Michael Chang (machang), Mitchell Lee (mklee1)

18-441

Project 2 Design Document

final design: packet format, state transition diagram, changes to protocol, comparing to the checkpoint.

Backend System Summary:

In general, our backend system communicated with front-end or other backend systems using packets and sockets that we created. Within the BackendServer class, we have two main functionalities: receiving and transmitting. The receiving functionality is found in the getContent function, which forwards the request for the file, while transmitting is found in the main function.

Prior to receiving and transmitting, however, the front-end server must initialize a BackendServer object and call the addPeer method from BackendServer. This specifies a host and port number to add a peer on from the backend side. This is later used in the BackendServer.

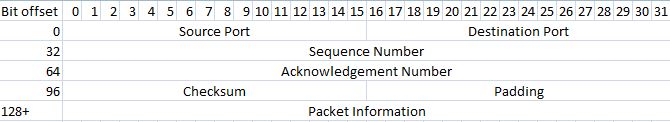
For the transmitting function (main), we wait until there is a request for a file, indicated by a packet that starts with "Send this file:" and then going through several initial connection set-up steps. Among these are attaining the bandwidth from the first backend server (which is connected to the front-end server. This bandwidth is obtained when the uri includes a config?rate=<byte/sec rate> and then is passed into the backend server to store and forward the information. Once the initial set-up is complete, we enter the transmit loop. Within this loop, we have a section that monitors the bandwidth. Every five seconds, the monitor checks the bandwidth. If the current transfer rate in the transport layer exceeds the given bandwidth, then we introduce a one-time delay until the next packet is sent. After the delay is added (or not), then the packet is sent (with a header indicating which index the array starts at) and the server waits for an ack packet. If the ack is received, then we can send the next packet. Otherwise, if there is a timeout, then we assume the packet was dropped and re-send the original packet.

For receiving the packet, we use a backend server which is connected to the front-end vodserver (which receives the HTTP requests). In this server, we request the file from a peer and receive a packet with the length of the requested file so that we know when to stop listening on the port. Lastly, we forward the maximum bandwidth specified by the Vodserver. Then, we enter the receive loop. Within this loop, we wait for a packet (denoted by a datagram packet starting with "startindex:"). This allows us to packet-drop correct by recognizing when a packet was dropped without having to send/receive another ack message. After the packet's data is received and copied, we then send and acknowledgement message to the peer server. When the loop is complete, we send the data back to the front-end (Vodserver) and view it through another Vodserver method.

Unfortunately, our implementation is limited to one thread. When we attempted to add multi-threading, we found that there was a race condition (that we couldn't quite solve) that caused the "receive" server to listen twice (once in the main function, and once in getContent) and were unable to solve the issue. Thus, we rolled back to a non-multithreaded solution and focused on other aspects of the project.

(Old) Packet Format:

Below, we have included our own packet format. It is similar to TCP, but is more of a barebones version. It includes less information, but we have chosen to retain some of it. For example, we chose to retain the source port and destination port for sending purposes. We also include the checksum for error correction. Lastly, we include the sequence and acknowledgement numbers to aid in setting up the connection. In order to maintain the format, we include 16 bits of padding after the checksum before sending the actual packet information.



(Final) Protocol Format:

Because we ended up (re)starting the final project 2, we ended up having to make several short-cuts. Among these were not implementing our own packet format. Instead of using a header that we designed on our own, we relied on UDP and simply added a simple "header" to the datagram packets, which were predefined. This included things such as "Bandwidth:", "Send this file:", "length:" and were used at specific times. As such, we did not use the sequence or acknowledgement number that could have been used in the three-way handshake method. Furthermore, we did not implement a checksum. However, if we had had the time, we would have checked if the first byte within a packet was transmitted correctly by sending an acknowledgement packet with the first byte and then checking it on the transmit side and returning an acknowledgement packet. Then, the transfer would continue.

(Old) State Transition Diagram:

Last\_Ack

Receive

Wait

TCP-lite Handshake Connection

Failed

Closed

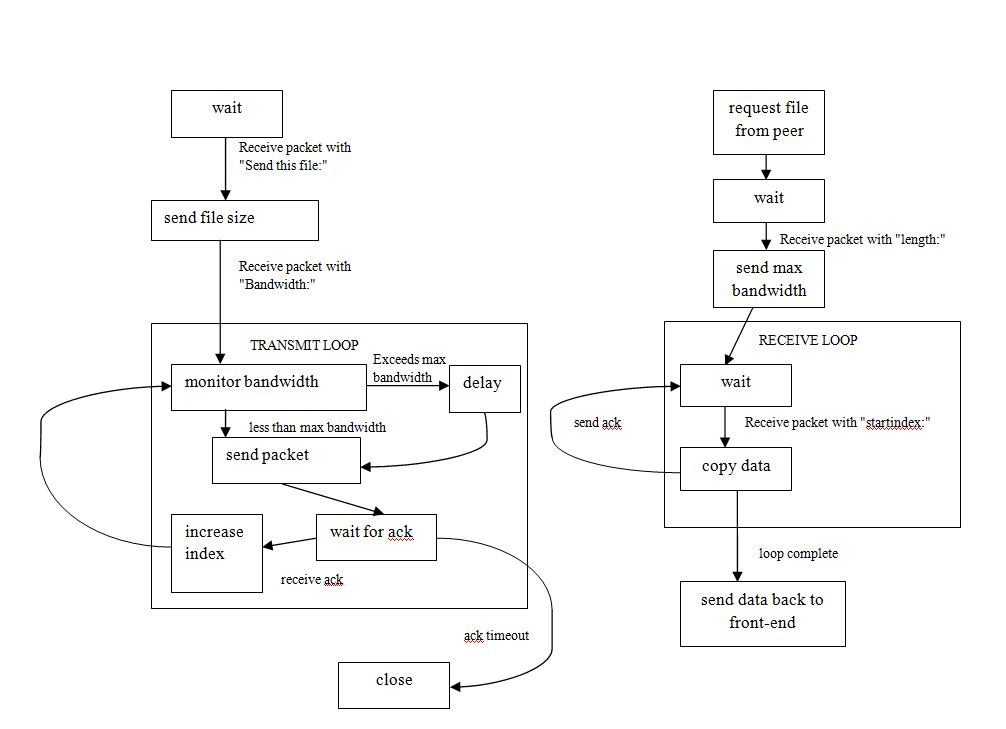
Ack Received

Ack sent

Syn Sent

Established!

Syn Received

(Final) State Transition Diagram

keep same index

BackendServer 2

BackendServer 1

Obviously, we can see from the changes in the state diagrams that the transitions are much different than what we anticipated at the checkpoint. We made a more clear distinction between the usages of the BackendServer class and also added clearer acknowledgement messages and packet drop correction.

Libraries Used:

DatagramSocket: used for opening and listening to a socket on a specified port.

DatagramPacket: used to send data back and forth, including connection set-up information, data, and acknowledgement messages

InetAddress: used for simple networking and determining the host names

CopyOnWriteArrayList: used for storing the added peers (via InetAddress)