Final Project Report

CS 330

Naing Lin Tun

14th December 2018

1 Introduction

As one of the students of CS 330: Computer Networking course in Fall 2018, I learned about each networking layer and how they work together. Particularly, what strikes me the most interesting is the two most common protocols from the transport layer: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). We were taught in details about how one protocol is different from another, its advantages and drawbacks, and how each protocol is used. Perhaps, what seems amazing to me is even for the same purpose of transferring packets from the client to the destination (and vice versa), the way how these two protocols do is very different that sometimes, a change in protocol could entirely break the functionality of the application. For this reason, as my final project, I had undertaken the task to change the original TCP protocol that is used in Craft (a simple Minecraft clone) to UDP. I also observed how latency and packet loss could effect differently for each protocol by running the game server and client on Emulab nodes with different network configurations.

2 A little bit about TCP and UDP

Since the main goal of this project is to explore the effect of TCP and UDP, the knowledge about those two protocols is undoubtedly a prerequisite.

Transmission Control Protocol (TCP) is a transport layer protocol that is used when a reliable transmission of packets is necessary. By "reliable" it means that the protocol guarantees the delivery of packets from the source to the destination and also supports in-order delivery of packets. It even does error-correction when some sort of errors are introduced in the packets along the transmission. TCP accomplices these reliable features with "connection establishment" via three-way handshake between the source and the client. The threeway handshake works as follows:

- The client sends SYN (synchronization) packet to the server
- When the server receives the SYN packet from the client, it responds back with SYN/ACK (acknowledgement) packet to the client
- The client sends back the SYN packet, and the connection is now established

From now onwards, every data sent from the client, the server will respond with an ACK. If the client does not receive the ACK packet from the server (either the server does not receive the packet that the client sends or the ACK packet gets lost), the client will send back the same data after some time (depending on the implementation). Thus, the guaranteed delivery of packets can be accomplished this way. In addition, every TCP packet has a 32-bit Sequence number in its header field. It sets a number for every packet that the client sends and the server responds with the ACK packet, including that sequence number. This ensures in-order delivery of packets.

However, all these reliable features of TCP comes with a great cost. The fact that the server has to respond with ACK packet every single time the client sends the data is expensive (real applications might tweak this “resending ACK” part though to increase efficiency). Also, because TCP guarantees the in-order delivery of packets, keeping track of sequence numbers and resending data can add a significant load on the transport side of the application. For these reasons, TCP tend to be slower than UDP.

On the other hand, UDP is completely different from TCP. All those reliable features that TCP supports, UDP does not have it at all. It does not establish any connection between the client and the server, does not have any sequence number, and does not do error correction. In simple terms, UDP just sends the data from the client, and it does not care whether the server side receives it or not. As a result, this makes UDP a faster transport protocol since all the heavy-features of TCP are not present there. It is a “best-effort” protocol.

3 Getting Started

Since one of my goals is to alter TCP to UDP in Craft, the first step of my project is to make sure that I can install, modify, and run the game on my machine. My laptop’s (basic) specifications are as follows:

- Windows 10 Pro 64-bit
- Intel Core i7 7th Gen with Nvidia GTX 1050
- 16 GB RAM with 512 TB SSD

Cygwin64 Terminal was used in order to access UNIX command lines. However, `telnet` is, by default, not ready to use on Windows Cygwin. Thus, I have to reinstall Cygwin with Telnet package in order to use `telnet` on the Cygwin terminal.

3.1 Installation and Dependencies

Installing and running Craft on a local machine require the following steps:

- Clone Craft’s github repository

- Install Cmake, MinGW, and Curl (depending on OS)
- Compile and run the game
- Test out multiplayer feature by running your own server in the terminal and connecting the game to the IP address of the server

All of the above steps are described in details in Craft's README file on Github. Also, because Craft's server is written in Python2 and client is written in C, it is also necessary to install Python2 and C in my machine. Therefore, knowledge of socket programming in both Python and C is also essential for this project.

3.2 Understanding Craft's protocol for player actions

In order to understand how Craft is processing the incoming and outgoing data for player movements, I need to understand how those graphical data are represented in the code first. Craft uses ASCII line based protocol to represent player's position and movements. Again, Craft's README file on Github provides details on the protocol, but I only deal with movements of the player for this project. To summarize, player's position are sent from the client to the server in the form of 6-tuple `P, x, y, z, rx, ry` where:

- P represents position. This is not a variable. P is fixed for any position-based data.
- x, y, z indicate the coordinate of player's location (z is the vertical vector. Craft needs this because players can fly and jump).
- rx, ry is the camera position of the player (Craft needs this since it is an FPS game)

On the server side, this string is slightly transformed into `P, pid x, y, z, rx, ry`, where *pid* is the player id. Because Craft is a multiplayer game, the server needs to distinguish between player, and it does this by attaching ID to each player.

