Programming Assignment 2

Implementing a Reliable Data Transport Protocol

Names :

Mohamed Ahmed Abd-Twab (57)
Mohamed Samir shaaban (60)
Mohamed Mahmoud Adel (79)

Problem Statement:

- The network communication in the upper two stages was provided through a reliable transfer protocol (TCP/IP).

- In this stage, We are required to implement a reliable transfer service on top of the UDP/IP protocol. In other words, we need to implement a service that guarantees the arrival of datagrams in the correct order on top of the UDP/IP protocol, along with congestion control.

- we have optional to implement in C/C++(Bonus :D) .

- we implement Stop and Wait and Selective repeat.

How to run code using commands :

- in client directory should contains file client.in contains :
    * Server IP “127.0.0.1”
    * Server Port use same Server port will define in server.in
    * client port not need it in Linux kernel
    * File Name we need it from server
    * initialize received cwnd
- run this command:
    g++ client.cpp -o c.out
    ./c.out
- in server directory should contains file server.in contains :
    * Server Port same port mentioned in client.in
    * max received cwnd
    * seed
    * PLP for simulation.
- Go to server path and type :
    g++ -pthread server.cpp -o s.out
    sudo ./s.out

- Go to client path and type :
    g++ client.cpp -o c.out
    ./c.out localhost 80

- you can see requests and responses in terminal or can see in each directory.

data structures:

- struct from ack , data and threads (args contain all data we need in threads):

```
/* Ack-only packets are only 8 bytes */
struct ack_packet {
    uint16_t cksum; /* Optional bonus part */
    uint16_t len;
    uint32_t ackno;
};
struct packet {
    /* Header */
    uint16_t cksum; /* Optional bonus part */
    uint16_t len;
    uint32_t seqno;
    /* Data */
    char data[500]; /* Not always 500 bytes, can be less */
```

```
};

struct send_data_thread_args
{
    sockaddr_in client; /*client socked details*/
    packet pck_file; /*data packed*/
    int loss_probability; /*loss probability used in PLP*/
    int seed;
};

struct send_pck_thread_args
{
    pthread_mutex_t * timer_mutex; /*mutex change packed time*/
    pthread_mutex_t * state_mutex; /*mutex change window and state*/
    priority_queue<Pair, vector<Pair>, PairCompare>* my_queue;
    /*to contain packet and time related with this packed*/
    int packets;
    bool * acknowledged;
    bool * finished; /*finished from send all file or not*/
    sockaddr_in client;
    int socket; /*communication socked*/
    double * window_size; /*window size*/
    int * current_state; /*current state  0,1 */
    double * ssthresh; /*threshould*/
    int * send_base; /*current send base*/
    int loss_probability;
    int seed;
};
/*to store each packed with related time */
class Pair
{
public:
    clock_t time;
    packet pck;
    Pair(clock_t time, packet pck)
        : time(time), pck(pck) {}
};

/*to sort each packet depending on it's time*/
class PairCompare
{
public:
    bool operator()(const Pair &t1, const Pair &t2) const
    {
        return t1.time > t2.time;
    }
};
```

- ordinary data structures like :
    * priority_queue<Pair, vector<Pair>, PairCompare>* my_queue : to store each packed(vector of data) with related time.

* Streaming to read and write in filename
* vector.

Code Organization :

client side (Stop&wait):

- open client.in file and read all data in string array then close it .
- call **open_connection()** function to Create a datagram/UDP socket and construct the server address structures no need to (bind , no hand shaking)
** IP/TCP book as a reference**
- call **send_file_name()** function to wait until find server to connect with it and we handle that in time out function will mentioned below then send request contain file name as a data packet.
- call r**ecv_file()** function to recv all data in file as packets each packet size is 500 and check validation on that will be discussed below then if file is received fine store it in vector else change sequence number and we assume there is no corruption and in all case send Ack to indicate we recv file even if is correct or not.

server side (stop&wait):

- open server.in file and read all data in string array then close it .
- call **Connection()** function to create socket for sending/receiving datagrams , Construct local address structure and Bind to the local address then waiting for receiving message from client and once it received create thread using previous args struct contains data needed about child (child address).
- call **sendall()** in thread to create new socket and to serve this client via this socket then read packet from file to send it to client.
- call **send_data_to_client()** to send packet to client and wait to recv ack and check if this ack is valid or not and if time out happen (will discuss that later).

client side (Selective Repeat):

- similar to client side in stop&wait and take in consideration recv base and window size.

server side (Selective Repeat):

- *priority_queue<Pair, vector<Pair>, PairCompare>* my_queue;
to order thr packets to be sent regarding to its timeout

*pthread_mutex_t * timer_mutex;
to solve race condition during update the my_queue

*double * window_size;
int * current_state;
double * ssthresh;
to indicate the current widow size, state and threshold.

