CS3102 P2: Practical Report

Reliable Data Transfer Using UDP

University of
St Andrews

190010906

01 April 2022

1 Introduction

Reliable Data Transport (RDT) is a unicast, connection-oriented protocol designed to provide reliable, ordered data transfer. This report will cover the detail, analyse and evaulate the design and implementation of RDT.

2 Design

Given the scope of the practical, simplicity was the main goal when designing the RDT protocol. This section will discuss design decisions and rationale behind them. The two attempted extension features, checksums and adaptive re-transmission timeouts, are also detailed here.

2.1 Packet Structure

RDT packets (see Figure 1) are composed of a constant 12 byte header and an optional data segment. The size of the data segment ranges from 0 to 1300 bytes, with 1300 bytes used as the maximum size so as to allow testing with `slurpe-3` without issue. Given that the TCP Maximum Segment Size (MSS) [2] is generally 1500 bytes, this was considered a reasonable choice. Larger packet sizes may also lead to fragmentation at the IP Layer, which was undesirable.

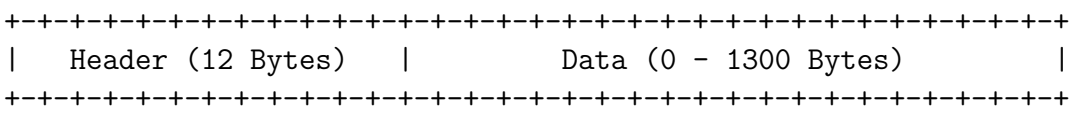

Figure 1: RDT Packet Structure

The RDT header (see Figure 2) is comprised of the following fields: a 32-bit `sequence` field, used for keeping track of the number of bytes sent by the client; a 16-bit `type` field, used to denote the packet function; a 16-bit `checksum` field, calculated over the header and data segment to detect bit-errors; a 16-bit `size` field denoting the size of the data segment (in bytes); and a 16-bit `padding` field to ensure 32-bit word alignment.

Several factors influenced the RDT header design. As file sizes are often calculated as 32-bit integers (such as bythe C library function `ftell`), a 32-bit `sequence` field was required to support the transmission of large files/amounts of data. For checksums, the given implentation of the IPv4 header checksum used returns a `uint16_t` value, thus necessitating a 16-bit field. As a maximum data segment size of 1300 bytes was required, at least 11 bits were required for the `size` field, however 16 bits were used for alignment. For the remaining `type` and `padding` fields, there were no other considerations for field size other than 32-bit alignment.

A single `type` field was chosen, rather than a set TCP-style flags, for simplicity. Given the minimal nature of the RDT protocol, it was faster simpler to enumerate all packet types (see below), rather than testing multiple flags.

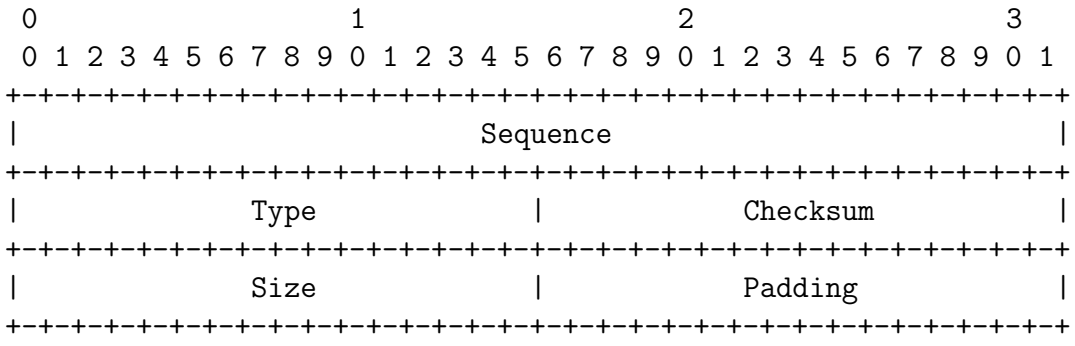

Figure 2: RDT Header

The type field supports the following types: `SYN` (0) and `SYN ACK` (1), used for the connection handshake; `DATA` (2) and `ACK` (3), used for sending and acknowledging data segments; `FIN` (4) and `FIN ACK` (5), used for graceful connection termination; and `RST` (6), used for abrupt connection termination.

