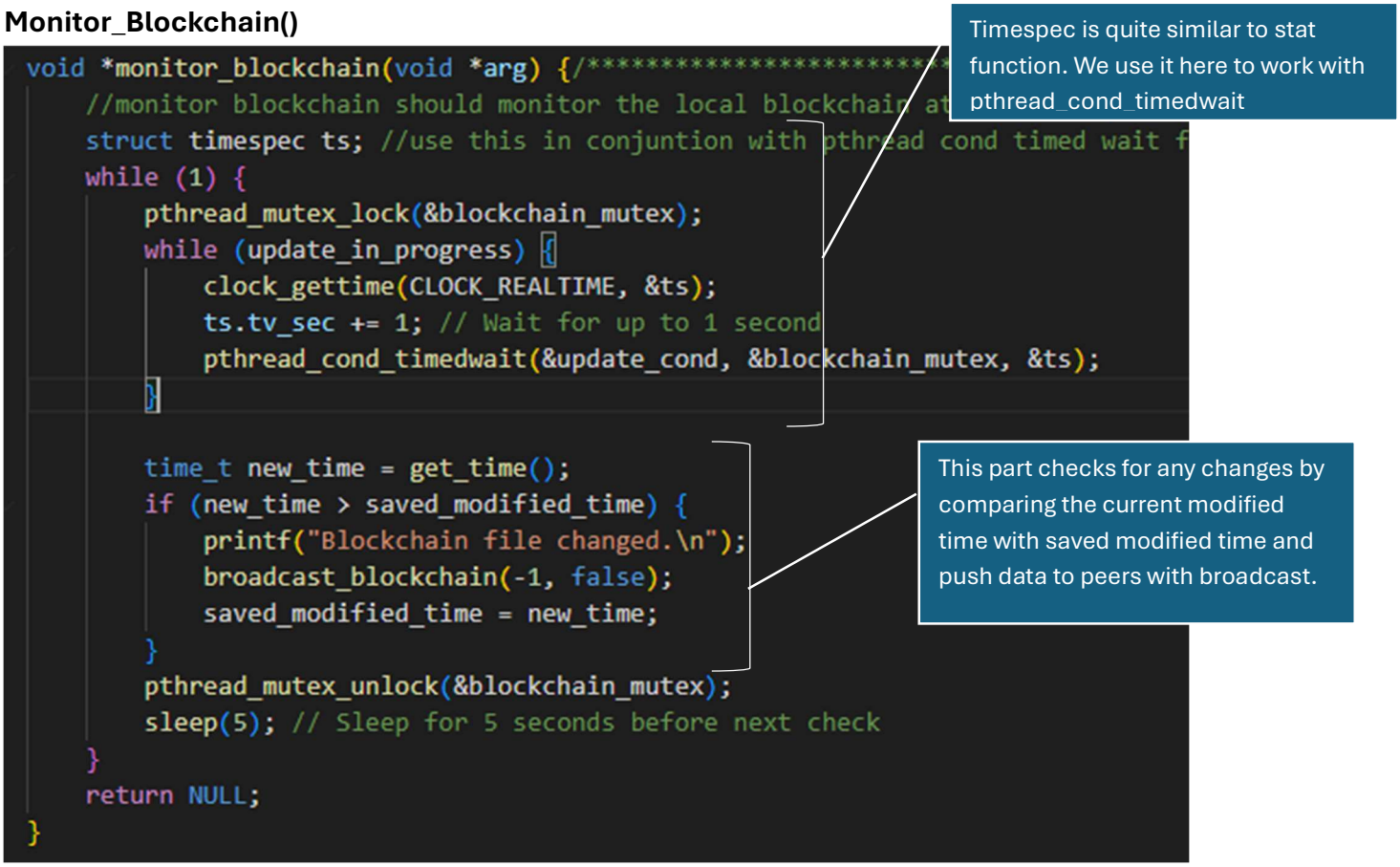
Get mutex lock to avoid race conditions.

Write the new data onto the blockchain.txt file,

Updates saved modified time so monitorBLKchain doesn't broadcast unnecessarily

The update_blockchain() fuction took in 2 parameters for data to add onto our blockchain.txt and sock_fd incase we need to broadcast to a specific file. Its main purpose isto add the incoming data from recv onto the blockchain.txt.

