210017213

CS3102 P2: Simple Reliable Transport Protocol

Overview

This practical required designing and implementing a simple, connection-oriented, unicast transport protocol, built on top of UDP. A Simple Reliable Transport Protocol (SRTP) that was required to provide reliable, ordered delivery between a single client and server, under loss conditions.

The initial stage of designing the SRTP was to create a Finite State Machine (FSM) for the connection management of the protocol. Furthermore, another design stage was to design a protocol for establishing and terminating a connection, as well as the reliable transfer which was based on idle-RQ.

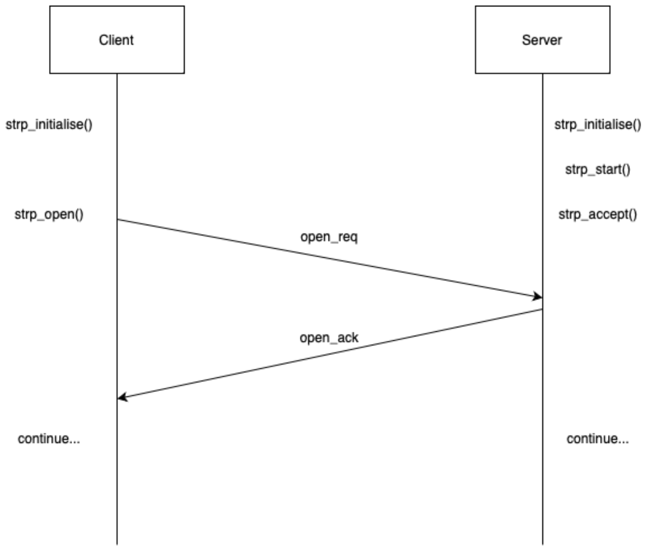
After designing the next stage was to implement the protocol in C using the API structure given and provide evidence of a successful SRTP implementation.

The final requirement was to change from fixed Retransmission Timeout (RTO) to adaptive RTO using Round Trip Time (RTT) measurements.

In my implementation and design which can be seen below I have attempted and completed all requirements. However, looking at my implementation compared to the requirements I have completely forgot to implement the max retries requirement. So please assume that I have not implemented this aspect to my implementation.

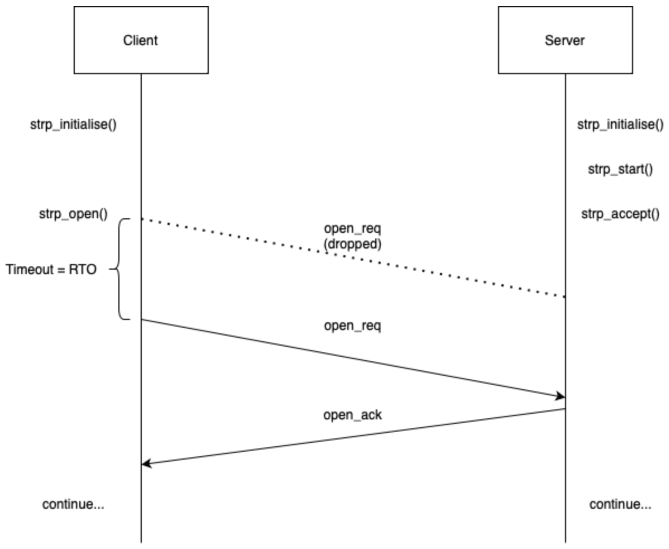
Design

Opening a connection:



To open a connection without any drop is simply achieved by the client sending an open request and receiving an open acknowledgement which can be seen in Figure 1.

Figure 1 Opening connection - no drop



When attempting to open a connection, it can be determined the open request sent by the client has been dropped if an open acknowledgement is not received to the client. To handle this event, I created a timeout on the client side to wait the given RTO value and if an open acknowledgement is not received the open request will be sent again until an open acknowledgement has been received.