The mechanism of these protocols can be easily tested out by listening to the server with `telnet`. First of all, I launched the server on my own laptop by typing `python2 server.py localhost 8000` (note: `python2` has to be replaced with `py` or `py2` or `python` depending on the Python installation) on the command prompt. Then, I run the game and typed `/online localhost 8000` to connect to the server (here, the game is the client). Afterwards, on Cygwin terminal, I listened to the server via `telnet localhost 8000`. As I made changes in the game, I could see ASCII strings printed out on `telnet` terminal. This aided me a lot in understanding the ASCII line based protocol of Craft (not that the README file does a poor job explaining; it's just that it personally helps me learn better when I get a handson experience on something).

4 Craft's TCP implementation

Next, I tried to understand TCP implementation in Craft. Below is the extract of `server.py` file where TCP implementation can be found:

```python
class Server(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    allow_reuse_address = True
    daemon_threads = True

class Handler(SocketServer.BaseRequestHandler):
    def setup(self):
        self.position_limiter = RateLimiter(100, 5)
        self.limiter = RateLimiter(1000, 10)
        self.version = None
        self.client_id = None
        self.user_id = None
        self.nick = None
        self.queue = Queue.Queue()
        self.running = True
        self.start()
    def handle(self):
        model = self.server.model
        model.enqueue(model.on_connect, self)

        try:
            buf = []
            while True:
                data = self.request.recv(BUFFER_SIZE)
                if not data:
                    break
                buf.extend(data.replace('\r\n', '\n'))
                while '\n' in buf:
                    index = buf.index('\n')
                    line = ''.join(buf[:index])
                    buf = buf[index + 1:]
                    if not line:
                        continue
                    if line[0] == POSITION:
                        if self.position_limiter.tick():
                            log('RATE', self.client_id)
                            self.stop()
                            return
```

```python
38                      else:
39                          if self.limiter.tick():
40                              log('RATE', self.client_id)
41                              self.stop()
42                              return
43                      model.enqueue(model.on_data, self, line)
44          finally:
45              model.enqueue(model.on_disconnect, self)
```

As we can see, Craft uses `SocketServer` module to implement TCP server. Here, we don't need to focus that much of `setup()` and the contents inside `while` loop at line 23 (it's just processing of the ASCII line based protocol, not TCP). What we need to know is that when the client first initiates a connection to the (TCP) server, the threeway handshake occurs. This is when `handle()` is called. At line 14, `model.enqueue(model.on_connect, self)` "connects/adds" the player into the game. Once the connection has been established, it waits for the client's data until the client disconnects from the server (TCP can detect this since connection is established between the server and client). The code `model.enqueue(model.on_data, self, line)` is used to update the player's position in the game model. When the client disconnects, the code from `finally` block is called. The code `model.enqueue(model.on_disconnect, self)` will "remove" the player object from the game model and thus, disconnects it.

In order to play around with Craft TCP's implementation, I wrote a simple client program that let me sent positions to the server. It is as follows:

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


#define PORT 8000 // change PORT as desired

int main(int argc, char const *argv[])
{
    struct sockaddr_in address;
    int sockfd = 0;
    struct sockaddr_in serv_addr;


    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("\n Socket creation error \n");
        return -1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr) <= 0) {
        perror("\nInvalid address/ Address not supported \n");
        return -1;
    }

    // TCP connection establishment
    if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
     ↪  < 0) {
        perror("\nConnection Failed \n");
        return -1;
    }


/*********************************************************/
/* This code reads the standard input of the user
 * for the player's position (in ASCII line based
 * protocol).
 * Test:
 *    input: P,45,17.97,1.9,2.0,-0.17
 *    and in the game, type "/list" to figure out
 *    the player name, and type "/goto playerName".
 *    The player block should be "visible" now.
 *
 *
 * When this code is replaced with the one below,
 */
```

```
/*    int n = 0;
    int maxlen = 100;
    char buffer[maxlen];

    while ((n = recv(sockfd, buffer, maxlen, 0)) > 0) {
        char *line = NULL;
        size_t leng;
        getline(&line, &leng, stdin);
        //puts(line);
        send(sockfd, line, strlen(line), 0);
        free(line);
    }*/

