

# INTERNET AND WEB ENGINEERING PROJECT

## TCP RELIABLE FILE TRANSFER OVER UDP

Diana Pasquali  
0240956

Livia Simoncini  
0255092

### System architecture and design choices

The goal of the system is to implement a reliable data transmission between a client and a server, based on the unreliable UDP protocol. To add reliability some basic functions of the TCP protocol, such as the use of *Acknowledgements* and retransmissions, were implemented in the application.

- **Software Platform**

The system was developed on Unix-like systems using Sublime Text and executed using the terminal. No special application is requested to correctly execute the program, except for the necessary use of *mate terminal* to print a progress bar during transmission.

- **Implementation**

The system is separated in a client part and a server part, both stored in different directories with their own files, to simulate to different hosts.

The server supports three different requests:

- LIST, which provides a list of the files the server can transmit to the client.
- GET, which permits the download of a certain file from the server to the client.
- PUT, which permits the upload of a new file from the client to the server.

Files are transmitted using sockets set up with the UDP protocol.

To provide a connection between the hosts, before the actual file transmission, a mechanism of 3-way handshake was implemented. After the handshake, the server adds a particular client to the queue of clients in service. One of the working threads of the server will take care of responding to the request of the new client.

Reliability is obtained using packets containing the type of the message, the data transmitted, the sequence number referring to the byte stream and the ack number. Packets transmitted get stored in a transmission window, whose base packet gets associated with a timeout struct to manage retransmissions.

The size of the transmission window was set to 10 packets, while the timeout interval was implemented as adaptive so that whenever an ACK is received for a known packet, the estimation of the RTT permits to recalculate the interval according to the traffic on the channel.

Retransmission also occurs when the sender receives three duplicated ACKs.

An artificial packet loss was also added in the implementation to simulate a real transmission channel. Since the system was developed and tested on one device, the effect of the artificial loss is added to the natural loss due to the speed with which the transmitted packets arrive at their destination.

The loss probability is a fixed value that is used along side the rand function: whenever a packet must be sent, the system checks whether a random value is a multiple of the loss probability and, if so, the packet is not transmitted.

When a transmission terminates, the connection gets closed with the final exchange of messages and the client is removed from the server's queue.

Both the client and the server are implemented with a multithreaded structure. The server spawns ten working threads that will wait for incoming requests. The client spawns a new thread for every concurrent request that it will send to the server. Every thread of the client will use a new socket to make the requests distinguishable on the server side.

### **Server:**

The server spawns ten working threads that will wait for incoming requests. After the handshake protocol that starts the connection between sender and receiver, a client node containing the client's information is added to the clients queue. Whenever a new client is added to incoming queue the main thread of the server signals the active working threads

to handle the incoming request. If one of these threads is not busy, it takes into charge the request and starts executing a different operation, according the request itself.

To perform the LIST operation, one of the server working threads reads the filenames of the files in the server directory, encapsulating them in packets and serializing those packets in buffer to send over the socket.

To perform the GET operation, one of the server working threads receives the filename of the file to transfer to the client and opens it if it exists in the file system, sends a packet containing the file size and then updates the transmission window starting the timer if the packet is the send base of the window. When the client is ready to receive data, the server wraps the file data in packets and serializes them in buffers of 65467 bytes and then sends data over the socket. At the same time it waits for the acks to update the transmission window.

To perform the PUT operation, the server receives first the file size to allocate enough space in memory, then it receives file data from client in chunks of 65000 bytes. If the packet is received in order, the data is written on the file and the `receive_next` index incremented by data size received. The server then checks if there are other buffered data to read in the receiver buffer and if necessary writes previously stored data packets to the target file and reorders the receiver buffer. If the packet is received out of order then it's stored into the receiver buffer until the packet with expected sequence number is received.

### **Client:**

When the user selects an operation to perform with the system, the client spawns a new thread that handles the request and starts the selected operation.

