Assignment 3 – Transport Layer

Assignment Notes and Instructions

This assignment is to be done in **pairs** only. You must upload all the necessary files in a ZIP file named after both of the student's ID. For example: $123456789\_987654321.zip$.

1) The assignment must be submitted by the date specified in the submission box.

2) All the assignment files (Code, Wireshark pcap files, PDF) must be submitted in a ZIP file. The PDF must contain all the necessary screenshots with a description for each one of them.

3) The assignment must be done in **pairs only**. In exceptional cases, you should send an email to the course coordinator.

4) You can use any reference material available through the course Moodle website or any material passed during the exercises.

5) You can use any reference from the web, but **copying codes from websites is forbidden**; this includes codes from other GitHub repositories. **A student who gets caught in the act will automatically fail this assignment (score 0).** Any website you've used to solve the assignment, every website you used to solve the assignment must be noted in the PDF.

6) No delays in submission without special permission. Lateness without approval will result in failure of the assignment (score 0).

7) All submissions will be submitted via the course's Moodle website. Email submissions will not be accepted, resulting in a failure in the assignment.

8) The order and design of the code must be carefully considered. Make sure that the program output is as clear as possible, comments in the code, and meaningful variable names. You must also submit a **makefile** that compiles all the programs.

9) **The assignment code must be written in C only**.

10) The assignment is personal for each pair, and you should not accept help from other people, whether outside the university or inside it. You can contact the course staff during reception hours for help or raise a question in the course forum. **Do not transfer code sections between students, upload solutions or parts of solutions to websites on the Internet or in various communication groups.**

11) Students who copy a solution will receive a 0 in all assignments in the course and a report will be made to the institutional disciplinary committee.

12) All code files of the assignment are required to both compile and run properly on the **Ubuntu 22.04 LTS** operating system. This is the operating system where the submissions will be tested. You **mustn't** code it in Windows, as it uses different API for sockets. WSL won't work, as the task needs a tool that works only in a fully Linux environment.

Good Luck!

Part A – Transmission Control Protocol (TCP) – 30 points

In this part, you'll write two program files: $\boldsymbol{TCP\_Sender.c}$ and $\boldsymbol{TCP\_Receiver.c}$.

The Sender will send a randomly generated file and the Receiver will receive it and measure the time it took for his program to receive the file. **The file size should be at least 2MB.**

When running the Sender and Receiver, they both should support two standard congestion control algorithms: **TCP Reno** and **TCP Cubic**.

Notes:

- The main goal of this part is to get familiar with the TCP concept of sending messages between two endpoints.
- For approaching this part of the assignment, please be sure that you are familiar with the congestion control algorithms and how to change them using the correct method in C (see **Appendix B** for more information).
- The receiver doesn't really care about saving the file itself (or its content). You've learned that TCP is reliable by design, so you can assume that all the data have been passed correctly, and thus, no need to recheck.
- You can support either IPv4 or IPv6, but you must note in your PDF in which IP version your assignment uses.

Usage:

- **Receiver:**

$$./TCP\_Receiver\ -p\ PORT\ -algo\ ALGO$$

- **Sender:**

$$./TCP\_Sender\ -ip\ IP\ -p\ PORT\ -algo\ ALGO$$

Where:

- $\boldsymbol{IP}$ – The IP address of the receiver.
- $\boldsymbol{PORT}$ – The TCP port of the Receiver.
- $\boldsymbol{ALGO}$ – The TCP congestion control algorithm that will be used by the party member (either $\boldsymbol{reno}$ for TCP Reno or $\boldsymbol{cubic}$ for TCP Cubic).

The programs will do the following:

Sender

1) Read the created file.

2) Create a TCP socket between the Sender and the Receiver.

3) Send the file.

4) User decision: Send the file again?
   a. If yes, go back to step 3.
   b. If no, continue to step 5.

5) Send an exit message to the receiver.

6) Close the TCP connection.

7) Exit.

Receiver

1) Create a TCP connection between the Receiver and the Sender.

2) Get a connection from the sender.

3) Receive the file, measure the time it took and save it.

4) Wait for Sender response:
   a. If Sender resends the file, go back to step 3.
   b. If Sender sends exit message, go to step 5.

5) Print out the times (in milliseconds), and the average bandwidth for each time the file was received.

6) Calculate the average time and the total average bandwidth.

7) Exit.