Monitor_Blockchain()



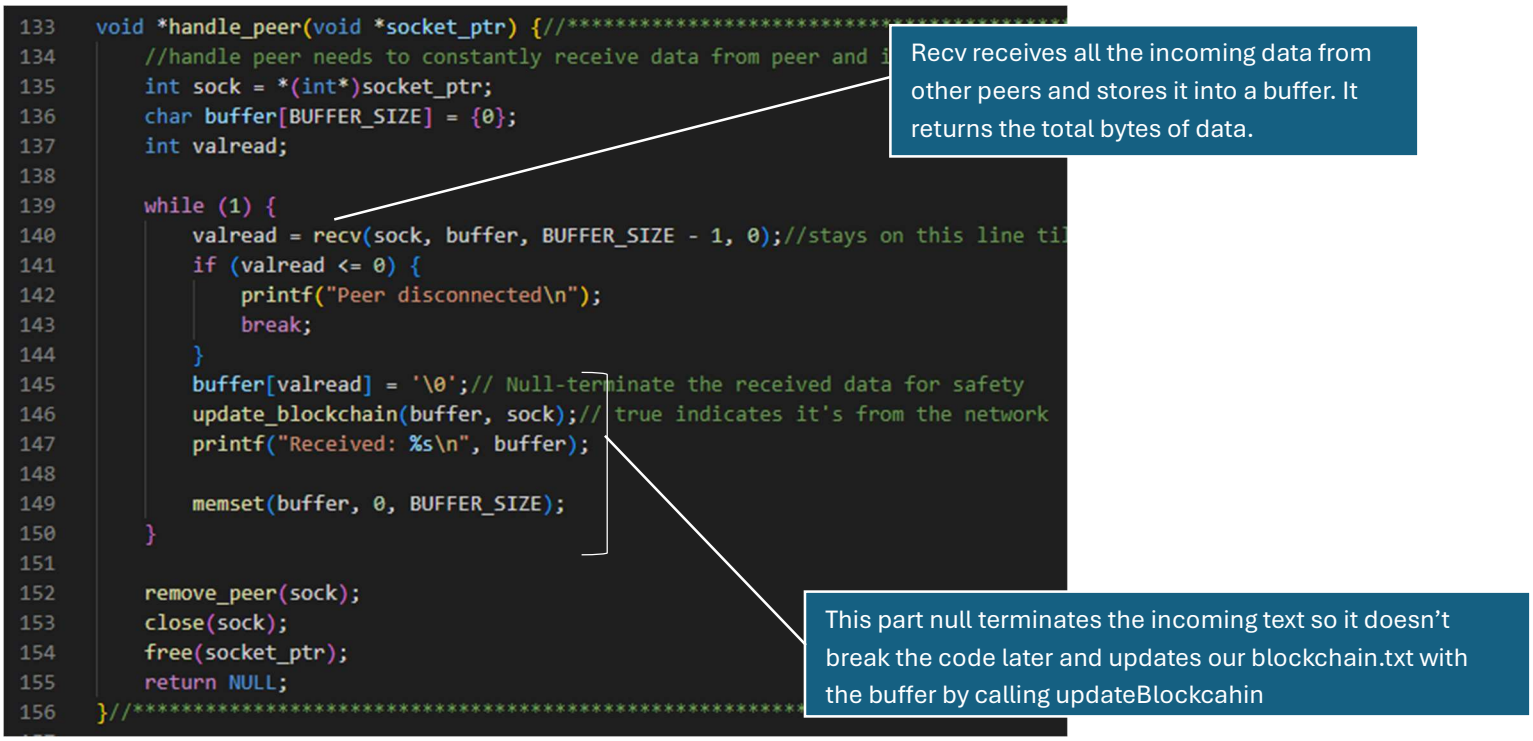

```
void *monitor_blockchain(void *arg) {/************************
    //monitor blockchain should monitor the local blockchain at
    struct timespec ts; //use this in conjuntion with pthread cond timed wait f
    while (1) {
        pthread_mutex_lock(&blockchain_mutex);
        while (update_in_progress) {
            clock_gettime(CLOCK_REALTIME, &ts);
            ts.tv_sec += 1; // Wait for up to 1 second
            pthread_cond_timedwait(&update_cond, &blockchain_mutex, &ts);
        }

        time_t new_time = get_time();
        if (new_time > saved_modified_time) {
            printf("Blockchain file changed.\n");
            broadcast_blockchain(-1, false);
            saved_modified_time = new_time;
        }
        pthread_mutex_unlock(&blockchain_mutex);
        sleep(5); // Sleep for 5 seconds before next check
    }
    return NULL;
}
```

The monitor_blockchain() functions purpose was to constantly monitor the existing blockchain.txt file for any changes. This function required mutual exclusions to work simultaneously with handle_peer function which will be explained next. When a change is detected using get_time() we broadcast to all peers.

