Communication Networks Transport Project Report, Group 86

Alkinoos Sarioglou, Johannes Schading, Fabian Stuber

June 3, 2021

1 Question 4.1 (Fabian)

To complete the SEND state, the GBN header had to be implemented. Therefore a new GBN header object was created with the following attributes: The sent packet is data, the SACK option is for now disabled, the length attribute is set to the length of the payload and the header length is six, as there are no optional header elements at the moment. Then the sequence number is set to the current one and the windows size to the objects attribute for the windows size. After this header was created, the packet consisting of the header and the payload is sent to the IP layer.

The ACK_IN state first sets the unack variable to a new value. The new value is exactly the sequence number contained in the received ACK packet, as cumulative acknowledgement is used, therefore this sequence number represents the next expected, therefore the next unacknowledged sequence number. Afterwards, the acknowledged packets are removed from the buffer. As sequence number overflow can occur, it was not possible to just remove all the packets with a sequence number smaller than the ACK number. Nevertheless, the maximum number of packets that can be removed from the buffer, is equal to the window size. Therefore all packets that are sent previous to, but still could be in the same window as, the last acknowledged packet, are removed from the buffer. It is possible, that some of these removed packets do have a bigger sequence number than the last acknowledged packet, but if one subtracts the maximum sequence number from this sequence number, they don't.

In the RETRANSMIT state, all elements from the buffer are sent again like in the SEND state, meaning a header is created with the corresponding attributes and the packet is sent to the IP layer.

1.1 Theoretical Question

If the packets are reordered, meaning the receiver receives packet 3 after packet 4 and 5, and additionally, the ACK that 3 was received, meaning an ACK with the number 6, is lost, the sender will receive two ACKs containing the number 3, even though packet 3 wasn't lost.

2 Question 4.2 (Johannes)

2.1 Receiver

The receiver has a state indicating the current expected sequence number from the sender. The number of this state defines the first number in the sliding window. With the state of the current window size ($self.win$) and the current expected packet ($self.next$) one can calculate the range of elements which should be buffered. If a sequence number is in the following range, it will be buffered

$$\{firstElement, firstElement + 1, ..., lastElement\} \tag{1}$$

where

$$\begin{aligned} firstElement &= self.next + 1, \\ lastElement &= self.next + self.win - 1. \end{aligned} \tag{2}$$

Packets with sequence numbers outside of this set will be dropped. Of course this set has to be shifted when the sequence number has been decreased by virtue of the modulo operation. So the packet with a decreased sequence number has to be in the following shifted window

$$\{firstElement - 2 * *self.n\_bits, firstElement + 1 - 2 * *self.n\_bits, ..., lastElement - 2 * *self.n\_bits\}. \tag{3}$$

The payload (*self.buffer*) as well as the sequence number (*self.buffer_seq*) of the payload are stored in a list. After increasing the expected next sequence number the receiver checks for available buffered payloads, which can be identified via the stored sequence number.

The following codes are the implementation of the check for matching buffered elements and of the buffering once an out of order packet inside the window is detected respectively.

Listing 1: Check for matching buffered payloads and buffering of in-window out of order elements

```
@ATMT.state()
def DATA_IN(self, pkt):
    """State for incoming data."""
    ...
    # segment was received correctly
    else:
        ...
        # check if segment is a data segment
        ...
        if ptype == 0:
            ...
            # this is the segment with the expected sequence number
            if num == self.next:
                ...
                #check buffer for buffered elements with correct sequence number
                while self.buffer_size > 0:
                    if self.next in self.buffer_seq:
                        for itemSeq, item in zip(self.buffer_seq, self.buffer):
                            if itemSeq == self.next:
                                log.debug("Delivered packet from buffer to upper layer: %s", itemSeq)
                                with open(self.out_file, 'ab') as file_help:
                                    file_help.write(item)
                                self.buffer_size -= 1
                                self.buffer.remove(item)
                                if itemSeq in self.buffer_seq:
                                    self.buffer_seq.remove(itemSeq)
                                self.next = int((self.next + 1) % 2**self.n_bits)
                    else:
                        break

            # this was not the expected segment
            else:
                log.debug("Out of sequence segment [num = %s] received. "
                          "Expected %s", num, self.next)
                #log.debug("window size is %s", self.win)

                #add to buffer iff doesn't exceed window size
                firstElement = self.next + 1
                lastElement = self.next + self.win - 1
                if num < self.next:
                    firstElement = firstElement - 2**self.n_bits
                    lastElement = lastElement - 2**self.n_bits
                if lastElement >= num >= firstElement:
                    if num not in self.buffer_seq:
                        self.buffer_size += 1
                        log.debug("Out of order sequence segment [num = %s] buffered. ", num)
                        self.buffer.append(payload)
                        self.buffer_seq.append(num)

        else:
            ...
```