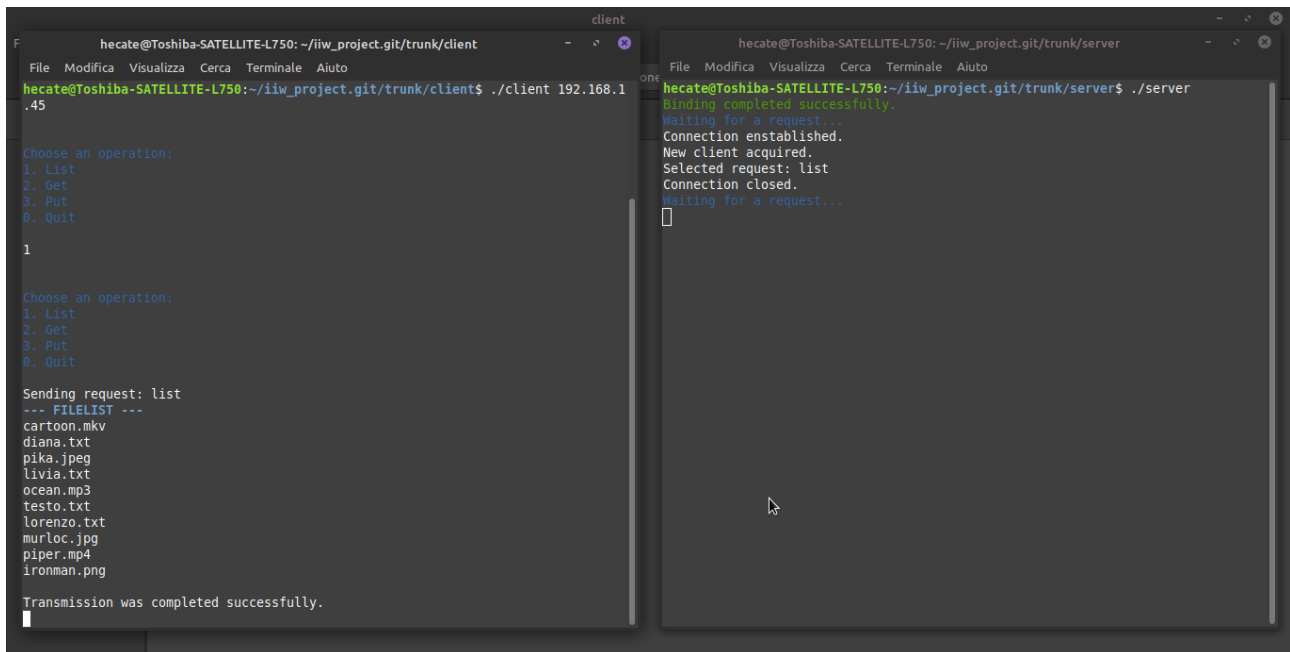
To perform the LIST operation, the client thread continuously receives packets serialized into buffers of 65000 bytes. If the packet received is in order, the client prints on screen the name of the file and checks if in the receiver buffer there are other buffered packets that still need to be read. If the packet is received out of order, the client stores it into the receiver buffer. In both cases, the client sends an ack for the last correctly received packet.

To perform the GET operation, the client receives from the server first the file size and then the file data to write in the previously open file. The client receives the file data in chunks of

65000 bytes, obtained serializing the packets in buffers. If the packet received is in order, then the client writes the received data to the file and checks if there are other subsequent packets stored in the receiver buffer. Else, if the packet is received out of order it's stored into the receiver buffer. In any case, after receiving the packet the client sends to the server an ack for the last correctly received packet (where the sequence number of the packet matches the sequence number expected).

To perform the PUT operation, the client thread spawns another thread to receive the acks, so that the working thread only transmits data to the server. The transmitting thread reads the selected file, sends data to the server wrapping it into a packet and serializing the packet into a buffer and updates the transmission window. A progress bar is showed on screen and the transmission window send base is incremented if the ack receiver thread receives the ack for the first packet in the window.

## Examples



The image shows two terminal windows side-by-side. The left window is titled 'client' and the right window is titled 'server'. Both are running on a machine named 'hecate@Toshiba-SATELLITE-L750'.

