COL334 Computer Networks SEM II 2023-24

Vatsal Jingar (2020CS50449)
Stitiprajna Sahoo (2020CS10394)
Tanish Gupta (2020CS10397)

Date: 8 September 2023

Assignment 2: Mimicking distributed file transfer

1 Implementation Cpp (g++ 10.4)

1. **clientburst.cpp**
   This file contains the code logic for receving packets continuously from the server.

2. **clientrecv.cpp**
   Client will receive the packets continuously from its peer cleint.

3. **cleintbroadcast.cpp**
   Client will broadcast the packets to its peer clients continuously.

4. **controller.cpp**
   Every client will keep count of how much lines it has received. When it is L = 1000, it will communicate it with the main thread.

5. **driver.cpp**
   Every client can run this file with their respective ip written in constants.h.

6. **constants.h**
   All the constants like Number of cleints, port number, ip address of clients are written here.

1.1 Code Structure

1. Central Struct : "Client_data"

This struct contains all the data that is required for the client to communicate with peers.

```
struct Client_data{
bool received[L];
string data[L];
bool complete;
int port[N];
const char *ips[N];
vector<int> broadcast;
int clientid;
};
```

received[**L**] : This array contains the information about which lines have been received.
data[**L**] : This array contains the data of the lines that have been received.
complete : This variable tells whether the client has received all the lines or not.
port[**N**] : This array contains the port numbers of the peers.
ips[**N**] : This array contains the ip addresses of the peers.
broadcast : This vector contains the information which line is yet to be broadcasted.
clientid : This variable contains the id of the client.

2. Driver.cpp

```
int main(){

// assert((int)client_ips.size() == N);
vector<pthread_t> clients(N);
vector<struct Client_data> args;
vector<int> ports;
for(int i = 0; i < N;i++){
    struct Client_data temp;
    args.push_back(temp);
    args[i].complete = 0;
    args[i].clientid = i;
    args[i].ips[i] = client_ips[i];
}
client((void*) &args[client_id]);
cout << "Completed Session\n";
return 0;
}
```

This file is driver for starting the cleint. It creates a thread with certain parameters like client id and ip address.

3. Client.cpp

3.1 Client Data Initialization

```
void* client(void* arg){
struct Client_data* args = (struct Client_data*) arg;
pthread_t clientburster, clientrecver, clientcontroller, clientbroadcaster;
for (int i = 0; i < L; i++) {
    args->received[i] = false;
    args->data[i] = "0";
    // args->broadcasted[i] = false;
}
args->complete = 0;
}
```

All the cleints will initialize their data and checkpoint array. And they will set the complete variable to 0.

3.2 Cleint Sub modules:

```
pthread_create(&clientburster, NULL, clientburst, (void*) &(*args));
pthread_create(&clientrecver, NULL, clientrecv, (void*) &(*args));
pthread_create(&clientbroadcaster, NULL, clientbroadcast, (void*) &(*args));
pthread_create(&clientcontroller, NULL, controller, (void*) &(*args));

pthread_join(clientburster, NULL);
pthread_join(clientrecver,NULL);
pthread_join(clientbroadcaster, NULL);
pthread_join(clientcontroller, NULL);
```

This file creates 4 submodules for the client.

3.2.1 Clientburst

This thread will continuously receive the packets from the server.

3.2.2 Clientrecv

This thread will continuously receive the packets from the peer client.

3.2.3 Clientbroadcast

This thread will continuously broadcast the packets to the peer clients.

3.2.4 Controller

This thread will continuously check whether the client has received all the packets or not.

4. Clientburst.cpp

4.1 Creation and Connecting Socket with Server

```
struct Client_data *needdata = (struct Client_data*) args;
int status, client_fd;
struct sockaddr_in serv_addr;
const char *sendline = "SENDLINE\n";
char buffer[BUFFER_SIZE];
if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    printf("\nSocket creation error \n");
    RETURN(2);
}
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(PORT);
if (inet_pton(AF_INET, serverIP, &serv_addr.sin_addr) <= 0){
    printf("\nInvalid address/ Address not supported \n");
    RETURN(2);
}
```

4.2 Sending SENDLINE to Server

```
while (!needdata->complete) {
    if(send(client_fd, sendline, strlen(sendline), 0) < 0){
        perror("send");
        continue;
    }
```

4.3 Receiving the Data from Server

```
while(count < 2){
    int x = recv(client_fd, buffer, BUFFER_SIZE, 0);
    if(x < 0){
        perror("recv");
        continue;
    }
    for(int i = 0; i < x; i++){
        if(buffer[i] == '\n') count++;
        mystring += buffer[i];
        if(count == 2) break;
    }
}
```

4.4 Parsing the Data

```
while (i < (int)mystring.size() && mystring[i] != '\n'){
    line_num = 10 * line_num + (mystring[i] - '0');
    i++;
}
if (line_num >= 0 && line_num < L && !needdata->received[line_num]){
    string data = "";
    for (int j = i + 1; j < (int)mystring.length(); j++){
        data += mystring[j];
        if(mystring[j] == '\n') break;
    }
    cnt++;
    // cout << line_num << endl;
    needdata->data[line_num] = data;
    needdata->received[line_num] = true;
    needdata->broadcast.push_back(line_num);
}
```

5. Clientrecv.cpp

Logic :

Every client will create a socket and bind it to its port. Then it will start receiving for the packets from its peer clients. When it receives the packets, it will parse the data and store it in its data array. It will also update the checkpoint array. The data which would received from the peer client would be in the form of "(line\nnumber

data)...(line\nnumber data)".
For every client, we will have N sockets which will try to connect their peer clients. The port number of the socket to which it will try to connect will be the func(client id of peer). We will maintain a vector of sockets which will be used to connect to the peer clients.