2.1.1 Theoretical Question

The answer to the theoretical question of why a buffer of infinite size is not beneficial is that the sequence numbers iterate through a cyclic group of addition modulo n with n a finite number. Therefore the same sequence number might appear repeatedly and the payload cannot be anymore identified uniquely. Hence, it is better to have a finite and small enough buffer.

2.2 Sender

For the implementation of the selective repeat we introduced a counter (*self.count_acks*) for ACKs arriving repeatedly. Once the counter reaches 3, the packet with the appropriate number of the arriving ACKs, and only this packet, will be resent. After re-transmission, the counter is set to zero. If an ACK with a new sequence number is received, the counter is set to 1. Moreover, once a timeout occurs, the counter is set to zero. Hence, the implementation of selective repeat comprised modifications in *ACK_IN*(*self*, *pkt*) and *timeout_reached*(*self*). Also the selective repeat can be executed only if $self.Q\_4\_2 == 1$ is true, otherwise the part which implements the selective repeat will never be considered.

The following code implements the selective repeat.

Listing 2: Selective repeat

```
@ATMT.state()
def ACK_IN(self, pkt):
    """State for received ACK."""
    # check if type is ACK
    if pkt.getlayer(GBN).type == 0:
        ...
    else:
        ...
        ack = pkt.getlayer(GBN).num

        #check if selective repeat needed and execute
        if self.Q_4_2 == 1:
            if ack == self.count_acks.__getitem__(1):
                self.count_acks.__setitem__(0, self.count_acks.__getitem__(0)+1)
                log.debug("Selective repeat counter: %s", self.count_acks.__getitem__(0) + 1)
            else:
                self.count_acks.__setitem__(0, 0)
                self.count_acks.__setitem__(1, ack)
                log.debug("Selective repeat counter: %s", 1)
            #resend iff equals 3, but only the one missing packet
            if self.count_acks.__getitem__(0) >= 2:
                log.debug("Selective repeat activated, resend packet %s", ack)
                header = GBN(type="data", options=0, len=len(self.buffer[ack]), hlen=6, ...
                ...  num=ack, win=self.win)
                send(IP(src=self.sender, dst=self.receiver) / header / self.buffer[ack])
                log.debug("Selective repeat finished")
                self.count_acks.__setitem__(0, 0)
                self.count_acks.__setitem__(1, -1)
```

3 Question 4.3 (Alkinoos)

3.1 Receiver

In the implementation of the Selective Acknowledgement (SACK) mechanism, the receiver needs to provide the suitable information in the SACK headers to the sender so that the appropriate packets can be re-transmitted. First of all, the receiver constructs a new type of header which has varying size according to the elements in the out-of-order segments' buffer. The algorithm written to provide the appropriate ranges in the SACK header is the following:

- First, the receiver checks if the sender uses SACK, hence if the `options` field of the received packet is '1'. If it is not, the receiver just sends an ACK with only the mandatory field of the header present. Otherwise, the receiver starts constructing the SACK header to include in the returned segment.
- The initial step includes reading the list (constructed in 4.2), which includes the sequence numbers of the out-of-order elements received. These elements are then sorted in order to deal with reordering in the network. However, it is also important to deal with the overflow. For example, in a case where the buffer includes [24,25,30,0,27], the desired sorted sequence would be [24,25,27,30,0], therefore a mere numerical sorting does not return the correct result. Here, the proposed solution includes:
  - Separate the sequence numbers in the buffer in bigger and smaller than the maximum value (2**n_bits) over 2 (max_value/2) and place them in two separate lists `under[]` and `over[]`. Then sort these lists in numerical order:

Listing 3: Separate out-of-order sequence numbers in groups

```
# Separate seq nums to groups of bigger and smaller than the half maximum value
for i in self.buffer_seq:
    if (i >= max_value/2):
        over.append(i)
    else:
        under.append(i)

# Sort the lists produced above in numerical order
over_sorted = sorted(over)
under_sorted = sorted(under)
```

- If both lists have elements, then overflow probably has happened and it needs to be dealt with. In this case, if the first element of the sorted `over` list has a bigger value than or it is equal to *max_value - window_size* and the first element of the sorted `under` list is smaller than or equal to *window_size*, then the two lists are concatenated in the opposite order (the `over` list is placed before the `under` list). Otherwise, the strict numerical ordering is followed. As mentioned in the example before, when `buffer_seq` = [24,25,30,0,27] with max_value=32 and win_size=10, then after separation we get over_sorted = [24,25,27,30] and under_sorted = [0] and the final sorted_buffer_seq will be [24,25,27,30,0], as desired. The code snippet that implements this is shown here:

Listing 4: Deal with overflow and reordering

```
# If elements in both lists then just return the correct order of ACKs
# dealing with overflow and reordering
if (over_sorted and under_sorted):

    # Deal with overflow and reordering of received segments
    # Only for the case when both under and over lists have elements
    # e.g. when buffer_seq = [24,25,30,0,27]
    # After separation: over_sorted = [24,25,27,30] and under_sorted = [0]
    # But desired result is sorted_buffer_seq = [24,25,27,30,0]
    # so pure sorting will not yield the desired result
    # The following function produces the result
    log.debug(self.win)
    if (over_sorted[0] >= max_value-(self.win) and under_sorted[0] <= (self.win)):
        sorted_buffer_seq = over_sorted + under_sorted
    else:
        sorted_buffer_seq = under_sorted + over_sorted
```

- In every other case, where only one of the two lists contains elements, the returned sorted_buffer_seq is the sorted corresponding list. Also, if none of the lists contains elements that means that the buffer is empty so an empty list is used:

Listing 5: Simple cases of sorting

```
# Else if only one list has elements return the sorted corresponding list
elif (over_sorted and not under_sorted):
    sorted_buffer_seq = over_sorted
elif (under_sorted and not over_sorted):
    sorted_buffer_seq = under_sorted
# If there are no elements in the buffer_seq list, return an empty list
elif (not under_sorted and not over_sorted):
    sorted_buffer_seq = []
```

- The following step includes finding the sequence numbers that are not consecutive with their neighbour sequence number in the sorted_buffer_seq list. In order to find the correct ranges, the following code snippet goes through the elements of the list and finds the beginning sequence numbers and the ending sequence numbers of each range found. It also deals with overflow by considering the consecutive sequence numbers of (2**self.n_bits-1) and 0 as belonging in the same range:

Listing 6: Detect beginning and end of ranges

```
ranges = sum((list(seq_num) for seq_num in zip(sorted_buffer_seq, sorted_buffer_seq[1:]) \
if (seq_num[0]+1 != seq_num[1] and seq_num[0] != (2**self.n_bits -1))), [])
```

```
# "iranges" is an iterator giving the starting and ending points of different blocks
# including the first and final blocks
iranges = iter(sorted_buffer_seq[0:1] + ranges + sorted_buffer_seq[-1:])

# Create a list that saves starting and ending points
start_end_ranges = []
for x in iranges:
    start_end_ranges.append(x)
```

