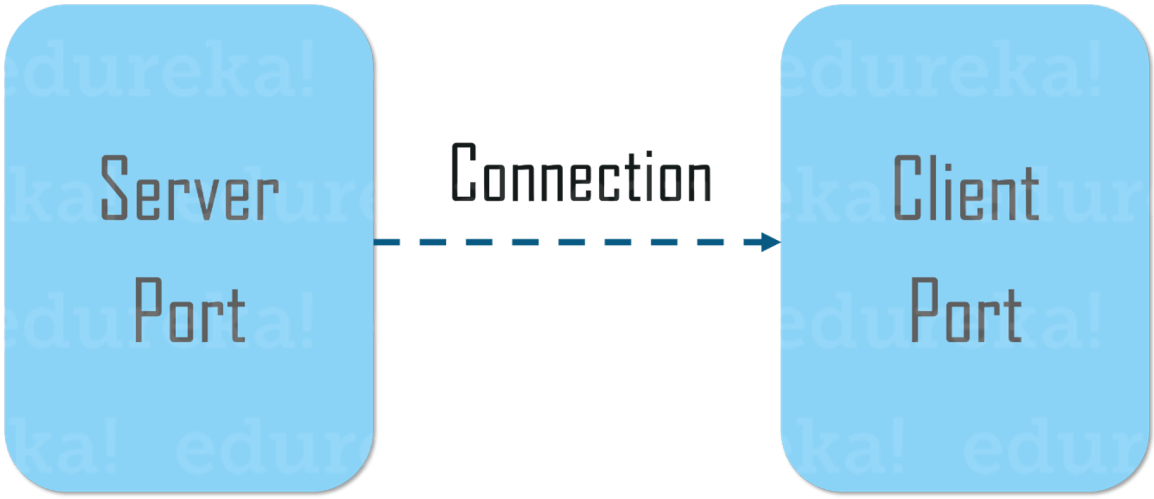
Networks

Implementing a Reliable Data Transport Protocol

Assignment #3



Github Repo :
https://github.com/khadijaAssem/Implementing-a-Reliable-Data-Transport-Protocol

Introduction

Implementing socket layer and reliable transport layer.

Implementing a reliable transfer service on top of the UDP/IP protocol. In other words, you 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.

Overall organization

There are two main files: client_runner.cpp and server_runner.cpp for both client_runner and server_runner there are client.cpp and server.cpp that use utility files (utilities file for operations in file, utilities file for sending and receiving packets/ACKS).

I also added make file to build and compile the client and servers and generate the executables for both server and client.

For client:

Client_runner.cpp: It is the main function that it is used to run the client and sends the required arguments to the client.

```
int main(int argc, char *argv[]) {
    client c;

    char PORT[] = "3490"; // set default value of PORT
    char HOSTNAME[] = "localhost"; // set default value of HOSTNAME
    char COMMANDFILE[] = "commands.txt"; // set default value of COMMANDFILE

    if (argc < 4) {
        printf("WILL USE HOSTNAME: %s\nPORTNUMBER: %s\nCOMMANDFILE: %s\n", HOSTNAME, PORT, COMMANDFILE);
        printf("IF YOU WANT TO CHANGE ONE OF THE DEFAULT RUN THE FOLLOWING\n");
        printf("./my_client hostname portNumber commands\n");
        printf("THANKS :)\n--------------------------------------------------------------------------------\n");
    }

    if (argc == 4) {
        strcpy(COMMANDFILE, argv[3]); // get command file from user
        strcpy(PORT, argv[2]); // get port number from user
        strcpy(HOSTNAME, argv[1]); // get host name from user
    }

    c.run(HOSTNAME, PORT, COMMANDFILE);
}
```

Client.cpp: It is the main client class that contains the overall logic of the client to construct messages and send them to the server.It is initiated when the client_runner calls it's function run with the required arguments.

```
int run(char HOSTNAME[], char PORT[], char COMMANDFILE[]);
```

For server:

Server_runner.cpp: It is the main function that it is used to run the server and sends the required arguments to the server.

```
int main(int argc, char *argv[]) {
    // RUN AS : my_server portNumber random_generator_seed_value probability_of_datagram_loss
    server s;
    // All ports below 1024 are RESERVED (unless you're the superuser)! You can have any port number abo
    char PORT[] = "3490"; // set default value of PORT
    float random_generator_seed_value = 1000;
    float probability_of_datagram_loss = 0.3; //(real number in the range [0.0 , 1.0])

    if (argc < 3) {
        printf("WILL USE PORTNUMBER: %s\n", PORT);
        printf("WILL USE RANDOM GENERATOR SEED VALUE: %f\n", random_generator_seed_value);
        printf("WILL USE DATAGRAM LOSS PROBABILITY: %f\n", probability_of_datagram_loss);
        printf("IF YOU WANT TO CHANGE THE DEFAULT RUN THE FOLLOWING\n");
        printf("./my_server portNumber random_generator_seed_value probability_of_datagram_loss\n");
        printf("THANKS :)\n----------------------------------------------------------------------------
    }
    if (argc >= 3){
        strcpy(PORT, argv[1]); // get port number from user
        random_generator_seed_value = std::stof(argv[2]);
        probability_of_datagram_loss = std::stof(argv[3]);
    }

    s.run(PORT, random_generator_seed_value, probability_of_datagram_loss);
}
```