**Client Terminal:**

```
hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/client
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/client$ ./client 192.168.1.45

Choose an operation:
1. List
2. Get
3. Put
0. Quit

1

Choose an operation:
1. List
2. Get
3. Put
0. Quit

Sending request: list
--- FILELIST ---
cartoon.mkv
diana.txt
pika.jpeg
livia.txt
ocean.mp3
testo.txt
lorenzo.txt
murluc.jpg
piper.mp4
ironman.png

Transmission was completed successfully.
```

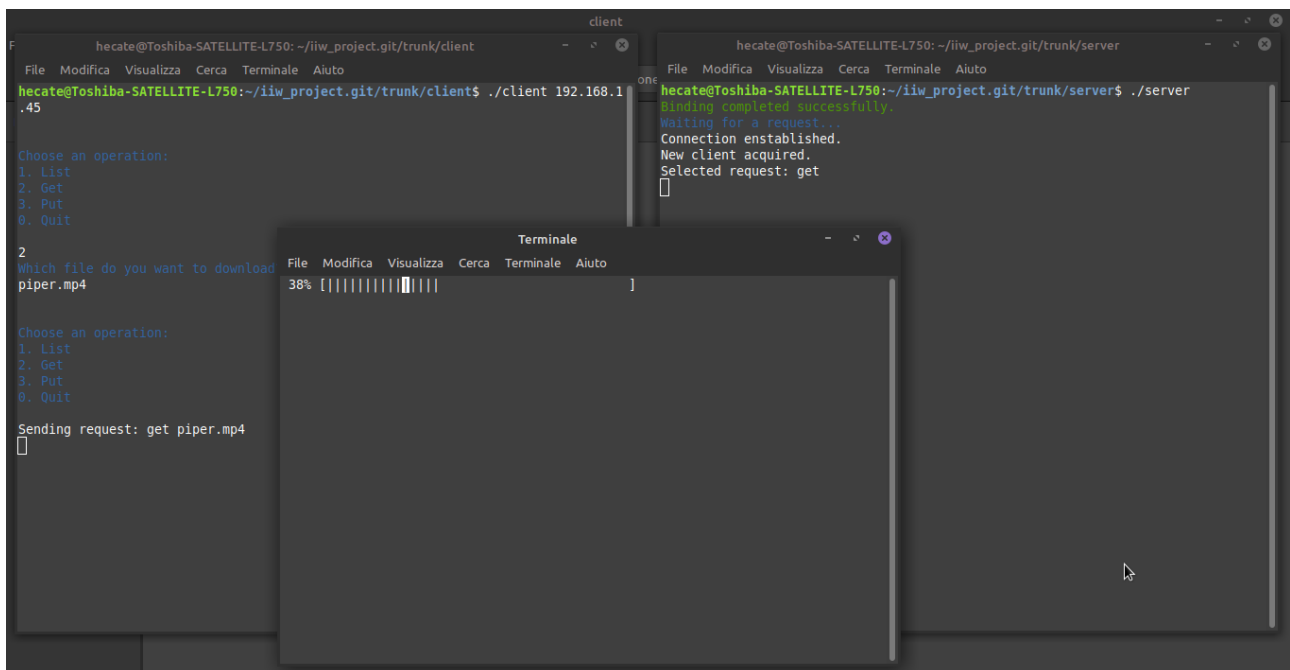
**Server Terminal:**

```
hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/server
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/server$ ./server

Binding completed successfully.
Waiting for a request...
Connection established.
New client acquired.
Selected request: list
Connection closed.
Waiting for a request...

```

### List



The image shows three terminal windows. The left window is the 'client' terminal, the right window is the 'server' terminal, and a new 'Terminale' window is in the foreground showing a progress bar.

**Client Terminal:**

```
hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/client
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/client$ ./client 192.168.1.45

Choose an operation:
1. List
2. Get
3. Put
0. Quit

2
Which file do you want to download
piper.mp4

Choose an operation:
1. List
2. Get
3. Put
0. Quit

Sending request: get piper.mp4

```

**Server Terminal:**

```
hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/server
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/server$ ./server

Binding completed successfully.
Waiting for a request...
Connection established.
New client acquired.
Selected request: get

```

**Terminale (Progress Bar):**

```
38% [||||||| ]
```

### Get

```
client
hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/client
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/client$ ./client 192.168.1.45
Choose an operation:
1. List
2. Get
3. Put
0. Quit
3
Which file do you want to upload?
got.mp3
Choose an operation:
1. List
2. Get
3. Put
0. Quit
Sending request: put got.mp3
Done uploading got.mp3
█

server
hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/server
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/server$ ./server
Binding completed successfully.
Waiting for a request...
Connection established.
New client acquired.
Selected request: put
Connection closed.
Waiting for a request...
█

Terminale
File Modifica Visualizza Cerca Terminale Aiuto
38% [||||| ]
```