/*************************************************************/

///******* Can replace with the above code **************************
    char *movement1 = "P,0.25,16.75,-1.22,0.28,-0.51\n";
    char *movement2 = "P,-53.15,28.35,-124.85,6.25,-0.66\n";
    char *movement3 = "P,0.41,20,-1.22,0.28,-0.51\n";
    char *movement4 = "P,0.25,16.75,-1.22,0.28,-0.3\n";
    char *movement5 = "P,45,17.97,1.9,2.0,-0.17\n";
    char *movement6 = "P,30,17.97,1.9,2.0,-0.17\n";
    char *movement7 = "P,30,13,1.9,2.0,-0.17\n";

    struct timeval start, stop;
    double secs = 0;


    int num_mov = 10;
    gettimeofday(&start, NULL);
    //while (1) {
    for (int i = 0; i < num_mov; i++) {
        send(sockfd, movement5, strlen(movement5), 0);
        printf("Client message: %s", movement5);
        usleep(500000);

        send(sockfd, movement6, strlen(movement6), 0);
        printf("Client message: %s", movement6);
        usleep(500000);


        send(sockfd, movement7, strlen(movement7), 0);
```

```
            printf("Client message: %s", movement7);
            usleep(500000);
        }
        gettimeofday(&stop, NULL);
        secs = (double)(stop.tv_usec - start.tv_usec) / 1000000 +
         ↪  (double)(stop.tv_sec - start.tv_sec);
        printf("%d movements sent\n", num_mov * 3);
        printf("time taken %f\n",secs);
        close(sockfd);


    //*********************************************************/


    /** Behaves like "telnet" in a way that the client console
     *  outputs the positions of player(s) as they change in the
     *  game. However, this code outputs the introductory lines
     *  such as "Welcome to Craft!" repetitively sometimes.
     *  (Telnet only outputs those introductory lines once when
     *  the connection is first made).
     *
     *  source:
  ↪  https://gist.github.com/suyash/2488ff6996c98a8ee3a84fe3198a6f85
     */
    /*********************************************************
        int n = 0;
        int len = 0, maxlen = 100000;
        char buffer[maxlen];
        char* pbuffer = buffer;

        while ((n = recv(sockfd, pbuffer, maxlen, 0)) > 0) {
            pbuffer += n;
            len += n;

            buffer[len] = '\0';
            buffer[len] = '\0';
            printf(buffer);

        }
    *********************************************************/
        return 0;
    }
```

5 Changing TCP to UDP in Craft

Initially, I thought changing the server from TCP to UDP was going to be a breezeway after going through socket programming tutorials. I thought changing `Server` class argument from `SocketServer.TCPServer` to `SocketServer.UDPServer` would just be enough. In fact, I thought UDP implementation would even be shorter since we do not have to do connection establishment. However, the actual outcome was a complete opposite of what I was expecting. The game implementation and protocols are so intertwined and reliant on TCP features that changing to UDP could break everything. It was excruciatingly challenging to change TCP to UDP since I ended up changing some of the game's protocol itself. In order to do that, I also had to understand how the game maintains players' data inside the `Model` class.

Of course, my first attempt in changing TCP to UDP was to change all syntaxes, such as `SocketServer.TCPServer` to `SocketServer.UDPServer`, `send` to `sendto`, etc, and see what happens. Then, I could not connect the game to the server anymore; it would just crash instead. So, I wrote my own simple UDP client where I can send movements to the UDP game server. My UDP client is the following:

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

#define PORT 8000 // change PORT as desired

int main(int argc, char const *argv[])
{
    int sockfd = 0;
    struct sockaddr_in serv_addr;


    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("\n Socket creation error \n");
        return -1;
    }

    memset(&serv_addr, 0, sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    serv_addr.sin_addr.s_addr = INADDR_ANY;


    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr) <= 0) {
        perror("\nInvalid address/ Address not supported \n");
        return -1;
    }

    char *movement1 = "P,0.25,16.75,-1.22,0.28,-0.51\n";
    char *movement2 = "P,-53.15,28.35,-124.85,6.25,-0.66\n";
    char *movement3 = "P,0.41,20,-1.22,0.28,-0.51\n";
    char *movement4 = "P,0.25,16.75,-1.22,0.28,-0.3\n";
    char *movement5 = "P,45,17.97,1.9,2.0,-0.15\n";
    char *movement6 = "P,30,17.97,1.9,2.0,-0.16\n";
    char *movement7 = "P,30,13,1.9,2.0,-0.17\n";

    int n = 0;
    int maxlen = 100;
    char buffer[maxlen];

    int num_mov = 10;
    for (int i = 0; i < num_mov; i++) {

        sendto(sockfd, (const char *) movement5, strlen(movement5), 0,
               (struct sockaddr *) &serv_addr, sizeof(serv_addr));
        usleep(500000);
        printf("Client message: %s", movement5);


        sendto(sockfd, (const char *) movement6, strlen(movement6), 0,
               (struct sockaddr *) &serv_addr, sizeof(serv_addr));
        usleep(500000);
        printf("Client message: %s", movement6);

        sendto(sockfd, (const char *) movement7, strlen(movement7), 0,
               (struct sockaddr *) &serv_addr, sizeof(serv_addr));
        usleep(500000);
        printf("Client message: %s", movement7);
    }
    char* logout = "logout\n";
    sendto(sockfd, (const char *) logout, strlen(logout), 0,
           (struct sockaddr *) &serv_addr, sizeof(serv_addr));
    printf("%d movements sent\n", num_mov * 3);
    close(sockfd);

    return 0;
}
```