Server.cpp: It is the main server class that contains the overall logic of the server to receive messages and send responses to the client.It is initiated when the server_runner calls it's function run with the required arguments.

```
public:
int run(char PORT[], float random_generator_seed_value, float probability_of_datagram_loss);
```

For network utilities:

It contains the important public main functions used by both server and client to send and receive packets/ ACKs.

```
public:
net_utilities();
net_utilities(float random_generator_seed_value, float probability_of_datagram_loss);
bool sendAck(int sockfd, uint32_t seqNum,
    struct sockaddr_storage toaddress, socklen_t addrlen); // For Server
bool sendAck(int sockfd, uint32_t seqNum,
    struct addrinfo *toaddress); // For Client

bool recieveAck(int sockfd, struct addrinfo *fromaddress, struct ack_packet *ACK); // For Client
bool recieveAck(int sockfd, struct sockaddr_storage fromaddress, socklen_t addrlen,
    struct ack_packet *ACK); // For Server

bool recievePacket(int sockfd, struct sockaddr_storage *fromaddress,
    socklen_t addrlen, struct packet *pkt); // For Server
bool recievePacket(int sockfd, struct addrinfo *fromaddress, struct packet *pkt); // For Client

std::string printIP(struct addrinfo *address); // convert IP address to string
std::string printIP(struct sockaddr_storage address);

bool sendPacket(int sockfd, struct packet *pkt, struct addrinfo *toaddress); // For Client
bool sendPacket(int sockfd, struct packet *pkt,
    struct sockaddr_storage *toaddress, socklen_t addrlen); // For Server

int createSocket(); // Create socket with timeout
int createSocket(int timeoutRecieve, int timeoutSend);
```

For other utilities:

It contains main data structures of both data packets and ACK packets

```
/* Data-only packets */
struct packet {
    /* Header */
    uint16_t cksum = 0; /* optional bonus part */ /* 2 bytes size */
    uint16_t len = DUMMY_VALUE; /* 2 bytes size */
    uint32_t seqno = DUMMY_VALUE; /* 4 bytes size */
    /* Data */
    char data[MAXDATASIZE]; /* Not always MAXDATASIZE bytes, can be less */
};

/* Ack-only packets are only 8 bytes */
struct ack_packet {
    uint16_t cksum; /* optional bonus part */ /* 2 bytes size */
    uint16_t len; /* 2 bytes size */
    uint32_t seqno; /* 4 bytes size */
};
```

Contains the message data structure used by both server and client and includes all data information about command

```
struct messege_content {

    char request[8]; // GET or Post
    char file_path[MAXFILEPATHSIZE];
    char host_name[MAXHOSTNAMESIZE];
    char port_number[MAXPORTNUMBSIZE];
    char request_msg[MAXDATASIZE]; // Whole request messege
    struct packet pckt; // Packet to be sent

};
```

Also contains main definitions used by both sender and receiver

```
#define BACKLOG 5    // how many pending connections queue will hold
#define MAXFILESIZE 1000000 // Maximum file to read
#define ACKPCKTSIZE 8
#define HEADERSSIZE 8
#define MAXDATASIZE 504
#define MAXPCKTSIZE MAXDATASIZE + HEADERSSIZE
#define MAXFILEPATHSIZE 50 // max number of bytes we can get at once
#define MAXPORTNUMBSIZE 20 // max number of bytes we can get at once
#define MAXHOSTNAMESIZE 50 // max number of bytes we can get at once

#define MICROSECONDS 1000000
#define SLEEP100 usleep(100 * MICROSECONDS) // For introducing some timeouts
#define SLEEP10 usleep(10 * MICROSECONDS) // For introducing some timeouts
#define SLEEP5 usleep(5 * MICROSECONDS) // For introducing some timeouts
#define SLEEP1 usleep(1 * MICROSECONDS) // For introducing some timeouts
#define TIMEOUT 5000000

#define SLOW_START 0 // IN GBN finite state machine
#define FAST_RECOV 1 // IN GBN finite state machine
#define CONG_AVOID 2 // IN GBN finite state machine

#define DUMMY_VALUE -1 // For initializing packet values
#define FIN_SEQNUM -2 // For fin datagram used for closing the connection
#define FIN_BIT -2
```

Main functions used by client and server

```
class utilities
{
    public:
    void save_data_to_path(char *buffer, std::string path);
    std::string read_data_from_path(std::string);
    struct messege_content request_postprocessing(char[]);
    std::vector<struct packet> create_file_packets(std::string);
};
```

Important Functions

For client:

```
int client::run(char HOSTNAME[], char PORT[], char COMMANDFILE[])
```

The function that does the whole logic of the client and uses some utility functions from the utilities class for some sub-tasks.

