Communication Networks

Group 85

Dominik Schindler
Student ID: 09-916-123

Yannick Huber
Student ID: 16-917-619

Dario Bolli
Student ID: 16-832-685

FS 2021
227-0120-00L: Communication Networks

Eidgenössische Technische Hochschule Zürich

Due Date: 04-06-2021

Taks 4.1: Complete the Go-Back-N sender

SEND State

```
# Function that is called in the GBNSender's state machine's SEND state.
def on_send(sender, payload):
    sequence_number = sender.current
    send_gbn(sender, payload, sequence_number)
```

```
def send_gbn(sender, payload, sequence_number):
    # Only send if the sender is the real sender. For a mock sender in the
    # unit tests, just do nothing.
    if isinstance(sender, GBNSender):
        header = create_gbn_data_header(sender, payload, sequence_number)
        send(IP(src=sender.sender, dst=sender.receiver) / header / payload)
```

```
def create_gbn_data_header(sender, payload, sequence_number):
    segment_type = "data"
    options = sender.SACK
    length = len(payload)
    assert 0 < length <= 64     # Segments are 64 bytes execpt for the last one.
    hlen = 6                    # Header length without optional header
    num = sequence_number
    win = sender.win
    return GBN(type=segment_type, options=options, len=length,
               hlen=hlen, num=num, win=win)
```

Figure 1: The following code segments show the code that is used to complete the GBN sender's SEND state implementation.

Figure 1 shows our implementation for the GBN sender SEND state. Most of the complexity of the task is setting the header fields correctly. The *create_gbn_data_header* function creates a GBN header:

- line 2: we make sure the sender's packet are considered a data packet.
- line 3: the sender informs the receiver if it is capable of dealing with *selective acknowledgements*
- line 4: we set the payload length.
- line 6: we set the header length. Data packets always have a header length of 6 bytes whereas ACK packets could have longer (SACK) headers,
- line 7: we set the sequence number of the data packet
- line 8: the sender communicates it's sender window. Note: this is not the same as the negotiated maximum number of segments in the send buffer.

ACK_IN State

In the ACK_IN State, we call the *on_ack_in* function to handle the received acknowledgements. The *on_ack_in* function solves the problem of removing the acknowledged packets from a high-level perspective. See Figure 2 for the details:

- line 5: we check if the sequence number corresponding to the previous segment is in the buffer
- line 6: if the sequence number is in the send buffer, we set the sender's *unACK* value to the acknowlegement's *ACK* (next expected segement) value.
- line 8: call *remove_all_acknowledged_messages_from_buffer* to remove all cumulatively acknowledged messages.

Also in Figure 2, the low-level details of the removal of the acknowledged packets from the send buffer is shown. If the acknowledged packet is in the send buffer, we check if we can also remove its predecessors sequentially. To handle sequence number overflows correctly, we implemented in the *previ-*

```python
# Function that is called in the GBNSender's state machine's ACK_IN state.
def on_ack_in(sender, ack, does_receiver_use_sack=0, ack_blocks=[]):
    # If we receive an ack for a message in the send buffer, we update the unack variable and
    # remove all the cumulative acknowledged messages. Also, we reset the ACK counter.
    if previous_sequence_number(ack, sender) in sender.buffer:
        sender.unack = ack
        sender.ack_count = 1
        remove_all_acknowledged_messages_from_buffer(sender)
    elif ack == sender.unack:
        # Increment the ACK counter for duplicated ACKs.
        sender.ack_count = sender.ack_count + 1
    # else: for unexpected ACKs (ACKs for which no data segment was sent), we do not modify the
    # unack variable nor do we reset the ACK counter.

    ...
```

```python
def previous_sequence_number(seq, sender):
    max_sequence_number = 2**sender.n_bits - 1
    return max_sequence_number if seq == 0 else seq - 1
```

```python
# Removes all cumulative acknowledged messages from the sender's buffer based on the sender's
# unack value.
def remove_all_acknowledged_messages_from_buffer(sender):
    prev = previous_sequence_number(sender.unack, sender)
    while prev in sender.buffer:
        del sender.buffer[prev]
        prev = previous_sequence_number(prev, sender)
```