- Then, depending on the elements of the `start_end_ranges` list, the header length, the block length, the left edges and the lengths of each range are calculated. Every element with an even index is a left edge of a block (up to 3 left edges) and the lengths are calculated by taking the difference between the element of the even index and its immediate neighboring element. Also, if the list has 2 elements, then there is one block available, if the list has 4 elements there are two blocks available and if the list has 6 elements then there are three blocks available. These approaches are reflected in the following code:

Listing 7: Extracting left edges, lengths, block length and header length

```
# 1 Block
if (len(start_end_ranges) == 2):

  header_length = 9
  block_length = 1
  first_1 = start_end_ranges[0]
  len_1 = start_end_ranges[1] - first_1 + 1

# 2 Blocks
elif (len(start_end_ranges) == 4):

  header_length = 12
  block_length = 2
  first_1 = start_end_ranges[0]
  len_1 = start_end_ranges[1] - first_1 + 1

  first_2 = start_end_ranges[2]
  len_2 = start_end_ranges[3] - first_2 + 1

# 3 Blocks
elif (len(start_end_ranges) >= 6):

  header_length = 15
  block_length = 3
  first_1 = start_end_ranges[0]
  len_1 = start_end_ranges[1] - first_1 + 1

  first_2 = start_end_ranges[2]
  len_2 = start_end_ranges[3] - first_2 + 1

  first_3 = start_end_ranges[4]
  len_3 = start_end_ranges[5] - first_3 + 1
```

- Attention is required in the case of calculating the length of a range when overflow occurs. In this case the length of the range would end up to be negative according to the above calculations. Therefore, in order to get the appropriate length we need to add the negative length to 2**(n_bits). For example in the case where sorted_buffer_seq = [30,31,0,1,2] and n_bits=5, then the calculated length is 2 - 30 + 1 = -27, so to get the right length the calculation becomes 32 + (-27) = 5. The following code illustrates how this is done:

Listing 8: Calculate the correct length in every case (overflow)

```
if (len_1 <0):
  len_1 = 2**self.n_bits + len_1
if (len_2 <0):
  len_2 = 2**self.n_bits + len_2
if (len_3 <0):
  len_3 = 2**self.n_bits + len_3
```

- Finally, the header of the segment is constructed and the packet is sent to the sender:

Listing 9: Header construction and packet integration

```
# SACK Segment
header_GBN = GBN(type="ack",
                 options=1,
                 len=0,
                 hlen=header_length,
                 num=self.next,
                 win=self.win,
                 blen=block_length,
                 left_1=first_1,
                 length_1=len_1,
                 padd_2=0,
                 left_2=first_2,
                 length_2=len_2,
                 padd_3=0,
                 left_3=first_3,
                 length_3=len_3
                 )

send(IP(src=self.receiver, dst=self.sender) / header_GBN,
     verbose=0)
```

Moving on into how the SACK header is constructed, its declaration is the following:

Listing 10: SACK header declaration

```
fields_desc = [BitEnumField("type", 0, 1, {0: "data", 1: "ack"}),
               BitField("options", 0, 7),
               ShortField("len", None),
               ByteField("hlen", 0),
               ByteField("num", 0),
               ByteField("win", 0),
               ConditionalField(ByteField("blen", 0), lambda pkt:pkt.hlen >= 7),
               ConditionalField(ByteField("left_1", 0), lambda pkt:pkt.hlen >= 8),
               ConditionalField(ByteField("length_1", 0), lambda pkt:pkt.hlen >= 9),
               ConditionalField(ByteField("padd_2", 0), lambda pkt:pkt.hlen >= 10),
               ConditionalField(ByteField("left_2", 0), lambda pkt:pkt.hlen >= 11),
               ConditionalField(ByteField("length_2", 0), lambda pkt:pkt.hlen >= 12),
               ConditionalField(ByteField("padd_3", 0), lambda pkt:pkt.hlen >= 13),
               ConditionalField(ByteField("left_3", 0), lambda pkt:pkt.hlen >= 14),
               ConditionalField(ByteField("length_3", 0), lambda pkt:pkt.hlen >= 15)
               ]
```

