

## TCP – Part A

**Names: Shira Yogev, Daniel Kuris**

**IDs: 214539397 , 325877208**

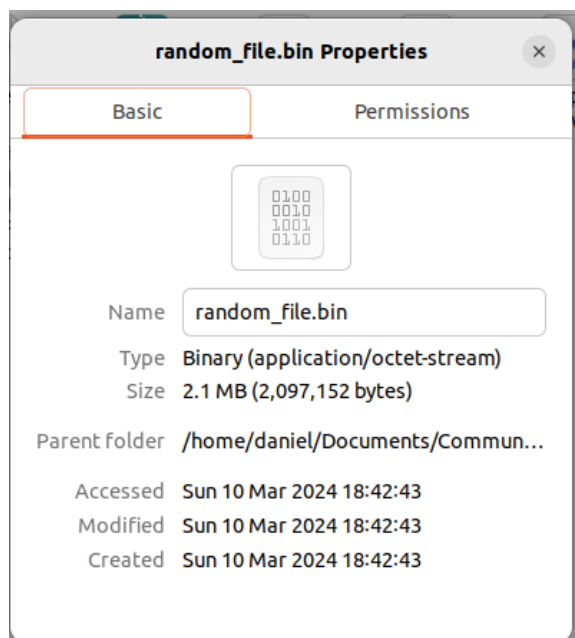
**#Note:** We chose **IPv4**.

**#Note:** If you're looking for how to run the code, **we added example screenshots of both algorithm's run** and an example for a disconnection that was caused due to one party closing the terminal.

### Sender:

We have 2 defines – file size and buffer size. The buffer size is the size of the packets that will be sent to the receiver until the file is sent completely. The file size is the amount of bytes in the file.

We made an helper function named “ generate\_random\_file “ that will write input in a file so that its size will be the the wanted size (in this case, the defined file size).



### Sender's main:

We read the command line from the terminal and check its correctness to our case. If it is not in the expected format, it will print the correct usage.

```
daniel@daniel-VirtualBox:~/Documents/CommunicationNetworks/communityNetwork_exe3
/partA$ ./TCP_Sender random message
Usage: ./TCP_Sender -ip <ip> -p <receiver_port> -algo <congestion_algorithm>
daniel@daniel-VirtualBox:~/Documents/CommunicationNetworks/communityNetwork_exe3
/partA$
```

We then proceed to create the socket in IPv4, TCP:

```
// Create a TCP socket
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == -1) {
    perror("Error creating socket");
    return -1;
}
```

We check if we were given the correct algorithm:

```
// Set TCP congestion control algorithm
if (strcmp(congestion_algorithm, "cubic") != 0 && strcmp(congestion_algorithm, "reno") != 0 ) {
    printf("Invalid congestion algorithm: %s\n", congestion_algorithm);
    close(sock);
    return -1;
}
```

And we set the socket accordingly:

```
if (setsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, congestion_algorithm, strlen(congestion_algorithm)) < 0) {
    perror("setsockopt() failed");
    close(sock);
    return -1;
}
```

We then connect to the receiver:

```
// Set up connection parameters
struct sockaddr_in server_address;
memset(&server_address, 0, sizeof(server_address));
server_address.sin_family = AF_INET;
server_address.sin_port = htons(receiver_port);
inet_pton(AF_INET, receiver_ip, &server_address.sin_addr);

// Connect to the receiver
if (connect(sock, (struct sockaddr *)&server_address, sizeof(server_address)) == -1) {
    perror("Error connecting to receiver");
    close(sock);
    return -1;
}
```

After we connect, we check the connection and establish that we are indeed connected by performing an “**hand shake**” with the receiver.

This procedure is done by:

1. Sending a message ‘H’ from the sender to the receiver.
2. Receiving ‘A’ from the receiver.

If all went well, we then print that the handshake was successful and that we are properly connected. We also print the connection details.

```
/partA$ make runc-cubic
./TCP_Sender -ip localhost -p 12345 -algo cubic
Handshake successful - connection established.
Connected to localhost:12345
```

Now that the connection is established, we enter the loop in which we repeatedly send the same randomly generated file until either the sender has decided ‘n’ , or one of the parties disconnected.