Figure 2: The following code segments show the code that is used to complete the GBN sender's ACK_IN state implementation. Note: the listing of the *on_ack_in* function was truncated by the code that is only relevant for later tasks.

ous_sequence_number function

$$\text{seq}_{\text{previous}} = \begin{cases} 2^{\text{N\_BITS}} - 1, & \text{if seq} = 0 \\ \text{seq} - 1, & \text{otherwise} \end{cases}$$

RETRANSMIT State

```python
# Function that is called in the GBNSender's state machine's RETRANSMIT state.
def on_retransmit(sender):
    if sender.Q_4_4:
        ...
    else:
        for seq, payload in sender.buffer.items():
            send_gbn(sender, payload, seq)
```

Figure 3: The following code segments show the code that is used to complete the GBN sender's RETRANSMIT state implementation. Note: the listing of the *on_retransmit* function was truncated by the code that is only relevant for task 4.4. For the function *send_gbn*, we reused the implementation from the SEND state implementation in Figure 1

.

In the RETRANSMIT State of the Automaton, we call the *on_retransmit* function, which simply resends all the messages with their corresponding sequence number that are in the sender's buffer. See Figure 3 for the code details.

Theoretical question - two times ACK, network condition?

We look now on a situation where it is not true that the data segment #3 was lost, while #4 and #5 made it to the receiver. We say that packets #3 up to #5 are transmitted. The normal Go-Back-N receiver in the first task accepts only segments in-sequence. It acknowledges correctly delivered segments by sending an

ACK segment back. As we are using cumulative ACKs, the ACK contains the sequence number of the next expected data segment.

The response of two times the ACK with packet #3 means that the receiver got the packets up to #2 and receives now two times a packet which differs from packet #3.

One possible reasoning could be, that the receiver gets two times out-of-order packets (for example 5-4-3). While receiving packet #5 and #4, the receiver sends ACKs with #3 to the sender. This doesn't implicate that the packet #3 got lost, but it has been delayed relative to the packets #4 and #5.

Task 4.2: Implement Selective Repeat

4.2.1 Receiver

```
            if ptype == 0:

                # check if last packet --> end receiver
                ...

                # this is the segment with the expected sequence number
                if num == self.next:
                    log.debug("Packet has expected sequence number: %s", num)
                    on_data_in_in_sequence(pkt, self)

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

            else:
                # we received an ACK while we are supposed to receive only
                # data segments
                ...

            # send ACK back to sender
            if random.random() < self.p_ack:
                # the ACK will be lost, discard it
                log.debug("Lost ACK: %s", self.next)

            # the ACK will be received correctly
            else:
                header_GBN = on_data_in_compute_acknowledgment(pkt, self)
                log.debug("Sending ACK: %s", self.next)
                send(IP(src=self.receiver, dst=self.sender) / header_GBN,
                     verbose=0)
```

Figure 4: The figure shows how we integrate our receiver code within the existing framework using the functions *on_data_in_in_sequence*, *on_data_in_out_of_sequence*, and *on_data_in_compute_acknowledgment*.

On the receiver side, in the DATA_IN State of the Automaton, if the received segment corresponds to the expected one, we write the data segment in the output file using the *deliver_packet_to_upper_layer* function, and delete the segment from the buffer.

If the received segment does not correspond to the one we expect, we check if the segment number falls within the receiver buffer with the *allow_packet_to_be_buffered* function, and then call the *buffer_packet* function *buffer_packet* to store the segment in the receiver buffer. To handle buffer overflows correctly in the *allow_packet_to_be_buffered* function, we use the *next_sequence_number* function that computes the overflown value with the modulo operator. Then, we send back the ACK. See Figure 4 on how we integrate our code within the existing framework and Figure 5 for our actual implementation.