Handle_peer()

The handle_peer() function took in a pointer to a connected peer pointer. The purpose of this function is to handle all connected peers and receive data from them using the recv function in an infinite while loop. This is one of the functions that need mutual exclusion and runs simultaneously with monitor_blockchain().




```
133  void *handle_peer(void *socket_ptr) {//**********************************
134      //handle peer needs to constantly receive data from peer and i
135      int sock = *(int*)socket_ptr;
136      char buffer[BUFFER_SIZE] = {0};
137      int valread;
138
139      while (1) {
140          valread = recv(sock, buffer, BUFFER_SIZE - 1, 0);//stays on this line til
141          if (valread <= 0) {
142              printf("Peer disconnected\n");
143              break;
144          }
145          buffer[valread] = '\0';// Null-terminate the received data for safety
146          update_blockchain(buffer, sock);// true indicates it's from the network
147          printf("Received: %s\n", buffer);
148
149          memset(buffer, 0, BUFFER_SIZE);
150      }
151
152      remove_peer(sock);
153      close(sock);
154      free(socket_ptr);
155      return NULL;
156  }//**********************************************************
```

Listen_for_peer()

listen_for_function is the longest of the bunch. Its main purpose is to listen for incoming connections or peers and pull in all the data about the socket such as address information. It also assign them a thread with handle_peer(). This also runs on a thread and is the first function the program calls when starting and runs indefinitely along side monitor_blockchain and handle peer.

```c
void *listen_for_peers(void *arg) {
    int sock_fd;
    struct addrinfo hints, *servinfo, *p;
    int yes = 1;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    if (getaddrinfo(NULL, PORT, &hints, &servinfo) != 0) {
        perror("getaddrinfo");
        return NULL;
    }

    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sock_fd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
            continue;
        }
        if (setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
            close(sock_fd);
            continue;
        }
        if (bind(sock_fd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sock_fd);
            continue;
        }
        break;
    }

    freeaddrinfo(servinfo);

    if (p == NULL) {
        fprintf(stderr, "Failed to bind\n");
        return NULL;
    }
    if (listen(sock_fd, 10) == -1) {
        perror("listen");
        return NULL;
    }
    printf("Listening for peers...\n");

    while(1) {
        struct sockaddr_storage their_addr;
        socklen_t addr_size = sizeof their_addr;
        int *new_sock = malloc(sizeof(int));

        *new_sock = accept(sock_fd, (struct sockaddr *)&their_addr, &addr_size);

        if (*new_sock == -1) {
            perror("accept");
            free(new_sock);
            continue;
        }
        char peer_ip[INET6_ADDRSTRLEN];
        inet_ntop(their_addr.ss_family,
            &(((struct sockaddr_in*)&their_addr)->sin_addr),
            peer_ip, sizeof peer_ip);
        printf("New peer connection from %s\n", peer_ip);

        add_peer(*new_sock);

        pthread_t thread_id;
        if (pthread_create(&thread_id, NULL, handle_peer, (void*)new_sock) < 0) {//assigns
            perror("Could not create thread");
            remove_peer(*new_sock);
            close(*new_sock);
            free(new_sock);
            continue;
        }
        broadcast_blockchain(*new_sock, true);//broadcast blockchain as soon as a new cli
        saved_modified_time = current_modified_time;
    }
}
```

Here we initialize the address info for the socket.

Here we get the new socket_fd, set socket options and bind the socket

This while loop runs indefinitely listening for new peers and accepts the socket. Also assigns handle peer for every new connection to get data.

Handle peer thread for all the new peers

Broadcast data to newly connected peer with bool true to indicate its to be sent to a specific peer.

Connect_to_peer()

```c
void connect_to_peer(const char *ip_address) { //****************************************
    int sockfd;
    struct addrinfo hints, *servinfo, *p;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if (getaddrinfo(ip_address, PORT, &hints, &servinfo) != 0) {
        perror("getaddrinfo");
        return;
    }
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
            continue;
        }

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            continue;
        }
        break;
    }
    if (p == NULL) {
        fprintf(stderr, "Failed to connect\n");
        return;
    }
    freeaddrinfo(servinfo);
    printf("Connected to peer %s\n", ip_address);
    add_peer(sockfd);

    int *new_sock = malloc(sizeof(int));
    *new_sock = sockfd;
    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, handle_peer, (void*)new_sock) < 0) {
        perror("Could not create thread");
        remove_peer(sockfd);
        close(sockfd);
        free(new_sock);
        return;
    }
}
```

Sets the socket for the host connection and connects to it. This is if we enter in connection mode by providing a valid IP.

