Ashley Teow and Justin Torre

CS4500: Assignment 5 [Part 2]: Networking

https://github.com/ashleyteow/sw-dev-a4

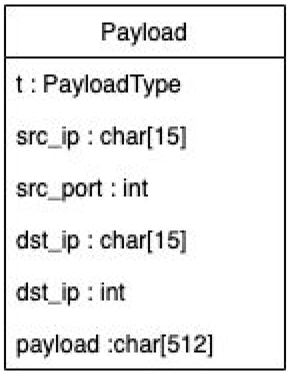
Introduction

The goal of this report is to explain our implementation details for the network layer prototype we designed for our upcoming project. Being able to send relevant and reliable messages between computers is important and the method described is reusable and applicable in many different applications. We will discuss the rough design of our API that contains our master server class as well as the client classes. Additionally, we will discuss some issues we had when trying to design this network layer API.

Design

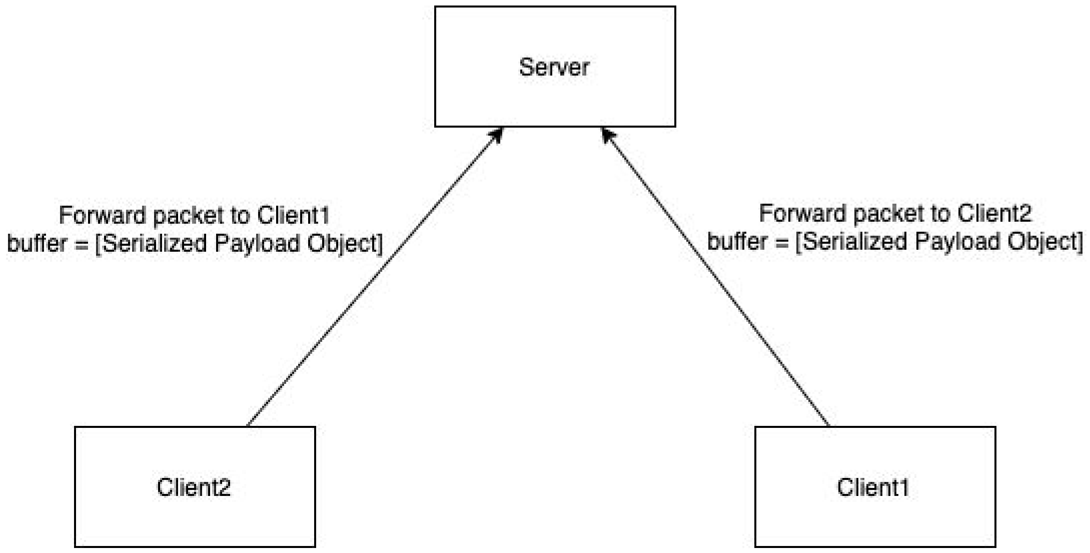
For registration, we have one single master class that stores a fixed number of clients. This master class is able to start the server and talk to all of its respective clients. There are then two steps in order to begin the server and wait for connections. The first is setting the server up to listen on a specific port, and the second is to actively wait for incoming connections.

The key to most of our implementation working properly is creating a Payload object that is very similar to a UDP packet. This Payload is capable of getting serialized and sent over the web socket then deserialize to be our data to process.

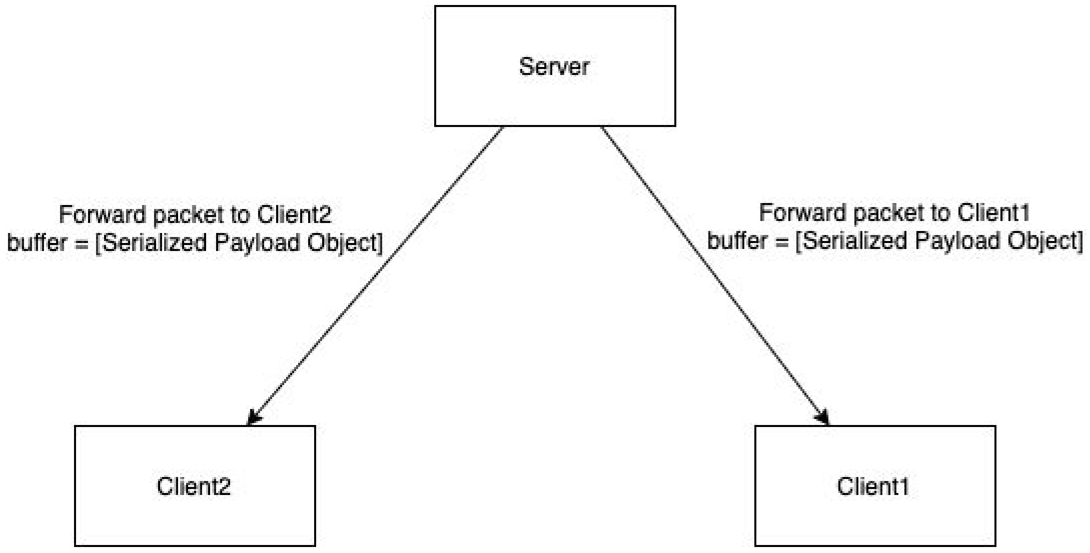


Please note variable `t`. This is an Enum that will represent what type our payload is. This will help the master understand what kind of message we are receiving. This will easily allows us to choose how to process messages in our master API.

Since the master is just forwarding packets it will act as a sort of router and forward packets.. The master will keep a list of seen ip's and what port they came from and manage all the clients appropriately. As shown below.


Server
Forward packet to Client1
buffer = [Serialized Payload Object]
Forward packet to Client2
buffer = [Serialized Payload Object]
Client2
Client1

Once a Payload object is received by the master it will then forward that packet to the destination specified in that packet. It will forward the entire Payload so that the receiving client can get the data needed by the sending client. Like so...


Server
Forward packet to Client2
buffer = [Serialized Payload Object]
Forward packet to Client1
buffer = [Serialized Payload Object]
Client2
Client1

Each client has a wait toggle that will allow the client to wait for a response. Here is some sudo code of what our API would look like.

```
Payload p = {
  t = FORWARD_PACKET,
  dst_ip = "192.168.1.1",
  dst_port = 8080,
  src_ip = "192.168.1.2",
  src_port = 8080,
  payload = "Hello from sender";
}

client_sender.send_message_to_master(payload = p, wait = true);
thread.spawn(client_receiver.process_response());
char* response = client_sender.get_response();
```

This example will send a packet from 192.168.1.2 to 192.168.1.1. The receiver will go into a listening mode, get the packet and process it accordingly then send a response back to the sender. Notice that the messages are sent to master. This is according to the spec.

Also the Master needs to keep a list of possible clients that can communicate with each other. The Master is also capable of sending messages in the form of Payloads to all their clients as well to update their local forwarding table. The way we are planning to handle this is to implement a thread that runs for each client that is listening for the master at all times.

Getting the client list to all the clients is done fairly easy. We would send a command to the master with the payload type equal to REQUEST_LIST which will send a list of ips back to the client that requested it with a list of all the IPs. The list is delimited by commas

Challenges and Open Issues

There are many challenges this implementation has to face. For instance, building a virtual network would have been very difficult if we didn't use Ip:port, but we were able to use it without diverging from the spec.
As of right now the master cannot handle multiple connections to it and only really forwards connections from one client to another as needed. Going forward, the proper way to handle this is when the master receives a message it spawns a new thread that handles the process of that message and then has the main thread continue listening for messages. We can maybe implement a message queue.