2.2 Connection Management

The operation of the RDT protocol can be modelled by the FSM in Figure 3. For connection management, a two-way handshake is used. As RDT only supports uni-directional communication, a two-way handshake (see Figure 4) is adequate for establishing and terminating connections. Adaptive re-transmission timeouts are used in both the handshakes and transmission of data segments.

During handshakes, timeouts are used, with an initial RTO value of 200ms. This is doubled successively (Section 2.4). After 5 timeouts, the client with aborts with an `RST` to prevent clients waiting on unavailable hosts or a dropped `FIN ACK`.

2.3 Data Transfer

RDT uses an Idle-RQ mechanism for sending data segments, and will wait for acknowledgement of each sent segment. If the sender receives an `ACK` with a sequence number that is greater than expected or an RTO is triggered, it will retransmit the same packet. If it receives an `ACK` with a sequence number lower than expected, it will retransmit from that segment as it assumes a data segment has been dropped or corrupted...

For the handshake, a random sequence number is chosen to prevent 'old' or 'stale' packets from causing errors, and to reduce predictability that could be exploited. The method for generating initial sequence numbers is pseudo-random and not cryptograhically secure.

2.4 Adaptive RTO

As an extension, adaptive re-transmission timeouts using measured RTT has been implemented. RDT's adaptive RDT is modelled on TCP's Retranmission Timer [4] [1], with the same initial RTO of 1s and maximum of 60s.

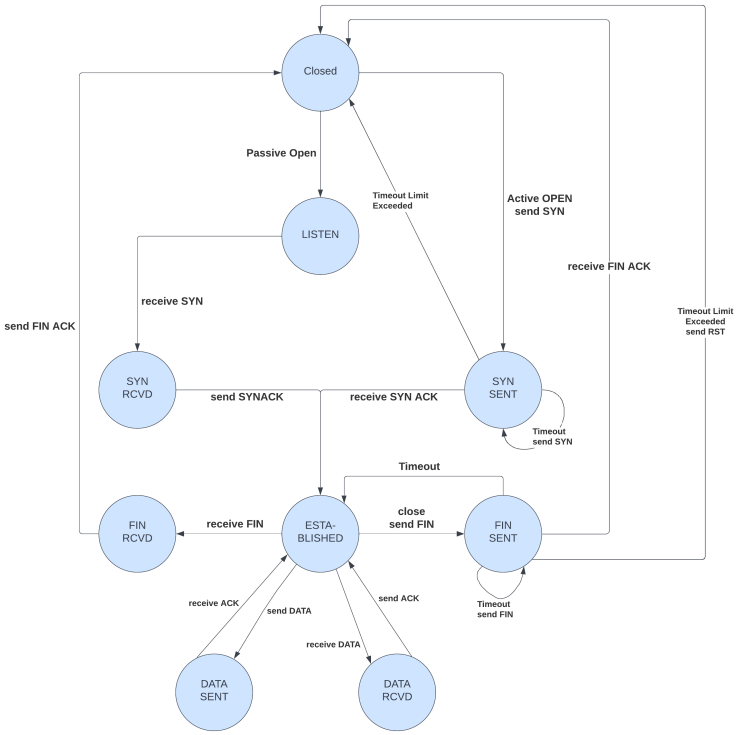

Figure 3: RDT Finite State Machine (see also A.1)