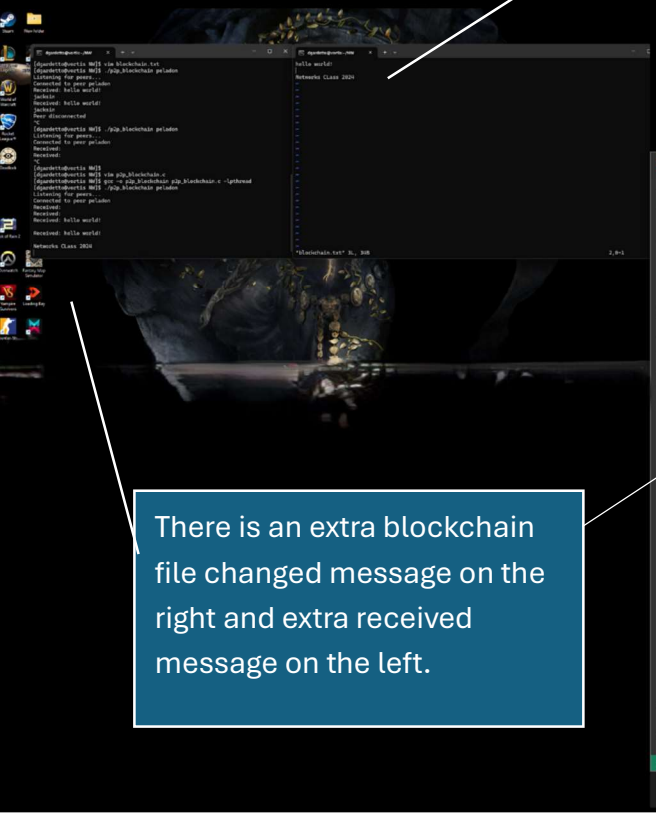
Assigns the connected host a handle peer thread to read in data from the host.

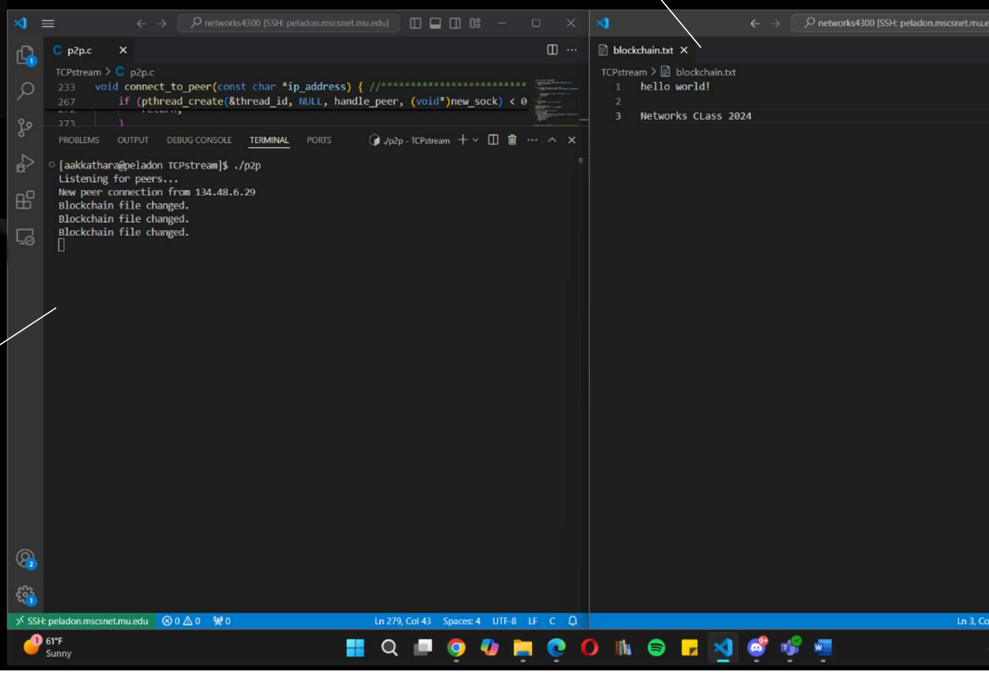
the main purpose of connect_to_peer() function is to allow peers to connect to a host or another peer. It takes in an IP address and connects to that particular socket essentially becoming a part of that network. This function then assigns the connected host or peer a thread with handle_peer() to receive data.

