

COSC - 4300 Network Design and Security

Dr. Debbie Perouli

Danny Gardetto & Allan Akkathara

Homework #2(Final)

Approach:

To tackle this challenging project, we spent a large amount of 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()`

```
2
3 int peer_sockets[MAX_PEERS];
4 int peer_count = 0;
5
6 time_t get_time() { //gets blockchain modified time.
7     struct stat st;
8     if (stat("blockchain.txt", &st) == 0) {
9         return st.st_mtime;
10    } else {
11        perror("stat_g_t"); // Prints an error if the stat call fails
12        return -1; // Return an invalid time in case of error
13    }
14 }
15
16 void add_peer(int socket) { // function to add peers or more connections.
17
18     if (peer_count < MAX_PEERS) {
19         peer_sockets[peer_count++] = socket; //save their sock_fd
20     }
21 }
22
23 void remove_peer(int socket) {
24     for (int i = 0; i < peer_count; i++) {
25         if (peer_sockets[i] == socket) { //Replaces the socket fd of the peer to be removed
26             peer_sockets[i] = peer_sockets[--peer_count];
27             break;
28         }
29     }
30 }
```

Array to hold all the
sockets_fd's of peers.

Here to remove a peer we just
replace the peer to be removed
with the last peer in the list

`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.

```

51 void broadcast_blockchain(int sender_socket, bool single) {/*****
52     //broadcast should only broadcast to everyone except the sender. it
53     //if the bool single is true then we are sending the buffer to one
54
55     FILE *file = fopen("blockchain.txt", "rb");
56     if (file == NULL) {
57         perror("Error opening blockchain.txt");
58         return;
59     }
60     fseek(file, 0, SEEK_END);
61     long file_size = ftell(file);
62     fseek(file, 0, SEEK_SET);
63
64     char *buffer = malloc(file_size + 1);
65     if (buffer == NULL) {
66         perror("Memory allocation failed");
67         fclose(file);
68         return;
69     }
70     fread(buffer, 1, file_size, file);
71     buffer[file_size] = '\0';
72     fclose(file);
73
74     if(single){ //if its true then I send the file to a single person.
75         send(sender_socket, buffer, file_size + 1, 0);
76     } else { //otherwise broadcast to everyone except sender.
77         for (int i = 0; i < peer_count; i++) {
78             if (peer_sockets[i] != sender_socket) { //if its -1, it'll send
79                 send(peer_sockets[i], buffer, file_size + 1, 0);
80             }
81         }
82     }
83     free(buffer);
84 }

```

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()

```

86 void update_blockchain(const char *data, int sock_fd) {
87     pthread_mutex_lock(&blockchain_mutex);
88     update_in_progress = true;
89
90     FILE *file = fopen("blockchain.txt", "w");
91     if (file != NULL) {
92         time_t now = time(NULL);
93         //fprintf(file, "%s", ctime(&now));
94         fprintf(file, "%s\n", data);
95         fclose(file);
96     }
97
98     current_modified_time = get_time();
99     saved_modified_time = current_modified_time;
100     update_in_progress = false;
101
102     pthread_cond_signal(&update_cond);
103     pthread_mutex_unlock(&blockchain_mutex);
104
105     if(sock_fd != -1){
106         broadcast_blockchain(sock_fd, false);
107     }
108 }

```

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()` function took in 2 parameters for data to add onto our `blockchain.txt` and `sock_fd` in case we need to broadcast to a specific file. Its main purpose is to 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 conjunction 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;
}
```

Timespec is quite similar to stat function. We use it here to work with `pthread_cond_timedwait`

This part checks for any changes by comparing the current modified time with saved modified time and push data to peers with broadcast.

The `monitor_blockchain()` function's 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         memset(buffer, 0, BUFFER_SIZE);
149     }
150
151     remove_peer(sock);
152     close(sock);
153     free(socket_ptr);
154     return NULL;
155 } //*****
156

```

Recv receives all the incoming data from other peers and stores it into a buffer. It returns the total bytes of data.