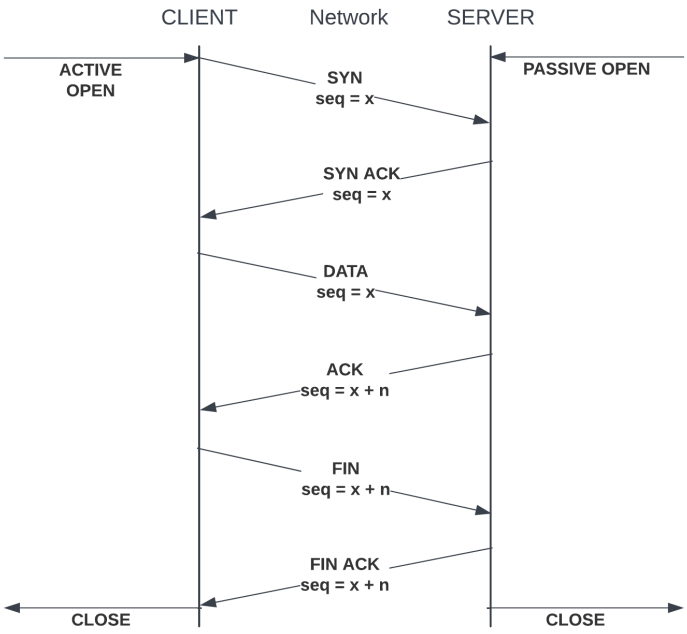

Figure 4: Connection Timeline Diagram (see also A.2)

2.5 Checksums

The RDT `checksum` field is calculated over the entire segment/packet (i.e. header and data) using the IPv4 Header Checksum Algorithm [1]. The implementation of the original source [3] to support the use of 8-bit byte values in the data segment. During checksum calculating, the `checksum` field itself is set to 0 for consistency.

3 Testing

To test and verify the correct operation of RDT, two test programs `RdtClient.c` and `RdtServer.c` were created to send an arbitrary file between two lab PCs using RDT.

3.1 Methodology

To test that RDT packets were delivered reliably and in-order, a 230KB JPEG file (*dog.jpg*[2]) was used as test file. This provided visible feedback as to the integrity of the data received by `RdtServer.c`. This integrity was also verified by calculating SHA1 checksums of both the sent and received files. `slurpe-3` was used to provide an emulated path with loss, delay and restricted data rate to test the reliability of RDT in degraded network conditions.

To collect RTT data for analysing the performance of RDT, two test programs `RdtClientRTT.c` and `RdtServerRTT.c` were created. These programs performed 10 rounds of transferring the same 230 KB file, to calculate a sample of 10 average RTT values. This was performed without slurpe-3 to give a true characterisation of performance. These results are included in `data/rtt-results.csv`.

3.2 Results

From the results in Table 1 and Appendix A.4 we can see the RDT performed as expected in a variety of degraded network conditions.

Scenario	In File	Out File	Match	Time
Control	*dog.jpg*	*dog-control.jpg*	✓	0.002336s
Delay	*dog.jpg*	*dog-d.jpg*	✓	12.037570s
Loss	*dog.jpg*	*dog-l.jpg*	✓	57.010318s
Constrained Rate	*dog.jpg*	*dog-cr.jpg*	✓	0.930083s
Loss and Delay	*dog.jpg*	*dog-ld.jpg*	✓	92.969945
Loss and Constrained Rate	*dog.jpg*	*dog-lcr.jpg*	✓	64.960451s
Delay and Constrained Rate	*dog.jpg*	*dog-dcr.jpg*	✓	13.948315s
Delay, Loss and Constrained Rate	*dog.jpg*	*dog-ldcr.jpg*	✓	64.960451s

