Networks I Project – Group 20

A Multi-Client-Synchronised, Reliable Data Transfer Protocol over UDP

Marcelina Gancewska
0232501102

Damien Bardina
0200061345

Bachelor in Applied Information Technology

University of Luxembourg

Prof. Radu State

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

Dr Stefan Hommes

R&D Department

ZF Friedrichshafen AG

The Architectural Design

The aim of this project consists in implementing a unidirectional file transmission service based on the application-layer *client-server* paradigm (server-to-client file transmission, the file being stored on the server), using the services of a custom, bidirectionally communicating transport-layer protocol based on the User Datagram Protocol (UDP).

Given that the underlying network-layer infrastructure is simulated to be *unreliable* via a tuneable communication failure probability, the custom transport-layer protocol implemented on top of the unreliable UDP protocol shall allow for a *reliable* communication service between the distributed entities (between a server and its max. 10 served client processes respectively). To do so, this custom protocol will have to address the following considered network flaws via reliability-enhancing mechanisms on top of UDP:

(1) *Transmitted data corruption* (packet data modification during packet transmission)
(2) *Data loss* (packet loss in unreliable network channel)
(3) *Data re-ordering* (sending sequence of packets is not preserved within network channel)

In any of the above cases, the acceptance of transmitted data will be communicated by the receiving side (downloading clients) to the sending side (file server) via the sending of an *acknowledgment message* (ACK), which is triggered if all of the following conditions are met:

(1) Data integrity, meaning that transmitted data was not corrupted by the underlying network channel, is verified via a *checksum*. There are several types of checksum, but we decided to use the *16-bit checksum* here, also known as the *Internet checksum*, as it is used for instance in the UDP header. The implemented computation of this checksum type and the data integrity validation procedure shall therefore be quickly explained.
In the project implementation, the message format (for file transmission or acknowledgment messages) is the following:

<checksum>:<sequence_number>:<payload_data>

where *checksum* (explained here), *sequence_number* (see explanation (3)) complement the IPv4 (containing the sender and receiver IPv4 addresses) and UDP (containing the sender and receiver ports, and optionally also a checksum) *headers* and *payload_data* (data in byte format) represents the segment *payload* (application-relevant file data or feedback message from the receiving side like "ACK").

Before transmission of a message, the data from <sequence_number> and <payload_data> is converted into a base 2 number (binary) respectively and concatenated to form a single (big) binary number. This number is then divided into chunks of 16-bit, which are subsequently added up via binary addition to give a cumulative sum (typically done in a 32-bit register to allow overflow). The possible leading carry bit of the sum is added back to the rest of the result ("carry wrap-around" to avoid overflow of the 16-bit checksum field) and 1's complement (exchanging 0 by 1 and vice versa) is taken – this the *sender-computed* checksum which is then put in the message as indicated above

Upon successful receipt of a message, the receiving side of the custom protocol repeats the procedure (excluding the transmitted checksum field) to compute the *receiver-computed* checksum. Finally, if the sender-computed and receiver-computed checksums are identical, the data is considered to not have been corrupted during transmission. If they differ, data is considered to have been modified and will thus be discarded, not being delivered to the application. (Remark: Strictly speaking, this very simple validation procedure is not "bullet-proof". For example, two bit-flip errors could annihilate their effect mutually and hence still yield the same checksum. Another example would be a bit-flip error in the checksum itself although the data has not been modified, which would result in not sending an ACK back to the sender although the data was not corrupted. For our purposes however, this shall suffice.)

Now several key aspects of the architecture shall be discussed in detail:

(1) Bootstrapping

Understood as taking measures to ensure that sending or receiving starts only when all processes have joined and the IP addresses are known to the other processes. It has been implemented in the following manner. Before sending any file the server waits for a message from all the clients that they are joining. Only after getting a message from all, can the sending proceed, moreover when sending the packets, the IP address of the clients is known (in the server file denoted as registered_clients_addr). A similar procedure is taken when receiving files since no client will receive the file before all have joined.

(2) Loss and protocol simulation

Loss has been simulated using the unreliable network file. The is_sent helper function simulates a Bernoulli trial with a specified failure probability. Randomness is simulated using the random module for deterministic, pseudo-random number generation of a uniform distribution between 0 and 1. Should the value lie in the failure range (smaller or equal to failure probability), return false indicating no loss simulation will occur. Analogically should the function return true the loss is simulated. The function is called once the packets are transferred to the clients (to simulate packet loss while transmitting the file).

Go back n has been implemented using the following strategy: the server (sender) maintains a sliding window, represented by window_base and window_end. Now iteration occurs during which the server sends packets within the current window to clients (where each packet includes a sequence number, a checksum, and the actual data payload). For each packet sent the server starts a timer with a set timeout. Should timeout occur the packet gets resent. Meanwhile, the server listens for acknowledgements and if received one marks the acknowledged packet as received. Only once all clients acknowledge sequence numbers higher or equal to the current window base the window can advance.