*pthread_mutex_t * state_mutex;
to solve the race condition during change the state.

we runs 3 threads:

* until send all file, check and update timeout, resend the loss packet agian after timeout and decrease the window size and threshold.

* until send all file, receive Ack from the client check sum this Ack, check if correct order, update the acknowleged and increase the window size with many ratio(linear, divide by 2 and exp.).

* until send all file, read date packet from file, try to send it , when ack, update window base and sequence number.

Handling Time out / loss packet / check sum:

- to type of time out :

* in first packet if client not yet connect with server so we use signal and time out in alarm function and errno to know if any interruption (EINTR) happen or not in both client in stop&wait and selective repeate.
* in stop&wait handle time out by setting time out to socket using **setscoket()** then in **recvfrom()** try to determine error type if errno = EWOULDBLOCK temporary unavailable so resend packet again and wait for recv Ack and so on.
* in selective repeat : Heeba

- loss packet :

* try to resend loss packet again and we do that for simulation by getting random number and compare it with **PLP** so determine if we send packet or not.

- check_Sum:

Note in calculation checksum we notice if we use struct of packet and ack_packet so :
cksum >> 2 B , len >> 2 B , ackno >> 4

so apply this role to find sum as decimal for each address as mentioned above then fit sum in 16 bit and take care of sign .

Then apply two’s complement.

I.e:
Ack seq = 0;
cksum = 0, len = 0 , ackno = 1;

sum = 1 and fit in 16 bit

apply two’s 1111 1111 1111 1110 >> 65534

Simulation:

- File size = 4MB

1%	Time (msec)	Throughput(B/msec)
Stop & wait	73.578	54364
Selective Repate	75.378	53065

5%	Time (msec)	Throughput(B/msec)
Stop & wait	82.605	12106
Selective Repate	73.126	13669

10%	Time (msec)	Throughput(B/msec)
Stop & wait	114.709	8718
Selective Repate	85.514	11694

30%	Time (msec)	Throughput(B/msec)
Stop & wait	247.561	4039
Selective Repate	89.551	11167

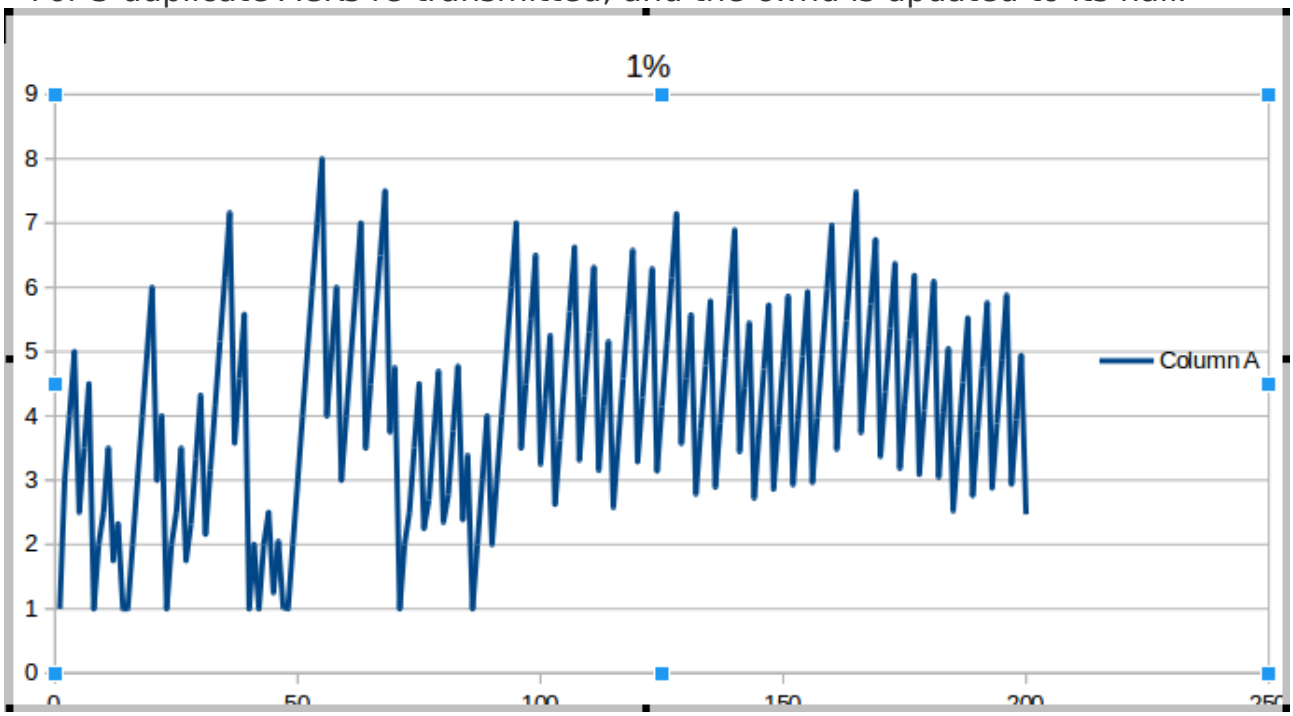
Note : Selective Repeat is better than stop and wait.

