CS118 Project 2
Reliable Data Transfer with GBN protocol

Authors
Stanley Hsu 704338787
Gary Zhang 504183404

Build and Run
To build: Run `make` in the source code directory
To run the server:
```
cd <source code directory>/test
./sender <serverPort> <windowSize> <pLoss> <pCorrupt>
```
To run the client:
```
cd <source code directory>
./receiver <hostname> <senderPort> <receiverPort> <windowSize>
            <filename> <pLoss> <pCorrupt>
```

*Note: the client needs to specify a windowSize because it acts as a sender when requesting the file (sending filename)

Implementation
High-Level Design
Sender.c and Receiver.c take care of the application level implementation of creating the socket, binding, sending/receiving, opening/closing files, and closing the socket.
rdt.c contains pseudo-transport layer data transfer functions that are used by both the client and the server, as defined in rdt.h

Packet Format
Maximum size of a packet is hardcoded to be 960 bytes. Actual packets may be less than 960 bytes; the tracking of packet size is left to UDP.
The first byte of the packet is the packet type. Packet type may be DAT (data from sender), END (no more data), ACK, or DNY (refuse packet).
The second byte of the packet is the sequence number, which ranges from 0 to 255.
The rest contains the data being transmitted. One packet may have up to 958 bytes of actual data

Reliable Data Transfer
We decided to use the GBN (Go-Back-N) protocol to implement reliable data transfer. In the GBN protocol, the sender has a window size greater or equal to 1, while the receiver has a window size of 1.

Sender pseudocode:

```
Loop forever:
    If an uncorrupted packet is received:
        If the packet does not come from receiver's address:
            Reply with DNY
        If the packet is an ACK with sequence number greater than the back of the command window:
            Update window back to the ACK's sequence number
            Reset timer
    If timer has expired:
        Reset timer
        Set sequence number to window back
    If connection timer has expired:
        Stop sending
    If END packet has been sent and ACKed:
```

Stop sending
Otherwise, if all data packets sent and ACKed:
Send END packet
Otherwise, until sequence number equals window front:
Send packet corresponding to sequence number
Increment sequence number

Receiver pseudocode:

Loop forever:
Block while waiting for uncorrupted packet
If the packet does not come from sender's address:
Reply with DNY
Otherwise, if it is a DAT packet and its sequence number matches requested sequence number:
Increment sequence number for next packet
Send ACK requesting sequence number of next packet
Append data to buffer
Otherwise, if it is an END packet:
Send ACK with incremented sequence number
Exit function

The server starts as a GBN receiver and the client starts as a GBN sender.
The client will first send a file request message to the server, which has been listening for incoming packets. The client ends the requesting by sending an END packet. When the server receives the END packet, it replies with an ACK and enters sender mode. When the client receives the said ACK, it enters receiver mode.

The file requested* is then being transferred from the server to the client using the GBN protocol. The server sends up to windowSize number of packets from the last acknowledged packet. When it receives an ACK that is in its window, it slides over the window. The client, upon receiving expected packets, sends back ACK to the server as well as appends the received data into a buffer. If the client receives duplicate packets, it sends the ACK for the next requested packet without appending the data to the buffer again. The client also just simply ignores wrong packets and out-of-order packets.

Similar as before, the server ends the sending by sending an END packet to signify the data has all been transferred. When the client receives the END packet, it replies with an ACK and saves the buffer to a file and exits. When the server receives the said ACK, it starts over again and listens to another file request. If the server doesn't receive ACK, it repeatedly sends END packet until connection timeout. At connection timeout the server listens to another request.

*If the file does not exist, nothing will be sent. The server will just send an END packet. If the file is empty, then an empty file will be sent (i.e. a DAT packet with no data).