(3) Verification if the file is intact (no corruption occurred)

For the purposes of verification if the file transmitted is still intact checksum method has been implemented. The functions compute the 16-bit checksum of data provided in byte format according to the Internet checksum specification in RFC 1071 (1988). The checksum is used both to verify if the file remains intact as well as whether the acknowledgements sent by the clients got corrupted. The details of this function have already been described in the introduction.

(4) Savings in bandwidth and delivery times

Bandwidth and time-minimizing mechanisms can be implemented through multiple measures. Firstly, the format of the data to be exchanged should be carefully chosen. In the case of the described project 'txt' file has been chosen. Another important step to take would be data compression, which can also increase savings in the bandwidth. Moreover, the use of efficient algorithms and data structures can improve on this aspect.

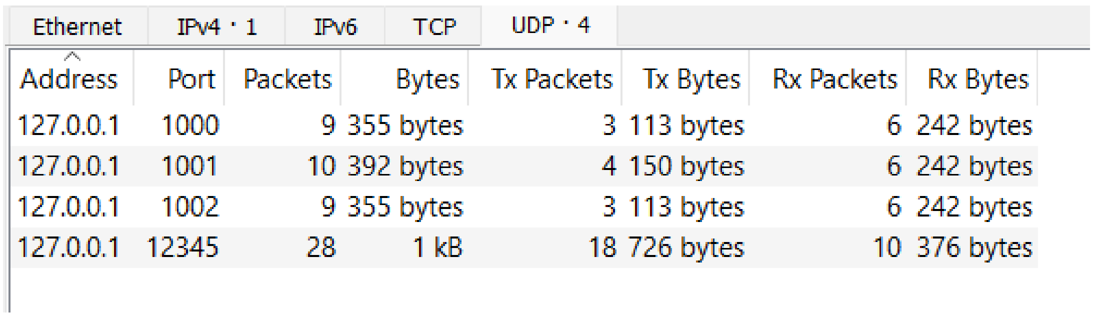
(5) Parameters choice

The choice of timeouts is crucial for proper execution of the code. Should it be too small, many timeouts will occur causing the program not to function correctly, on the other hand, if the chosen value is too big, in the case of an error the program will wait too long before taking any actions. Therefore, it is important to find the right value for each application. In the case of our program, the timeouts have been chosen to be around a few milliseconds for each packet transmission(for example 0.02s). Window size is another parameter which needs to be predetermined. A greater window size may imply greater efficiency however requires a greater buffer as well. Thus, should be appropriately chosen depending on the application. A higher window size would be preferable for bigger files (hence window size 10 for 1GB file) while a smaller window size should be chosen while working with unreliable networks (quicker loss detection and recovery). The probability has been chosen depending on the case examined, while verifying how the protocol works with many errors a higher one has been chosen, and analogically a smaller one for a stable network.

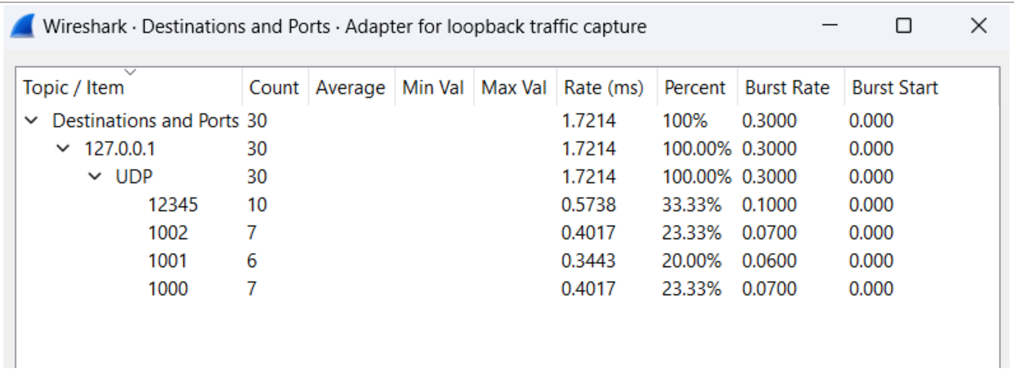
Evaluation

The following screenshots have been taken from Wireshark while testing the code. They present the transmission of two different files of slightly different size.