We send the ACK to the sender using the *on_data_in_compute_acknowledgement* function, which creates a GBN header (by calling *create_gbn_ack_header* function, and then sends it using the Scapy *send* function.

```python
# For in-sequence packets we deal with delivering the data.
def on_data_in_in_sequence(pkt, receiver):
    num = pkt.getlayer(GBN).num
    payload = bytes(pkt.getlayer(GBN).payload)

    # Deliver current packet.
    deliver_packet_to_upper_layer(payload, num, receiver)

    # Deliver all consecutive packets if they are available in the receiver buffer.
    while receiver.next in receiver.buffer:
        i = receiver.next
        deliver_packet_to_upper_layer(receiver.buffer[i], i, receiver)
        # Remove delivered packets from the receive buffer.
        del receiver.buffer[i]
```

```python
# For out-of-sequence packets we deal with buffering the data.
def on_data_in_out_of_sequence(pkt, receiver):
    num = pkt.getlayer(GBN).num
    window = pkt.getlayer(GBN).win
    payload = bytes(pkt.getlayer(GBN).payload)

    # The sender should never pick nor communicate a window that is larger than the receiver
    window.
    # A window that is larger than half the maximum sequence number puts the reliable transport
    # at risk. Thus, we limit the window size here.
    if window > receiver.win:
        log.error(f"Limiting communicted transmission window from {window} to {receiver.win}. "
                  "Check that the sender picks and communicates the correcct window size.")
        window = receiver.win

    if allow_packet_to_be_buffered(num, window, receiver):
        buffer_packet(payload, num, receiver)
```

```python
# For any packet - in-sequence or not - we deal with the acknowledgement.
def on_data_in_compute_acknowledgment(pkt, receiver):
    use_sack = pkt.getlayer(GBN).options == 1
    if use_sack:
        return create_gbn_sack_header(receiver)
    else:
        return create_gbn_ack_header(receiver)

# Part of the code in this function was copied from the original assignment framework and
# isolated here for easier reuse.
def deliver_packet_to_upper_layer(payload, num, receiver):
    # append payload (as binary data) to output file
    with open(receiver.out_file, 'ab') as file:
        file.write(payload)
    log.debug("Delivered packet to upper layer: %s", num)
    receiver.next = next_sequence_number(receiver.next, receiver)
```

```python
def buffer_packet(payload, num, receiver):
    log.debug("Buffer packet: %s", num)
    receiver.buffer[num] = payload
```

```python
def allow_packet_to_be_buffered(num, window, receiver):
    for i in range(1, window):
        if num == next_sequence_number(receiver.next, receiver, steps=i):
            return True
    return False
```

```python
def create_gbn_ack_header(receiver):
    return GBN(type="ack",
               options=0,
               len=0,
               hlen=6,
               num=receiver.next,
               win=receiver.win)

def next_sequence_number(seq, receiver, steps=1):
    return int((seq + steps) % (2**receiver.n_bits))
```

Figure 5: The implementation details of our receiver buffer code. The *allow_packet_to_be_buffered* shows how we check if the received data packet is one of next expected packets within the receiver window. The *next_sequence_number* function shows how we handle sequence number overflows with the modulo operator.

Theoretical question - buffering every out-of-order segment?

If we store every out-of-order segment and do completely ignore the current sender and receiver window, we face the issue of re-using sequence numbers in an ongoing transmission. If we do have large amount of data to send, we use large quantities of packets. Once, we have used up all the sequence numbers, we have to re-use them (we do have an overflow). If we buffer now everything, we could store delayed packets in the buffer which do not belong to the current state of transmission (previous old packets using same number). Also delayed packets from a different previous transmission (between sender and receiver) could be received and stored, when they have "expected" sequence numbers.