Example output of the Receiver (this is just an example, your output can look different, as long as it’s readable and understandable):

```
foo@bar:~$ ./TCP_Receiver -p 1234 -algo reno
Starting Receiver...
Waiting for TCP connection...
Sender connected, beginning to receive file...
File transfer completed.
Waiting for Sender response...
Sender sent exit message.
----------------------------------
-         * Statistics *         -
- Run #1 Data: Time=0.2ms; Speed=1024.00MB/s
-
- Average time: 0.2ms
- Average bandwidth: 1024.00MB/s
----------------------------------
Receiver end.
```

Part B – Reliable User Datagram Protocol (Reliable UDP or RUDP) – 40 points

You've learned that while TCP is reliable, it lacks speed and has an overhead data of 20 bytes, compering to UDP that needs only 8 bytes of overhead data.

Reliable UDP or RUDP, is a protocol that uses the UDP protocol for a faster transmission of data, while adding a reliability layer to it, but without adding too much overhead data like TCP.

A simple sketch for a RUDP connection would look something like this:

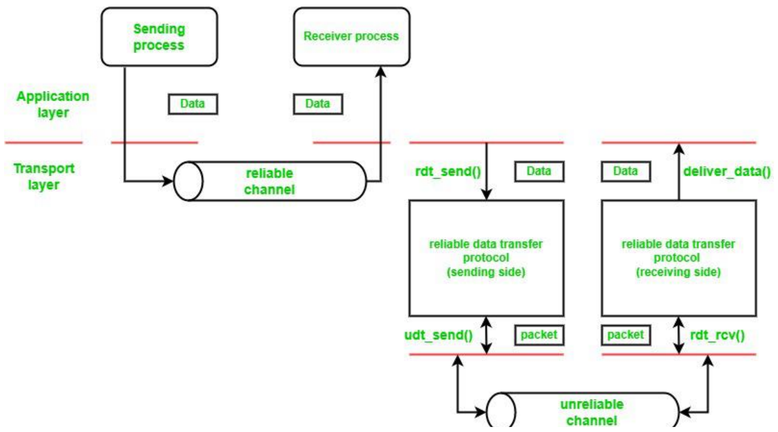


Figure 1 - RUDP Protocol Architecture, Geeksforgeeks

Your task in this part is to implement a Reliable protocol using UDP only. For simplicity, your implementation should only support **a handshake, closing scheme, error checking, acknowledgment of received packets, and retransmission of lost packets**.

Guidance:

- You can use $setsockopt(2)$ for detecting a timeout.

- You need to build your own protocol, as Reliable UDP isn't a standard protocol (see a draft of RUDP in **this link**), and no official header is provided. You can use this sketch for a header, but you can come up with your own protocol, as long as it works.

Reliable UDP Header proposed sketch			
Byte 0	**Byte 1**	**Byte 2**	**Byte 3**
Length (2 Bytes)		Checksum (2 Bytes)	
Flags (1 Byte)			

  Where:
  - $\boldsymbol{Length}$ is the length of the data itself, without the RUDP header.
  - $\boldsymbol{Checksum}$ is a 16bit number that validates the correctness of the data.
  - $\boldsymbol{Flags}$ is a special byte where we classify the packet itself (SYN, ACK, etc.).

- In your PDF, you must explain how your implementation of RUDP works – How does it detect a timeout to do retransmissions, how does the handshake work, what is the header overhead, etc.

- For this part, you'll write three files: $\boldsymbol{RUDP\_API.c}$, $\boldsymbol{RUDP\_Sender.c}$ and $\boldsymbol{RUDP\_Receiver.c}$.

Notes:

- The Sender will send a random generated file and the Receiver will receive it and measure the time it took for his program to receive the file. **The file size should be at least 2MB.**

- The receiver doesn't really care about saving the file itself (or its content). But, as the RUDP is based on UDP, it's not reliable. You must use some form of error checking (like checksum) to ensure that the packet wasn't alerted during the transfer.

- You can support either IPv4 or IPv6, but you must note in your PDF in which IP version your assignment uses.

- The RUDP needs support only 1 sender and 1 receiver, for simplicity purposes.

Usage:

- **Receiver:**

$$./RUDP\_Receiver\ \ -p\ PORT$$

- **Sender:**

$$./RUDP\_Sender\ \ -ip\ IP\ \ -p\ PORT$$

Where:

- $\boldsymbol{IP}$ – The IP address of the receiver.
- $\boldsymbol{PORT}$ – The TCP port that both the Sender and the Receiver will use.