Table 1: Results for varying network conditions

4 Analysis

4.1 Performance in Different Network Scenarios

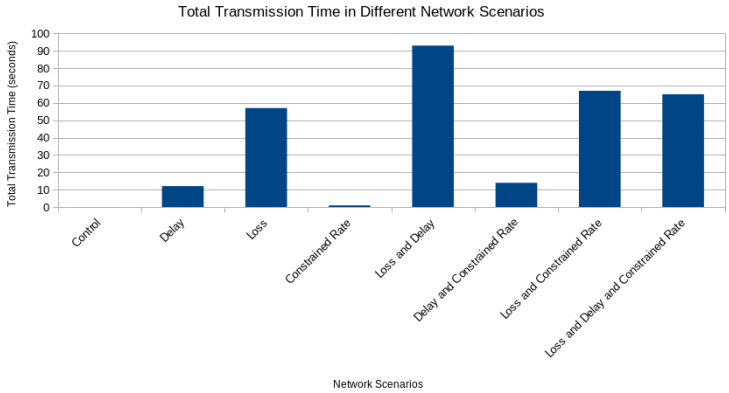

Figure 5: Total Transmission Time in Different Network Scenarios for 230 Kb file.

From Figure 5 we can see that the network characteristic that has the biggest impact on the performance RDT is packet loss.

4.2 Bandwidth Delay Product and Link Utilisation

The Bandwith-Delay Product (BDP) the 'capacity' of a link and is calculated as:

$$BDP = \text{Bandwidth (bits per second)} \times \text{Round Trip Time (seconds)}$$

For the link between `pc7-082-l` and `pc7-085-l` the observed average RTT with RDT was 0.106ms. This therefore yields a BDP of for the link.

As RDT only sends 1300 bytes = 10400 bits in each packet, this means that RDT achieves a link utilisation of $1040/BDP =$.

4.3 Maximum Theoretical Data Rate

4.4 Idle-RQ Performance

$$U_{RDT_IRQ} = \frac{N_p T_x}{T_x + 2T_p}$$

where:

- N_p is the number of packets sent

- T_x is the tranmission delay
- T_p is the propogation delay

4.5 RDT Packet Data Size

4.6 Wastage due to Control Information

$$\rho = \frac{i - c}{i + a} = \frac{1312 - 12}{1312 + 12} = 0.982 \text{ (3 s.f.)}$$

5 Evaluation

5.1 Extension Features

Given that RDT is implemented using UDP and uses the same checksum algorithm used by UDP, it is unlikley that RDT will ever encounter an incorrect checksum. This makes the use of a checksum in RDT mostly redundant. However the inclusion of a checksum provided some utility. Firstly, it helped to catch several errors in the initial implementation of RDT, and secondly it provided a useful. opportunity to understand the function of the Internet Checksum.

The inclusion of adaptive RTO was more useful however. From our analysis in Section 4.1, we concluded that the amount of retransmission required has the biggest impact on RDT performance. Whilst RDT do anything to minimise packet, it is able to control the number of retransmissions due to RTO via the adaptive RTO mechanism. Using the adaptive RTOs, RDT can account for varying amounts of network delay. Therefore we can say that adaptive RTO has had a positive impact on the efficiency and performance RDT in scenarios where network delay is present.

5.2 Further Extension

While the implementation of bi-directional communication and Continuous-RQ was not implemented, it is useful to consider these features in evaluating the desing of RDT.

In it's current design, RDT would be unable to support simulatenous bi-directional communication, due to its use of a two-way handshake. For bi-directional communication to occur, both parties are required to choose and synchronise an 'Initial Sequence Number', which is not possible with only a two-way handshake. Therefore, to support bi-directional communication RDT would require a significant re-design.

However, RDT would not require a fundamental re-design to support Continous-RQ. Continous-RQ with Go-Back-N would solve RDT's fundamental issue of low transmission rate due to link under-utilisation (see **section**).

6 Conclusion

As demonstrated in this report RDT provides reliable, ordered, uni-directional data transfer on top of UDP. RDT provides reliable performance in heavily degraded network environments. However in its current implementation RDT is extrememly inefficient due to its Idle-RQ transmission mechanicsm. RDT performance suffers particularly when there is heavy packet loss that causes retransmission. This efficiency could have been addressed, if given more time, by replacing Idle-RQ with Continuous-RQ with Go-Back-N.