It can be seen that the conditions for every conditional field depend on the length of the header specified from the SACK algorithm. Every individual field has a different required header length because it is added in a sequence, but essentially the only possible header lengths are 6 (no blocks advertised), 9 (1 block advertised), 12 (2 blocks advertised) and 15 (3 blocks advertised). In the case of sending a simple ACK, the header length is 6 so none of the additional fields is added.
Besides this specific SACK header design, there are other designs that could be used:

- Instead of the two padding fields in the existing header, these could be replaced with two new fields: total number of blocks available in the out-of-order segments' buffer, number of elements in the out-of-order segments' buffer. That way the receiver can also inform about additional blocks available, besides the three advertised and the sender can infer some more information about other segments received successfully.
- There could also be another design where the receiver selects to advertise different blocks besides just the three first. This could mean that in different segments, the receiver adds SACK headers to advertise any range of blocks and not necessarily the first three blocks available in the buffer. For example, if 4 blocks are available, the first segment could advertise the blocks 1-3, but the second segment could advertise blocks 2-4 in the buffer. That way the sender can shape a more detailed view of the receiver's buffer and send the required packets faster. The indexes of blocks can be introduced into the header by replacing the padding fields with two new fields: starting block index, ending block index.

3.2 Sender

The implementation of the sender includes reconstructing the view of the receiver's buffer according to the SACK header received and then comparing it with the sender's buffer (which keeps the sent but not yet ACKed packets). The packets that lie out of the boundaries of the received blocks are re-transmitted, apart from the ones that have bigger sequence number than the right edge of the last block.
More specifically, the sender uses the left edge of every block and its length to add to a list the range of numbers in between them. Here, the algorithm also deals with overflow by first adding the range between the left edge until the max_value (2**n_bits) and then the range from 0 until the right edge. This process is repeated for all the blocks and then the resulting list of ranges is flattened to reflect the received ACKs.
Following this, a new list is constructed to include the keys of the sender's buffer dictionary. Using this, we can find the index of the last ACK received and keep only the elements of the sender's buffer before that, because we do not want to re-transmit any elements after the right edge of the last block as mentioned before (sender's reduced buffer).
Then, the two arrays of receiver's buffer and sender's reduced buffer are compared. For every element that is not included in the receiver's buffer, a new SACK header is constructed and the corresponding payload is sent again. There are three different cases depending on the block length of the SACK header received but the following code snippet only includes the mechanism's implementation for block length of 3 (see the other cases in the actual code files):

