Where did the bulk of logic occur?

In my implementation of the RDTLayer class, the bulk of the logic occurs within the process_send() function. The process_receive() function comes in a close second. Both functions were heavily reworked, and the original skeleton code was mostly scrapped.

Furthermore, the process_receive() function was updated to be called prior to the process_send() function. This was done in order to facilitate flow control. This approach allows incoming segments to be received, processed, and acknowledged before sending subsequent segments.

Finally, to make the code easier to troubleshoot and implement, several additional functions were created to be called within the process_send() and process_receive() functions. These functions include the send_segment(), sort_segments(), reassemble_segments(), and process_acks().

How were the timeouts resolved? What happened if timeouts have been resolved?

On the client end, my code resolves timeouts by tracking all outgoing payload segments within the sent_packets dictionary. The dictionary key for each segment is its sequence number, and the dictionary value is the number of iterations since the segment was sent. As respective acks are received by the client, corresponding segments are removed from the dictionary. If an ack is not received due to packet loss, delay, or data errors, the timeout value is incremented by 1. If the timeout value is greater than 1, a timeout occurs, and the client resends the segment and resets its timeout value to 0. Eventually, this results in all segments being sent.

On the server end, all incoming payload segments are sorted, then added to the pending_packets list, which is also sorted. These segments are also confirmed to be error-free. The server uses the self.acknum value to track which segment contains the next in-order payload. Then, the server loops through the pending_packets list and searches for the segment. If the segment is found, it's payload is added to self.received_data and the value of self.acknum is updated to the next expected sequence number.

In the case that a packet is dropped, delayed, etc., self.acknum will not update and inhibit new data from being recorded until the required packet is received. Due to the client's timeout property, a packet is typically recovered within two iterations: the first iteration being the one that resulted in the missing packet, and the second waiting for the timeout to expire (timeout value reaches > 1). That being said, it's possible more than two iterations are required to retrieve a segment due to further packet or ack loss. Once a timeout has been resolved, it's value is recorded and self.acknum is increased. Any downstream segments waiting on the missing packet are then recorded.

How was the packet dropping handled?

Packet dropping is handled via the same process as timeouts, explained above. The server acks any packets it has received. If a packet was dropped or delayed, the server will not send an ack. On the client end, any non-acked packets are re-sent after a timeout. This includes non-acked payload packets due to packet drops, as well as dropped or delayed acks from the server to the client.

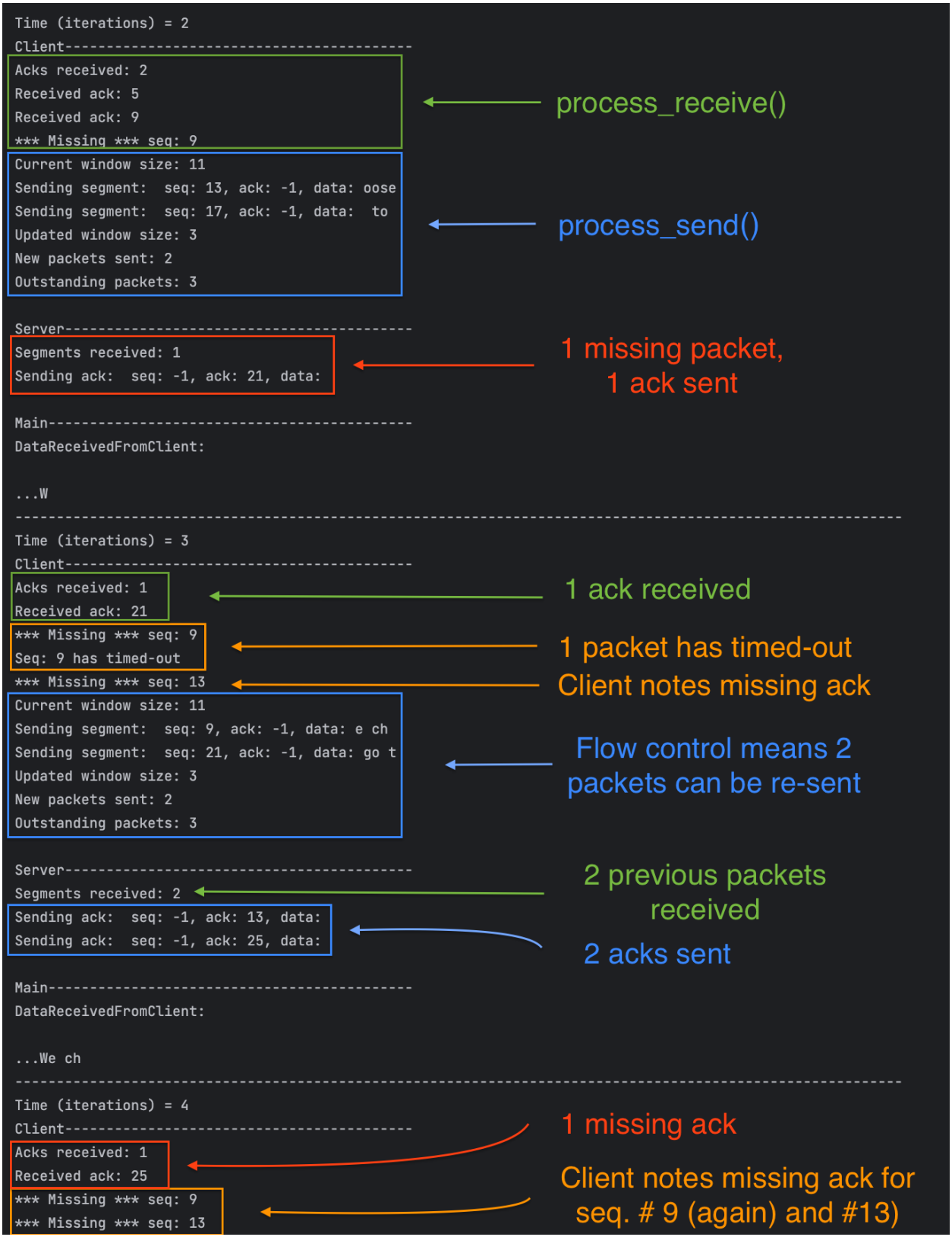
How was the retransmission policy implemented?