Overall this practical provided an interesting opportunity to design and implement a protocol and analyse protocol performance.

Notes

[1]Clock granularity is not considered, however RTO values are calculated in microseconds and the School Lab PCs have a granularity of 1 nanosecond.

[2]The image was obtained under a royalty-free licence. Credit: Chayathon Wong Getty Images

References

[1] Internet Protocol. RFC 791, September 1981.

[2] Transmission Control Protocol. RFC 793, September 1981.

[3] Saleem Bhatti. Simple IPv4 Checksum Calculation Example, January 2022.

[4] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.

A Appendix

A.1 Finite State Machine

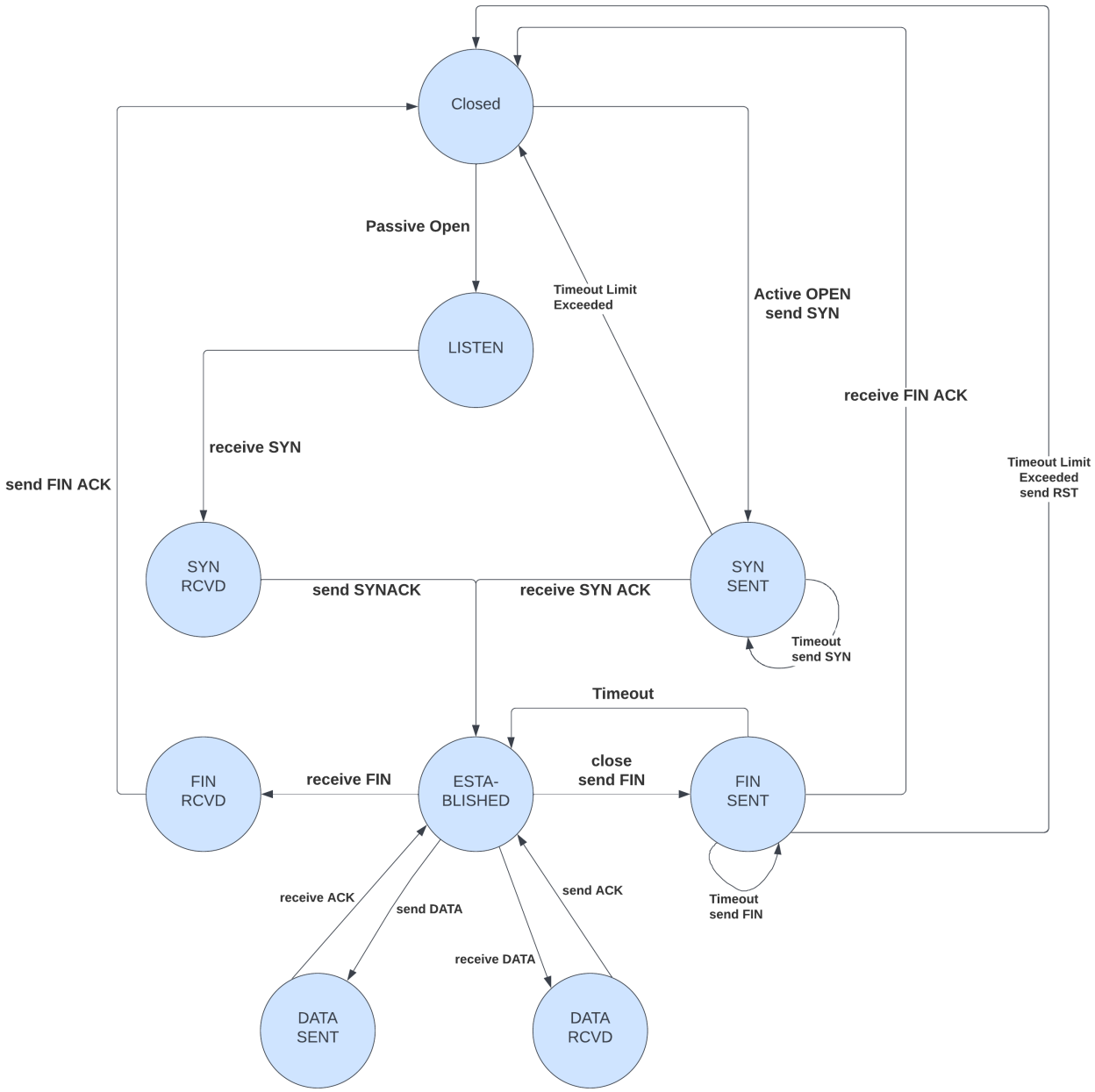

Figure 3: RDT Finite State Machine

