

ECE 650 project

Yiyi Liu

Student ID: 20625436

E-mail: y864liu@uwaterloo.ca

December 2016

1 Introduction

This report introduce the project of a client/server road map system. Since the project consists of two parts: client and server, the report would be divided into two parts.

The first part is server which executes all of the operation from client, including managing the road map system (i.e. changing road map system, storing and retrieving road map system) and using system. Thus, the server is the most critical part of the system.

The second part of the system consists of two types of clients: manager client and user client. Manager client, as its name showing, used to executing all of operation to road map system while user client, merely for users to use the system.

The whole system runs over the connection between server and clients, in this system, it set up TCP/IP connection.

1.1 TCP

TCP (Transmission Control Protocol) is a standard that defines how to establish and maintain a network conversation via which application programs can exchange data. TCP works with the Internet Protocol (IP), which defines how computers send packets of data to each other. Together, TCP and IP are the basic rules defining the Internet.

TCP is a connection-oriented protocol, which means a connection is established and maintained until the application programs at each end have finished exchanging messages. It determines how to break application data into packets that networks can deliver, sends packets to and accepts packets from the network layer, manages flow control, and handles retransmission of dropped or garbled packets as well as acknowledgement of all packets that arrive.

In this project, TCP is used to provide communication between server and clients for managing and accessing system.

2 Server

The server does all operation over the system. Once the server runs, the server starts **socket** and **bind** function waiting for calling clients. The server set a port number for **socket()** through which the clients can send requests to manage and access the system, as well as getting response from server. Indeed, the server does nothing but completing the operation of clients and maintain the connection from clients.

2.1 Socket

The **socket** function implements the **listen** function at a specific port, waiting for the connection of client. Once receiving the connection of client, accept it and response, a stable TCP connection set up then **socket()** provides the a specific and unique pipe for communication between two or more ends, thus, it would maintain a unique communication without disturbing other work in the same machine. Also, it would not disturb other communication, that means, in the same machine, it is plausible to set up more similar communication which makes multiple-end communication possible. Thus, it is most popular and usable method for possible and potential Internet communication.

The usage of **socket()** is mainly about **bind()** function. Through this function, the server and client are bound with each other with the port number of server and addresses of both.

2.2 Socket of server

In C/C++, there is **struct sockaddr_in** for socket address. The server, to maintain a connection with client, should be aware about address of client. Thus, server sets two **sockaddr_in** maintain connection. The **serv_addr** is for own port and address binding in the session, it means, through binding with a **servsock**, all of message to the port number and address would be assigned to this session.

Code for socket of server as following:

```
1 int servsock, portno;
2 struct sockaddr_in serv_addr;
3 servsock = socket(AF_INET, SOCK_STREAM, 0);
4 if (servsock < 0)
5     error("ERROR opening socket");
6     bzero((char *) &serv_addr, sizeof(serv_addr));
7     portno = 60000;
8     serv_addr.sin_family = AF_INET;
9     serv_addr.sin_addr.s_addr = INADDR_ANY;
10    serv_addr.sin_port = htons(portno);
11    if (::bind(servsock, (struct sockaddr *) &serv_addr,
12        sizeof(serv_addr)) < 0)
13        error("ERROR on binding");
```

After binding port and address of server, the server would be keep listening to the connection request from client.

```
1 listen(servsock,5);
```

Code as above make the socket of server keep listening potential connection. The second parameter 5 is the possible maximum connection for this socket.