Retransmission was implemented using a selective retransmit approach, explained above.

On the client end, the sequence number for every outgoing packet is tracked via the sent_packets dictionary. All packets received by the server are acked back to the client. The client removes sequence numbers from the dictionary that correspond to any of the acks received. If an ack is not immediately received, the sequence's timeout value is incremented by 1. Once the timeout value reaches > 1), the pack is selectively retransmitted and it's timeout value is reset to 0. Only non-acked packets are re-sent. This process repeats until the packet's ack is received.

On the server end, all error-free incoming packets are acked, sorted, and stored in a buffer called the pending_packets list. The server then parses the list looking for the next expected sequence number, which is the cumulative ack number variable self.acknum. Since it's possible that acks from the server to the client were previously delayed or dropped, any incoming packets that have a sequence number lower than the current cumulative ack number self.acknum are acked then discarded. The server will hold any downstream packets in the buffer until the lowest missing packet is received. At that point, any packets waiting on the missing packet will be officially recorded in-order. The variable self.acknum will then be updated to the next expected sequence number.

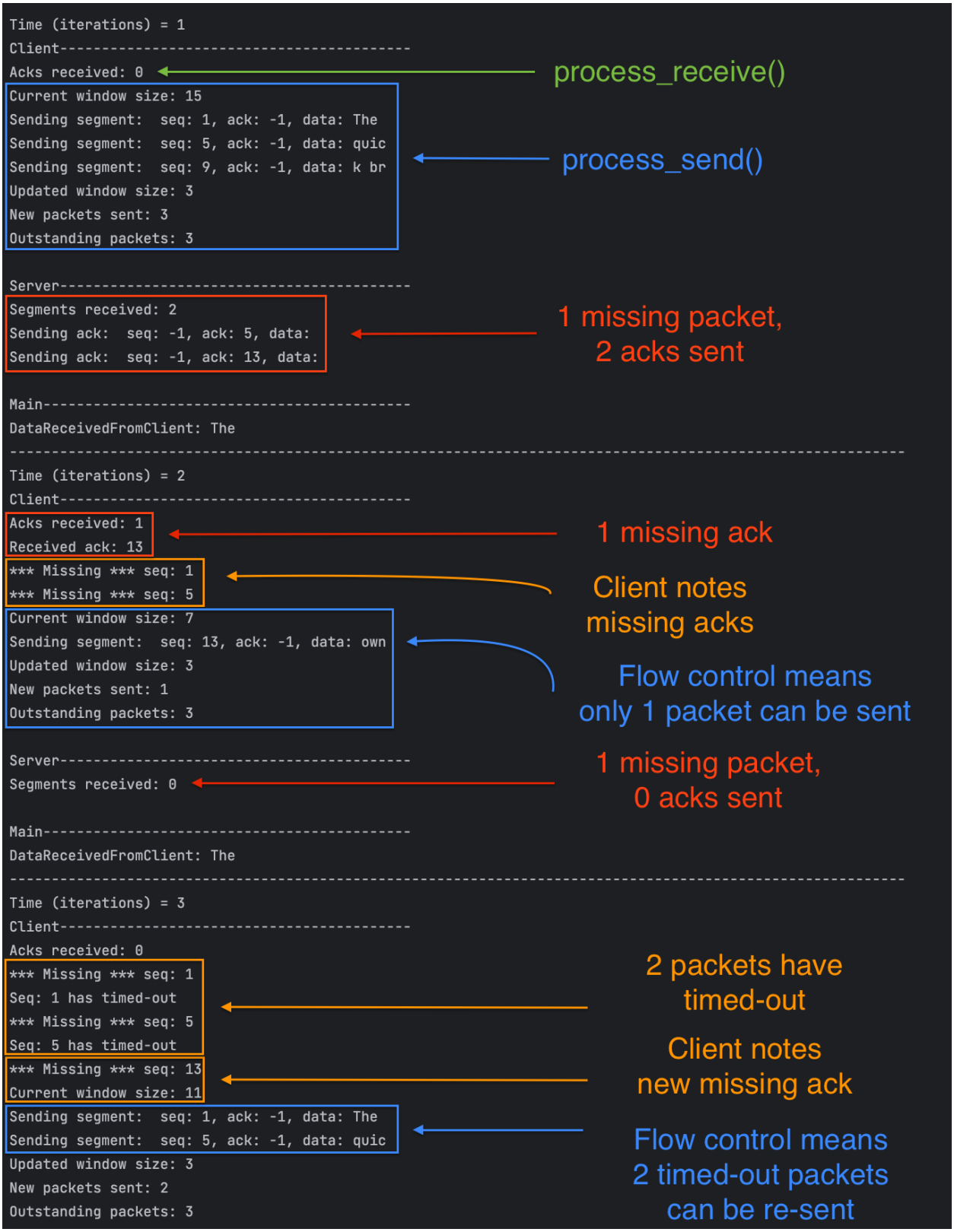
Screenshot 1: Larger text



Screenshot 2: Larger text

```
Updated window size: 7
New packets sent: 1
Outstanding packets: 2

Server----------------------------------------
Segments received: 1
Sending ack:  seq: -1, ack: 1241, data:

Main----------------------------------------
DataReceivedFromClient:

...We choose to go to the moon. We choose to go to the moon in this decade and do the other things, not
 because they are easy, but because they are hard, because that goal will serve to organize and measure the
 best of our energies and skills, because that challenge is one that we are willing to accept, one we are
 unwilling to postpone, and one which we intend to win, and the others, too.

...we shall send to the moon, 240,000 miles away from the control station in Houston, a giant rocket more
