CS118 Winter 2018, Project 2

Kuan Xiang Wen, 00446154

Josh Camarena, 004454718

Implementation:

We implemented a file transfer server and client applications, which connects using a UDP socket and uses application-level code to mimic a Selective Repeat protocol of reliable data transfer.

Structs:

Our implementation revolves around a WindowFrame object that mimics the form of a TCP packet with its both its header and payload, plus a few other variables for our own use:

1) Our first struct Packet represents the TCP packet, taking up 1024 bytes. Its variables are:

- "seq_num" represents a sequence number and is not used in the client program.
- "ack_num" represents the ACK number and is not used in the server program.
- "length" represents the bit-size of the payload.
- "flags": Represents several flags, using only the 4 least significant bits. 1st bit is the SYN flag, the 2nd bit is the FRAG flag, the 3rd bit is the FIN flag, and the last bit is the ACK flag.
- "payload" represents the TCP packet's payload of size 1016. The other variables represent metadata.

2) The second struct is the WindowFrame. We only use this struct for sending packets that we expect the other party to reply with an ACK. Its variables are:

- "packet" is a struct Packet and represents TCP packet.
- "sent" is a flag that is 1 if a packet exists in this frame and is sent, and is 0 otherwise.
- "ack" is a flag that is 1 if this packet's ACK is received, and is 0 otherwise.
- "timesent_tv" is the timestamp of which this packet was last sent, including its retransmissions. We use this last variable as a timestamp for our timeout functionality explained later.

In the server, we also use an array window[5] of 5 WindowFrame objects. This represents the sliding window of the TCP protocol. In the client, we use a single window1 object for the same purpose, since the client can wait for at most one ACK (during SYN and FIN phases).

Helper functions:

- error: If called, prints an error description and exits with code 1.
- get_packet: Only called when an input is available, as controlled by poll() in main(). Uses recvfrom and writes the buffer into a Packet object that is passed by reference.
- send_packet: Creates a TCP packet based on parameters 2 through 9. The first parameter is a WindowFrame* object that is normally a null pointer. If this packet is expecting an ACK, a WindowFrame is passed in and its variables will be set.
- retransmit: Called when a packet in a WindowFrame object needs retransmission.
- File_transfer: Server only. Called if we need to transfer a file (eg. Begin transfer, or we received a fragment ACK). This function first finds the packet in the window[] array to be ACKed and marks it. It then looks at this array: If the first x packets is ACKed, moves the last 5-x objects forward, and fills the last x objects with new file fragments.
- empty_window: "Empties" the window[] array by setting their "sent" flags to 0.
- timeout_remaining: Calculates, based on a 500ms timeout, how much time a packet has left to timeout.
- refresh_timeout: Server only. Based on timeout_remaining, find out how much time is left for any packet in the window[] buffer to timeout. Saves that value into the global_timeout variable that is used as the timeout value for poll(). Also, if anything in the array is timed-out, retransmits that packet.
- check_timeout: Retransmits any packet in window[] that has timed out.
- Respond: Responds to a received packet.

Functionalities:

UDP Connection:
Server binds a UDP socket (set with SOCK_DGRAM). Client then connects with a SYN packet with sendto(). Server can retrieve the message and the client's address with recvlen, and can then return messages with this address using sendto(). Based on example found at https://www.cs.rutgers.edu/~pxk/417/notes/sockets/udp.html.

File transfer protocol:
Based on the arguments passed when executing the program, the client sends an ACK to the server with the requested filename. The server will respond with a SYNACK packet: If the file exists, its payload will contain the size of the file. If file does not exist, the FIN flag will be set, and the FIN protocol begins.

During file transfer, the server first sends at most 5 packets to the client, and those 5 packets will be stored in window[5] to await their ACKs. Whenever an ACK is received, that packet will be marked as ACKed. If the first x packets of window[] is ACKed, moves the last 5-x objects forward, and fills the last x objects with new file fragments and sends those packets as well.

After all fragments are ACKed, a FIN is sent. The client replies with a FINACK, and for RetransmissionTimeOut*2. If the server receives the FINACK, it closes immediately. If the client waits out for the above duration, it closes. If it receives a duplicate FIN packet from the server, it resends the FIN and restarts the duration.