Listing 11: Sender Retransmission Algorithm

```python
# Declare a list which will reflect the ACKs received from the ranges in the received SACK packet
rec_acks = []
flat_rec_acks = [] # to combine all ranges for block length > 1

# Block length 3
if (pkt.getlayer(GBN).blen == 3):

    # Create the reflection of the received ACKs in the list from the range received

    # Deal with overflow first - Block 1
    if (pkt.getlayer(GBN).left_1 + pkt.getlayer(GBN).length_1 >= 2**self.n_bits):
        rec_acks.append(range(pkt.getlayer(GBN).left_1, 2**self.n_bits))
        rec_acks.append(range(0, \
        pkt.getlayer(GBN).length_1 - len(range(pkt.getlayer(GBN).left_1, 2**self.n_bits))))
    else: # Normal case
        rec_acks.append(range(pkt.getlayer(GBN).left_1, \
        pkt.getlayer(GBN).left_1 + pkt.getlayer(GBN).length_1))

    # Deal with overflow first - Block 2
    if (pkt.getlayer(GBN).left_2 + pkt.getlayer(GBN).length_2 >= 2**self.n_bits):
        rec_acks.append(range(pkt.getlayer(GBN).left_2, 2**self.n_bits))
        rec_acks.append(range(0, \
        pkt.getlayer(GBN).length_2 - len(range(pkt.getlayer(GBN).left_2, 2**self.n_bits))))
    else: # Normal case
        rec_acks.append(range(pkt.getlayer(GBN).left_2, \
        pkt.getlayer(GBN).left_2 + pkt.getlayer(GBN).length_2))

    # Deal with overflow first - Block 3
    if (pkt.getlayer(GBN).left_3 + pkt.getlayer(GBN).length_3 >= 2**self.n_bits):
        rec_acks.append(range(pkt.getlayer(GBN).left_3, 2**self.n_bits))
        rec_acks.append(range(0, \
        pkt.getlayer(GBN).length_3 - len(range(pkt.getlayer(GBN).left_3, 2**self.n_bits))))
    else: # Normal case
        rec_acks.append(range(pkt.getlayer(GBN).left_3, \
        pkt.getlayer(GBN).left_3 + pkt.getlayer(GBN).length_3))

    # Make flat list of ranges
    for i in range(len(rec_acks)):
        for k in range(len(rec_acks[i])):
            flat_rec_acks.append(rec_acks[i][k])

    # Keep seq num of elements in sending buffer in a list
    seq_numbers_send = list(self.buffer.keys())
    # Specify seq num of last packet ACKed to set the limit for the retransmission window
    # No retransmission allowed for packets after the last SACK block
    final_index = seq_numbers_send.index((flat_rec_acks[-1]))

    # Keep only the part of the sending buffer inside the allowed range
    # set by the last ACKed packet in the SACK message
    buffer_appl = seq_numbers_send[:final_index]

    # Check all the seq nums in the allowed range of the sender buffer and retransmit only the ones
    # that are not included in the received ACKs of the SACK message
    for seqnum_appl in buffer_appl:
        if (seqnum_appl not in flat_rec_acks):
            log.debug("Sender SACK activated, resending packet %s", seqnum_appl)
            header = GBN(type="data", options=1, len=len(self.buffer[seqnum_appl]), \
            hlen=6, num=seqnum_appl, win=self.win)
            send(IP(src=self.sender, dst=self.receiver) / header / self.buffer[seqnum_appl])

    log.debug("Sender SACK retransmit finished for blocks 1,2,3")
```

Another small change is added in the `RETRANSMIT` state, where after a timeout if the SACK option is enabled the header has an `options` field of '1'.

Listing 12: Modified RETRANSMIT state

```
@ATMT.state()
def RETRANSMIT(self):
    """State for retransmitting packets."""

    ##################################################
    # TODO:                                          #
    # retransmit all the unacknowledged packets      #
    # (all the packets currently in self.buffer)     #
    ##################################################
    #go through the buffer, create a header for each element and retransmit it

    for seqnum in self.buffer:

        if (self.SACK):
            header = GBN(type = "data", options = 1, len = len(self.buffer[seqnum]), \
            hlen = 6, num = seqnum, win = self.win)
            send(IP(src = self.sender, dst = self.receiver)/header/self.buffer[seqnum])
        else:
            header = GBN(type = "data", options = 0, len = len(self.buffer[seqnum]), \
            hlen = 6, num = seqnum, win = self.win)
            send(IP(src = self.sender, dst = self.receiver)/header/self.buffer[seqnum])

    # back to SEND state
    raise self.SEND()
```

The SACK option is enabled in the sender by setting the parameter `Q_4_3`. Then, it is asserted that parameters `Q_4_2` and `Q_4_3` are not active at the same time. When SACK is activated, the sender sends a packet with the `options` field set to '1' and waits for the received SACK packets from the receiver. If the receiver at any point sets the `options` field to '0' then the sender also stops using SACK and sends the rest of the packets with the `options` field set to '0'. But if the receiver replies with a SACK header then the SACK mechanism continues being used between them:

Listing 13: SACK negotiation

```
# Check if Receiver is using SACK otherwise deactivate it
if (pkt.getlayer(GBN).options == 0):
    self.SACK = 0
    # break
elif (pkt.getlayer(GBN).options == 1):
    self.SACK = 1
```

It is true that with this implementation many unnecessary packets are re-transmitted. To improve this drawback, another possible implementation of the sender would be:

- Receive the first SACK header and determine the packets that need to be re-transmitted.