*Put*

```
Terminale
File Modifica Visualizza Cerca Terminale Aiuto
38% [||||| ]

hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/server
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/server$ ./server
Binding completed successfully.
Waiting for a request...
Connection established.
New client acquired.
Selected request: get
Connection established.
New client acquired.
Selected request: put
Connection closed.
Waiting for a request...
█

hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/client
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/client$ ./client 192.168.1.45
Choose an operation:
1. List
2. Get
3. Put
0. Quit
2
Which file do you want to download?
piper.mp4
Choose an operation:
1. List
2. Get
3. Put
0. Quit
Sending request: get piper.mp4
3
Which file do you want to upload?
cartoon.mkv
Choose an operation:
1. List
2. Get
3. Put
0. Quit
Sending request: put cartoon.mkv
Done uploading cartoon.mkv
█
```

*Concurrent requests (Put and Get)*

```
hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/client
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/client$ ./client 192.168.1.45

Choose an operation:
1. List
2. Get
3. Put
0. Quit

2
Which file do you want to download?
piper

Choose an operation:
1. List
2. Get
3. Put
0. Quit

Sending request: get piper
An error was encountered:
    The file does not exist.

hecate@Toshiba-SATELLITE-L750: ~/iiw_project.git/trunk/server
hecate@Toshiba-SATELLITE-L750:~/iiw_project.git/trunk/server$ ./server

Binding completed successfully.
Waiting for a request.
Connection established.
New client acquired.
Selected request: get
The file does not exist.
Connection closed.
Waiting for a request...
█
```

## Error handling

## Performance evaluation

The system uses three crucial parameters:

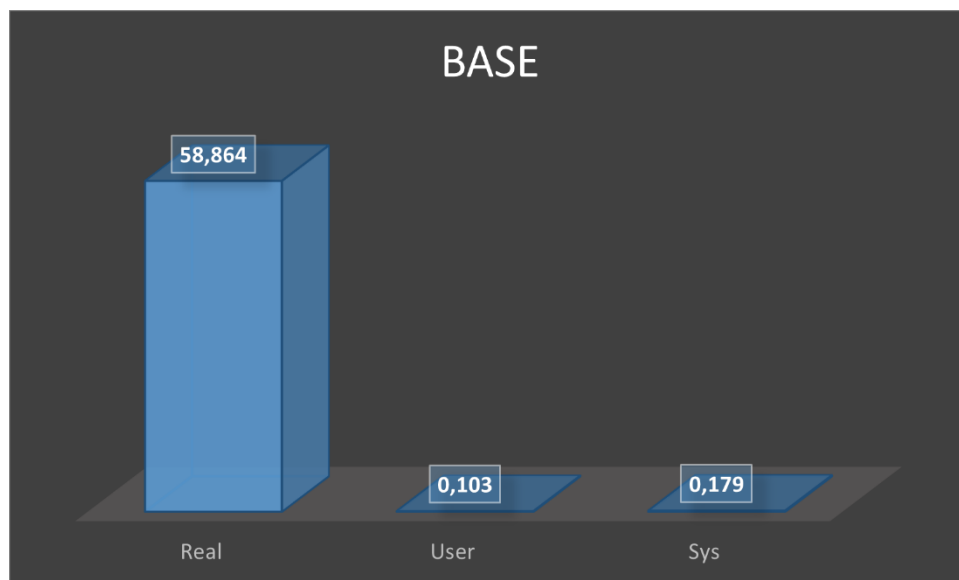
- Size of the transmission window ( $N$ ), which is used to store the packets “in flight”.
- Loss probability ( $P$ ), which is used to simulate the transmission over a lossy channel.
- Timeout ( $T$ ), which is used to implement the retransmission mechanism.

While  $N$  and  $P$  are fixed values,  $T$  was implemented as adaptive along with a periodic RTT estimation.

As these parameters vary, the performances of the system will vary with them.