Example

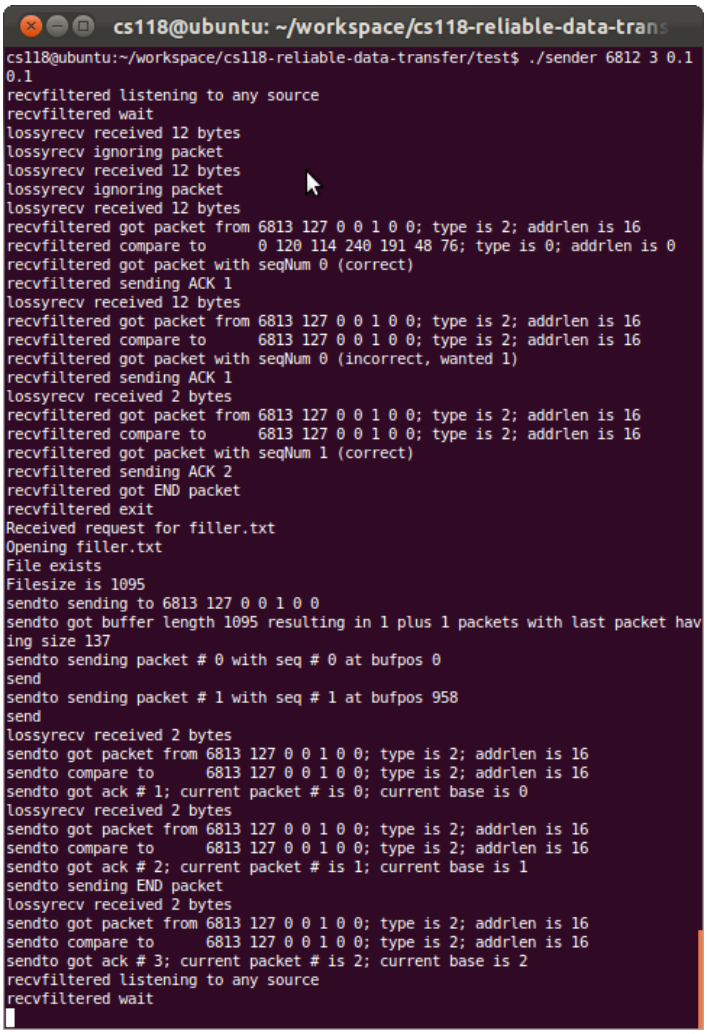
The following example shows a client trying to get the file "test/filler.txt" from a server. pLoss and pCorrupt are both set to 0.1, windowSize is set to 3 on both sides, and the server and client port numbers are 6812 and 6813 respectively.

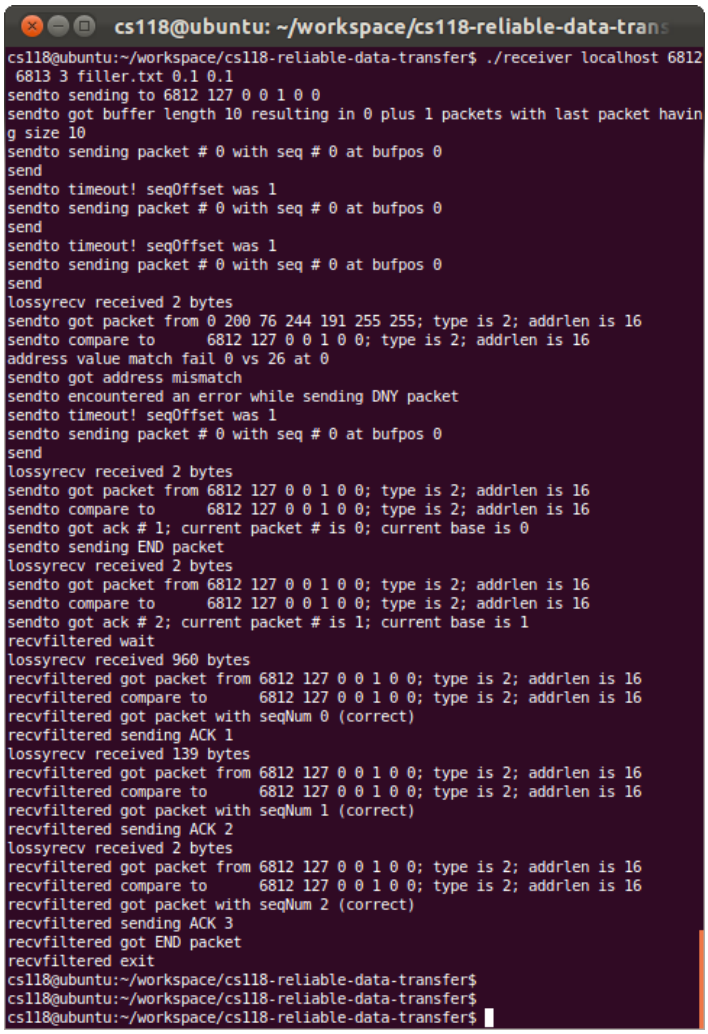
Server terminal:

```
./sender 6812 3 0.1 0.1
```

Client terminal:

```
./receiver localhost 6812 6813 3 filler.txt 0.1 0.1
```





Server-side terminal output (sender)

Client-side terminal output (receiver)

As seen in the images above, both sides behave as expected with appropriate sent packet and received packet messages. Note that lossyrecv is the function that ignores or corrupts packets at random.

Difficulties

Since the project is very open-ended, we had to figure out the exact interface of RDT such as the variable to pass into the functions, and the format of the packet header. Also, because our high level design specifies that the functions are used for both the sender and the receiver, we had to integrate it by keeping it as generic as possible while having a parameter to determine which side it is being used on.

Other than that, there were the usual bugs such as negative numbers in modular arithmetic, and off-by-one errors. However they were solved easily by looking at print statements and examining the source code more carefully.