Then, I complied my C client and ran the command `./client_udp localhost` (`client_udp` is the program name and UDP server is already running on `localhost`). As expected, the game was not working as it should. After the client sent its first movement, the server got stuck in the infinite loop, and it did not proceed to receiving the second movement. So, I took out the `while True:` part from the server `handle()` function and observed what happened too. So, my server `handle()` became as follows:

```python
def handle(self):
    model = self.server.model
    model.enqueue(model.on_connect, self)

    try:
        buf = []
        data = self.request[0]

        if not data:
            break
        buf.extend(data.replace('\r\n', '\n'))

        while '\n' in buf:
            index = buf.index('\n')
            line = ''.join(buf[:index])
            buf = buf[index + 1:]
            if not line:
                continue
            if line[0] == POSITION:
                if self.position_limiter.tick():
                    log('RATE', self.client_id)
                    self.stop()
                    return
            else:
                if self.limiter.tick():
                    log('RATE', self.client_id)
                    self.stop()
                    return
            model.enqueue(model.on_data, self, line)

    finally:
        model.enqueue(model.on_disconnect, self)
```

The result was quite interesting that it made me understand more about the differences between TCP and UDP even more. Everytime the client sends a movement to the server, `handle()` gets called, in contrast to TCP where `handle()` is only called once during threeway handshake. What ends up happening is that every movement adds a new player to the game. So, sending 20 movements creates 20 new players at their respective position. The reason UDP is calling `handle()` for every single send is because there is no connection establishment for the server to realize that the connection is from the client that has already connected to the server before. Thus, the server treats every send as a new connection and triggers the `model.enqueue(model.on_connect, self)` code, which creates a new player.

I did not want to create a new player for every single send; instead, I just want the currently-existing player's position to get updated. Thus, I will have to know what exactly is the Model's function `on_connect` is doing. The following is the `on_connect` function from `Model` class:

```python
def on_connect(self, client):
    client.client_id = self.next_client_id()
    client.nick = 'guest%d' % client.client_id
    log('CONN', client.client_id, *client.client_address)
    client.position = SPAWN_POINT
    self.clients.append(client)
    client.send(YOU, client.client_id, *client.position)
    client.send(TIME, time.time(), DAY_LENGTH)
    client.send(TALK, 'Welcome to Craft!')
    client.send(TALK, 'Type "/help" for a list of commands.')
    self.send_position(client)
    self.send_positions(client)
    self.send_nick(client)
    self.send_nicks(client)
```

Somehow, I needed to modify `on_connect` function to not connect the player if the connection was from the already-existing player in the game. One way to accomplish this was to store the set of players' IP addresses, and if the incoming connection was from that set, we knew that it was from the same client. Thus, I created a global variable `clients_ip = set()`, and at the start of `on_connect`, I added a condition to only "add" the player to the game `client.client_address[0]` was not in the `clients_ip` set. When adding the player to the game, the function would also its IP address to `clients_ip` set. The caveat with this approach, though, was that there could not be different players from the same IP address, such as launching multiple clients from the same machine.

Just modifying `on_connect` was still not enough for fully functioning UDP server. For TCP, if the client manually disconnected from the game, the server could detect this thanks to the connection establishment. But for UDP, the server could not do this. Thus, I need to change the game protcol slightly. I made the client side send "logout" as the final packet, and when the server receives the packet "logout", it would call `on_disconnect`. This is still not enough. The `on_disconnect` call (from line 32 of `handle()` function) also accepts `self`, client socket object, as its parameter to disconnect from the game. So, I needed to make sure that `on_disconnect` called the correct object. Thus, I created a dictionary `client_obj` that mapped `IP address` to `client socket object` and cached the entry when the player was first created by `on_connect` function. Then, the disconnect function call in line 32 of `handle()` function became
`model.enqueue(model.on_disconnect, client_obj[self.client_address[0]])`.

There was still one problem that I needed to fix. If the existing player in the game disconnects and reconnects back, `on_connect` does not add that player, and thus, `on_disconnect` also fails (it cannot disconnect the player that is not in the game). This was because I forgot to delete the IP of disconnected player from `clients_ip` set. Thus, inside `on_disconnect` function, I added `clients_ip.remove(client.client_address[0])`. After fixing this issue, everything worked as it should (at least for position movements and with my simple UDP client, not with the game client).