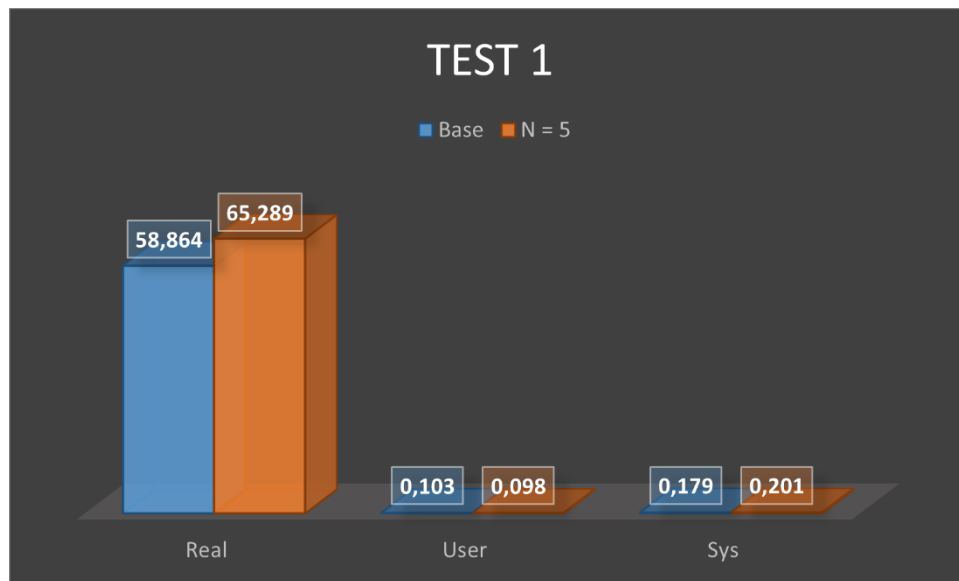
To test this variation, different situations were considered. The function used to test the system is GET, which resulted in being the slowest one.

- **Base:** values used normally ( $N = 10$ ,  $P = 50$ ,  $T$  adaptive)