In case of unexpected disconnection (one party closed the connection prematurely), we made sure to close and free all relevant variables.

Premature disconnection:

```
daniel@daniel-VirtualBox:~/Documents/CommunicationNetworks/communityNetwork_exe3
/partA$ make runc-cubic
./TCP_Sender -ip localhost -p 12345 -algo cubic
Handshake successful - connection established.
Connected to localhost:12345
Sending 2097152 bytes of data...
Successfully sent 2097152 bytes of data!
Time taken: 1.06 ms
Do you want to send more data? (y/n): y
Waiting for the server to be ready...
Server localhost:12345 disconnected.
daniel@daniel-VirtualBox:~/Documents/CommunicationNetworks/communityNetwork_exe3
/partA$
```

In case we are successfully running the loop until the ‘n’ message was received, we send the file over to the receiver. We print how many bytes should’ve been sent and how many actually were.

We then print how long it took to send the file.

After the sending itself is done, the user (in sender) will be required to give either ‘y’ or ‘n’ answer. In case the user answers unexpectedly, he will be required to answer correctly.

```
while (decision != 'y' && decision != 'n') {
    printf("Invalid answer, do you want to send more data? (y/n) ");
    scanf(" %c", &decision);
}
```

It doesn't matter which decision the sender made, we we'll always send its decision to the receiver end to be processed.

In case we sent 'n', the sender exits the loop, which then closes the connection.

```
if (decision == 'n')
    break;
```

```
printf("Closing connection...\n");
// Close the TCP connection
free(buffer);
close(sock);

return 0;
```

If the sender chose 'y', the sender will be waiting to receive a sync byte from the receiver. So that it knows that the receiver got the message to keep receiving the files.

```
else if (decision == 'y'){
    printf("Waiting for the server to be ready...\n");

    // Receive the sync byte from the receiver
    char sync_byte;
    ssize_t bytes_received = recv(sock, &sync_byte, sizeof(sync_byte), 0);
    if (bytes_received < 0) {
        perror("Error receiving sync byte from receiver");
        free(buffer);
        close(sock);
        return -1;
    } else if (bytes_received == 0) {
        printf("Server %s:%d disconnected.\n", receiver_ip, receiver_port);
        free(buffer);
        close(sock);
        return 0;
    }

    // Check if the received byte is the sync byte
    if (sync_byte == 'S') {
        printf("Server accepted your decision.\n");
    } else {
        printf("Error: Unexpected response from server.\n");
        free(buffer);
        close(sock);
        return -1;
    }
}
```

After the sender receives the sync byte correctly, both parties are ready to send over the file again, and so the loop repeats.

## Receiver:

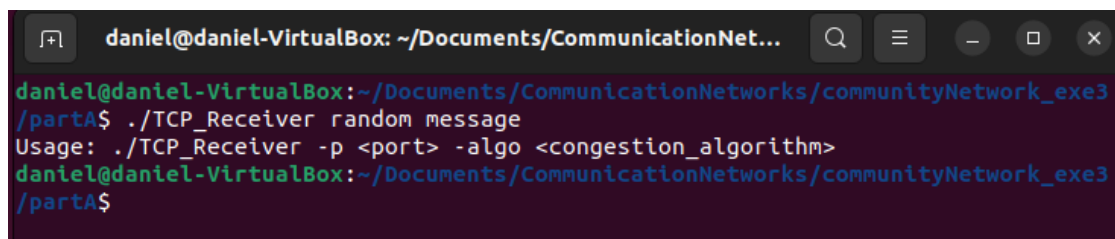
We defined 3 constants, similarly to the sender. Statistics size, file size and buffer size. The buffer size is the size of the packets that will be received from the sender until the file is sent completely. The file size is the amount of bytes in the file. And **statistics size is the maximum amount of times the sender is able to send the file.**

**#Note:** We made the maximum amount 100 instead of dynamically increasing the array that saves information because we assumed there won't be that many repeated sends of the same file. So 100 seemed reasonable.

The receiver also has couple helper functions to deal with saving the data and presenting it correctly.