4.2.2 Sender

```python
# Function that is called in the GBNSender's state machine's ACK_IN state.
def on_ack_in(sender, ack, does_receiver_use_sack=0, ack_blocks=[]):
    # If we receive an ack for a message in the send buffer, we update the unack variable and
    # remove all the cumulative acknowledged messages. Also, we reset the ACK counter.
    if previous_sequence_number(ack, sender) in sender.buffer:
        sender.unack = ack
        sender.ack_count = 1
        remove_all_acknowledged_messages_from_buffer(sender)
    elif ack == sender.unack:
        # Increment the ACK counter for duplicated ACKs.
        sender.ack_count = sender.ack_count + 1
    # else: for unexpected ACKs (ACKs for which no data segment was sent), we do not modify the
    # unack variable nor do we reset the ACK counter.

    # If applicable, resend missing data segments based on the selective acknowledgement.
    if sender.SACK:
        ...

    # If applicable, resend data segments based on the selective repeat method.
    elif should_selective_repeat(sender):
        if sender.unack in sender.buffer:
            send_gbn(sender, sender.buffer[sender.unack], sender.unack)
        else:
            log.error(f"Sequence number {sender.unack} not found in send buffer "
                      f"{sender.buffer.keys()} for selective repeat.")
```

```python
def should_selective_repeat(sender):
    return sender.Q_4_2 and sender.ack_count > 0 and sender.ack_count % 3 == 0
```

Figure 6: As extension to the code from Figure 2, the code shown in this figure implements the *selective repeat* method in the GBN sender implementation for Task 4.2.2.

When receiving an ACK packet from the receiver (ACK_IN State), we first check if the previous sequence number's packet is in the buffer. If it is the case, we delete the packet from the buffer, and reset the ACK counter to one. Otherwise, if the *ack* value equals the sender's *unack* value but the corresponding segment is already removed from the buffer, it means that we already deleted it the last time we received an ACK. It implies that we are receiving the same ACK again, and in this case, we increase the ACK counter. As a third case, if the ACK is outside the sender buffer, we have to assume that it is an arbitrarily old ACK that was delayed. In that case, we do not reset the ack counter. See Figure 6) for the code details.

In selective repeat, we want to send the packet again if we received three times the same ACK. This is done with the last condition in the *on_ack_in* function, which sends the packet again if the ACK counter equals three and the mode is selective_repeat (see the *should_selective_repeat* function).

Task 4.3: Implement Selective Acknowledgement

4.3.1 Receiver

The goal of the Selective Acknowledgement (SACK) mechanism is to return a more precise feedback to the sender on which segments were correctly delivered and which not, for the whole window at once.

```
1     fields_desc = [BitEnumField("type", 0, 1, {0: "data", 1: "ack"}),
2                    BitField("options", 0, 7),
3                    ShortField("len", None),
4                    ByteField("hlen", 0),
5                    ByteField("num", 0),
6                    ByteField("win", 0),
7                    ConditionalField(ByteField("block_length", 0), lambda pkt: pkt.hlen >= 7),
8                    ConditionalField(ByteField("block_1_start", 0), lambda pkt: pkt.hlen >= 9),
9                    ConditionalField(ByteField("block_1_size", 0), lambda pkt: pkt.hlen >= 9),
10                   ConditionalField(ByteField("block_2_padding", 0), lambda pkt: pkt.hlen >= 12),
11                   ConditionalField(ByteField("block_2_start", 0), lambda pkt: pkt.hlen >= 12),
12                   ConditionalField(ByteField("block_2_size", 0), lambda pkt: pkt.hlen >= 12),
13                   ConditionalField(ByteField("block_3_padding", 0), lambda pkt: pkt.hlen == 15),
14                   ConditionalField(ByteField("block_3_start", 0), lambda pkt: pkt.hlen == 15),
15                   ConditionalField(ByteField("block_3_size", 0), lambda pkt: pkt.hlen == 15)]
```

Figure 7: In order to support *selective acknowledgement*, *field_desc* of the *GBN* class was updated with conditional fields.