2.3 Socket of client

After binding itself, there should be one data structure to accept the connection from client. The **cliensock** is same structure with **servsock**, however, it is for save related information of client when accepting the its request of accessing. Code as following:

```
1 cliensock = accept(servsock, (struct sockaddr *) &cli_addr, &clilen  
    )
```

The next step is keep receiving message from client socket. Once received, the function **recv()** would return a positive integer if succeed, otherwise it failed. The code as following:

```
1 char buffer_c[256];  
2 n=(int)recv(cliensock, buffer_c,255,0);
```

The **buffer_c** is char array for saving message from client. Then, processing the message to needed format and executing the operation from request of client. Once session finished, the server need to close two socket separaetly as following:

```
1 close(cliensock);  
2 close(servsock);  
3
```

2.4 Processing message

Another critical function is for processing message from client. Since all of message are sent with format of **char**, parsing every parts of message to needed format separately is the key step before executing the command. In this project, the command from client is merely command of operation(i.e.the function operation). Thus, the parsing function just need to get the function name and its parameter. Moreover, the function needs to transfer different command parameter to needed format, such as, index to int, string to self-defined data structure and etc.

3 Client

3.1 Client Socket

The socket of client is similar to the server one, but only one **sockaddr_in** needed since the client merely need to save server address and port number. Code as following:

```
1 struct sockaddr_in serv_addr;
2 struct hostent *server;
3 portno = 60000; //server port number
4 sockfd = socket(AF_INET, SOCK_STREAM, 0);
5 if (sockfd < 0)
6     error("ERROR opening socket");
7 //get host by name using localhost
8 server = gethostbyname("localhost");
9 serv_addr.sin_family = AF_INET;
10 bcopy(server->h_addr,
11        (char *)&serv_addr.sin_addr.s_addr,
12        (size_t)server->h_length);
13 serv_addr.sin_port = htons(portno);
```

After setting server address and port number, the client would try to connect with server. The function would return a positive number if succeed. Code as follows:

```
1 if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
2     < 0)
3     error("ERROR connecting");
4 else printf("Connect success!\n");
```

Setting up a stable connection, client would keep communicate with server. Client and server would have the same method to passing the message to socket, thus, **send()** and **recv()** would be used also in client for sending and receiving message.

The code as follows:

```
1 n = (int)send(sockfd, buffer_c, strlen(buffer_c), 0);
2 if (n < 0)
3 {
4     perror("ERROR writing to socket:");
5     exit(0);
6 }
7 bzero(buffer_s, 255);
8 n = (int)recv(sockfd, buffer_s, 255, 0);
9 printf("Server: %s", buffer_s);
```

3.2 Manager client

The manager client does works for managing system, such as add vertice, store road map updates and retrieve the road map system. For this role of client, test case would be those operation involving revising road map system.

4 User client

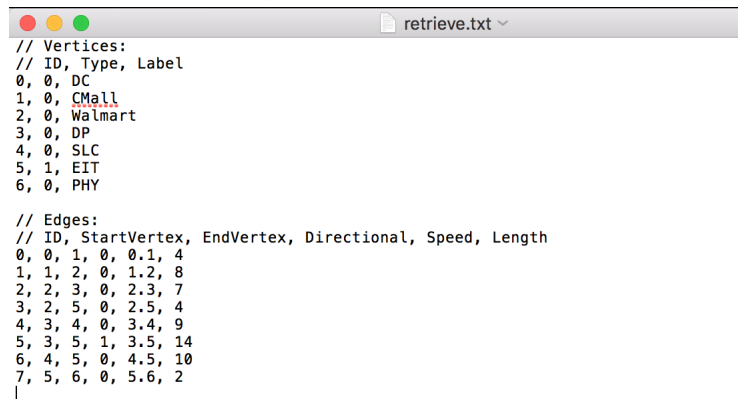
The user client is for users of the road map system. In this project, the user client and manager client share one program. One can run different role in the same program with choosing different test case. In user client, the client can send any message to the server, as well as command to get road map information and using it.

5 Test Case

This following text case, since the project is from assignment 3, adapts the same test case of assignment 3.