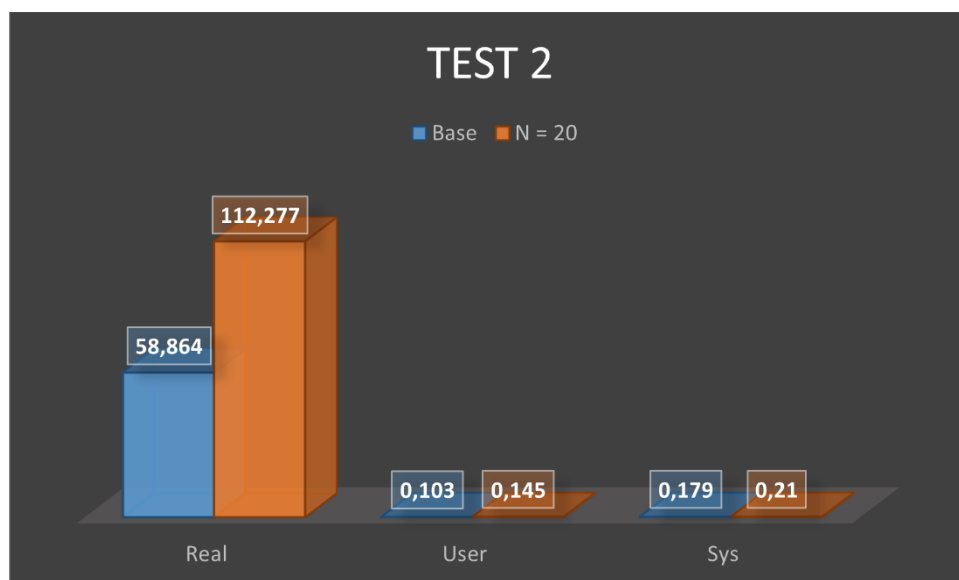




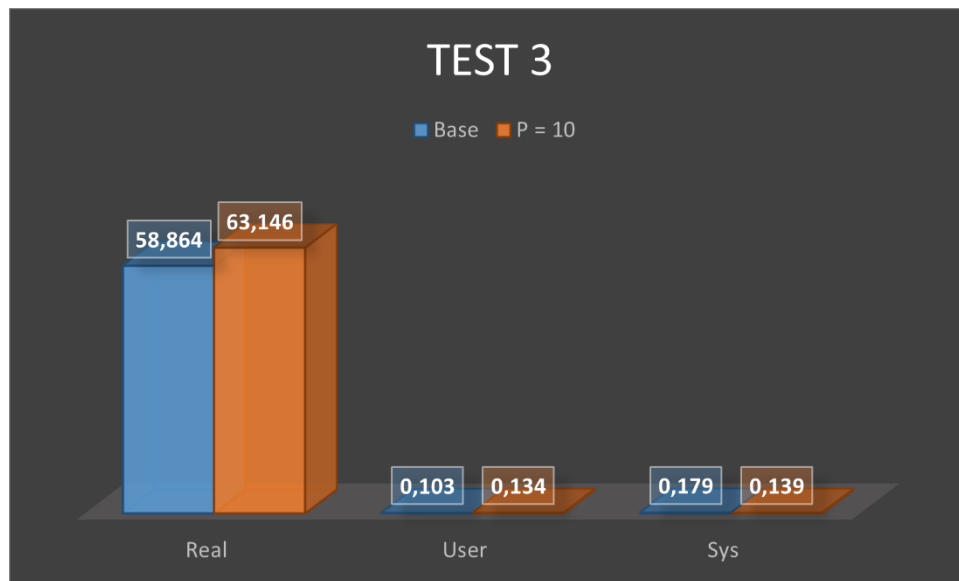
- **Test 1:** lower dimension of the transmission window ( $N = 5$ ,  $P = 50$ ,  $T$  adaptive)



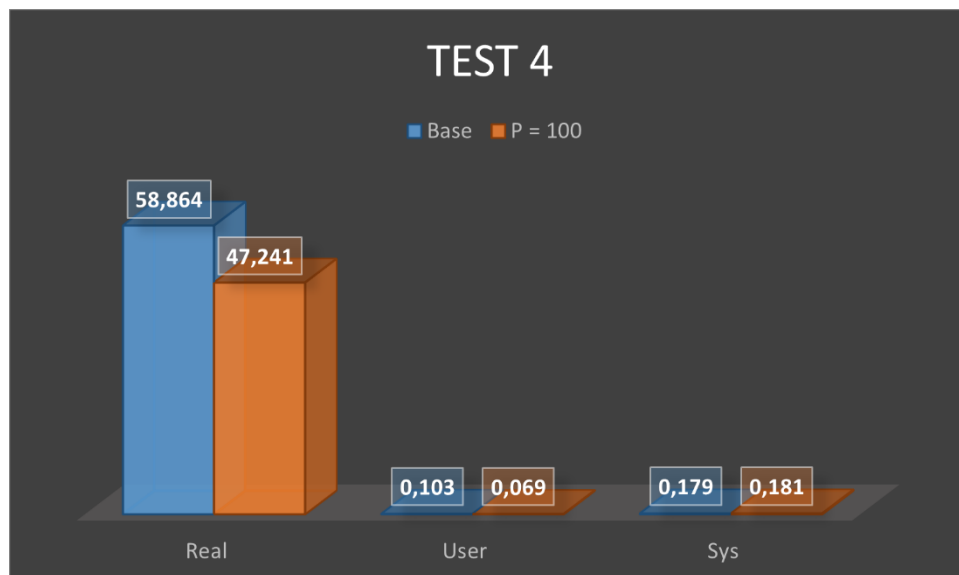
- **Test 2:** higher dimension of the transmission window ( $N = 20$ ,  $P = 50$ ,  $T$  adaptive)