We added 9 conditional fields (corresponding to the optional header). We base our condition on the header solely on the header length because it contains all the information necessary to compute the optional fields – at least for the current protocol. Hypothetical future versions might break our assumptions. But currently, header that is larger than the mandatory part implies that it is an ACK header and that it has selective acknowledgement blocks. See the Figure 7 for our exact conditions. Note that on line 7, we allow communication partners to falsely include the block length variable without setting any *block_start* and *block_size* header fields (that would result in a 7 byte header, whereas this is actually not allowed by the protocol).

As is shown in Figure 8, we again use our *on_data_in_compute_acknowledgement* function to create the GBN header and send it, but this time, as use_sack is True, we call *create_gbn_sack_header* function, which creates a GBN header with the mandatory and the optional fields.

To create the optional fields, we need to know the number of blocks (between 1 and 3), the sequence number of the first segment of a contiguous block that has been correctly received (Left edge, or block_start), and the length of this contiguous blocks (Length of blocks, or block_size). To determine those, we use the *compute_sack_blocks* function. Then, according to the number of blocks, we assign block_start and block_size to the corresponding fields of the GBN sack header.

To compute the block_start and block_size, we first sort sequence numbers in the buffer taking sequence number overflows into account. To this end, we call the *sorted_sequence_numbers* function (see 9 for the detailed function). We fix both block_start and block_end at the first sequence number (line 14, line 15). We iterate over the sorted sequence numbers.
Then, if the next received sequence is the right one, we update block_end to the received sequence number (line 20). Otherwise, it means that it is the end of one contiguous block, so we calculate the block_size (block_start - block_end), and append the start sequence number and size to the blocks array (line 23), and set block_start and block_end to the actual sequence number.
Finally, we return at most three contiguous blocks.

Theoretical question - optional header design

- The receiver could send a GBN header with the sequence numbers of packets which were not correctly received (up to window size). This would be efficient for a situation where we do have few transmission's failures. However, with increasing transmission failures, the efficiency steadily decreases compared to the current implementation.
- Another option would be to transmit the start- and endpoints of a block of packets which were not correctly received (instead of the correctly received ones with Left edge and Length of block in the current implementation). This wouldn't lead to any improvements neither disadvantages compared to the actual SACK implementation.

```
# For any packet - in-sequence or not - we deal with the acknowledgement.
def on_data_in_compute_acknowledgement(pkt, receiver):
    use_sack = pkt.getlayer(GBN).options == 1
    if use_sack:
        return create_gbn_sack_header(receiver)
    else:
        return create_gbn_ack_header(receiver)
```

```
def create_gbn_sack_header(receiver):
    blocks = compute_sack_blocks(receiver)

    block_length = len(blocks)
    hlen = 6 + 3 * block_length
    args = {"type" : "ack",
            "options" : 1,
            "len" : 0,
            "hlen" : hlen,
            "num" : receiver.next,
            "win" : receiver.win}

    if block_length >= 1:
        args["block_length"] = block_length
        args["block_1_start"] = blocks[0].start
        args["block_1_size"] = blocks[0].size

    if block_length >= 2:
        args["block_2_start"] = blocks[1].start
        args["block_2_size"] = blocks[1].size

    if block_length == 3:
        args["block_3_start"] = blocks[2].start
        args["block_3_size"] = blocks[2].size

    return GBN(**args)
```

```
def compute_sack_blocks(receiver):
    sequence_numbers = sorted_sequence_numbers(receiver)
    if len(sequence_numbers) == 0:
        return []

    class SackBlock:
        def __init__(self, start, end):
            self.start = start
            self.size = sequence_numbers_difference(end, start, receiver) + 1

    blocks = []

    # Start the first block.
    block_start = sequence_numbers[0]
    block_end = sequence_numbers[0]

    for seq in sequence_numbers[1:]:
        if seq == next_sequence_number(block_end, receiver):
            # Extend the current block by shifting the block_end to the next sequence number.
            block_end = seq
        else:
            # Non-consecutive seqence numbers indicate that current block is complete.
            blocks.append(SackBlock(block_start, block_end))

            # Start a new block.
            block_start = seq
            block_end = seq

    # The lack of further seqeunce numbers in the buffer indicates that last block is complete.
    blocks.append(SackBlock(block_start, block_end))

    # Truncate the output to the number of blocks that can be transmitted by the protocol.
    if len(blocks) > 3:
        return blocks[:3]
    return blocks
```