The $\boldsymbol{RUDP.c}$ file will contain all the implementations of the RUDP API functions (That is, to prevent reusing the same chuck of code both in the Sender and the Receiver.).

The API should support at the bare minimum the following functions (you decide what parameters to implement, and what are the names of the functions, as long as the API is explained in the PDF):

- $rudp\_socket()$ – Creating a RUDP socket and creating a handshake between two peers.

- $rudp\_send()$ – Sending data to the peer. The function should wait for an acknowledgment packet, and if it didn't receive any, retransmits the data.

- $rudp\_recv()$ – Receive data from a peer.

- $rudp\_close()$ – Closes a connection between peers.

The programs will do the following:

Sender

1) Read the created file.

2) Create a UDP socket between the Sender and the Receiver.

3) Send the file via the RUDP protocol.

4) User decision: Send the file again?
    a. If yes, go back to step 3.
    b. If no, continue to step 5.

5) Send an exit message to the receiver.

6) Close the TCP connection.

7) Exit.

Receiver

1) Create a UDP connection between the Receiver and the Sender.

2) Get a connection from the sender, by the custom RUDP protocol you've built.

3) Receive the file, measure the time it took and save it.

4) Wait for Sender response:
    a. If Sender resends the file, go back to step 3.
    b. If Sender sends exit message, go to step 5.

5) Print out the times (in milliseconds), and the average bandwidth for each time the file was received.

6) Calculate the average time and the total average bandwidth.

7) Exit.

Example output of the Receiver (this is just an example, your output can look different, as long as it's readable and understandable):

```
foo@bar:~$ ./RUDP_Receiver -p 1234
Starting Receiver...
Waiting for RUDP connection...
Connection request received, sending ACK.
Sender connected, beginning to receive file...
File transfer completed.
ACK sent.
Waiting for Sender response...
Sender sent exit message.
ACK sent.
----------------------------------
-         * Statistics *         -
- Run #1 Data: Time=0.2ms; Speed=1024.00MB/s
-
- Average time: 0.2ms
- Average bandwidth: 1024.00MB/s
----------------------------------
Receiver end.
```

Part C – Research – 25 points

In this part, you'll conduct research about your implementations of part A and part B.

In the research itself, each one of the versions (TCP version and UDP version) will be run at least 5 times (meaning that the file transfer will occur at least 5 times) by the following conditions:

- For TCP, you'll run it in TCP Reno and TCP Cubic, alternating each time, for each scenario, meaning that you run 5 times with TCP Reno in both Sender and Receiver, and 5 times with TCP Cubic in both Sender and Receiver.

  Bonus (5 points): Create a very large data set – 5 times with TCP Reno in both Sender and Receiver, 5 times with TCP Reno in Sender and TCP Cubic in Receiver, 5 times with TCP Cubic in Sender and TCP Reno in Receiver, and 5 times with TCP Cubic in both Sender and Receiver. This will give you a data set large enough, which allows you to work with, compare with and to reach a definite conclusion.

- For RUDP, you will run it 5 times for each scenario.

- For both TCP and RUDP, you will run them on those scenarios of packet loss:
  - **0% packet loss** – Control group, simulating perfect connection.
  - **2% packet loss** – Simulating unstable connection that commonly occur.
  - **5% packet loss** – Simulating bad connection.
  - **10% packet loss** – Simulating extremely poor connection.

For each scenario, you must provide a filtered PCAP file and relevant screenshots that explain each one of the steps in the process. **(10 points)**

After you've gathered the data set for both TCP and Reliable UDP, answer the following questions:

1) In TCP, which congestion control algorithm gave better results overall? TCP Reno or TCP Cubic? Which one gave better results on high packet loss? Explain how you came to this conclusion based on the data set you've gathered. **(5 points)**

2) How did your implementation of Reliable UDP perform overall compared to regular TCP? Which one is better for high packet loss? **(5 points)**

3) According to the data you've gathered, in which scenarios and applications would we prefer to use TCP and in which situations would we prefer to use reliable UDP? Explain your answer. **(5 points)**

ענו על השאלות הבאות (5 נקודות):

- ב"רשת אמינה" הכוונה היא לרשת שבה מעט מאוד חבילות הולכות לאיבוד. ב"קשר ארוך" הכוונה היא לקשר TCP שיש בו הרבה מאוד מידע לשלוח. מוצע להגדיל ב-TCP את SSThreshold בתחילת הקשר. באיזה מהמקרים הבאים השינוי הזה עשוי להועיל במידה המירבית? נמק.
  1. בקשר ארוך על גבי רשת אמינה עם RTT גדול.
  2. בקשר קצר על גבי רשת לא אמינה עם RTT גדול.
  3. בקשר ארוך על גבי רשת לא אמינה עם RTT גדול.
  4. בקשר קצר על גבי רשת אמינה עם RTT קטן.
  5. בקשר ארוך על גבי רשת אמינה עם RTT קטן.
  6. בקשר קצר על גבי רשת לא אמינה עם RTT קטן.
  7. בקשר ארוך על גבי רשת לא אמינה עם RTT קטן.
  8. בקשר קצר על גבי רשת אמינה עם RTT גדול.