1. Process the commands file and create the message requests
2. Create socket for the client
3. Connects with the server on the servers socket
4. Send the requested file name to the server
5. Wait for receiving data packets containing the file text
6. Ack each received packet considering the order of packets

For server:

```
int server::run(char PORT[])
```

The function that does the whole logic of the server and uses some utility functions from the utilities class for some sub-tasks.

1. Create socket for the server
2. Bind to the server socket

3. Start listening on the socket so incoming connections will wait on the queue waiting to be accepted by the server
4. The main accept loop that receive the first packet from the client containing the required file name
5. Creating a process for handling the file sending process either by go back N or by stop and weight protocol

For utilities:

A function that saves the passed data to a file on a local directory with the given path.

```
void save_data_to_path(char *buffer, std::string path);
```

A function to read data from the passed path and return it.

```
std::string read_data_from_path(std::string);
```

Used by client to transfer the given command from commands file to messages and form the packet:

```
struct messege_content request_preprocessing(char[]);
```

Uses read_data_from_path to read the file and divide it to fixed size packets except last packet

```
std::vector<struct packet> create_file_packets(std::string);
```

Data Structures

```c
/* Data-only packets */
struct packet {
    /* Header */
    uint16_t cksum = 0; /* optional bonus part */ /* 2 bytes size */
    uint16_t len = DUMMY_VALUE; /* 2 bytes size */
    uint32_t seqno = DUMMY_VALUE; /* 4 bytes size */
    /* Data */
    char data[MAXDATASIZE]; /* Not always MAXDATASIZE bytes, can be less */
};
```

A structure to save the packet and server/client use this data structure as the unit of sending and receiving data

```c
/* Ack-only packets are only 8 bytes */
struct ack_packet {
    uint16_t cksum; /* optional bonus part */ /* 2 bytes size */
    uint16_t len; /* 2 bytes size */
    uint32_t seqno; /* 4 bytes size */
};
```

For ACK packets

Network system analysis (For 4MB file)

For probability of datagram loss = 1%

GBN	Stop and wait
1st run: 28010891	1st run: 147016791
2nd run: 55267419	2nd run: 147176404
3rd run: 69991705	3rd run: 147405160
4th run: 58414285	4th run: 147410983
5th run: 60106830	5th run: 147046427

For probability of datagram loss = 5%

GBN	Stop and wait
1st run: 38312978	1st run: 819229261
2nd run: 39073250	2nd run: 819736999
3rd run: 45873806	3rd run: 816327125
4th run: 57496002	4th run: 816670501
5th run: 72051761	5th run: 816247295

For probability of datagram loss = 10%

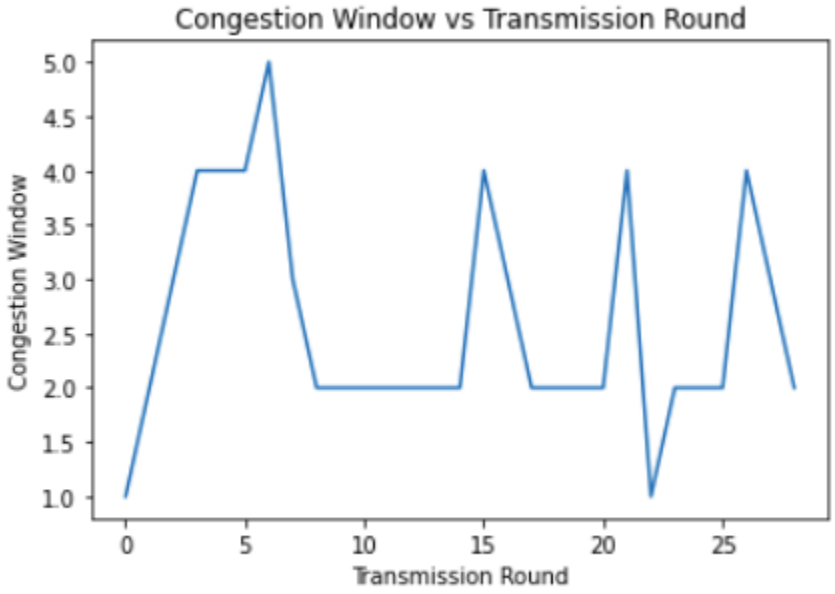
GBN	Stop and wait
1st run: 57394047	1st run: 1828186114
2nd run: 61858653	2nd run: 1827246017
3rd run: 74703006	3rd run: 1831014225
4th run: 76360257	4th run: 1830196132
5th run: 74275478	5th run: 1828703905

For probability of datagram loss = 30%

GBN	Stop and wait
1st run: 83190017	1st run: 1824508698
2nd run: 83432470	2nd run: 1824156785
3rd run: 83725130	3rd run: 1824786064
4th run: 63003549	4th run: 1834807002
5th run: 78322131	5th run: 1842530979

Network system analysis (Graphs)

Graph for 64K file with loss probability 30%



Graph for 164K file with loss probability 30%