Timeout:
Reliable data transmission in our project is achieved with the global_timeout variable, that is passed into the poll() function so that we can have a timeout detection mechanism. By default, if nothing is found in the window[] array, the global_timeout equals zero. Otherwise, the timeout value will equal the shortest timeout value among all entries in window[]. This value is calculated whenever a packet is received or transmitted, and we do this by placing it at the end of the respond() function. See refresh_timeout() and file_transfer() in Helper Functions section above for more information.

Challenges Faced

1) Our first decision was to choose between C++ and C. We began with C++ because we wanted to make use of the class object and the internal functionality it provides, to make design of the structs more easy. However, the file transfer operation was very sensitive to object size and we also faced issues trying to use binary operations among differing types. Also we had problems with compiler and library standards. The biggest factor of all was the difficulty rewriting from scratch (since our first project was in C), so in the end we decided to switch back to using C.

2) This ties in to our challenge faced during header design. We were trying to design a PacketHeader struct to contain just the header. The rationale was so that we could read-from-buffer the header into the PacketHeader and read the values according to the struct. We noticed that although we used 3 unsigned shorts and one char (for the flag), which takes a total of 7 bytes, the class used 8 bytes due to how memory allocation works in C, which forced us to face segmentation faults early on until we realized it was 8 bytes. Later, we decided to reorganize our send_packet function so we could change PacketHeader to a Packet struct, which also contains the payload.

3) We had a lot of difficulty copying files initially. Originally, the send_packet function did not contain a "datalen" (representing length of payload) input argument. Instead, we tried to use sizeof() on the "input" (representing payload data) argument to get its size. However, this caused file names in the client's SYN request that were longer than 8 characters long to be cut off, producing 404 errors. This happened because the file names were stored string-style in a char* argument. When passed in, sizeof() read the pointer size, which was always 8. We figured this

out and switched to use strlen() instead of sizeof(). This time, when we were trying to transfer binary files, the strlen() would not read the whole buffer and would stop prematurely due to '\0' characters which denoted end of string, so the file was always poorly read. Ultimately, we decided to have "datalen" as an argument so that the caller could choose between sizeof() and strlen().

4) We noticed that our packet sizes were always 1024, even if the payloads were only a few bytes large. This happened because we always allocated a full 1024-size array for convenient interfacing with our Packet structs. We decided to forego non-1024 increments entirely, as technically we were indeed sending 1024-sized packets. Before we decided to stick to this, we faced buffer overflow problems. For example, the "filebuf" was originally given a size exactly equalling the size of the file. However, reading the last fragment, which was smaller than size 1024, caused buffer overflows. We avoided this by increasing the filebuf size to the next 1024-multiple, but used the actual file size when calling write().

5) We had trouble figuring out the spec, as this was a non-traditional Selective Repeat protocol. For example, we had trouble deciding if a size 5 window meant that if the frames 1 and 5 were not ACKed but 2,3,4 were ACKed, does that mean we are waiting for more ACKs or we should move frame 5 forward and add 3 new frames. Also, in normal SR, the client initiates FIN. We had to spend extra time asking around to clarify the spec.

6) A design challenge was to figure out how to transition between the SYN-phase, the file transfer phase, and the ACK-phase, for a server that operates based on events. With asynchronous event loops, this problem might have become trivial, but we had little time to figure that out in C. We decided to go with a "stateflag" argument to clearly mark between the phases, so that the response() helper function will act differently. In addition, we used poll() so that we could mimic event-based execution. This became a lifesaver later when we needed to start implementing the timeout functionality.

7) As a follow-up to point 6, it took us some time to figure out how to trigger timeouts in a non-asynchronous program. Also, before we used poll(), we were trying to implement a timeout feature shoddily by having it loop infinitely. For example, if the server was not being ran, we wanted the client to keep sending SYN. But something unknown was blocking our infinite loop, despite the fact that recvfrom() does not block, and our response() function should have returned immediately. We were never able to figure out the cause, but when we changed to use poll() every problem we had regarding blocking went away.

8) An option we tried for alarms was to use alarm() and SIGALRM, but we quickly abandoned it when we found that it caused race conditions.

9) It was troublesome figuring out how to use the network emulator "tc" in Ubuntu. For example, my Virtualbox doesn't have "eth0", and I had to use "lo" instead. We also haven't been able to figure out "tc qdisc del". Also, emulating terrible conditions also causes git's network traffic to be affected, causing it to time out often.