 than 300 feet tall, the length of this football field, made of new metal alloys, some of which have not yet
  been invented, capable of standing heat and stresses several times more than have ever been experienced,
 fitted together with a precision better than the finest watch, carrying all the equipment needed for
 propulsion, guidance, control, communications, food and survival, on an untried mission, to an unknown
 celestial body, and then return it safely to earth, re-entering the atmosphere at speeds of over 25,000
 miles per hour, causing heat about half that of the temperature of the sun--almost as hot as it is here
 today--and do all this, and do it right, and do it first before this decade is out.

JFK - September 12, 1962

$$$$$$$$ ALL DATA RECEIVED $$$$$$$$
countTotalDataPackets: 441
countSentPackets: 825
countChecksumErrorPackets: 58
countOutOfOrderPackets: 22
countDelayedPackets: 80
countDroppedDataPackets: 36
countAckPackets: 365
countDroppedAckPackets: 24
# segment timeouts: 365
TOTAL ITERATIONS: 226

Process finished with exit code 0
```

Screenshot 3: Smaller text



Screenshot 4: Smaller text

```
Main-------------------------------------------
DataReceivedFromClient: The quick brown fox jumped o
-----------------------------------------------------------------------------------------------------------
Time (iterations) = 7
Client-----------------------------------------
Acks received: 2
Received ack: 29
Received ack: 37
*** Missing *** seq: 29
Seq: 29 has timed-out
Current window size: 15
Sending segment:  seq: 29, ack: -1, data: ver
Sending segment:  seq: 37, ack: -1, data: lazy
Sending segment:  seq: 41, ack: -1, data:  dog
Updated window size: 3
New packets sent: 3
Outstanding packets: 3

Server-----------------------------------------
Segments received: 3
Sending ack:  seq: -1, ack: 33, data:
Sending ack:  seq: -1, ack: 41, data:
Sending ack:  seq: -1, ack: 45, data:

Main-------------------------------------------
DataReceivedFromClient: The quick brown fox jumped over the lazy dog
$$$$$$$$ ALL DATA RECEIVED $$$$$$$$
countTotalDataPackets: 15
countSentPackets: 26
countChecksumErrorPackets: 1
countOutOfOrderPackets: 1
countDelayedPackets: 2
countDroppedDataPackets: 0
countAckPackets: 13
countDroppedAckPackets: 2
# segment timeouts: 10
TOTAL ITERATIONS: 7

Process finished with exit code 0
```