COMP 7005
Project
Report

Sami Roudgarian, A01294122

Harmanbir Dhillon, A00994245

December 1, 2023

Purpose

To design a reliable network protocol using UDP and simulate a lossy network

Requirements

TASK	STATUS
Create a reliable protocol based on UDP (The UDP data portion holds your protocol)	Fully implemented
You must support IPv4 and IPv6	Fully implemented
Create a sender that reads from the keyboard and write to a UDP socket	Fully implemented
Create a receiver that reads from a UDP socket and writes to the console	Fully implemented
Create a proxy that sits between the two hosts that: ● Randomly drops data from the sender ● Randomly drops acks from the receiver ● Randomly delays packets from the sender or the receiver	Fully implemented
The receiver must send an acknowledgement to the sender	Fully implemented
If the sender doesn't receive an acknowledgment in a reasonable time (as determined by you), it resends the packet	Fully implemented
A GUI that graphs the data the sender, receiver, and proxy	Fully implemented
All 3 programs must maintain a list of statistics that show how many packets have been sent or received	Fully implemented
The statistics must be stored in a file	Fully implemented
(bonus) Dynamically change the percent chance to drop or delay at runtime	Fully implemented
(bonus) implement a window based protocol instead of send and wait	Fully implemented
(bonus) Sender calculates and includes a checksum in the packet header. Receiver checks checksum, if checksum does not match it drops the packet.	Fully implemented
(bonus) Proxy can randomly corrupt the data in a packet	Fully implemented
(bonus) Client can receive Cumulative ACK	Fully implemented

Platforms

This project has been tested on:

- macOS Sonoma 14.0

Languages

- ISO C17 (client, server, proxy)
- Python3 (GUI)

Findings

We better understood how to make a protocol reliable. We implemented our reliable protocol by using Sequence and Acknowledgement numbers and using the SYN, ACK, PSH flags. We took inspiration from TCP and we implemented a protocol that is very similar. After the 3-way handshake we increment our Sequence number by the bytes of data that were in the previous packet. On the server side, we Acknowledge each packet by telling the Client what byte number we expect next, i.e the expected Sequence number.

We also implemented a Window. Our window size specifies the number of packets we can send to the Server before we have to receive an ACK for the first packet we sent. We implemented the Window by using a ring buffer. The head was pointed to the first empty packet, basically the next time we send a packet it would be at the index that the head is pointing too. The tail pointed to the first packet that was sent that did not receive an ACK. The head and the tail cant overlap each other in our implementation of the ring buffer.

We also learned how a checksum works by implementing our own basic version of it. We just add up all the bytes multiplied by 34. We then add that sum to an XOR of the sum of all the bytes. The reason we used two checksum algorithms is because if we only used one, there was a chance that even after the data was corrupted the checksum would be the same. But when we use these two different algorithms, there is a lower chance of that happening.

We also better understood the need of keeping statistics logged because it makes the debugging process much easier. Instead of having a bunch of print statements littered in the code, you can just log everything into a file and if you want, write a program that will comb through the logs.