Below is the code for `handle()` (the important parts that I changed are indicated with yellow highlight):

```
1  ######################### Three global variables ####################
2  clients_ip = set()
3  client_obj = {}
4  count = 0 # to track the number of packets received (the number of
   ↪ requests triggered)
5  #####################################################################
6  def handle(self):
7      global count
```

```
        model = self.server.model
        model.enqueue(model.on_connect, self)
        data = self.request[0]
        if data != "logout\n":
            count += 1
            print(data[:-1])
            model.enqueue(model.on_data, self, data[:-1])
            sock = self.request[1]

        else:
            print(data[:-1])
            print("%d packets received" % count)
            model.enqueue(model.on_disconnect,
             ↪  client_obj[self.client_address[0]])
```

For `on_connect()` from `Model` class:

```
    def on_connect(self, client):
        global clients_ip
        global client_obj
        global clients_conn_status
        if client.client_address[0] not in clients_ip:
            client_obj[client.client_address[0]] = client;
            clients_conn_status[client.client_address[0]] = True
            client.client_id = self.next_client_id()
            client.nick = 'guest%d' % client.client_id

            clients_ip.add(client.client_address[0])

            log('CONN', client.client_id, *client.client_address)
            client.position = SPAWN_POINT
            self.clients.append(client)
            client.send(YOU, client.client_id, *client.position)
            client.send(TIME, time.time(), DAY_LENGTH)
            client.send(TALK, 'Welcome to Craft!')
            client.send(TALK, 'Type "/help" for a list of commands.')
            self.send_position(client)
            self.send_positions(client)
            self.send_nick(client)
            self.send_nicks(client)
```

Below is the code for `on_disconnect()` from `Model` class:

```python
    def on_disconnect(self, client):
        clients_ip.remove(client.client_address[0])
        log('DISC', client.client_id, *client.client_address)
        self.clients.remove(client)
        self.send_disconnect(client)
        self.send_talk('%s has disconnected from the server.' % client.nick)
        global count
        count = 0
```

To recap, I altered `on_connect` function in a way that in only added the player once, not for every single movement. I also made the server detect the disconnection of the player by sending a "logout" string from client to the server. Then, I changed `on_disconnect` function to disconnect the correct player by passing the right client socket object. Also, I fixed the issue where the disconnected player should be able to reconnect again.

6 Carrying out experiment with Craft's TCP and UDP

I first ran experiments on loopback configuration (without bandwidth constraint, packet loss, latency) to see how the game "normally" performs. The visual effects of all experiments mentioned from here onwards were also recorded in my public Google Drive folder. Later, I set different network configurations (with Emulab, which is explained in next subsection) to see how the performance changes. Below is the terminal scripts for client and server side for TCP:

```
naing@Naing /cygdrive/c/Users/naing/Documents/msi GE60 2PC _backup/Naing 2/IWU/6th semester/
    CS 330/workspace_c/test
$ ./client2 127.0.0.1
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
30 movements sent
time taken 15.012940
```

and for TCP server:

```
naing@Naing /cygdrive/c/Users/naing/Documents/msi GE60 2PC _backup/Naing 2/IWU/6th semester/
    CS 330/workspace_c/test
$ ./client2 127.0.0.1
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.17
Client message: P,30,17.97,1.9,2.0,-0.17
Client message: P,30,13,1.9,2.0,-0.17
30 movements sent
time taken 15.012940
```

The visual effects of these data tranmssion could be observed from the game window as well, please see the file `tcp_loopback_game.mp4`.

Below is the transcript for UDP client:

```
naing@Naing /cygdrive/c/Users/naing/Documents/msi GE60 2PC _backup/Naing 2/IWU/6th semester/
    CS 330/workspace_c/test
$ ./client_udp 127.0.0.1
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
30 movements sent
```

and for UDP server:

```
naing@Naing /cygdrive/c/Users/naing/Documents/msi GE60 2PC _backup/Naing 2/IWU/6th semester/
    CS 330/workspace_c/test
$ ./client_udp 127.0.0.1
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
Client message: P,45,17.97,1.9,2.0,-0.15
Client message: P,30,17.97,1.9,2.0,-0.16
Client message: P,30,13,1.9,2.0,-0.17
30 movements sent
```

All packets arrived in order without any loss or delay, at least for now with loopback configuration.

For all the experiments, I tried to record the local time when the client sent the data and also record the local time when the server received it. Then, compare those two times to see how much delay there were between sends and receives. But it seemed Emulab nodes had a different time zone, and thus, my plan did not work. I tried setting start and end time for client: the start time was when the client sends the first packet, and the end time was when the server sends the disconnect packet. Then, subtract those times to report the time that the client was run. I also did the same for the server (the starting point being the first recv call, and the end point was when the receives a disconnect packet from the client) and compared the time that the client and server was run. I was hoping that those two times would be different if there is a delay between each send from the server side (at least for TCP). However, that did not seem to work too since both client and server seemed to have same runtimes even though when delays (latency) were introduced. I don't know why, but it just doesn't work.

