Communication Network TCP Project

Mathis Lindner - Sven Gutjahr- Martin Stecher

1.Complete the GO-Back-N sender:

Send state:

To send a packet, we defined our header_GBN as type data, with the correct segment and header lengths, the corresponding sequence number and window size by using the predefined values.

ACK_IN state:

For the ack_in state we are iterating from our first unacked packet through the whole buffer until we reach our received ack. While doing so we delete every unacked packet in the buffer because we are using cumulative acks.

Retransmit state:

For retransmitting packets due to a timeout, we are iterating again through the buffer of unacked packets. For every element in the buffer, we again create a new header with the corresponding values of the unacked packet and send it again to the receiver.

Theoretical question:

This conclusion is not true. Packets could be sent via different routes and arrived in a different order than they were sent. For example, the receiver received first packet 5 and then 4 and at last packet 3, so now the receiver sends first ACK 3 two times and then ACK 6 but ACK 6 gets lost. The receiver actually received all packets but an ACK was lost by the network.

2.Implement Selective Repeat

Receiver:

Implementation:

If the incoming packet is the expected packet, we process it and directly check if we have the following packets stored in our buffer. If so, we also add them to the output file. If it is not the expected Packet, we save it in the buffer and go back to the IN_DATA state until we process all the expected packets.

Theoretical question:

It is not always beneficial, because if we have acks, which are not in our current window size, these should not be considered because they are an obvious mistransmission. If we do not filter them out, we could store redundant information.

Sender:

Implementation:

The dictionary acks_received{} keeps track of how many times we received an ACK packet. We increase our counter when we get a new ACK. As soon as we've received an ACK 3, we resend the unacknowledged segment and set the value of the acks_received[ack] back to 0. If we receive a correct ACK we detect the entry in the dictionary.

3. Implement Selective Acknowledgement

Receiver:

Implementation:

We are iterating from our next expected ack to the highest ack number allowed from our window size. We know that there must have been a loss of an ack before our first element in the buffer, otherwise, cumulative ack would have taken care of it. So, our first block has to start there. After that we only need to iterate through the buffer until a subsequent ack does not have an acknumber that is one larger than the previous one, hence there will be our next loss. We repeat this until we have reached a block_number of three. Through the steps, we save the left_edge of each block and the lengths in a list. We take care of the overflow by using the modulo operator, so we don't have to consider special cases because our acknumber will always be in our "2**n_bits" range. The conditions for the receiver to add the optional fields to the header are bound to our list, where we saved the left_edges and the lengths of the block. We simply look at the length of our list, for example if the list length divided by 2 is 2, we know that we have 2 blocks and hence need 6 optional fields (block_number, 2x left edges, 2x lengths and 1 padding).

Theoretical question:

We could take the most basic implementation and just send every ACK we are missing. Problem: the header could get extremely big and the sending effort would mostly be for sending the buffer state instead of the payload. The second approach could be to send also kind of block_numbers. But instead of sending the left edge and the length of each block, we could alternately send the length of received acks and then the length of not received acks, with a basic case, e.g., -1 to state the end of the sequence.

Sender:

Implementation:

The approach was to find the missing acks and resend them. To achieve that, we iterated through our list, where we saved the sack information from the header. After a left edge value, we only had to add the length of that block length and then the next value is the start of our first lost ACK segment. After that, all ACKs were lost until the next left edge value. After finding the missing acks, we simply resent them if

they were in our buffer, to avoid again wrong ack information (e.g., out of window values)

Theoretical question:

The sender could, before sending, check what the last retransmitted packets were. If the next header requests to resend a packet, which was recently resent, the sender would stop that sending process. Certainly, the sender had to allow the sending of that packet again after a certain amount of time, because the ACK could be lost twice.