A.2 Connection Management Timeline Diagram

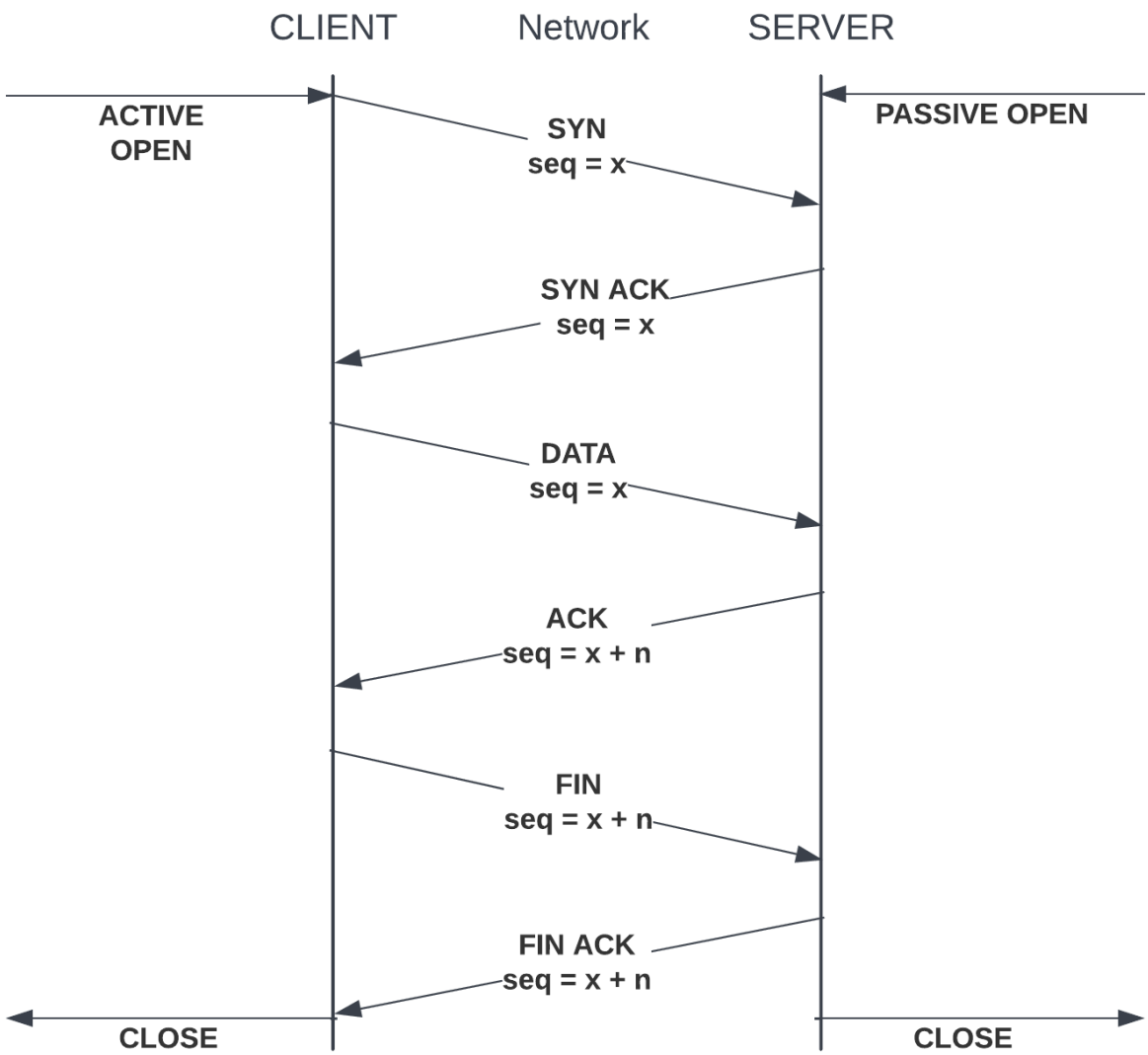

Figure 4: Connection Timeline Diagram

A.3 Performance in Different Network Scenarios

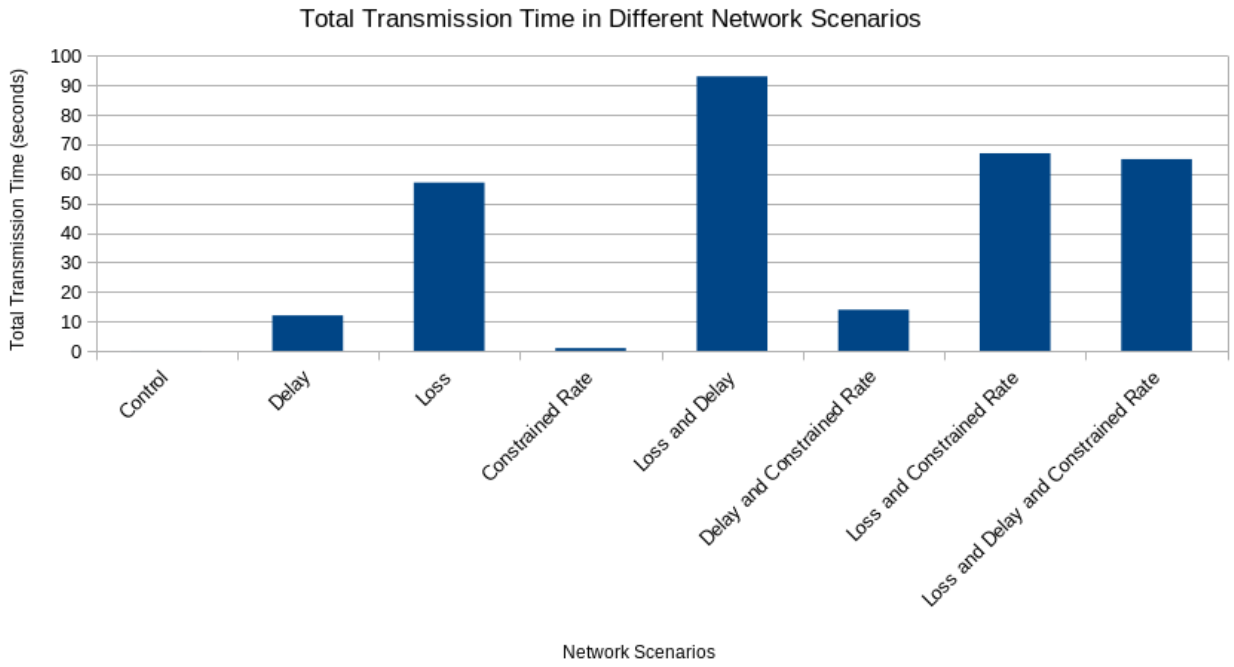

Figure 5: Total Transmission Time in Different Network Scenarios for 230 Kb file.

A.4 Testing Screenshots

Testing screenshots can be found inside the `data/screenshots` directory.