Testing procedure:

Testing code was quite challenging when working on this project individually because using the same file on the mscsnet network became tricky. It became quite straight forward when working with a partner since we have 2 separate files. We were able to connect to each other and send our updated blockchain.txt files successfully. We also tested multiple connections to multiple peers but only got to test it on one mscsnet account. The connections were all successful but cant really tell if the blockchain was properly being synchronized.

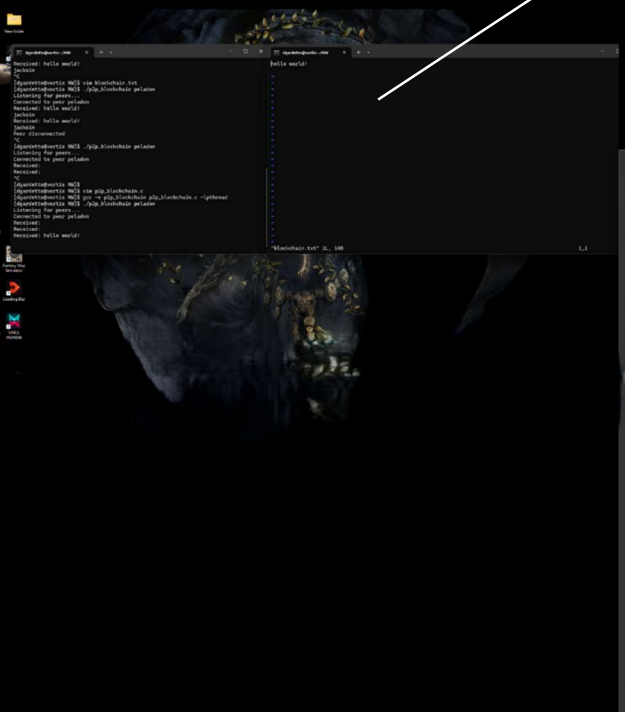
Here both blockchain.txt files are the same and have been successfully shared.

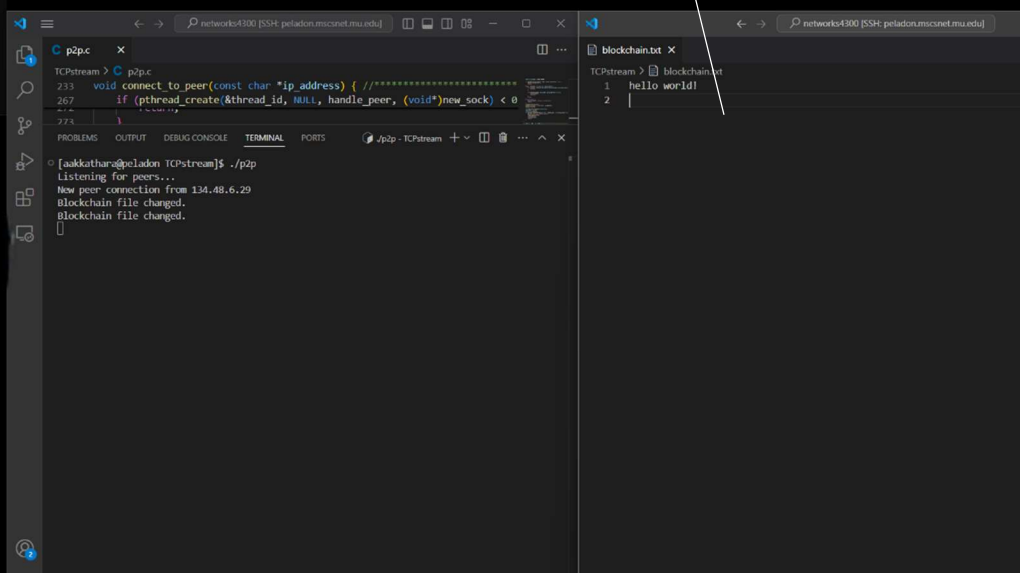




There is an extra blockchain file changed message on the right and extra received message on the left.

Again we have the same updated blockchain file on two different computers and accounts.





Challenges:

We faced so many challenges during this project but the ones worth mentioning are connection issues, synchronization issues, we had to learn many new built in function, mutex issues, and issue that took most of our time, the feedback loop. With the help of the TA, we were able to solve most of the issues. Some of the solutions included adding a FD list for the new connections and adding mutex for all the threads, avoiding race conditions. Testing was also very tricky since all my files are synced across mscsnet.