Figure 2 Opening connection - with drop

Reliable Data Transfer - both ways design:

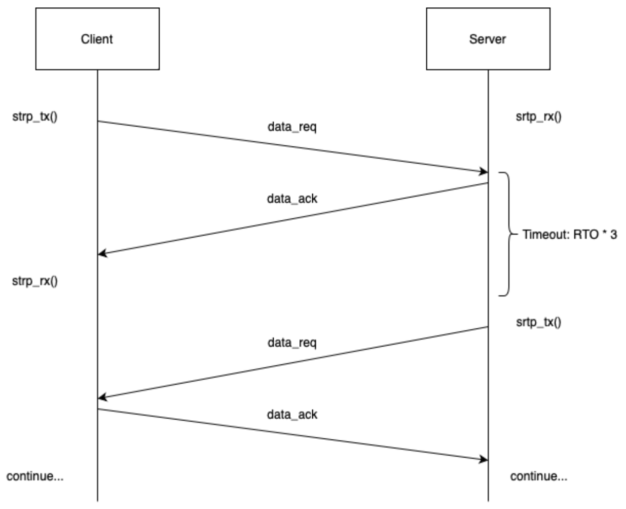


Figure 3 Data transfer both ways - no drop

When sending data both with no drop, as can be seen above after the server has received the data request there is a timeout of RTO * 3. This is to account for the event of the data acknowledgment being dropped, as the data request on the client side would have been retransmitted in this time. This timeout allows the server to be sure the data acknowledgement has been received successfully by the client. So therefore can move into srtp_tx() and continue with the server-side sending and client-side receiving.

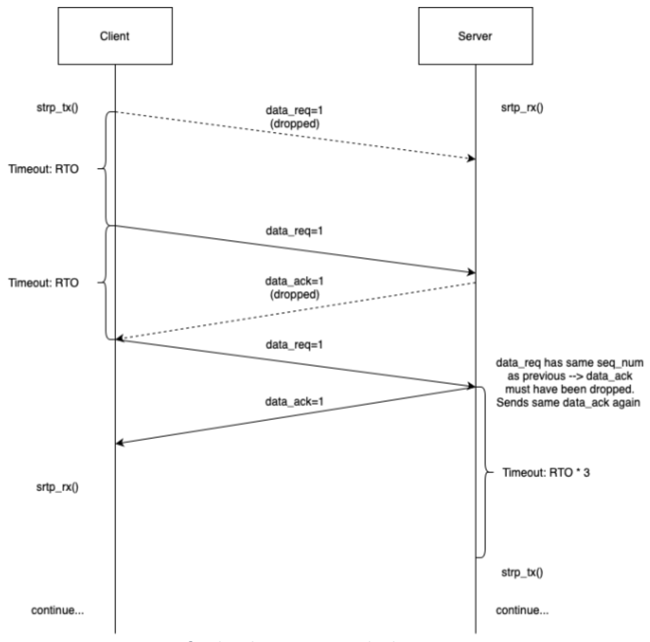


Figure 4 Data Transfer both ways - with drop

Furthermore, when sending data both ways with dropped packets. As seen in Figure 1 with opening a connection, the same approach is taken when dropping data request packets. The data request is retransmitted after the given RTO value. To account for the drop in data acknowledgements required a different approach compared to when dropping data requests. If a data acknowledgement is dropped, srtp_tx() will retransmit the data request packet. The srtp_rx() will receive the data request and observe that the sequence number attached to the data request packet is the same as the last packet to be received. This shows that the data acknowledgement was therefore dropped.

As a retransmission has occurred. The srtp_rx() function will handle this by sending the same data acknowledgement again. The method for determining the data acknowledgement has been successfully transmitted for data transfer both ways, is that the timeout of RTO * 3 occurs. Which leads the server in this case to enter srtp_tx() and the client srtp_rx().