- A שולח ל-B חבילות מידע באמצעות קשר TCP. נתון כי הקשר מתחיל עם חלון בגודל $1 \cdot MSS$. הקשר מסתיים בפעם הראשונה שהוא מגיע ל-SSthresh. במהלך הקשר לא אובדות חבילות. נסמן בקיצור: SSthresh=S*MSS.
נתון כי לאורך כל הקשר rwnd < S*MSS.
מהי התפוקה של הקשר מ-A ל-B (ב-Bytes / sec) בזמן הנ"ל?
  א. בערך $2S\frac{MSS}{\lg S \cdot RTT}$
  ב. בערך $2S\frac{MSS}{\sqrt{S} \cdot RTT}$
  ג. בערך $S\frac{MSS}{RTT}$
  ד. בערך $S^2 \cdot MSS/RTT$

- שתי תחנות מתקשרות באופן אלחוטי בשיטת Go Back N.

נסמן ב-X את ספרת הביקורת של הת.ז. שלך. אם שני שותפים מגישים את התרגיל יחד, X הוא ספרת הביקורת בעלת הערך הנמוך יותר מבין השתיים.

קצב התקשורת הוא 8Gbps, וגודל כל חבילה הוא X*KByte. גדלי ה-headers וה-Acks זניחים.

קצב ההתפשטות הוא $2 \cdot 10^8\ m/sec$, והמרחק בין התחנות הוא 1Km.
זמן עיבוד הנתונים בתחנות זניח.
אף חבילה ואף Ack לא הולכים לאיבוד.

- מהו X שלך?
- מה צריך להיות גודל חלון המשלוח כדי להבטיח תפוקה מקסימלית?

Bonus – Improving the Reliable UDP – 15 points

<u>Note</u>: This part isn't mandatory and is a bonus only!

In this part, you'll improve your RUDP implementation by adding the concepts of flow control and congestion control from TCP. You're free to choose which algorithms to implement, as long as you explain in your PDF which algorithms you've used. You'll probably need to make changes in the original design of the RUDP. Changes only need to be to the API itself, $\boldsymbol{RUDP.c}$.

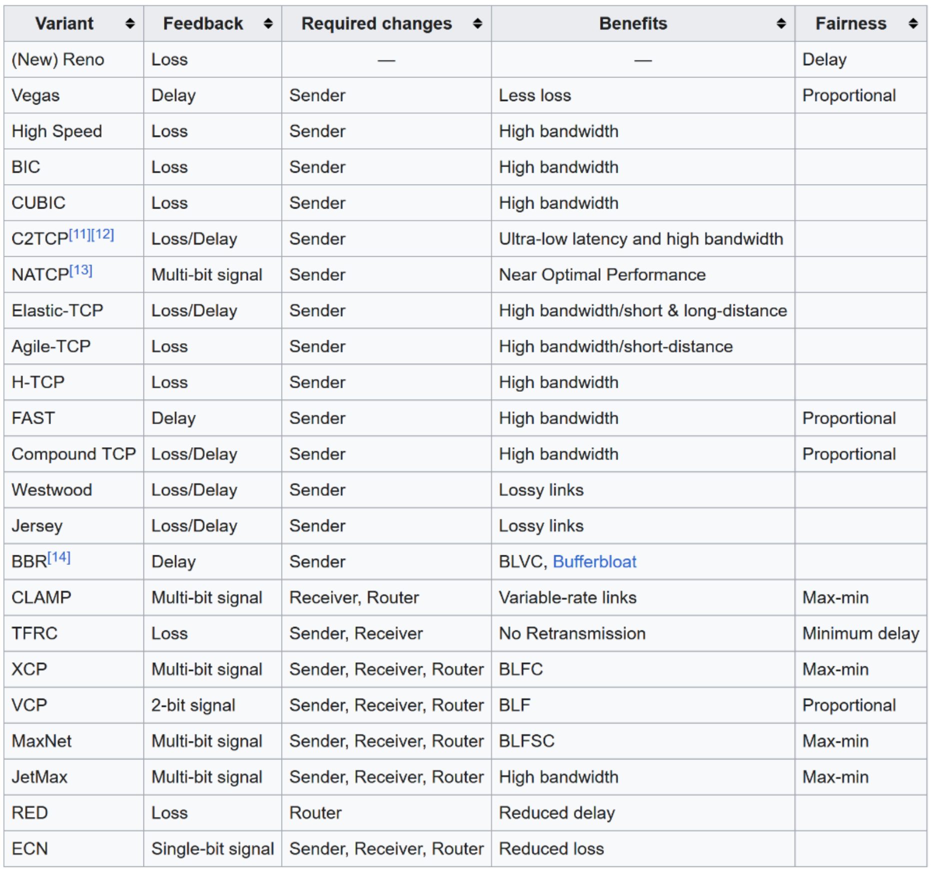
Points distribution:

- **Flow control improvement** – 7 Points
- **Congestion control improvement** – 8 points.

Guidance:

- For flow control, you must use one of the Sliding window techniques: **stop-and-wait ARQ**, **Go-Back-N** or **Selective repeat**.

- List of known TCP congestion control algorithms, which you can use any one of them for this bonus task, or makeup your own:



Variant	Feedback	Required changes	Benefits	Fairness
(New) Reno	Loss	—	—	Delay
Vegas	Delay	Sender	Less loss	Proportional
High Speed	Loss	Sender	High bandwidth	
BIC	Loss	Sender	High bandwidth	
CUBIC	Loss	Sender	High bandwidth	
C2TCP[11][12]	Loss/Delay	Sender	Ultra-low latency and high bandwidth	
NATCP[13]	Multi-bit signal	Sender	Near Optimal Performance	
Elastic-TCP	Loss/Delay	Sender	High bandwidth/short & long-distance	
Agile-TCP	Loss	Sender	High bandwidth/short-distance	
H-TCP	Loss	Sender	High bandwidth	
FAST	Delay	Sender	High bandwidth	Proportional
Compound TCP	Loss/Delay	Sender	High bandwidth	Proportional
Westwood	Loss/Delay	Sender	Lossy links	
Jersey	Loss/Delay	Sender	Lossy links	
BBR[14]	Delay	Sender	BLVC, Bufferbloat	
CLAMP	Multi-bit signal	Receiver, Router	Variable-rate links	Max-min
TFRC	Loss	Sender, Receiver	No Retransmission	Minimum delay
XCP	Multi-bit signal	Sender, Receiver, Router	BLFC	Max-min
VCP	2-bit signal	Sender, Receiver, Router	BLF	Proportional
MaxNet	Multi-bit signal	Sender, Receiver, Router	BLFSC	Max-min
JetMax	Multi-bit signal	Sender, Receiver, Router	High bandwidth	Max-min
RED	Loss	Router	Reduced delay	
ECN	Single-bit signal	Sender, Receiver, Router	Reduced loss	

Figure 2 – List of known TCP congestion control algorithms, Wikipedia

Appendix A – Linux local packet loss tool

To simulate a packet loss in a local network, we'll use a Linux tool that's called **Linux Traffic Control** (tc). This requires a full Linux installation, **so working with WSL won't work**.

If the tool isn't installed in your machine, you can install it by using the following commands:

```
sudo apt update
sudo apt install iproute
```

If the installation didn't work, you can use the following command:

```
sudo apt install iproute2
```

To create a packet loss scenario, type the following command:

```
sudo tc qdisc add dev lo root netem loss XX%
```

Where XX represents the wanted packet loss percentage.

After creating a packet loss scenario, you can change the percentage by typing the following command:

```
sudo tc qdisc change dev lo root netem loss XX%
```

Where XX represents the wanted packet loss percentage.

To remove the packet loss scenario, write the following command:

```
sudo tc qdisc del dev lo root netem
```

Appendix B – $man(2)$ system calls

This appendix's purpose is to give you a better understanding of socket handling in C.

A recommended reference is **Beej's Guide to Network Programming**, by Brian "Beej Jorgensen" Hall, which gives a lot of information about sockets in C and functions.

Functions and documentation links:

- $\boldsymbol{socket(2)}$ – A Factory method system call that creates a socket with the given parameters.