6.1 Setting up different network configurations in Emulab

Since I want to explore the effect of throughput, packet loss, and latency on TCP vs UDP, I need to be able to set different network configurations. With the help of Professor Liffiton, Emulab allows me to set up a profile with two nodes, which is connected via a link. I can modify such link to have the configuration that I desire. Multiple profiles can also be generated for multiple configurations.

After setting up the configuration for the link between two Emulab nodes, Professor Liffiton provided me with the script to forward the incoming packets from `nodeA` to `nodeB` via that configured link. The script is as follows:

```
#!/bin/sh

PUBLIC_IP=155.98.39.112
PORT=8000

sysctl -w net.ipv4.conf.all.forwarding=1

# Flush nat table
iptables -F -t nat

# Forward port £PORT to 192.168.1.2, both DNAT and SNAT for
 ↪  bidirectional, both TCP and UDP
iptables -t nat -A PREROUTING -p tcp -d $PUBLIC_IP --dport $PORT -j DNAT
 ↪  --to-destination 192.168.1.2
iptables -t nat -A POSTROUTING -p tcp --dport $PORT -d 192.168.1.2 -j SNAT
 ↪  --to-source 192.168.1.1
```

```
iptables -t nat -A PREROUTING -p udp -d $PUBLIC_IP --dport $PORT -j DNAT
  ↪ --to-destination 192.168.1.2
iptables -t nat -A POSTROUTING -p udp --dport $PORT -d 192.168.1.2 -j SNAT
  ↪ --to-source 192.168.1.1
```

`PUBLIC_IP` value should be set to `nodeA`'s public IP, which can be found by using `ifconfig` command. The above script will forward incoming packets of `nodeA` on port 8000 to `nodeB` via the Emulab link. Thus, if I set the packet loss of the link to 30%, it will forward packets with a loss of 30%.

The bash script should be transferred to `nodeA` via `scp`. Afterwards, on `nodeA`, the command `chmod +x ./start_routing.sh` should be run to make the script executable. Then, execute the script with `sudo ./start_routing.sh` command for packet forwarding to start.

Afterwards, I transferred the whole Craft repository to the nodes, and also my own TCP and UDP client (how to run and compile the game is outlined in Craft's README file). I ran the server on `nodeB` using the command
`python server.py 192.168.1.2 8000`. `192.168.1.2` is a local IP address of `nodeB` which `nodeA` can access to via Emulab link. `server.py` can also be replaced with other server files, such as `server_udp.py`. Then, I ran the client from my machine that sends data to `nodeA`'s public IP on port 8000, which in turns forward the packets to `nodeB`.

The reason why this packet forwarding is convenient is because I can run the server on `nodeB` while run the client just on my laptop (connecting to `nodeA`'s public IP on port 8000). This means I can also run the actual game window from my laptop and make it connect to `nodeA` on port 8000 as well. Thus, running the game window is also possible – without packet forwarding, I would have to run the game on `nodeA` (rather than from my laptop), but it's impossible to visually see the game window launched on `nodeA` (not that I know of).

6.2 The effect of throughput in Craft's TCP and UDP

First of all, I was just sending the positions as data from the client to the server. So, the packet for each position would be very small, and I would have to change the bandwidth of Emulab link within a small enough range to see its effect.

Initially, I set the bandwidth of the link to 1 Kbps. It did not seem to have any effect (I got the same results as a local throughput test). Thus, I reset the bandwidth value to 0.1 Kbps and less, but Emulab did not allow me to do that. I tried setting the bandwdith to "1/10" instead of 0.1. Emulab would no longer show me error, but I could not sense any difference with the loopback configuration too. I thought may be “1/10” was still large; so, I set it to “1/1000000”. Still, there was no difference in the outcome. Thus, I assume it was because 1 Kbps is the lower limit for bandwidth value, but I could not find any documentation to backup this (there are only documentations about packet loss).

