CS 118 Computer Network Fundamentals

Project 2: Window-Based Reliable Data Transfer Over UDP

Dylan Gray	UID: 804-127-949	SEASnet ID: dylang
Sharon Hsieh	UID: 804-184-690	SEASnet ID: hsieh

Implementation Description

Header Format

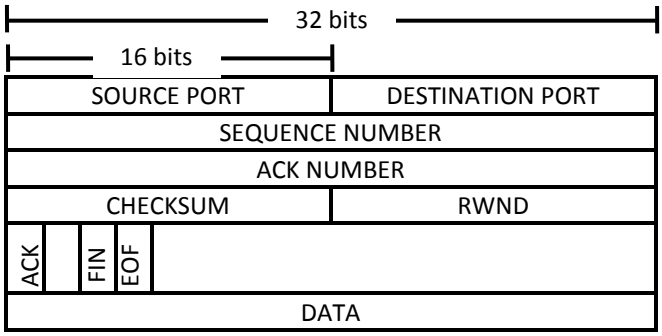

Figure 1: Header Format Implementation
The header size is 20 bytes. The source port, destination port, checksum, and receiver window size (rwnd) are all 16 bits each. The sequence number and ACK number are both 32 bits each. There are 3 flag bits, one each for ACK, FIN, and EOF.

The header values are stored in a buffer, which is shown in Figure 1. The header fields in order are: source port [16 bits], destination port [16 bits], sequence number [32 bits], ACK number [32 bits], checksum [16 bits], receiver window size (rwnd) [16 bits], ACK flag [1 bit], FIN flag [1 bit], and EOF flag [1 bit].

The source port is the port number from which the packet originates. The destination port is the port number to which the packet must travel. Source and destination ports are needed for multiplexing and demultiplexing. The source port is needed because when a packet arrives, the recipient must know the source to send an ACK back to. The destination port is needed because the host may have multiple instances of the program using different ports, and it needs to know which port the packet is intended for.

The sequence number is used to track the order of the packets sent. The sender and receiver both keep their own separate sets of sequence numbers and ACK numbers. The ACK number is used to track which packets have been successfully received (not lost or corrupted). Sequence numbers and ACK numbers are based on bytes.

The checksum is used to determine if there is bit corruption in the rest of the header and the payload (data). This is necessary to implement reliable data transfer.

The receiver window size (rwnd) must be known by the sender because the sender cannot send more packets rwnd, even if its congestion window size (cwnd) is greater than rwnd, otherwise there is potential for buffer overflow on the receiver's side, meaning packets will be dropped.

The ACK flag is set to true by the receiver and sent to the sender upon successfully receiving a non-corrupt data packet. The field between ACK and FIN was originally a SYN flag used for TCP handshake, but since the TA informed us the handshake is not required, this field is unused. The FIN flag is set to true by the receiver when it sends the ACK upon successfully receiving the last packet. The sender waits for the FIN flag before exiting. The EOF flag is set to true by *deconstruct_header()* when the sender calls *rdt()* upon *bytes_read* < *MSS* (1000 bytes). Then, all received data is stored in the file name defined by RESULT_FILE[] in constants.h.

Window-Based Protocol: Go-Back-N and Reliable Data Transfer

Our window-based protocol is built on top of Go-Back-N. *cwnd* is set by the sender in sender.cpp. The window size is the minimum of *cwnd* and *rwnd*. This is because if the receiver's window size is smaller than *cwnd*, packets sent could be dropped if the receiver's buffer is full.

We create packets and store them in a list before we send them. Once an ACK is received, we iterate through the list, removing all packets with sequence numbers less than the ACK that was just received. Packets are only sent if the number of unACKed packets is less than the minimum of *cwnd* and *rwnd*. This reduces the possibility that the packet sent will be dropped by the receiver because the receiver is still processing other packets and does not have room in its buffer.

Both the sender and receiver keep their own separate sequence numbers and ACK numbers in terms of bytes. These numbers are initialized to random values. We multithread such that one thread sends ACKs and the other thread handles them. The sender always has two threads: one sending data and one receiving and handling ACKs. The receiver has a thread which receives the data from sender and checks for corruption. It then spawns multiple threads which are responsible for sending ACKs. The sender and receiver increment their sequence numbers by the number of bytes of data sent every time they send a packet. They increment their ACK numbers upon receiving a packet if the sequence number of that packet matches the ACK number they are expecting:

```
// if checksum valid and ack_num == seq_num_from_sender
//    ack_num += max(data_size, 1)
```

If the packet is an ACK, the data will be 0B and the ACK number will only be incremented by 1.

To implement the checksum, we first fill the checksum field in the header with 0. Then we XOR all sets of 16 bits in the packet to get the checksum. Last, we drop the checksum into the checksum field in the header and send the packet on its way.

Timeouts

To implement timeouts, we first use a *timeval* struct and the *setsockopt()* function to set the timeout value to a constant TIMEOUT_KILL defined in constants.h. *recvfrom() and sendto()* will time out after the number of seconds indicated by this constant because we assume that the network has lost connection if one side does not hear from the other in a certain amount of time. Next, we created a constant called TIMEOUT in constants.h, and in sender.cpp, we give each packet a time stamp value called *time_sent*. These values are used when sending packets in the following manner:

```
// if time_sent + TIMEOUT > time(NULL)
//    if tries < MAX_TRIES
//          tries++
//          retry
//    else
//          exit
```

The variables tries and MAX_TRIES are needed because if the network was too congested and packets were being dropped or the connection was lost, we would not want to keep trying and timing out. Instead, every time we send the packet, we increment the number of times we've tried to send the packet and give up when we've hit MAX_TRIES, which is defined in constants.h.

Congestion Control (Extra Credit)

Our code implements congestion avoidance, slow start, fast retransmit/fast recovery, and congestion control timeouts. For congestion avoidance, we add $\frac{1}{cwnd}$ every time we get a new ACK if *cwnd > ssthresh*. For slow start, we increment *cwnd* by 1 MSS (1000 bytes) for every new ACK if *cwnd <= ssthresh*. If we receive 3 duplicate ACKs in a row, we start fast recovery, where *ssthresh* = $\frac{cwnd}{2}$, *cwnd = ssthresh + 3* (we add 3 for the 3 duplicate ACKs we got), and resend all packets in the sender window. For every duplicate ACK after this point, *cwnd += 1* MSS. For the first nonduplicate ACK, we set *cwnd = ssthresh* and exit fast recovery, then enter slow start and *cwnd += 1 MSS*. For congestion control timeouts, we *set ssthresh = max(*$\frac{cwnd}{2}$*, 2*MSS)* and reset *cwnd = 1 MSS* when a timeout occurs. Then we retransmit all packets in the sender window.

Difficulties Faced

- Multithreading race conditions led to problems with checksum and buffer
    - Because we had two threads editing the list of unACKed packets, the iterator sometimes became invalid, so we accessed memory outside the list and sent junk data or segfaulted.
    - Example Race Condition: The iterator was incremented when the list was empty, but this could be done by multiple duplicate ACKs, leading to segfaults when the iterator was incorrectly incremented and ran off the end of the list. Our solution was to check if the list was empty before incrementing the iterator.

- Iterator ran off end of the list when deleting last element in list
    - Attempt 1: Keep an empty packet in the list. Iterator will point to this when we delete the last element.
    - Attempt 2: Point iterator to next packet to send, not the last one sent (Final Solution).