CS 118 - Project 2

Alex Crosthwaite – Jacob Nisnevich – Jason Yang

June 7, 2016

1 Design

As a group, we implemented two classes to create our TCP packet abstraction, `Packet` and `Packet_info`. We also made the decision to *not* implement classes for the client and server for this project. Rather, we decided to simply execute the client and server code line-by-line in `client.cpp` and `server.cpp` respectively.

1.1 Packet

In our project, we used the `Packet` class for as our TCP packet abstraction. The class contained `SYN`, `ACK`, and `FIN` flags, in addtion to member variables for the sequence number, acknowledgment number, receive window, and the data contained in the packet. The class could either be initialized with each of these values or with default values.

1.2 Packet_info

We also implemented a `Packet_info` class that we used to keep track of data length, the time the packet was sent, and the timeout time for the packet.

1.3 Client

Overall, our implementation of the client was very much by-the-book and straightforward. As such, this will be a general run-through of our implementation of the client.

The first step in our client execution is to set-up the UDP socket. To do this, we created a helper function, `set_up_socket`, that takes in the command-line arguments and returns the socket file descriptor. This helper function sets up an `addrinfo` struct and finds and connects to an available socket.

The client then selects a random initial sequence number, initializes a `Packet` with this sequence number and a `SYN` flag, and sends it along the UDP socket that was previously set up and increments the sequence number. The client then sits in a loop until a `Packet` with `SYN` and `ACK` flags arrives, and then increments the sequence number. Then, the client sends a `Packet` with an `ACK` flag and again increments the sequence number.

Now that the connection has been established, the client initializes a window, which is implemented as an `unordered_map` of `uint16_t` to `Packet_info`, and enters a loop that continues until a `Packet` with a `FIN` flag has been received. Within the loop, the client discards all invalid `ACK` packets, updates the window with new packets, and either sends an acknowledgment if a data packet was received or a `FIN/ACK` if a `FIN` packet was received. The received data is written to the "received.data" file and the socket is closed when the client receives the final `ACK` packet.

1.4 Server

The server implementation also mainly followed the general specification of a TCP server although we did use the book's finite-state machine abstraction to implement TCP Tahoe congestion control.

The server, like the client, begins by initiating the UDP socket connection with a call to the server's version of the `set_up_socket` function, which takes the port number as a parameter and returns a socket file descriptor. This function simply finds an available socket and binds to it. The server also initiates a window, which like on the client is implemented as an `unordered_map` of `uint16_t` to `Packet_info`. The final step in setting up the server is to select a random initial sequence number.

The next step in the server's execution is to wait for a `Packet` with a `SYN` flag to come through the socket. Once this packet arrives, the server sets the acknowledgment number to be one greater than the received sequence number and sends a `SYN/ACK` packet with the appropriate sequence and acknowledgment numbers. Then, the server increments its sequence number. The server then waits for an `ACK` packet to come in and begins the file transfer process.

Before the file transfer begins, the server intializes a congestion window, a slow start threshold, a receive window, counters for packets sent, and boolean variables for keeping track of slow start, congestion avoidance, fast recovery, and fast retransmit. At this point, the server enters the file transfer loop, which it will stay in until the window is full or the entire file has been sent. The server first checks if the retransmission variable is set to true. If it is, the server will re-send the `base_num` packet and set retransmission to false. Otherwise, the server will transmit new data packets until the congestion window is full.

The server then wait to receive an acknowledgment packet. If the wait times out, the slow start threshold will be set to half the congestion window and the congestion window will be set to the maximum segment size. Furthermore, the slow start and retransmission values will be set to true and the congestion avoidance and fast recovery values will be set to false. If retransmission is true at this point, the server will skip the next part of the loop.

If the server reaches this point in the loop, then an acknowledgment packet from the client has been correctly received by the server. The server will then check if the acknowledgment number is a duplicate. If it is, the server will increase the congestion window by the maximum segment size if it is in fast recovery mode or increment the duplicate ack counter otherwise. Furthermore, If the server has received 3 duplicated acknowledgments, the slow start threshhold will be set to half the congestion window and the congestion window will be set to the slow start threshhold plus three times the maximum segment size. Furthermore, the fast recovery and retransmission values will be set to true and the congestion avoidance and slow start values will be set to false.