Figure 8: Code for the computation of the *selective acknowledgement* blocks. In the function *on_data_in_compute_acknowledgement* the SACK negotiation part of the receiver is implemented. Further see Figure 9 for lower level functions that are used in this code.

```
# Sorts the sequence numbers whereas the oldest sequence number is placed first.
# The sorting takes sequence number overflows into account.
def sorted_sequence_numbers(receiver):
    if (len(receiver.buffer) == 0):
        return []

    # We could just pick any sequence number that is in the buffer for the sorting job.
    # However, using the minimum makes debugging (and understanding) of the code easier.
    base_seq = min(receiver.buffer.keys())

    # As the sequence numbers are not sorted directly by their values, a sort function is
    # defined. With the value of base_seq from above, 0 is associated to the smallest sequence
    # number. If there is an overflow, sequence numbers from before the overflow are associated
    # to negative numbers.
    def sort_value(seq):
        return sequence_numbers_difference(seq, base_seq, receiver)

    sorted_seqs = sorted(receiver.buffer.keys(), key=sort_value)
    return sorted_seqs
```

```
# Like a normal difference seq1 - seq2 but taking sequence numbers overflows into
# account. From the infinite number of solutions the one with the smallest
# absolute value is returned.
def sequence_numbers_difference(seq1, seq2, receiver):
    no_overflow_diff = seq1 - seq2
    n = 2**receiver.n_bits

    if no_overflow_diff >= 0:
        overflow_diff = no_overflow_diff - n
    else:
        overflow_diff = no_overflow_diff + n

    if abs(no_overflow_diff) <= abs(overflow_diff):
        return no_overflow_diff
    else:
        return overflow_diff
```

```
def next_sequence_number(seq, receiver, steps=1):
    return int((seq + steps) % (2**receiver.n_bits))
```

Figure 9: Low-level details for the helper functions that are used for computation of the *selective acknowledgement* blocks in Figure 8. Of particular interest is the *sorted_sequence_numbers* function that is a corner stone of our algorithm to correctly compute the selective acknowledgement blocks in the presence of sequence number overflows.

4.3.2 Sender

```
            ###############################################################
            # TODO:                                                       #
            # remove all the acknowledged sequence numbers from the buffer #
            # make sure that you can handle a sequence number overflow    #
            ###############################################################
            on_ack_in(sender=self, ack=ack, does_receiver_use_sack=pkt.getlayer(GBN).options,
                      ack_blocks=extract_selective_acknowledgment_blocks(pkt))
```

```
# Function that is called in the GBNSender's state machine's ACK_IN state.
def on_ack_in(sender, ack, does_receiver_use_sack=0, ack_blocks=[]):

    ...

    # If applicable, resend missing data segments based on the selective acknowledgement.
    if sender.SACK:
        if does_receiver_use_sack:
            for key, payload in unacknowledged_messages(sender, ack_blocks).items():
                send_gbn(sender, payload,  key)

    ...
```

Figure 10: The code listings show how we integrate our *selective acknowledgement* sender functions, *extract_selective_acknowledgement_blocks* and *unacknowledged_messages*, in the existing framework as well as our sender code from Task 4.1 (Figure 2). Note the variables does_receiver_use_sack and sender.SACK that implement part of the SACK negotiation on the sender side.

