**Simple Window-based Reliable Data Transfer**

**CS 118 Project 2**

Aman Agarwal (604183535)

Siddhant Khemka (704161175)

**Project Summary**

The basic purpose of this project was to use User Datagram Protocol (UPD) socket programming, along with C, to establish a system of reliable data transfer between a sender and a receiver. The sender acts as what is conventionally called a server, and delivers packets to the receiver, or client, which established their correctness and acknowledges their receipt to the sender.

We utilized a window-based go-back-N (GBN) protocol, in which the receiver asks the sender for a file, and the sender sends it over in 512-byte packets, along with header data. This is done in a window, or size of the number of packets to be sent over at once.

Along with this, data loss and data corruption are simulated by including probabilities for loss and corruption to occur while starting the sender and the receiver, and randomizing this via the rand() function.

**Implementation**

Our implementation has four major facets that are discussed in greater detail in the bullet points below:

* **Packet/Header Format**

Each packet is composed of several variables and up to 512 bytes of data from the original file. They are contained in a struct as defined in protocol.h.

5 integer variables are used to transmit essential data about the packet and the transfer process:

1. seq: This tracks the sequence number of the packet being sent
2. ack: This is used by the receiver to send positive acknowledgment of a packet being correctly received.
3. fin: This is used to convey that this is the last packet being sent.
4. len: As the name implies, this holds the length of the data.
5. crc: This is used to indicate whether the packet is corrupt or not

Aside from this, numbytes and numbytesValid are used to check for the integrity of the data.

* **Messages**

The program prints messages at several points of execution.

In sender, the first messages printed deal with incorrect usage of the arguments, for when the correct number has not been used or when the probabilities for loss and corruption are out of bounds. Next, while the specified socket is being bound, an error or success regarding this will be printed. Following this, the file is attempted to be read, and indication of this success is printed or, the reason for failure is.

As the packets begin to be sent, the sequence number of the ones being transferred is printed, along with whether they were lost, corrupted or successfully sent. Next, the ACKs received for this, if not corrupted are printed to the terminal to indicate success, followed with a final FINACK to indicate that the file has been successfully transferred.

In receiver, similar errors for incorrect arguments and being unable to create or find the specified file are printed. During packet receipt, confirmation of a packet being received or its corruption are printed. Similarly, confirmations of sent ACKs or those lost in transmission are also printed, along with a confirmation of the FINACK being sent.

* **Timeout**

The sender waits for up to 3 seconds to receive ACKs from the receiver. If the ACKs are not received in this timeframe, the loop goes on to retransmit the same packets to ensure that they have been received by the receiver.

* **Packet Loss & Corruption**

As the user enters these values as probabilities, they are stored as doubles. In the sendPacket() function, these are converted to integer values, and random integer is generated using the rand() function along with a mod 100 operator. If the generated number is less than the probability, the packet is either lost or corrupted based on the respective probabilities.

As there is a chance for both loss and corruption to occur simultaneously, loss takes precedence as the corruption status does not matter for a lost packet, given that the receiver would never even be able to tell that it was corrupt before being lost.

* **Windowing**

As required by the project specification, one major feature of this code is to window the amount of data being sent. Thus, instead of only sending one packet at the time, the program takes in a window size, and divides it by 512 to determine what the maximum number of packets that can be sent at once is. Once that is determined, the loop in the sender main() function sends that many packets in one go before collecting any ACKs. By default, 512 is set as the minimum window size if Cwnd is less than that.

Once it has waited for 3 seconds to collect all the ACKs it gets, it then checks to see which are in order, and correspondingly shifts the window from the previous starting points, based upon how many sequential ACKs were received by the sender. This helps the efficiency of the program, as more packets are sent at once asynchronously. After one cycle is complete and the counter is incremented, the next Cwnd set of bytes is sent to the receiver.

**Difficulties**

We did not face many major difficulties in completing this project, though we did face some smaller ones along the process. Particularly, thinking of how to incorporate windowing was challenging and took some trial and error on our end, as well as thinking of how the math would work in corner cases.

**Compiling and Running**

We have included a Makefile which can be used to easily compile the program, so only make needs to be run to compile.

After compiling, the usage for sender and receiver are as follows:

./sender <portnumber> <Cwnd> <Pl> <Pc>

./receiver <hostname> <portnumber> <filename> <Pl> <Pc>

Port number is any valid portnumber not being used by other processes.

Cwnd is the size of the window.

Pl is the probability of losing a packet.

Pc is the probability of corrupting a packet.

Hostname is the name of the sender host.

Filename is the name of the file to be transferred.

The new file is set up with ‘new\_’ prefixed to it. Thus, sending test.txt would result in the new file being created as new\_test.txt.

**Conclusion**

The project was overall very insightful and taught us a lot about how data transfer protocols are designed and implemented.