If the packet received was not a duplicate, then the server will check if it is in slow start, congestion avoidance, or fast recovery mode. If slow start is true, the congestion window will be incremented by the maximum segment size and the duplicate ack counter will be reset. Furthermore, if the new congestion window value exceeds the slow start threshold, the server sets slow start to false, and enters congestion avoidance mode. If congestion avoidance is true, it keeps track of packets sent and increments the congestion window by a ratio of maximum segment size to `cwd_pkts`. If fast recovery is true, the congestion window is set to the slow start threshold, the duplicate ack count is reset, fast recovery mode is disabled, and congestion avoidance is set to true.

Finally, at the end of the file transfer loop, the server makes sure that the congestion window is greater than the maximum segment size and less than half the maximum sequence number. The server also ensures that the slow start threshold is at least the maximum segment size.

After the file transfer phase is complete, the server sends a `FIN` packet to the client and increments the sequence number. The server then waits to receive the `FIN/ACK` packet. Upon receiving the packet, the server increments the acknowledgment number and sends the final `ACK` packet. The server then waits to make sure that the client received the final acknowledgment. If the client sends another `FIN/ACK`, then the server will again respond with an `ACK` packet.

2 Problems and Solutions

2.1 Binary File Transfer Issues

One issue we had initially was with sending binary files over our client/server connection. For example, we attempted to send the built `client` executable file using our TCP connection but this file transfer ultimately failed. The issues, as it turns out, was that using `<<` and `std::string` cut off at null bytes.

2.2 Bit Vector Issues

Another problem we had at first was sending the bit flags in our packet abstraction over the UDP socket properly. We were initially using a bit vector, which sent the bits as 8 bytes rather than 3 bits. The solution to this issue was switching to a bit field.

2.3 Congestion Control Issues

For a long time, we were unable to properly set up TCP Tahoe due to the structure of our server implementation. The notes suggest that we should ensure that all sent packets are ack'd before incrementing the congestion window when in congestion avoidance, but with our server structure it would we be too difficult to keep track of acknowledging each packet before moving on to incrementing the window. As such, we decided to use the finite-state machine abstraction for congestion control that the textbook suggested.

2.4 Debugging Issues

One of the main challenges in creating our version of TCP was debugging and ensuring that the code executed properly. Our first instinct was to simply run `gdb` on each of the executables but we quickly realized that the server and client would simply timeout if we paused the execution at any point. Our debugging solution ended up simply being to use as many debugging `cout`'s as possible.

2.5 Extra Bits at the End of Packets

Another minor problem we ran into while creating our project was random extra bits at the end of certain packets. The problem ended up simply being that no null bytes was added when the data was being read into a buffer on the server. The solution was to change everything to `strncpy` or other functions that required us to pass in a buffer size variable.

3 Extra Credit

For the extra credit portion of the project, our group chose to implement TCP Reno fast retransmission and fast recovery as well as adaptive RTO.

4 Build Instructions

For Project 2 we did not modify the Vagrantfile and just used the suggested virtual machine and g++ version. We did however opt to use the Project 1 Makefile. Additionally, we changed the makefile to add "received.data" as a `make clean` target. To build the project, simply initiate the two Ubuntu-based virtual machines with:

```
vagrant up
```

Then, `ssh` into both the client and server machines with:

```
vagrant ssh client
vagrant ssh server
```

Finally, build the project with:

```
make client
make server
```

5 Test Cases

5.1 No Loss

For the no packet loss testing, we tried sending four different files:

- `small.txt`
- `server.cpp` (medium-size)
- `large.txt`
- `client` (binary)

5.2 10% Loss on Client/Server

We also tested the TCP connection with 10% loss on both the client and the server. We tested this using the given command to emulate a network with 10% loss as follows:

```
tc qdisc add dev eth1 root netem loss 10%
```

Using this emulated network, we sent the same files as with the no packet loss testing.

5.3 20% Loss on Client/Server

In addition to testing with no loss and 10% loss, we tested with an emulated network with 20% loss. This was done with the following command:

```
tc qdisc add dev eth1 root netem loss 20%
```

Again, we sent the same four files as with the previous two testing scenarios.

5.4 Packet Reordering

The last test case we used was packet reordering. To test this scenario, we simply used the suggested command as follows:

```
tc qdisc change dev eth1 root netem gap 5 delay 100ms
```

In this case, we also tested the same four files as with the other three test cases.

6 Contributions

6.1 Alex Crosthwaite

6.2 Jacob Nisnevich

6.3 Jason Yang