We have to define the same GBN header as on the receiver side, so that the sender can decode the incoming bytes (Figure 7).

When receiving an acknowledgement from the sender (we call the *on_ack_in* function). If sender.SACK is True, we check for the sequences number of the packet that were not received according to the selective acknowledgement using the *unacknowledged_messages* function, and send those packets again (see Figure 10).

In the *unacknowledged_messages* function (Figure 11), we extract all the packets which are in the buffer, and store them in the output table. Then we remove from this output table the packets that have been correctly acknowledged. That is all the packets between the start index and the start+size index that we got back from the optional fields of the GBN SACK header (see Figure 11 on how we extract those information from the bytes that the receiver sent). Finally, we also remove the packets which are after the last contiguous block, as we do not know if it has been correctly received or not, and thus do not want to retransmit them yet.

Depending on the block_length, we extract the Left edge(s) and the Length of block(s) (block_start and block_size) from the bytes we receive, and store them in ack_blocks.

Whether to use SACK or not is negotiated between sender and receiver as follows:

- If the sender wants to use SACK, it sets the option field in the data header to 1 (Figure 1)
- If the receiver learns that the sender wants to use SACK, it may generate a SACK header if applicable (our receiver does). If the receiver decides to use SACK, it should also set the option field in the ACK headers to 1.
- Now the sender learns from the receiver that it uses SACK and resends the dropped packets accordingly.

Theoretical question - reduce number of retransmitted segments

When receiving a SACK, the sender could use an individual time-stamp in the unacknowledged-buffer to label packets which have been unacknowledged at this moment and fall outside of the contiguous blocks. Then it could resend this labeled packets and wait a certain amount of time ignoring further SACK

```python
# Returns a list of SACK blocks
def extract_selective_acknowledgment_blocks(pkt):
    # The receiver communicates that it does not use SACK. Then we don't use SACK even if
    # the SACK blocks are present because it is more robust to not use SACK in presence of
    # a whacky receiver.
    if pkt.getlayer(GBN).options == 0:
        return []

    # If the receiver communicates to use SACK but no SACK block fits into the header, return
    # no SACK blocks.
    hlen = pkt.getlayer(GBN).hlen
    if hlen < 9:
        return []

    class SackBlock:
        def __init__(self, start, size):
            self.start = start
            self.size = size

    # In case of conflicting hlen and block_length variables, be conservative and choose
    # the lower number of blocks.
    block_length = min(pkt.getlayer(GBN).block_length, int((hlen - 6) / 3))

    ack_blocks = []
    if block_length >= 1:
        ack_blocks.append(SackBlock(pkt.getlayer(GBN).block_1_start,
                                    pkt.getlayer(GBN).block_1_size))
    if block_length >= 2:
        ack_blocks.append(SackBlock(pkt.getlayer(GBN).block_2_start,
                                    pkt.getlayer(GBN).block_2_size))
    if block_length == 3:
        ack_blocks.append(SackBlock(pkt.getlayer(GBN).block_3_start,
                                    pkt.getlayer(GBN).block_3_size))
    return ack_blocks
```

```python
# Compute unacknowledged messages from selective acknowledgements blocks.
def unacknowledged_messages(sender, ack_blocks):
    if len(ack_blocks) == 0:
        return {}

    # Copy all possible segements for resending into the output variable. Afterwards, remove
    # what is already known to be acknowledged by the SACK blocks.
    output = sender.buffer.copy()

    # Do not retransmit messages inside the ack blocks.
    for block in ack_blocks:
        seq = block.start
        for i in range(0, block.size):
            if seq in output:
                del output[seq]
            else:
                log.error(f"Could not find {seq} in send buffer {sender.buffer.keys()} "
                          f"when handling the acknowledgement blocks {ack_blocks}.")
            seq = next_sequence_number(seq, sender)

    # Do not retransmit messages after the last ack block.
    seq = next_sequence_number(ack_blocks[-1].start, sender, steps=ack_blocks[-1].size)
    while seq in output:
        del output[seq]
        seq = next_sequence_number(seq, sender)

    return output
```

```python
def next_sequence_number(seq, sender, steps=1):
    return int((seq + steps) % (2**sender.n_bits))
```