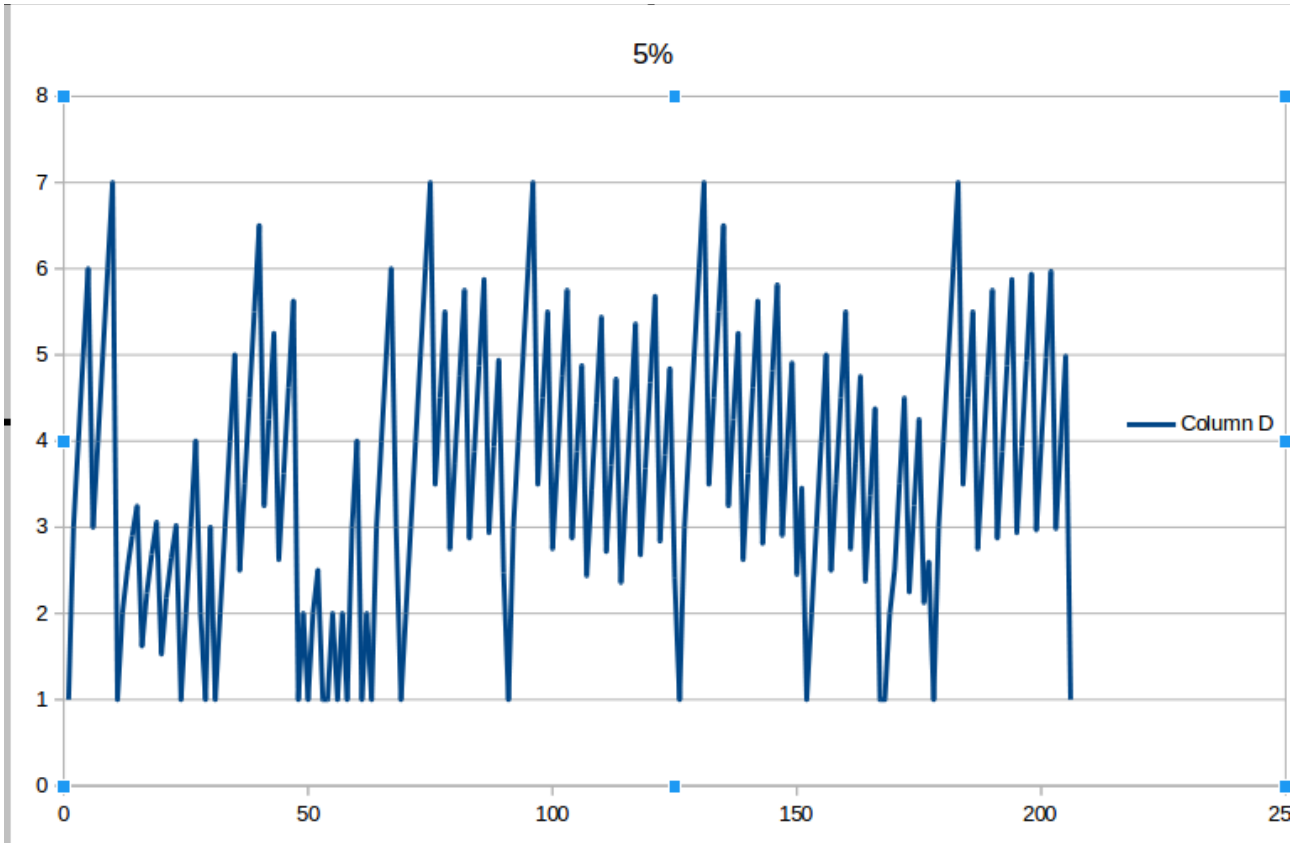
Congestion control:

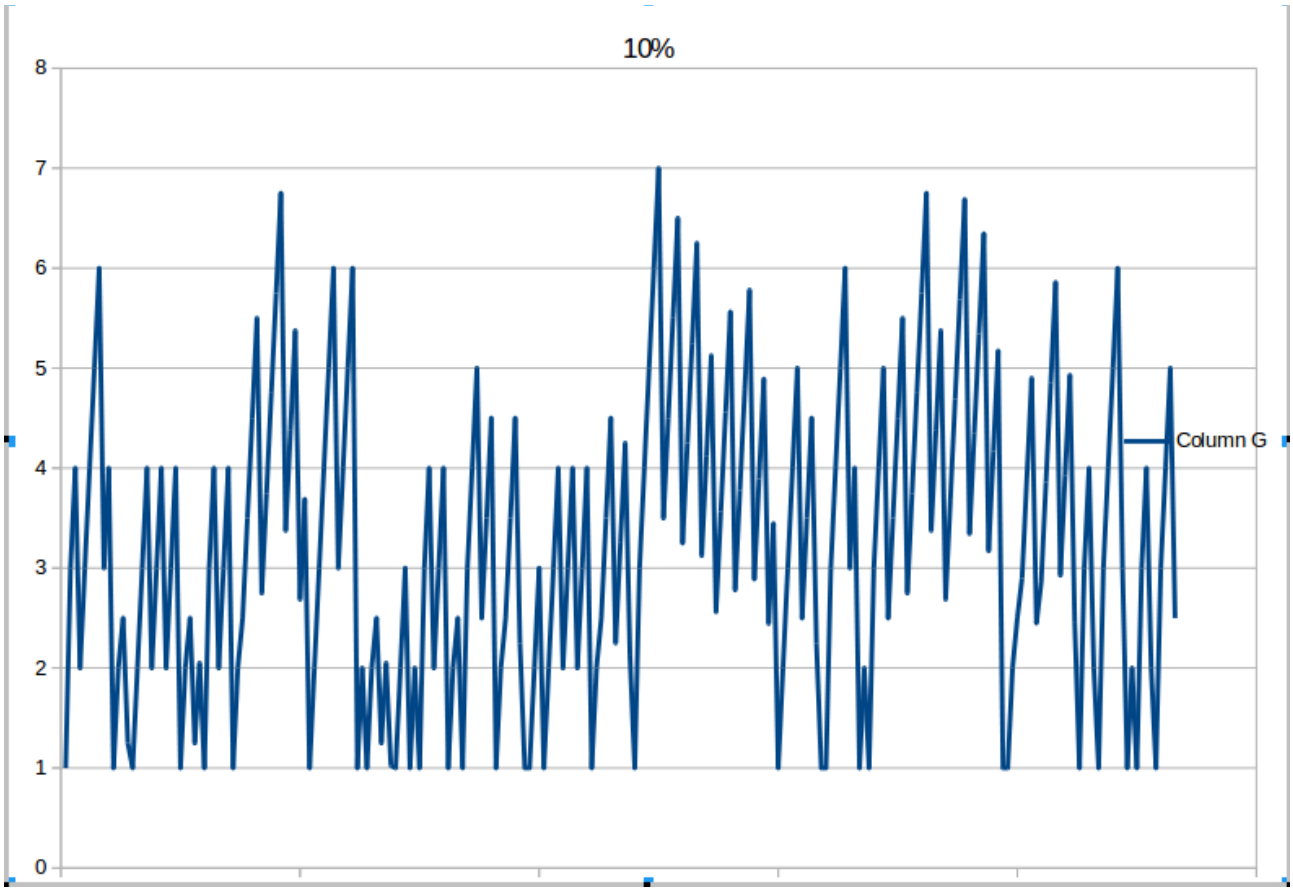
- two cases :