File 1:



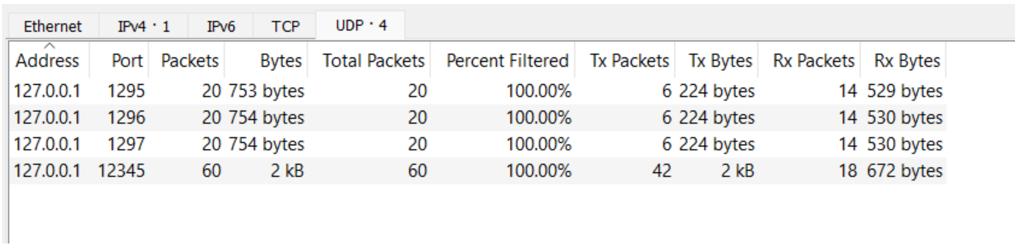
Address	Port	Packets	Bytes	Tx Packets	Tx Bytes	Rx Packets	Rx Bytes
127.0.0.1	1000	9	355 bytes	3	113 bytes	6	242 bytes
127.0.0.1	1001	10	392 bytes	4	150 bytes	6	242 bytes
127.0.0.1	1002	9	355 bytes	3	113 bytes	6	242 bytes
127.0.0.1	12345	28	1 kB	18	726 bytes	10	376 bytes



Wireshark · Destinations and Ports · Adapter for loopback traffic capture

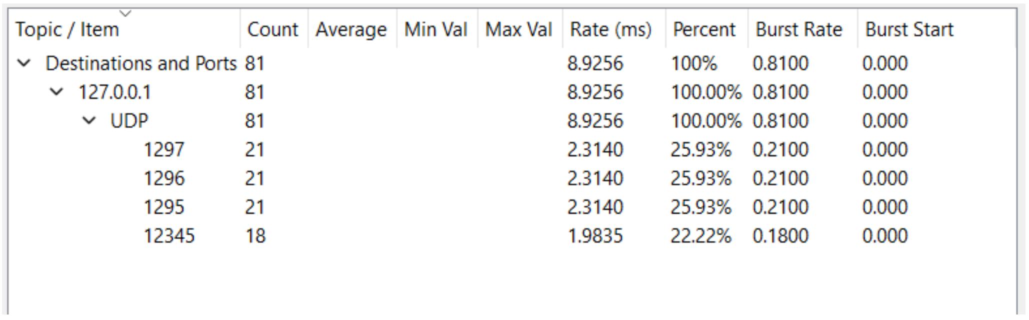
Topic / Item	Count	Average	Min Val	Max Val	Rate (ms)	Percent	Burst Rate	Burst Start
Destinations and Ports	30				1.7214	100%	0.3000	0.000
127.0.0.1	30				1.7214	100.00%	0.3000	0.000
UDP	30				1.7214	100.00%	0.3000	0.000
12345	10				0.5738	33.33%	0.1000	0.000
1002	7				0.4017	23.33%	0.0700	0.000
1001	6				0.3443	20.00%	0.0600	0.000
1000	7				0.4017	23.33%	0.0700	0.000

File 2:



Ethernet | IPv4 · 1 | IPv6 | TCP | UDP · 4

Address	Port	Packets	Bytes	Total Packets	Percent Filtered	Tx Packets	Tx Bytes	Rx Packets	Rx Bytes
127.0.0.1	1295	20	753 bytes	20	100.00%	6	224 bytes	14	529 bytes
127.0.0.1	1296	20	754 bytes	20	100.00%	6	224 bytes	14	530 bytes
127.0.0.1	1297	20	754 bytes	20	100.00%	6	224 bytes	14	530 bytes
127.0.0.1	12345	60	2 kB	60	100.00%	42	2 kB	18	672 bytes



Topic / Item	Count	Average	Min Val	Max Val	Rate (ms)	Percent	Burst Rate	Burst Start
Destinations and Ports	81				8.9256	100%	0.8100	0.000
127.0.0.1	81				8.9256	100.00%	0.8100	0.000
UDP	81				8.9256	100.00%	0.8100	0.000
1297	21				2.3140	25.93%	0.2100	0.000
1296	21				2.3140	25.93%	0.2100	0.000
1295	21				2.3140	25.93%	0.2100	0.000
12345	18				1.9835	22.22%	0.1800	0.000

What the screenshots prove is that indeed only UDP protocol has been used for the transfer. Additional information which can be interpreted from the screenshots is: rate of transfer, port, addresses and number of bytes/ packets transferred.

Sources:

https://peps.python.org/pep-0257/

https://moez-62905.medium.com/the-ultimate-guide-to-command-line-arguments-in-python-scripts-61c49c90e0b3#:~:text=In%20Python%2C%20command%2Dline%20arguments,arguments%20passed%20to%20the%20script.

https://www.theserverside.com/tip/What-does-the-Python-if-name-equals-main-construct-do#:~:text=to%20the%20console.-,The%20if%20__name__%20%3D%3D%20%22__main__%22%3A,it%20would%20not%20execute%20automatically.

https://docs.python.org/3/library/subprocess.html

https://docs.python.org/3/library/functions.html

https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

https://docs.python.org/3/library/socket.html

https://www.datacamp.com/tutorial/a-complete-guide-to-socket-programming-in-python

https://docs.python.org/3/library/random.html

https://docs.python.org/3/library/os.html#os.urandom

https://www.linkedin.com/advice/3/what-best-practices-optimizing-code-reduce-bandwidth-qr5qe?trk=public_post