Communication Networks
Transport project

Group 26
Jérémie Amstutz
Joseph Gallmann
Tiancheng Chen

June 2021

1 Go-Back-N

1.1 Sender

1.1.1 SEND state

The SEND state is where packets are being sent. To prepare for sending a packet we need first the create a new GBN header. This header can then be passed to the sender after the IP, but before the payload.

This header contains multiple fields that we need to set up :

- The option field is set to `self.SACK`, which is 1 when Selective Acknowledgement is used
- `len(payload)` returns number of bytes of the payload, which is essential for the receiver to determine the end of transmission.
- The proper window size, which is the min of sender and receiver's window size.

After sending the packet, `self.current` increase by 1 and we use the modulo to ensure it stays in the valid range of sequence number $[0, 2^{self.nbits})$. WE DID NOT DO THAT

```python
# send with the proper window size
proper_win = min(self.win,self.receiver_win)
# set the header for data segment, options=1 of SACK is used.
header_GBN = GBN(type="data",
               options=self.SACK,
               len=len(payload),
               hlen=6,
               num=self.current,
               win=proper_win)
send(IP(src=self.sender,dst=self.receiver) / header_GBN / payload)
# sequence number of next packet
self.current = int((self.current + 1) % 2**self.n_bits)
```

1.1.2 ACK IN state

When an ACK is received, it enters this ACK IN state. Based on this ACK, we can deduct which packet was received, update the `self.unack`, and remove the packet from the buffer.
To make the protocol robust against ack lost, we iterate through the possible range of number, that is $[unack - possible\_win, unack)$, use modulo % to deal with number overflow. Pop out the payload of the number if it's in the buffer. `possible_win` will be explained below, normally it's `min(self.win, self.receiver_win)`.

```python
# update unack
self.unack = ack
# remove acked packets from buffer (may be multiple ones)
for index in range(self.unack-possible_win, self.unack):
    # deal with overflow
    index_mod = index % 2**self.n_bits
    if index_mod in self.buffer:
        self.buffer.pop(index_mod)
```

Considerations:

1. **Out-of-date acks**

```python
# deal with delayed ack
if self.unack + possible_win + 1 > 2**self.n_bits:
    #overflow
    effective_ack = ack >= self.unack or ack < self.unack + possible_win + 1
else:
    # no overflow
    effective_ack = ack >= self.unack and ack < self.unack + possible_win + 1

# Determine if the ack is out of date.
if effective_ack == 0:
    # back to SEND state
    raise self.SEND()
```

It's possible in real world that delayed acks exist. Though in test scripts only ack loss is tested. We still include this part of code.
We should only accept ack if it's in the desired range $[unack, unack + possible\_win + 1)$. Otherwise this ack should be discarded as it contains no new information about the receiver's state. `effective_ack` is used to track this, and is set to 0 if this ack is out-of-date.

2. **Proper window size**
Think about this case: if sender window is much larger than receiver window, and RTT is large. Then the sender will send all packets in window size min(self.win, self.receiver_win), which is equal to sender window size at the beginning (because of the way we initialize `self.receiver_win`).
Problem is raised when sender receives acks. It deletes acked packets from the buffer. We use "for index in range(self.unack-min(self.win, self.receiver_win), self.unack):" to iterate in the buffer. However, the current min(self.win, self.receiver_win) is smaller (updated when receiving the first ack), which might cause some packets not removed from buffer. The test scripts does not consider this. But it's an actual problem.
The following if clause is used to identify if sender receives the receiver window size.

```python
# if window size set correctly and safely (win_safe_to_use_min=1), ignore the if clause.
if self.win_safe_to_use_min == 0:
    if self.receiver_win > self.win:
        # if receiver window is larger, then no problem in the first window size packets.
        self.win_safe_to_use_min = 1
    elif ack > self.win:
        # at the first time ack > sender window, certainly min(self.win, self.receiver_win)
            is safe, see the report for detailed explanation
        self.win_safe_to_use_min = 1

if self.win_safe_to_use_min == 1:
    possible_win = min(self.win, self.receiver_win)
else:
    # use sender window, actually same as max(self.win, self.receiver_win)
    possible_win = self.win
```

Solution: `self.win_safe_to_use_min` and `possible_win` are used to determine the correct window size. If the actual receiver window size is larger than sender window size, then the problem is avoided. If not, the first time sender receives $ack > self.win$, it's safe to use the min window. Because this indicates sender must have received at least an ack for payload in the first sender window. So that 1. `self.receiver_win` is correctly set, no payload will be sent until free space in window. 2. The acked payload in the front are popped out from buffer.

1.1.3 RETRANSMIT state

We retransmit all payloads in the buffer.

```
# loop through all the packets inside the buffer
for index, payload in self.buffer.items():
    header_GBN = GBN(type="data",
        options=self.SACK,
        len=len(payload),
        hlen=6,
        num=index,
        win=min(self.win,self.receiver_win))
    send(IP(src=self.sender,dst=self.receiver) / header_GBN / payload)
```

Theoretical question
Assume the sender just transmitted data segment 3, 4 and 5 and received as response two times an ACK with number 3. You conclude that data segment 3 was lost and 4 and 5 reached the receiver. Describe, for example, a network condition under which this conclusion is not true.
Answer
Sender receives 2 acks, means receiver received at least two segments. It is possible that segment 3 is on the fly and segment 4 is received twice (duplicated segment), segment 5 on the fly or lost. Then the sender will receive two ack 3 even though segment 3 is not lost (yet).