6.3 The effect of packet loss in Craft’s TCP and UDP

The effects of packet loss on Craft’s TCP had some interesting results. Below was the terminal transcript from the server side run on `nodeB` when the client from my laptop connected to `nodeA`, which in turn forwarded the packets to `nodeB` with 30% packet loss link:

```
ntun@node2:~/Craft % python server.py 192.168.1.2 8000
2018-12-11 11:01:28.445746 SERV 192.168.1.2 8000
Called
P,45,17.97,1.9,2.0,-0.17

2018-12-11 11:01:29.227013 CONN 1 192.168.1.1 55404
P,30,17.97,1.9,2.0,-0.17

P,30,13,1.9,2.0,-0.17

P,45,17.97,1.9,2.0,-0.17

P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17

P,45,17.97,1.9,2.0,-0.17

P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17


7
----------------------------------------
Exception happened during processing of request from ('192.168.1.1', 55404)
Traceback (most recent call last):
  File "/usr/lib/python2.7/SocketServer.py", line 596, in process_request_thread
    self.finish_request(request, client_address)
  File "/usr/lib/python2.7/SocketServer.py", line 331, in finish_request
    self.RequestHandlerClass(request, client_address, self)
  File "/usr/lib/python2.7/SocketServer.py", line 652, in __init__
    self.handle()
  File "server.py", line 120, in handle
    data = self.request.recv(BUFFER_SIZE)
error: [Errno 104] Connection reset by peer
```

```
49 ----------------------------------------
50 2018-12-11 11:01:44.242521 DISC 1 192.168.1.1 55404
51 2018-12-11 11:01:44.245183 guest1 has disconnected from the server.
52 2018-12-11 11:01:44.247172 Time taken 15.0201809406
```

From the terminal log, we could see that the (TCP) server did indeed receive the packets in order. However, the way that “packet retransmission” in case of packet loss was interesting. So, when there was a packet loss, I was expecting it for the client to send the loss packet back *before* sending the next packet. However, what I think actually happened was that the client resent the loss packet along with the next packet in line. I had set up the server in a way that it would print a movement on each line and then add additional new (blank) line before printing out another movement (in a single request). But as we can see, some of the movements came in together as a group in a single request (for instance, line 18-34). As I was watching the server log, I could also see those set of movements coming in at the same time, rather than one at a time like those that were not affected by packet loss. This effect is demonstrated in the file `tcp_loss30_fixed.mp4` file. The visual (game) representation of packet loss can be seen in `tcp_loss30_loop.mp4` file.

The dropped packets (due to loss) also only resent when there was an additional `send` lining up from the client side. The problem with this was that when the client became disconnected, it would no longer send anything to the server side and the server would never receive the loss packet along with all the later packets, even the “client is disconnected” packet. As a consequence, the player could be stuck in the game forever (until the server shuts down) even if he has actually disconnected from the game. It seems that TCP in Craft is so determined in sending the packets in the correct order that it would not send any packets that comes after the dropped packet (unless the resend is successful), even if it means the server will not get several packets.

I repeated the experiment by setting the client to send the movements in an infinite loop and visually observed the effects from the game window. I launched Craft game and typed in `/online [nodeA's public IP] 8000`. The presence of 30% packet loss did not definitely create a smooth experienece in the game. The movements were very abrupt: the character would just stand in one position for quite a time, then suddenly move from one position to another in a very fast speed, but still in the correct order. This was clearly the behavior of lag that we can usually see in games.

For UDP, I used the same network configuration (30% packet loss), but I set different movements just to visually see that in-order delivery was not supported. These three movements were sent in order from the client side:

- `P,45,17.97,1.9,2.0,-0.15`
- `P,30,17.97,1.9,2.0,-0.16`
- `P,30,13,1.9,2.0,-0.17`

Notice the last tuple of each movement. If the position packets arrived in order, the last tuple of each packet should be in -0.15, -0.16, -0.17 order. I ran UDP server on `B` and connected the client to it, the same way I did for TCP. Below was my terminal transcript for the server side:

```
ntun@node2:~/Craft % python server_udp.py 192.168.1.2 8000
2018-12-11 12:04:31.121654 SERV 192.168.1.2 8000
P,30,13,1.9,2.0,-0.17
2018-12-11 12:04:34.456793 CONN 1 192.168.1.1 13084
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
 P,30,17.97,1.9,2.0,-0.16
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,45,17.97,1.9,2.0,-0.15
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
logout
21 packets received
2018-12-11 12:04:48.504614 DISC 1 192.168.1.1 13084
2018-12-11 12:04:48.507918 guest1 has disconnected from the server.
```

As we can see clearly from the log, the packets did not arrive in order (the last tuple were not in -0.15, -0.16, -0.17 order). We can evidently see the packet loss too. Out of 30 packets sent by the client, only 21 arrived. UDP just tried to send whatever data it had currently without emphasis on retransmission or in-order delivery. It is in this sense UDP is a "best-effort" protocol. The problem with this UDP protocol, though, was that the "logout" packet itself could get lost, and the player would be stuck in the game forever (until the server shuts down), just like in TCP.