Reliable Data Transfer – one way design:

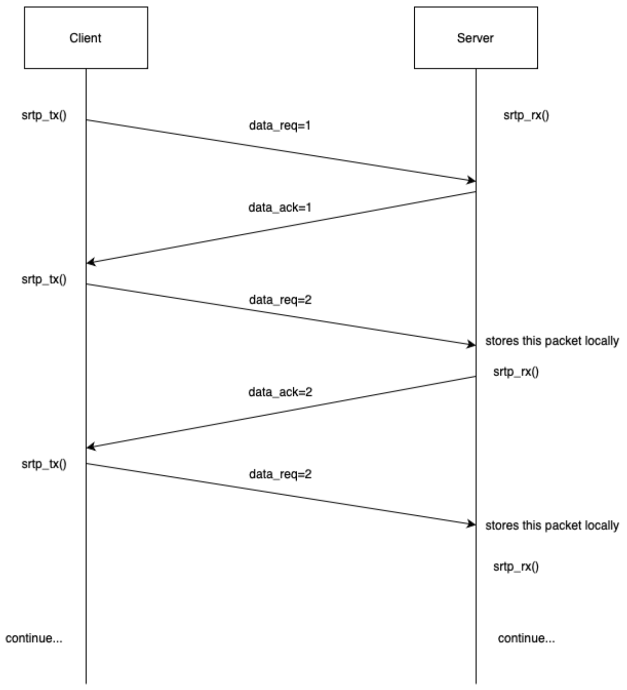


Figure 5 Data Transfer one way - no drop

In the case of data transfer one way, when no dropped packets occur I have designed the srtp_rx() function to for the first packet receive the first data request, send the data acknowledgement, and wait for the second data request which I store locally. The reason for this decision as the sequence number of the packet stored locally is used to determine whether a data acknowledgement is dropped. In this situation where no drop occurs, there is a conditional statement comparing the sequence numbers of the data request packet stored locally and the data request packet just received, as these sequence numbers will be different the data acknowledgement has not been dropped. This mechanism continues until the close packets are sent and received, and below when talking closing a connection I will mention the issues this raised and how I handled them.

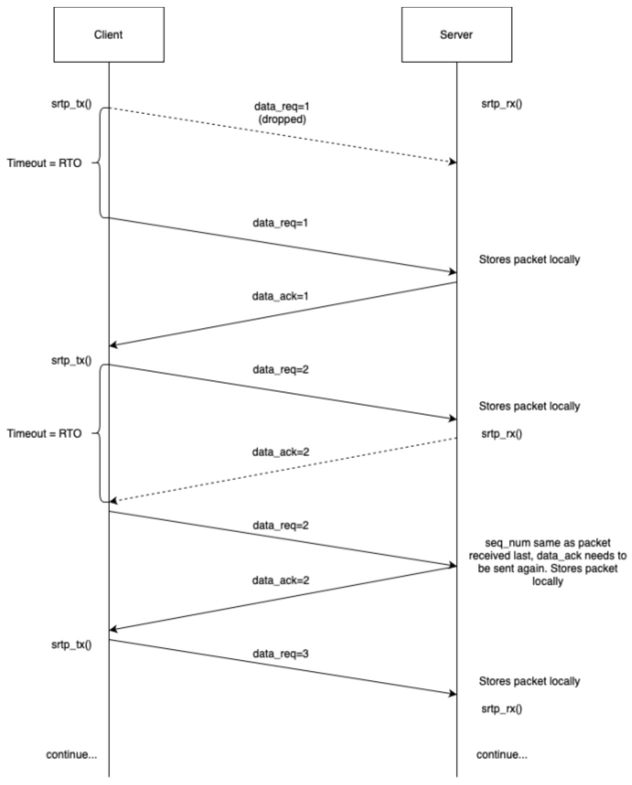
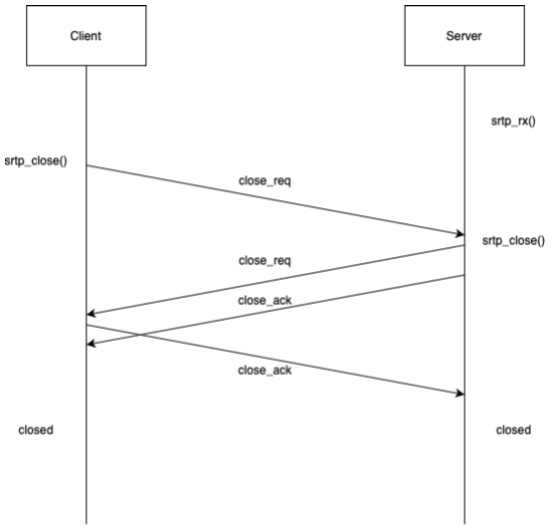


