

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.)

*++Pipelining via Go-Back-N* (both sender and receiver have sliding window each) represents a performance improvement compared to Stop-and-Wait where the sender sends a message, then waits for an ACK before sending the next packet++

++ retransmission is decided based on a timer associated to each sent packet (sequence number): expiration of timeout without receiving ACK signal triggers retransmission, receival of ACK for sequence number within timeout triggers ??? ++

++ Don’t forget packet EXTRACTION INTO FILE at client process ++

out-of-order packets

<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>