- Send the packets that are not ACKed by the SACK header and save them locally in the sender too.
- If further SACK headers require the sender to send the same packets again, do not do it and only send the unACKed packets that fall outside the new blocks.
- Only resend the already re-transmitted packets if they are not included in 3 consecutive SACK headers.

4 Question 4.4 (Fabian&Johannes)

Every time before a packet is sent in the send state, the sending window size self.win is set to the minimum of the receiving window size and the congestion window. With this it is ensured, that the receiver is not overloaded and that there is proper reaction to congestion. The algorithm to adjust the congestion window is chosen to be the same as in TCP, namely first a slow start phase with exponential growth. After a threshold is reached, the increase becomes linear. In this phase additive increase and multiplicative decrease (AIMD) is used. As it was shown in the lecture this would result in a fair allocation of bandwith when there are multiple senders. The threshold is set to infinity at the start, therefore the sending window is increased quickly until congestion occurs. When a rough value for the available bandwith is known, the threshold is set to this value divided by two. A CWND evolution is shown in figure 1. As the loss of packets and acks is random in this test, the timeouts or duplicate acks are random as well, which results in a chaotic behaviour. Nevertheless the linear increase can be seen from timestep 60 to 90.

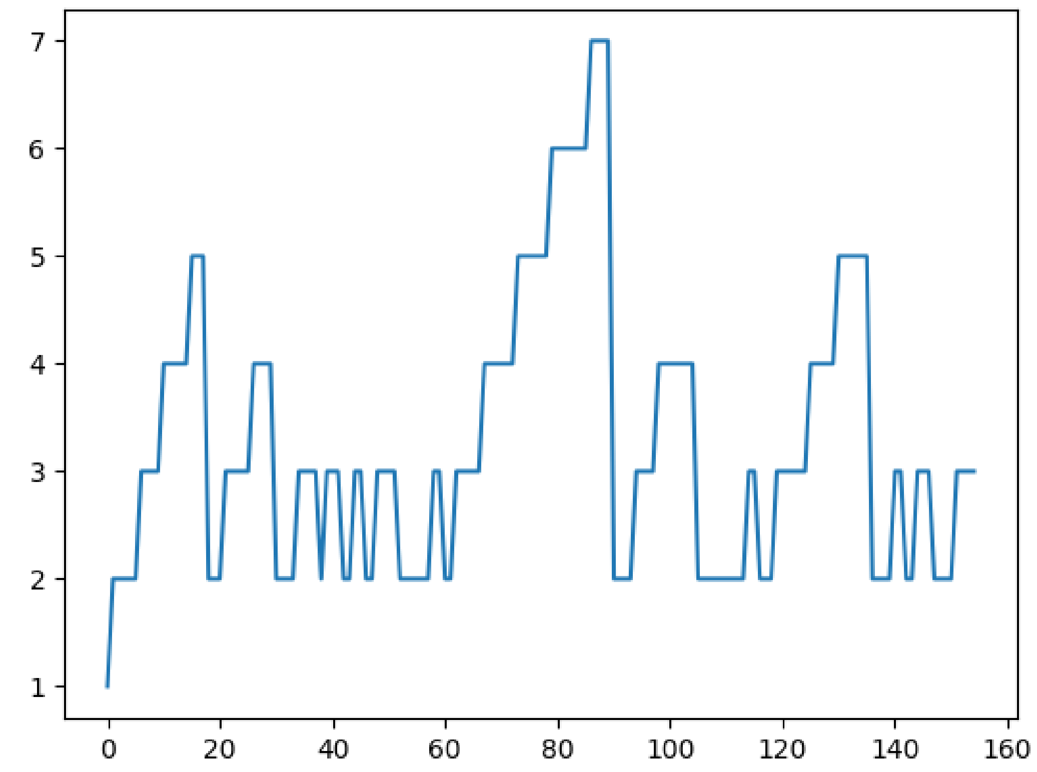

Figure 1: CWND evolution