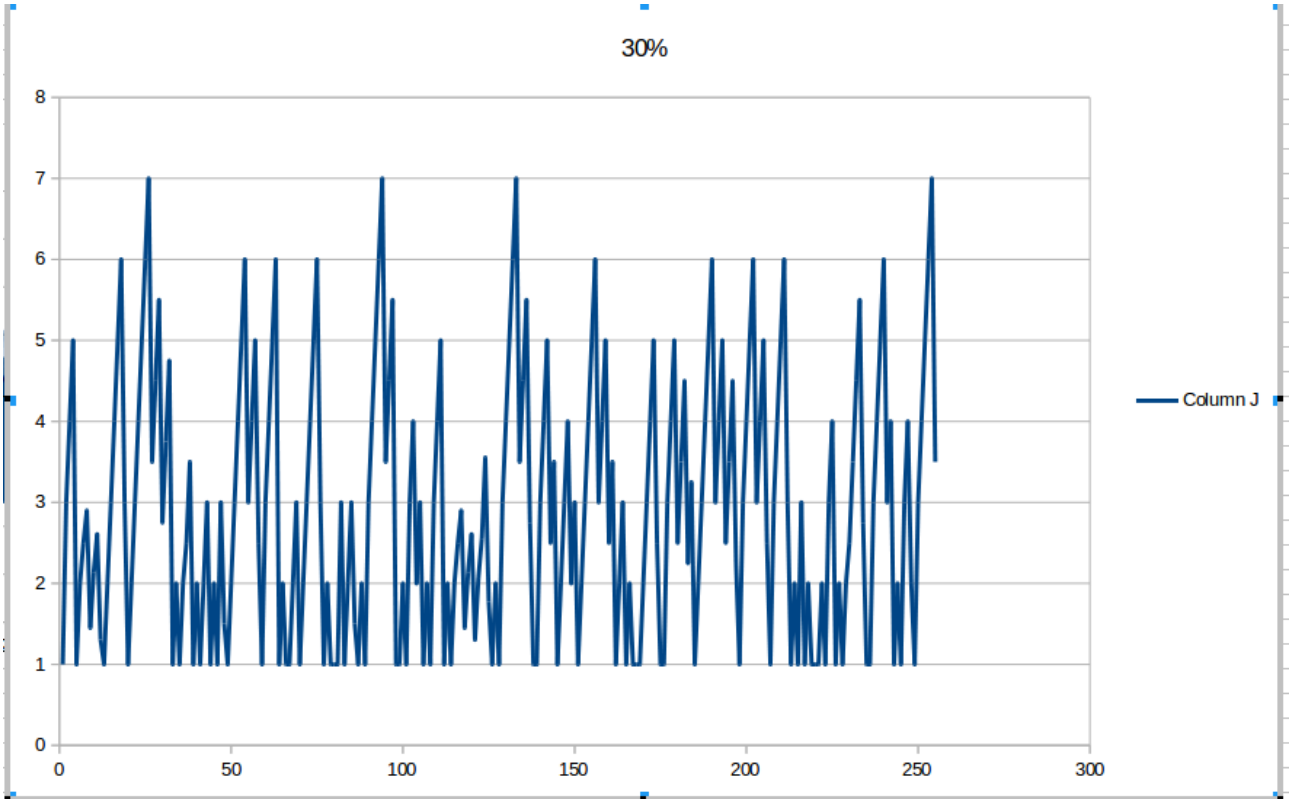
* the packet that timed out (lost)to be re-transmitted, and the cwnd is updated to 1.

* For 3 duplicate ACKs re-transmitted, and the cwnd is updated to its half.






10%
8
7
6
5
4
3
2
1
0
Column G


30%
8
7
6
5
4
3
2
1
0
0
50
100
150
200
250
300
Column J