5.1 retrieve, addVertex, addEdge

The test case is written in the **main.cpp**. First, we retrieve some vertices and edges from the file “retrieve.txt”, whose content is shown in Figure 1. Then we add additional vertices and edges using **addVertex** and **addEdge**, and add some edges events and roads using **edgeEvent** and **road**.



```
// Vertices:
// ID, Type, Label
0, 0, DC
1, 0, CMall
2, 0, Walmart
3, 0, DP
4, 0, SLC
5, 1, EIT
6, 0, PHY

// Edges:
// ID, StartVertex, EndVertex, Directional, Speed, Length
0, 0, 1, 0, 0.1, 4
1, 1, 2, 0, 1.2, 8
2, 2, 3, 0, 2.3, 7
3, 2, 5, 0, 2.5, 4
4, 3, 4, 0, 3.4, 9
5, 3, 5, 1, 3.5, 14
6, 4, 5, 0, 4.5, 10
7, 5, 6, 0, 5.6, 2
|
```

Figure 1: Retrieve

5.2 vertex, trip

We create three test cases for **vertex()**: **g.vertex(“DC”)**, **g.vertex(“DP”)** and **g.vertex(“Toronto”)**. The return objects of the first two call serve as the parameter of subsequent **trip(v1, v2)** call, while third call tests what happens when someone queries an unknown label.

When testing **trip**, we create two test cases:

1. All paths include edges with event on it

2. Only the shortest path include edges with event on it

We print out the returned paths in main for the first test case. The results show that our **trip** keeps searching for a no-event shortest path. If all paths between the two vertices include events, then “There is no other path from ... to...” is printed to notify programmer. Each path is represented in two ways: a sequence of vertex IDs; a sequence of edge IDs and, if any, road names of these edges. All paths are returned to main function, so that users can choose by themselves whether to go the destination via a path with events. From figure2, the result can be seen.

```

C:\MINGW\bin\gcc.exe *.c
main: is not known: C:\MINGW\bin\gcc.exe *.c
trip--Shortest path from 0 to 3 (Vertex ID): 0, 1, 2, 3, (Edge ID): 0, 1 University Ave, 2 Columbia,
Event 3 on edge 0, Event 0 on edge 2
trip--Shortest path from 0 to 3 (Vertex ID): 0, 7, 6, 5, 4, 3, (Edge ID): 8, 11, 7 University Ave, 6 Columbia, 4 University Ave,
Event 2 on edge 4
There is no other path from 0 to 3
main routine--Shortest path from DC to DP
(Vertex ID): 0, 1, 2, 3, (Edge ID): 0, 1 University Ave, 2 Columbia,
(Vertex ID): 0, 7, 6, 5, 4, 3, (Edge ID): 8, 11, 7 University Ave, 6 Columbia, 4 University Ave,
trip--Shortest path from 4 to 2 (Vertex ID): 4, 5, 2, (Edge ID): 6 Columbia, 3 University Ave,
Event 1 on edge 5
trip--Shortest path from 4 to 2 (Vertex ID): 4, 5, 6, 3, 2, (Edge ID): 4 Columbia, 3 University Ave, 12, 10,

```

Figure 2: Trip

5.3 removeEvent, store

We remove the event “ACCIDENT” on edge 2, and then store the current map state to “store.txt”.

```

store.txt
1, 0, CMall
2, 0, Walmart
3, 0, DP
4, 0, SLC
5, 1, EIT
6, 0, PHY
7, 0, RCH
8, 0, ES

// Edges:
// ID, StartVertex, EndVertex, Directional, Speed, Length
0, 0, 1, 0, 0.1, 4
1, 1, 2, 0, 1.2, 8
2, 2, 3, 0, 2.3, 7
3, 2, 5, 0, 2.5, 4
4, 3, 4, 0, 3.4, 9
5, 3, 5, 1, 3.5, 14
6, 4, 5, 0, 4.5, 10
7, 5, 6, 0, 5.6, 2
8, 0, 7, 0, 0.7, 8
9, 1, 7, 1, 1.7, 11
10, 2, 8, 0, 2.8, 2
11, 6, 7, 0, 6.7, 1
12, 6, 8, 1, 6.8, 6
13, 7, 8, 0, 7, 7

// Events:
// Edge, Type
0, 3

```

Figure 3: Store