As the data received from peer is of the complex form, cleintrecv will try to convert it into the form of "line\ndata".

Converting Data into the required form

```
reading = "";
while(count < 2){
    int x = recv(sock, buff, BUFFER_SIZE, 0);
    if(x < 0){
        perror("recv");
        continue;
    }
    for(int i = 0; i < x; i++){
        if(buff[i] == '\n') {
            count++;
        }
        else{
            count = 0;
        }
        reading += buff[i];
        if(count == 2){
            break;
        }
    }
}
```

We are creating a string reading. We will keep on appending the data to the string reading. When we encounter two consecutive new lines, we will break the loop. It is assumed that the other peers will send the data with appending one extra \ n at the end.
We know that the underlying protocol TCP is reliable but it uses streams. Hence it could happen that data writtem in kernel socket file is not complete and received in chunks. So to assure that we had recevied all the data, we will keep writing on the client buffer until we encounter two consecutive new lines.
Data is parsed in the same way as it was done in clientburst.

6. Clientbroadcast.cpp

Logic :

We have created a queue of lines which are yet to be broadcasted. Whenever we will

see that the queue is empty we will record the length of queue at that time. To achieve the functionality of iterating over queue, we have implemented it using a C++ vector. We will iterate over the queue till the length we saw and create a string temp which will contain the data in the form of "line\ndata....line\ndata". We will then broadcast this data to all the peer clients.

```
if(data->broadcast_idx < (int)data->needed_data->broadcast.size()){
    string temp = "";
    int lengthofvector = data->needed_data->broadcast.size();
    while(data->broadcast_idx  < lengthofvector){
        int i = data->needed_data->broadcast[data->broadcast_idx];
        data->broadcast_idx++;
        temp += (to_string(i) + "\n" + data->needed_data->data[i]);
        // upperboundcount++;
    }
    temp += "\n";
    const char *temp1 = temp.c_str();
    if(send(sock, temp1, strlen(temp1), MSG_NOSIGNAL) < 0){
        perror("send");
        continue;
    }
    data->sent = true;
}
```

7. Calculating the peer ports

Every client will create N sockets. One socket will always be empty. All the other sockets would be to send the data to other cleints.

The ports of the sockets would be calculated as follows :
$PORTS + (needddata- > clientid) * (N) + (i)$;

Every cleint will connect to the ports :
$(PORTS + (i) * (N) + (need\_data- > clientid))$;

This will ensure that every client will connect to every other client.

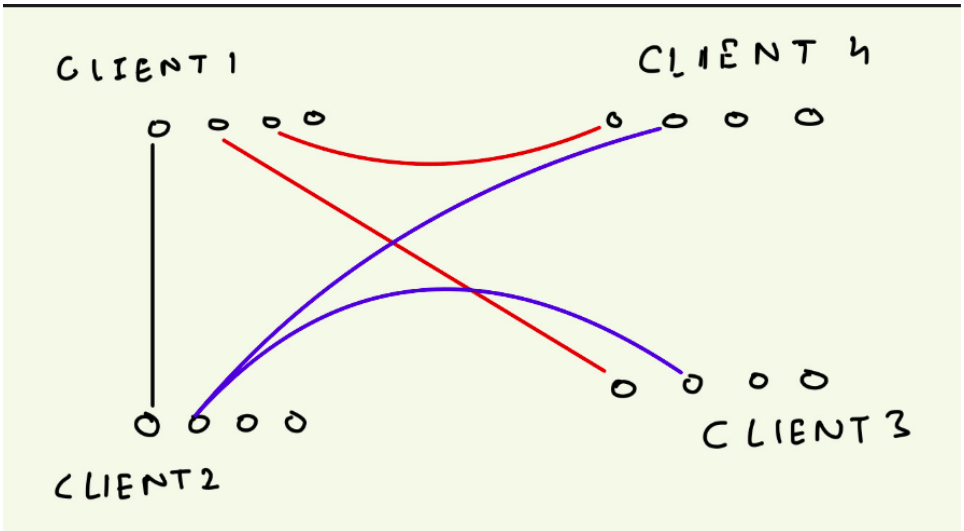

Figure 1: Client 1 and Client 2 Connections

The figure show that how Client 1 and Client 2 are connected from all the other peers.

1.2 Fault Recovery and Consistency

2 Analysis

2.1 N = 1

2.2 N = 2

2.3 N = 3

2.4 N = 4

2.5 Relation between Number of Clients and Time