Figure 6 Data Transfer one way - with drop

The mechanism for handling dropped data requests has been mentioned in the previous sections. To handle the dropping of data acknowledgements, when the srtp_rx() function receives what should be the 'next' data request. The sequence number of the locally stored data request packet is compared to the data request packet that has just been received. It can be determined the data acknowledgement packet was dropped if the two sequence numbers are the same, as the data request packet has been retransmitted, so received again.

Terminating a connection:



Regarding closing the connection between client and server. Due to the design of the srtp_rx() function in that it technically receives the 'next' data request packet as a method for handling data acknowledgement loss. This results in a close request being received while in the srtp_rx() function. This is identified within the srtp_rx() function, and when a close request is received the srtp_rx() function returns and move to the srtp_close() call. Which performs the two-way handshake as can be seen in Figure 7.

Figure 7 Closing a connection

Justifications:

The reasoning for storing the last data request received locally is that it doesn't take up too much space locally and provides a reliable and successful method to ensure that when data acknowledgement packets are dropped it can be handled while also making sure that the data received is still in the correct order and that no data is dropped.

When data is being transferred both ways the reasoning for waiting the timeout of RTO * 3 after sending the acknowledgement is this ensures that there is no retransmission of the data request packet. The reason for multiplying the RTO by three specifically is to account for extra delay and the time taken for retransmission to occur.

Packet Design:

Packet header diagram:

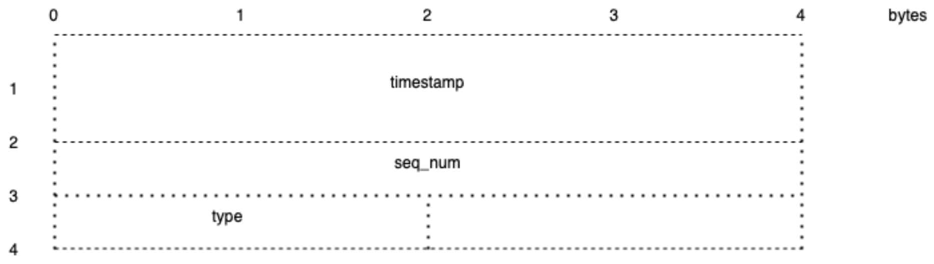


Figure 8 Packet Header diagram

FSM Diagram:

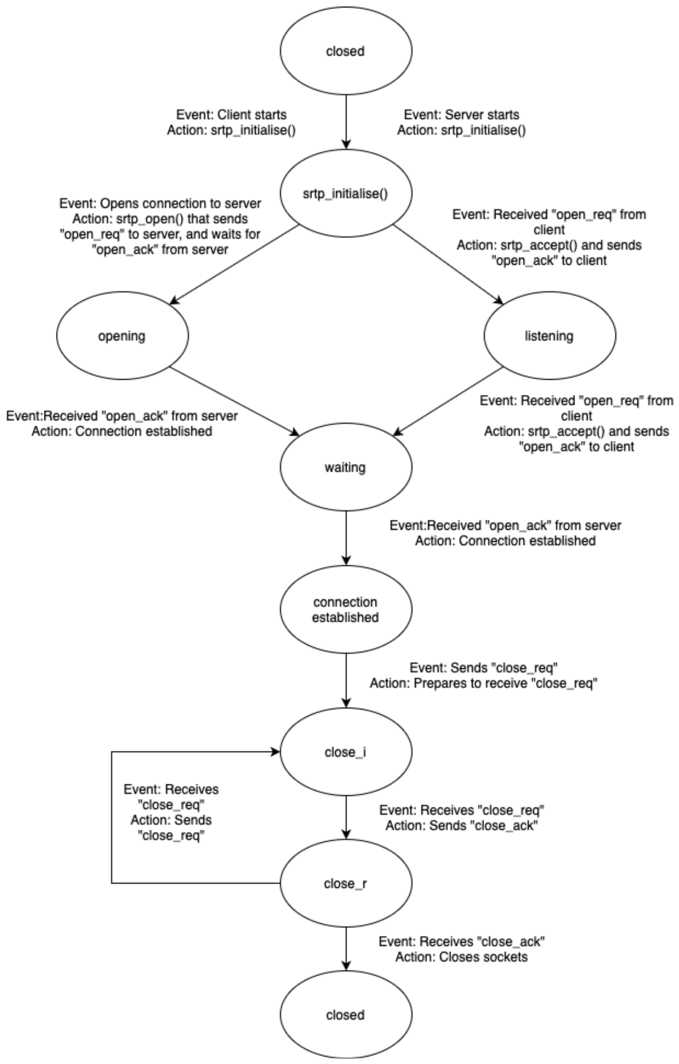

closed
Event: Client starts
Action: srtp_initialise()
Event: Server starts
Action: srtp_initialise()
srtp_initialise()
Event: Opens connection to server
Action: srtp_open() that sends "open_req" to server, and waits for "open_ack" from server
Event: Received "open_req" from client
Action: srtp_accept() and sends "open_ack" to client
opening
listening
Event:Received "open_ack" from server
Action: Connection established
Event: Received "open_req" from client
Action: srtp_accept() and sends "open_ack" to client
waiting
Event:Received "open_ack" from server
Action: Connection established
connection established
Event: Sends "close_req"
Action: Prepares to receive "close_req"
close_i
Event: Receives "close_req"
Action: Sends "close_req"
Event: Receives "close_req"
Action: Sends "close_ack"
close_r
Event: Receives "close_ack"
Action: Closes sockets
closed

Figure 9 FSM Diagram

The above FSM diagram details how the program moves through each stage when opening a connection, the connection is established and closing a connection with event and action descriptions on the FSM diagram. I decided not to include a table with the events and actions, as this has been included as part of the FSM diagram.

Implementation

Packet implementation:

The layout of my packet header design can be seen in Figure 8. In the packet header the timestamp is needed to calculate the RTT value which is needed when implementing the adaptive RTO which I will mention later. The sequence number is used as an identification for each packet, so that it can be determined whether packets have been sent or received twice. This is essential to deal with packet loss and retransmission mechanisms. The final part of the packet header I decided to use is a type, this stores a 16-bit integer value which is used to easily identify what type of packet has been sent and received. Furthermore, this proves very useful when sending different types of packets such as open, data and close, requests and acknowledgements. I ordered my packet header in terms of size as this I helps with the aligning of the packet and minimises any padding caused.

Adaptive RTO:

Adaptive RTO is a technique used to optimise the sending and receiving of packets. It performs a calculation using the RTT value when a data acknowledgement packet is received. The RTT value is usually only measured when there haven't been any retransmissions as this would skew the result calculated. To counteract this each time, I retransmit a packet I recalculate the timestamp value meaning the measurements are not skewed by retransmissions.

Through trial and error, I have implemented a smallest value for the RTO as I found that allowing the RTO value to drop as low as possible caused errors with retransmission and other aspects. Enforcing a minimum RTO value eradicated these issues.

When performing the RTO calculation, I used a dynamic estimate of the RTT value for a packet to travel from the sender to receiver and back again. The estimate is due to the external factors which can impact the RTT value. Additionally, I implemented a value to measure the deviation of the RTT values. These two values used together to calculate the RTO value allows for coverage of external factors and allow for unexpected delays.

Dropping packets:

To implement the dropping of packets I used a random generator using the current time as the seed for the random function. To aid with debugging and seeing what is going on within my program I still 'receive' packets that are dropped but I do nothing with them. This was solely to see what occurred within the program when a packet was dropped.