This behavior of UDP is usually preferable in (PvP) games because they are time sensitive and faster delivery is prioritized over reliable transmission. In fact, TCP's reliable behavior of Craft can even be bad in the presence of high packet loss; it could never update the player with the packets that come in after the loss packet. Thus, it is as though TCP itself is creating a chain of packet loss itself. While TCP may seem to solve the issue of packet loss, it is in fact making it worse (as mentioned above, it could even not send the "client is disconnected" packet as if the packet was lost). On the other hand, UDP attempts to send whatever data the client has in a much faster manner, and it does not make the issue worse than it already is. To see how client and server logs "almost instantly" sends and receives data even in the presence of packet loss, see `udp_loss30.mp4` file

I repeated the experiment with several packet loss values, but the behaviors were generally similar, just in less or greater frequency depending on the loss percentage. Of course, the lesser packet loss percentage it is, the lesser "loss" effect we can see on both TCP and UDP, and vice versa. Loss value less than 10% were hard to notice; I had to repeat the experiment several times for the packet loss to significantly take effect. On the other hand, packet loss higher than 30% were hard to experiment (especially for TCP) because most of the packets would get loss; it was as though there were little to no packets sent from the client.

6.4 The effect of latency in Craft's TCP and UDP

I set the latency of Emulab link between `nodeA` and `nodeB` to 10 secs. Below is part of the transcript from TCP server among sending three movements `P,45,17.97,1.9,2.0,-0.17`, `P,30,17.97,1.9,2.0,-0.17`, `P,30,13,1.9,2.0,-0.17`:

```
ntun@node2:~/Craft % python server.py 192.168.1.2 8000
2018-12-14 05:29:27.415406 SERV 192.168.1.2 8000
Called
P,45,17.97,1.9,2.0,-0.17

2018-12-14 05:29:32.545887 CONN 1 192.168.1.1 7252
P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17

P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17

P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17

P,30,17.97,1.9,2.0,-0.17
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.17
P,30,17.97,1.9,2.0,-0.17
```

The result with 10 sec latency is somewhat similar with 30% packet loss for TCP server. I was expecting that in the presence of latency, the sender would not send any additional new packet until the delayed packet is received on the server side since TCP supports in-order delivery and guaranteed transmission. However, just like with packet loss, the delayed packet arrived together with some other packets behind it in *one singe request* (as mentioned before, I made the TCP server adds a blank line between two `recv` calls). So, when positions are printed all together in one lump without any blank lines between them, this indicated to me that all those positions were not one by one separately. The visual representation of this effect can be found in the following files

- `tcp_latency_10s_game.mp4`: this displays how the latency effect (lag) can look in game visually
- `tcp_latency_10s_terminal.mp4`: this shows the delay between the sends and receives by client and server terminals

On the other hand, the results with UDP were quite different and interesting. I sent the same set of three movements that I did with packet loss experiment above for UDP. Below was the result I received from the UDP server:

```
ntun@node2:~/Craft % python server_udp.py 192.168.1.2 8000
2018-12-14 05:49:55.903755 SERV 192.168.1.2 8000
P,45,17.97,1.9,2.0,-0.15
2018-12-14 05:50:06.333942 CONN 1 192.168.1.1 37768
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
P,45,17.97,1.9,2.0,-0.15
P,30,17.97,1.9,2.0,-0.16
P,30,13,1.9,2.0,-0.17
logout
30 packets received
2018-12-14 05:50:21.320322 DISC 1 192.168.1.1 37768
2018-12-14 05:50:21.323203 guest1 has disconnected from the server.
```

There were no packet loss, and every packet was in order. Even though I set the latency to 10 seconds, the delay was way lower than that, only about 2 seconds. I repeated the experiment multiple times, and there were a few packet losses (potentially delayed packet that was dropped), but the rate at which the server receives the packet was way faster than UDP, as if the server and client were running on loopback configuration.

I repeated the experiment with different latency values for both TCP and UDP. The effects were basically the same, just in more or less frequency. According to the data, it seems UDP performs way better than TCP in the presence of latency.

7 Conclusion

Honestly, the journey throughout the whole project to explore Craft's implementation and changing it had been brutal, yet very enjoying as well. I learned very much about how changing protocol from TCP to UDP in an application was not a simple task, unlike in those socket programming tutorials that can be found online. Everything in Craft is so tight-knit in a way that changing one simple thing in the protocol could break everything. Dealing with these transport protocols is definitely either a hit or miss: the application either works or it crashes and burns. I also learned how sending, receiving, and modelling data work in a real application (though Craft is not the most realistic application out there). In addition, I got an opportunity to gain a hands-on experience on how packet loss and latency can effect TCP and UDP differently (at least in Craft). Overall, I wouldn't have chosen a more fun project, and I didn't regret choosing a difficult one too as I gained way more knowledge than I had originally anticipated.