This part null terminates the incoming text so it doesn't break the code later and updates our blockchain.txt with the buffer by calling updateBlockchain

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 assigns 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 alongside monitor_blockchain and handle peer.

```

58 void *listen_for_peers(void *arg) {
59     int sock_fd;
60     struct addrinfo hints, *servinfo, *p;
61     int yes = 1;
62
63     memset(&hints, 0, sizeof hints);
64     hints.ai_family = AF_UNSPEC;
65     hints.ai_socktype = SOCK_STREAM;
66     hints.ai_flags = AI_PASSIVE;
67
68     if (getaddrinfo(NULL, PORT, &hints, &servinfo) != 0) {
69         perror("getaddrinfo");
70         return NULL;
71     }
72
73     for(p = servinfo; p != NULL; p = p->ai_next) {
74         if ((sock_fd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
75             continue;
76         }
77         if (setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
78             close(sock_fd);
79             continue;
80         }
81         if (bind(sock_fd, p->ai_addr, p->ai_addrlen) == -1) {
82             close(sock_fd);
83             continue;
84         }
85         break;
86     }
87
88     freeaddrinfo(servinfo);
89
90     if (p == NULL) {
91         fprintf(stderr, "Failed to bind\n");
92         return NULL;
93     }
94     if (listen(sock_fd, 10) == -1) {
95         perror("listen");
96         return NULL;
97     }
98     printf("Listening for peers...\n");
99
100    while(1) {
101        struct sockaddr_storage their_addr;
102        socklen_t addr_size = sizeof their_addr;
103        int *new_sock = malloc(sizeof(int));
104
105        *new_sock = accept(sock_fd, (struct sockaddr *)&their_addr, &addr_size);
106
107        if (*new_sock == -1) {
108            perror("accept");
109            free(new_sock);
110            continue;
111        }
112        char peer_ip[INET6_ADDRSTRLEN];
113        inet_ntop(their_addr.ss_family,
114            &(((struct sockaddr_in*)&their_addr)->sin_addr),
115            peer_ip, sizeof peer_ip);
116        printf("New peer connection from %s\n", peer_ip);
117
118        add_peer(*new_sock);
119
120        pthread_t thread_id;
121        if (pthread_create(&thread_id, NULL, handle_peer, (void*)new_sock) < 0) {
122            perror("Could not create thread");
123            remove_peer(*new_sock);
124            close(*new_sock);
125            free(new_sock);
126            continue;
127        }
128        broadcast_blockchain(*new_sock, true);
129        saved_modified_time = current_modified_time;
130    }
131 }

```

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()

```
83 void connect_to_peer(const char *ip_address) { //*****
84     int sockfd;
85     struct addrinfo hints, *servinfo, *p;
86
87     memset(&hints, 0, sizeof hints);
88     hints.ai_family = AF_UNSPEC;
89     hints.ai_socktype = SOCK_STREAM;
90
91     if (getaddrinfo(ip_address, PORT, &hints, &servinfo) != 0) {
92         perror("getaddrinfo");
93         return;
94     }
95     for(p = servinfo; p != NULL; p = p->ai_next) {
96         if ((sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
97             continue;
98         }
99
100        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
101            close(sockfd);
102            continue;
103        }
104        break;
105    }
106    if (p == NULL) {
107        fprintf(stderr, "Failed to connect\n");
108        return;
109    }
110    freeaddrinfo(servinfo);
111    printf("Connected to peer %s\n", ip_address);
112    add_peer(sockfd);
113
114    int *new_sock = malloc(sizeof(int));
115    *new_sock = sockfd;
116    pthread_t thread_id;
117    if (pthread_create(&thread_id, NULL, handle_peer, (void*)new_sock) < 0) {
118        perror("Could not create thread");
119        remove_peer(sockfd);
120        close(sockfd);
121        free(new_sock);
122        return;
123    }
124 }
```

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.