### Receiver's main:

We read the command line from the terminal and check its correctness to our case. If it is not in the expected format, it will print the correct usage.



```
daniel@daniel-VirtualBox: ~/Documents/CommunicationNet...
daniel@daniel-VirtualBox:~/Documents/CommunicationNetworks/communityNetwork_exe3
/partA$ ./TCP_Receiver random message
Usage: ./TCP_Receiver -p <port> -algo <congestion_algorithm>
daniel@daniel-VirtualBox:~/Documents/CommunicationNetworks/communityNetwork_exe3
/partA$
```

We create the socket similar to the sender, but we also add the 'listen' and 'accept' in order to accept the sender's 'connect'.

```
if (listen(listening_socket, 1) == -1) {
    perror("listen() failed");
    close(listening_socket);
    return -1;
}

printf("Waiting for incoming TCP connections...\n");

struct sockaddr_in client_address;
socklen_t client_address_len = sizeof(client_address);
int client_socket;

// Accept the connection from the sender
memset(&client_address, 0, sizeof(client_address));
client_address_len = sizeof(client_address);

client_socket = accept(listening_socket, (struct sockaddr *)&client_address, &client_address_len);
if (client_socket == -1) {
    perror("accept() failed");
    close(listening_socket);
    return -1;
}
```

After we connect, we check the connection and establish that we are indeed connected by performing an “hand shake” with the receiver.

This procedure is done by:

1. Sending a message ‘H’ from the sender to the receiver.
2. Receiving ‘A’ from the receiver.

If all went well, we then print that the handshake was successful and that we are properly connected. We also print the connection details.

```
daniel@daniel-VirtualBox:~/Documents/Communicati
/partA$ make runs-reno
./TCP_Receiver -p 12345 -algo reno
Waiting for incoming TCP connections...
Handshake successful - connection established.
Connected to 127.0.0.1:12345
```

Now that the connection is established, we enter the loop in which we repeatedly receive the same randomly generated file until either the sender has decided ‘n’ , or one of the parties disconnected.

In case of unexpected disconnection (one party closed the connection prematurely), we made sure to close and free all relevant variables.

Premature disconnection:

```
daniel@daniel-VirtualBox:~/Documents/CommunicationNetworks/communityNetwork_exe3
/partA$ make runs-reno
./TCP_Receiver -p 12345 -algo reno
Waiting for incoming TCP connections...
Handshake successful - connection established.
Connected to 127.0.0.1:12345
Receiving file from client...
Received 2097152 bytes of data from 127.0.0.1:12345.
Waiting for client's decision...
Client 127.0.0.1:12345 disconnected.
daniel@daniel-VirtualBox:~/Documents/CommunicationNetworks/communityNetwork_exe3
/partA$
```

In case we are successfully running the loop until the ‘n’ message was received, we receive the file from the sender. We print how many bytes we’ve received.

After the receiving itself is done, the user (in receiver) will be waiting to get either ‘y’ or ‘n’ answer from the sender.

In case the sender answers 'y', the receiver will send a sync byte over to the sender. So that it confirms for sender end that they're ready for the next transfer. Then, the loop continues.

```
Waiting for client's decision...
Client responded with 'y', sending sync byte...
```

In case the sender answers 'n', the receiver will close the connection and print the statistics:

```
else if (decision == 'n') {
    printf("Client responded with 'n'.\n");
    printf("Closing connection...\n");
    // Close the TCP connection
    free(buffer);
    close(client_socket);
    close(listening_socket);
    print_statistics(timeTaken, transferSpeed, congestion_algorithm, iteration);
    free(timeTaken);
    free(transferSpeed);
    return 0;
}
```

```
- - - - -
-          * Statistics *          -
- Algorithm: reno
- The file was sent 4 times
- Average time taken to receive the file: 2.62 ms.
- Average throughput: 787.70 Mbps
- Total time: 10.49 ms

Individual samples:
- Run #1 Data: Time=2.14 ms; Speed=932.84 Mbps
- Run #2 Data: Time=3.17 ms; Speed=630.12 Mbps
- Run #3 Data: Time=2.17 ms; Speed=922.93 Mbps
- Run #4 Data: Time=3.01 ms; Speed=664.89 Mbps
- - - - -
Receiver end.
```