CS118
Project 2 Report

Akshay Smit (804641231)
Nikola Samardzic (904799099)

Overview and Build

The basic part of the project is contained in server.cpp, client.cpp, and rdt.h . The make command will compile executables called server and client.
The server can be run by specifying the port:

```
./server 15000
```

The client can be run by specifying the hostname, port, and filename:

```
./client localhost 15000 output.dat
```

The extra credit part is contained in server_tcp.cpp, client_tcp.cpp, and rdt_tcp.h . The make command will also compile executables called server_tcp and client_tcp . The server and client can be run similar to above:

```
./server_tcp 15000
./client_tcp localhost 15000 output.dat
```

The basic server and client executables implement selective repeat, where the window size is fixed at 5120 (5 full packets). The server_tcp and client_tcp implement TCP, with cumulative ACK, flow control, and congestion control. The timeout is always fixed at 500ms.

Packet Design

The packet struct is defined inside rdt.h . It has the following fields and format:

Sequence Number	- 8 bytes
Ack Number	- 8 bytes
Receiver Window	- 2 bytes
ACK flag	- 1 byte
SYN flag	- 1 byte
FIN flag	- 1 byte

Sequence Number field is only meaningful if SYN flag is set. Ack number field is only meaningful if ACK flag is set. The FIN flag is used during the close connection procedure. Receiver Window is just rwnd, used for TCP-style flow control.

Timers

The timers were implemented using the <chrono> module from the C++11 standard. This module is a big improvement over <time.h> which is now obsolete.

The <chrono> module offers multiple timers, but we use stead_clock. Here is an example from server.cpp :

```
std::chrono::time_point<std::chrono::steady_clock> *timers;
```

The above time_point object array, which uses stead_clock, contains a timer for each packet the serve wishes to send. Thus we can time each packet separately.

```
timers[i] = std::chrono::steady_clock::now();
```

The above shows how one can update the timer for a particular packet.

Threads

There are a number of threads used in our implementation. The most important thread is the ACKreader thread inside server and server_tcp. The ACKreader thread is created at the beginning of file transmission, and is responsible for reading all incoming ACK's and moving the window accordingly. max_index and min_index are global variables that determine the location of the window, and only ACKreader can change them.

Of course, there is a mutex lock on the window called master_lock. ACKreader acquires this lock before moving the window. Below is an example from server_tcp.cpp, where ACKreader moves the window by calling move_window:

```
else if (status == FR) {
   if (ind >= FR_trigger_index) {   //new ACK, window size <- SSTHRESH + 1*MSS
     min_index = ind+1;
     max_index = min_index + SSTHRESH/MAX_PACKET_SIZE + 1;
     status = CA;
   }
   else
     max_index++;                  //a packet left network, expand window by 1MSS
 }
```

Here "ind" is the index of the packet whose ACK was received by ACKreader. "FR_trigger_index" is the index of the packet that had caused fast retransmission. So in the above code we are checking if we should leave fast recovery or not.

There are separate threads to handle reading of SYN/SYNACK/ACK and the FIN/FINACK/ACK .

send_file

The main thread is responsible for iterating through the window and sending the corresponding packet. send_file first acquires the master_lock, after which it loops through the window to see if there are packets to be sent, or if a packet has timed out. All this is implemented in the send_file function of server.cpp and server_tcp.cpp .

So ACKreader and send_file run in different threads, and coordinate their actions using master_lock. The master_lock also controls printing to stdout, so that the two threads do not interleave their prints.

Window Movement

For the basic server and client, the window follows selective repeat protocol. So there is buffering on the client side, non-cumulative ACK's, and individual timeouts for packets.

For the extra credit, the window movement follows TCP. So there is buffering on the client side, the sender window increases according to the specifications of Slow Start/ Congestion Avoidance/Fast Recovery. To support this the server keeps track of SSTHRESH as well as the current state (slow start, congestion avoidance, fast recovery). But sender window never exceeds rwnd, to implement flow control.

Difficulties Faced

The first difficulty was finding a good timing mechanism that allows easy conversion to milliseconds. After grappling with time.h, we learned about chrono which we will always use from now on.

The second difficulty is how to separate window movement and sending packets. This was achieved by ACKreader thread being independent of the main thread. Only ACKreader can move the window. Only the main thread can send packets.

Another issue was client buffering since the packets arrive in no particular order. So we set up supporting variables indicating the offset of the packet's data. When we write data to the file, we just use these variables to find the offset into the file.

Another difficulty was implementing wraparound of sequence number. The issue was how the client knows the order of the data into the file. Then we realized that the window size is only 5, so there can only be one wraparound in that window. So we can always figure out the correct order of buffered packets through simple calculations (at most there are only 4 buffered packets).