- $\boldsymbol{bind(2)}$ – Whenever we create a new socket, we need to bind it to a specific port in our machine, "to catch a port". The port number is managed and used by the kernel to link incoming packets to the corresponding application socket. The port numbers are usually fixed. The $addr$ parameter must match the current socket that is used.

- $\boldsymbol{listen(2)}$ – The $listen(2)$ system call doesn't do anything in the mean of communication but only memory allocation (malloc) for stream sockets only. If we won't allocate memory, any incoming connection will fail.

- $\boldsymbol{connect(2)}$ – The $connect(2)$ system call starts the link between a client and a specific IP address and port number. The connection would complete when the other side will call the $accept(2)$ system call.

- $\boldsymbol{accept(2)}$ – The $accept(2)$ system call accepts the incoming stream connection request from the other side, by the listening socket. This system call completes the three-way handshake of stream socket (like TCP) and opens a new socket that has a direct communication channel only with the other party in the handshake.

- $\boldsymbol{send(2)}$ and $\boldsymbol{sendto(2)}$ – The $send(2)$ and $sendto(2)$ system calls are an analogy for the $write(2)$ system call, specifcly for sockets, with flags set to 0. Those system calls return the number of bytes that were sent through the socket itself, as the number of bytes can vary. The $send(2)$system call designed for stream sockets (like TCP), while the $sendto(2)$ system call designed for datagram connection-less sockets (like UDP). When the $send(2)$ system call returns 0, it means that the other party closed the connection.

- $\boldsymbol{recv(2)}$ and $\boldsymbol{recvfrom(2)}$ – The $recv(2)$ and $recvfrom(2)$ system calls are an analogy for the $read(2)$ system call, specifically for sockets, with flags set to 0. Those system calls return the number of bytes that were received through the socket itself, as the number of bytes can vary. The $recv(2)$ system call designed for stream sockets (like TCP), while the $recvfrom(2)$ system call designed for datagram connection-less sockets (like UDP). When the $recv(2)$ system call returns 0, it means that the other party closed the connection.

- ***close*(2)** and ***shutdown*(2)** – The $close(2)$ and $shutdown(2)$ system calls close the socket. While $close(2)$ system call closes the whole socket, the $shutdown(2)$ system call partially closes the socket for read/write.

Appendix C – Random data generator

The data that you are going to send in this assignment is randomly generated. You can use your own file with appropriate size or use the following function (or a function of your own) to generate the data:

```
/*
 * @brief        A random data generator function based on srand() and rand().
 * @param size    The size of the data to generate (up to 2^32 bytes).
 * @return        A pointer to the buffer.
*/
char *util_generate_random_data(unsigned int size) {
    char *buffer = NULL;

    // Argument check.
    if (size == 0)
        return NULL;

    buffer = (char *)calloc(size, sizeof(char));

    // Error checking.
    if (buffer == NULL)
        return NULL;

    // Randomize the seed of the random number generator.
    srand(time(NULL));

    for (unsigned int i = 0; i < size; i++)
        *(buffer + i) = ((unsigned int)rand() % 256);

    return buffer;
}
```

Appendix D – Checksum function

A checksum is a small-sized block of data derived from another block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage. By themselves, checksums are often used to verify data integrity but are not relied upon to verify data authenticity.

Below is a simple yet efficient implementation of the checksum function, you can use it for the assignment, but you can implement your own checksum function.

```c
/*
 * @brief    A checksum function that returns 16 bit checksum for data.
 * @param data    The data to do the checksum for.
 * @param bytes    The length of the data in bytes.
 * @return    The checksum itself as 16 bit unsigned number.
 * @note    This function is taken from RFC1071, can be found here:
 * @note    https://tools.ietf.org/html/rfc1071
 * @note    It is the simplest way to calculate a checksum and is not very strong.
 *        However, it is good enough for this assignment.
 * @note    You are free to use any other checksum function as well.
 *        You can also use this function as such without any change.
*/
unsigned short int calculate_checksum(void *data, unsigned int bytes) {
    unsigned short int *data_pointer = (unsigned short int *)data;
    unsigned int total_sum = 0;

    // Main summing loop
    while (bytes > 1) {
        total_sum += *data_pointer++;
        bytes -= 2;
    }

    // Add left-over byte, if any
    if (bytes > 0)
        total_sum += *((unsigned char *)data_pointer);

    // Fold 32-bit sum to 16 bits
    while (total_sum >> 16)
        total_sum = (total_sum & 0xFFFF) + (total_sum >> 16);

    return (~((unsigned short int)total_sum));
}
```