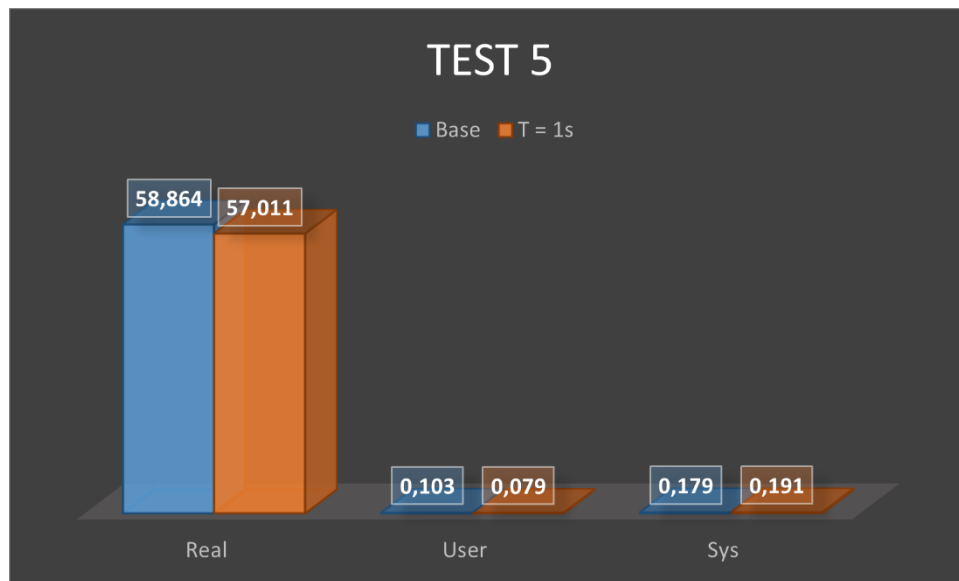
- **Test 3:** higher probability of packet loss ( $N = 10$ ,  $P = 10$ ,  $T$  adaptive)



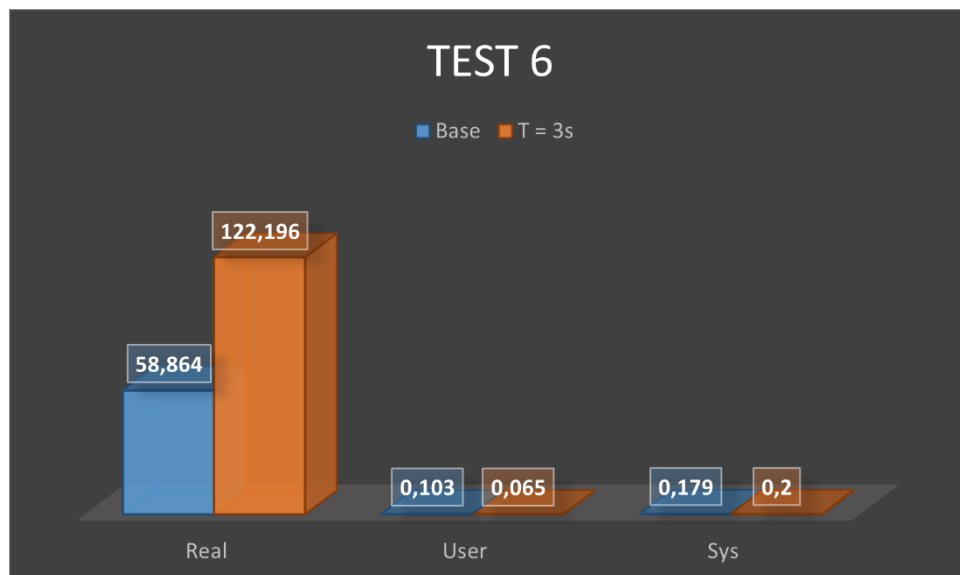
- **Test 4:** lower probability of packet loss ( $N = 10$ ,  $P = 100$ ,  $T$  adaptive)



- **Test 5:** fixed timeout interval of 1 second ( $N = 10$ ,  $P = 50$ ,  $T = 1s$ )



- **Test 6:** fixed timeout interval of 3 seconds ( $N = 10$ ,  $P = 50$ ,  $T = 3s$ )



## Instructions

The source code is divided into two folders. To run the program, two different shells must be opened, one in the client directory and the other in the server directory. Both have their own Makefile to compile every module.

The server must be the first to be executed, using the command `./server` after compiling with `make`. After this, the client will be executed in the other shell with `./client <local IP-address>`.

The client will choose which service to request using an integer from 1 to 3 (0 to exit the program) and the server will respond consequently. When requesting the upload or download of the file, the name of the file must be provided, including its extension.

To avoid having errors caused by the management of the progress bar used during the PUT and GET requests, *mate-terminal* must be used on the device that executes the program.