Figure 11: Implementation of *selective acknowledgement* sender functions for Task 4.3.2, *extract_selective_acknowledgement_blocks* and *unacknowledged_messages*

messages. After this certain time period, the sender could look at the unacknowledged-buffer and resend all the packets with the right time-stamp.

Task 4.4: Bonus question: Implement congestion control

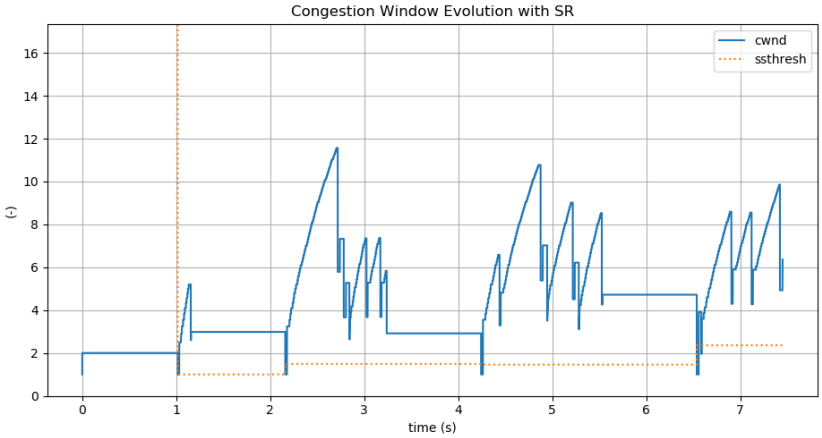

Figure 12: Evolution of the congestion window and sstresh. Our evolution of the CWND is not as linear as in the text book. We hypothesise that the cause is that our network delay is not dominated by the propagation delay as it is in the example graphs.

Note: We had to include "import time" in order to generate the CWND-plot.

Generally, the receiver window (RWND) advertises the amount of data that the destination side can receive. This variable is advertised through the header and indicates the amount of data the receiver can handle and store in the buffer.

The congestion window (CWND) limits the amount of data that can be send to the network before receiving an ACK. The CWND is lower or equal to the sender window. Together, the RWND and CWND are parameters used to steer data flow in TCP connections. The CWND and RWND have a certain dependence. The RWND enables the CWND to determine the amount of data it can reliably transmit without an ACK. Once received, the data is stored at the receivers buffer and the receiver sends an ACK or multiple ACKS depending on the amount of received data.

The CWND size keeps growing up to the maximum RWND, or until the data flow reaches the network's limit. The amount of sending data is always bound to the CWND. Even when the RWND is large, the CWND might be considerably smaller - this is the case when the network has limited throughput.

Congestion is detected by receipt of a duplicate ACK or timeout signal. The sender decreases the send rate by decreasing the maximum CWND (= ssthresh) size by a factor determined by design choice (we use 1/2). The maximum amount of data the sender can transmit is the lower one of the two windows.

Our mechanisms to increase and decrease the CWND:

- At the beginning, the maximum CWND (= ssthresh) is set to approximately infinite (in our implementation a high value of $2^{\mathrm{N\_BITS}}$).
- We use "slow start" congestion control, as explained in the lecture, to increase the CWND exponentially to reach the maximum transfer rate as fast as possible. When the network is able to handle the data without errors, the CWND is increased, but limited by the maximum RWND.
- When reaching the ssthresh, we increase the CWND linearly to avoid congestion.
- Receiving three duplicated ACKs, we multiply the CWND by 1/2.
- Reaching a timeout, we set ssthresh to CWND/2 and start with CWND = 1 again.