Simple Window-Based Reliable Data Transfer in C/C++

Computer Science 118
Spring 2014

Nathan Tung
004-059-195
SEASnet Login: nathant

Mark Iskandar
704-050-889
SEASnet Login: iskandar

Task Description

The purpose of this project is to use User Datagram Protocol (UDP) socket programming to establish reliable data transfer. This RDT implementation involves two main parties over the UDP connection: the sender acts like a server by delivering files, while the receiver acts like a client by requesting files to be transmitted.

We will be implementing the window-based Go-Back-N (GBN) protocol. First the receiver sends a filename to the sender. Upon receiving the filename, the sender can transmit the specified file through multiple (depending on the congestion window) 1024-byte-max packets. Notice that these packets are preceded by a header that includes information like sequence numbers, acknowledgement numbers, etc.

Though not a reliable data transfer protocol in theory, UDP does not seem to lose or corrupt packets in our tests, so packet loss and packet corruption must be simulated. The sender and receiver must specify Pl and Pc values – the probabilities of loss or corruption, respectively – in order to demonstrate how GBN operates when retransmission is necessary.

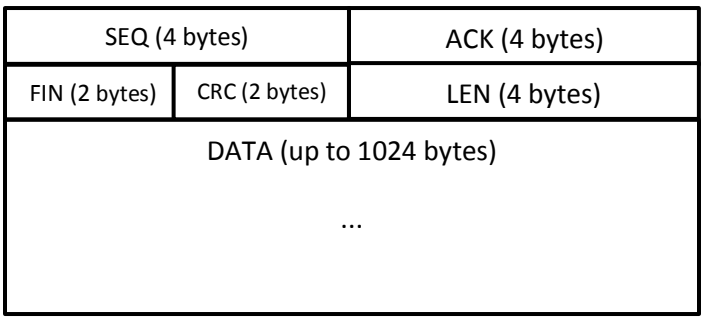
Implementation

The sender and receiver we constructed are based off of skeleton code from Beej's Guide to Network Programming. IP addresses, port numbers, requested file, congestion window size, and probabilities of loss or corruption are specified when the sender and receiver programs are run.

Header Format:

The packets sent consist of 16-bytes of header information: sequence (SEQ) number, acknowledge (ACK) number, FIN, CRC, and LEN. Up to 1024-bytes of data follow the header.

SEQ and ACK are rather self-explanatory. The SEQ tells the receiver the order of the data, while the ACK updates the sender on which files have been successfully received. FIN is usually 0. When FIN reaches 1, it means we have either received the last packet of the file or received the ACK sending that last packet. CRC is a slight misnomer in this case. Rather than doing an actual Cyclic Redundancy Check, we are using the CRC field as an indicator on whether the packet is corrupt or not. CRC is usually 0, but when a packet's CRC is 1, that packet is corrupted and is therefore ignored. LEN is the length of the data in the packet. This can be useful for updating the ACK number and prevent errors that might otherwise occur in binary files due to `strlen()`.



Messages:

Messages regarding packet status are outputted to the console. Let us pretend N is some number. On the sender side, we see "`Sent…sequence #N`" when a packet is dispatched. If the packet is lost, we see a "`Packet with sequence #N lost!`" and likewise for corrupted. When an expected ACK is received, "`ACK #N received`" is displayed; otherwise, we see "`Timed out while waiting for ACK!`"

On the receiver side, we see "`Received…sequence #N`" when a packet is received. If the packet is in good condition and expected, we send back an ACK: "`Sent ACK #N`". If the packet is corrupted or out of order, we display "`Packet is corrupted! Ignoring it.`" Or "`Packet is out of order! Ignoring it.`".

At the very end, both sides show "`FIN sent/received`" and "`FINACK sent/received`" prior to finishing file transmission and program termination.

Timeout:

Our timeout and retransmit implementation involves "`time.h`" and `FD_SET()`, or a file descriptor. A `timeval struct` is provided with specified time fields `tv_sec` and `tv_usec` that show how long we wait before timing-out, but the `select()` function actually enables our sender to know when receiving an ACK has timed out. If the ACK has timed out, we break out of the loop that is receiving ACKs and retransmit the next packets depending on the window size.

Packet Loss & Corruption:

Packet loss and corruption are handled very similarly in our implementation. Notice that the user specifies these probabilities as doubles. In our `send2()` function, we convert these to integer probabilities out of 100, then use `rand()` to generate a number from 1 to 100. If the random number falls under the integer probability of packet loss, then that packet is "lost" and is therefore not sent. If the random number falls under the integer probability of packet corruption, then that packet is marked as corrupt. Otherwise, the packet is sent normally. On the receiving end, packets that are corrupt are ignored and treated as if they were lost.

Window-Based Protocol:

For GBN, there is only one congestion window that monitors packets being dispatched from the sender. The value `CWnd`, as specified by the user, denotes the number of packets that can be sent at once without checking for ACKs. During the loop that is sending packets, the sender checks to make sure there is still more data to transmit from the file and we don't send more than `CWnd` packets at once. Each time an expected ACK is reached, we increment the window, since that last packet will never need to be retransmitted. The next cycle, we can then send the next `CWnd` packets to the receiver.

Difficulties Faced

One of the hardest parts of Project 2 was figuring out how to get started. With Beej's tutorial, it wasn't difficult to understand how UDP works in C, but it took quite a few trial-by-error attempts in order to figure out how to send and receive from the same socket in both the sender and receiver. Furthermore, the continuous sending/receiving had to be based on some loop. Our solution was to utilize a while-loop until the FIN or FINACK was received.

Perhaps one of the more annoying bugs we faced was sending binary data, or otherwise non-text files. Often times there are consecutive zeroes in binary data that effectively act like null-bytes and can prematurely end your data transmission. To remedy this, we got rid of any problem-causing string-referencing methods like `strlen()` and treated null-bytes just like other characters to a certain extent.

Compiling and Running

The included `Makefile` means compiling the data transfer program is very easy. One simply needs to type "`make`" (no quotes) into the console in the directory containing `sender.c` and `receiver.c`. Notice that our sender and receiver files also rely on functions found in the `packetTransfer.h` and `packetTransfer.c` files.

To run the sender, enter the console command:
`./sender <portnumber> <Cwnd> <Pl> <Pc>`
Remember to input port number, congestion window size, and probabilities of loss or corruption.

To run the receiver, enter the console command
`./receiver <hostname> <portnumber> <filename> <Pl> <Pc>`

Throughout the project, we use the local host, or `127.0.0.1`, in order to execute our program. The port number can be any 4-digit integer that isn't used for other purposes, like 5000 or 9999. The probability parameters are doubles that must range from 0 to 1. Finally, filename is the name of the file requested. The transmitted file is preceded by "new_". That means if we request test.txt, the transmitted file created is called new_test.txt.

For example, assume we have a window of 5 packets being sent at once with 10% packet loss and 5% packet corruption on both the sending and receiving sides with port 9000. If we want to request a local "penguin.jpeg" to generate a "new_penguin.jpeg," we would input:

```
./sender 9000 5 0.1 0.05
./receiver 127.0.0.1 9000 penguin.jpeg 0.1 0.05
```

Conclusion

Although the learning curve for this project made it more time consuming than Project 1, this project clearly demonstrated the usefulness of data retransmission for GBN. Overall, designing a GBN-based reliable data transfer protocol in C was both a unique and rewarding experience.