As mentioned in my overview, I did not implement the maximum retries functionality when a packet was dropped. This means that when imitating packet loss, in theory there could be a sizeable number of packets that have been dropped in a row. This will have impacted slightly on the results I will receive when running tests of my program.

Furthermore, had I implemented the maximum retries I could have implemented controlled loss such that if there has already been two retries the third packet would be guaranteed to be successfully transmitted. This would minimise issues that would occur otherwise and would allow for the testing results to be in line with a SRTP where all the data has been received, in order with no missing data packets.

Timeout:

To implement the timeout used within the designs of my protocol, I used pselect(). This is blocking so will not perform further operations until the timeout has triggered. This proved useful for implementing retransmission of data requests and ensuring that data acknowledgements were received by using a timeout of RTO * 3.

Testing

All results of experiments carried out can be found in the /data directory of my submission. When carrying out any experiments with 'loss' I have used a loss rate of 10%. Additionally, due to time constrictions to run experiments some tests have not sent the full 100,000 packets. I will clearly label which experiments this applies to, and results will be scaled up so that considering the most accurate results can be determined.

File name (in /data)	**Script used**	**Drop**	**Adaptive RTO**	**Duration(s) to 3dp**	**Mean data rate (bps)**	**Packets sent**
Test-client-1-no_drop_no_adaptive	test-client-1.c	No	No	13.906	73,639,910	100,000
Test-client-1-no_drop_adaptive	test-client-1.c	No	Yes	13.847	73,949,119	100,000
Test-client-1-drop_no_adaptive	test-client-1.c	Yes	No	5,615.263	182,360	100,000
Test-client-1-drop_adaptive	test-client-1.c	Yes	Yes	169.676	603,502	100,000
Test-server-1-no_drop_no_adaptive	test-server-1.c	No	No	13.905	73,641,367	100,000
Test-server-1-no_drop_Adaptive	test-server-1.c	No	Yes	13.847	73,950,897	100,000

Test-server-1-drop_no_adaptive	test-server-1.c	Yes	No	5,616.263	182,360	100,000
Test-server-1-drop_adaptive	test-server-1.c	Yes	Yes	169.676	603,504	100,000
Test-client-3-no_drop_no_adaptive	test-client-3.c	No	No	301,238.469*	3399	1,000
Test-client-3-no_drop_adaptive	test-client-3.c	No	Yes	9,060.227*	11,302	500
Test-client-3-drop_adaptive	test-client-3.c	Yes	Yes	9,374.105*	12,016	250

Figure 10 Testing table

(* values which have been scaled up to the approximate value when 100,000 packets are used)

As seen above all test results can be viewed in the /data of my submission. In all experiments carried out using loss, adaptive RTO it can be seen that all data transferred returns (good) which shows that my solution successfully handles loss and doesn't provide any errors.

Analysis and Conclusion

From my testing table in Figure 10 some key findings can be made. The first is that when loss is experienced, across the board this has a profound impact on the duration and mean data rates that can be achieved. For example, when using the test-client-1 program when loss occurs the is significantly higher. This outlines the importance of handling packet loss efficiently as it can have a major impact on how long it takes data to get from A to B. Furthermore, the amount of data that can be transferred when there is no loss compared to when loss is significant.

The use of adaptive RTO can be seen to have a positive impact when used, which is to be expected. It can be seen for test-client-1 when loss is implemented and adaptive RTO is activated the amount of time it takes for all the data to be transmitted is significantly lower, also leading to a higher mean data rate. This shows that the adaptive RTO mechanism I have implemented successfully aids in counteracting the consequences of packet loss.

Moreover, it can be seen that for test-{client, server}-1 in every experiment performs better compared to test-{client, server}-3. This shows that sending large amounts of data in one direction, then the other direction rather than one packet each way would result in a much more efficient system. This is however not possible in every situation, but where possible sending in one direction then swapping once all data has been sent would be advisable and this can be seen from the results in Figure 10.