2 Selective Repeat

2.1 Receiver

Same as in GBN, the sender does not know the window size of receiver at the beginning, it might send more segments than actual receiver window. Our design choice is to buffer all of them.

```
send_win = pkt.getlayer(GBN).win # the sender window
if send_win > self.win:
    # sender does not know the correct window size yet, use a larger window to buffer
    possible_win = max(send_win, self.win)
else:
    # sender knows the correct window size which is min(send_win, self.win)
    possible_win = min(send_win, self.win)
```

Then if

1. num is equal to `self.next`:
   Deliver the segment, increase `self.next` by 1, and loop in the buffer to see if there's out-of-order segments we can restore next.

2. num is not equal to `self.next`:
   If the num is in the range $[self.next + 1, self.next + possible\_win)$, then buffer it.

3. For the rest conditions, we discard the out-of-window segment.

```python
# detect if we need to buffer, if num is within desired range, num_in_pwin=1 else 0.
if self.next + possible_win > 2**self.n_bits:
    num_in_pwin = num >= self.next + 1 or num < (self.next + possible_win) % 2**self.n_bits
else:
    num_in_pwin = num >= self.next + 1 and num < self.next + possible_win
if num == self.next:
    # the packet is exactly what receiver expect
    # check if last packet --> end receiver
    # append payload (as binary data) to output file
    # move the self.next to next expected packet
    ...
    while self.next in self.buffer:
        # the next expected packet is buffered
        pl = self.buffer.pop(self.next)
        # check if last packet --> end receiver
        # append payload (as binary data) to output file
        # move the self.next to next expected packet
        ...
elif num_in_pwin:
    # buffer this payload
    self.buffer[num] = payload
else:
    # Discard packets that we don't need
    ...
```

Theoretical question:
Assume you have a receiver with unlimited buffer space. Why is it not always beneficial to buffer every out-of-order segment, completely ignoring current sender and receiver windows?
Answer
Because segments maybe duplicated and delayed during transmission.
Consider following case: assume that the range of valid sequence number is small, say 32. The TIMEOUT value (time before packets are retransmitted) is large, RTT is low. Segment k is duplicated during transmission into k_1,k_2. k_1 is received in time, k_2 is delayed. Then when k_2 arrives, the receiver may think this packet is the 32nd packet after k_1 and buffer it (receiver ignores window sizes). As a result, it may deliver a segment of wrong data.

2.2 Sender

2.2.1 Fast retransmit in case of duplicated ACKs

Notice that in our design, Q4.4 and Q4.2 shares the same duplicate ack counter. So that congestion control is available whether selective repeat is on or not.

First we determine if the ack is in the desired range $[self.current - possible\_win, self.current)$. If not, this out-of-window ack is **not considered**. (The definition of `possible_win` is in previous sections, normally it's min of sender and receiver window sizes.)
Use `self.prev_ack` to record the previously received ack.
Then if

- The ack number equals to `self.prev_ack`
  Add the counter by 1. If the counter reaches 3, then resend the segment and set the `self.prev_ack` to -1, so that it goes to the "not duplicated" else clause when receiving next valid ack.
- In case of "not duplicated":
  We set `self.prev_ack` to this ack, and counter to 1.

```python
# Q4.2 and Q4.4 share the same duplicate ack counter
if self.Q_4_2 == 1 or self.Q_4_4 == 1:
    # determine if the ack is in the desired range [self.current-possible_win, self.current)
    # If yes, ack_in_win = 1, else 0
    if self.current < possible_win:
        ack_in_win = ack >= (self.current-possible_win) % 2**self.n_bits or ack < self.current
    else:
        ack_in_win = ack >= self.current-possible_win and ack < self.current
    if ack_in_win == 1:
        if ack == self.prev_ack:
            # duplicated ack
            self.duplicated_times += 1
            # branching of Q4.4 omitted here
            ...
            if self.Q_4_2 == 1:
                # resend if duplicated = 3
                if self.duplicated_times == 3:
                    pl = self.buffer[ack]
                    header_GBN = GBN(type="data",
                        options=0,
                        len=len(pl),
                        hlen=6,
                        num=ack,
                        win=min(self.win,self.receiver_win))
                    send(IP(src=self.sender,dst=self.receiver) / header_GBN / pl)
                    # reset record
                    self.prev_ack = -1
                    self.duplicated_times = 1
        else:
            # not duplicated, reset record
            self.prev_ack = ack
            self.duplicated_times = 1
```

3 Selective Acknowledgment

3.1 Receiver

3.1.1 Design of SACK header

We determine the conditional fields added to our header by the header length. If the header length=6, no additional fields. If header length > 6, then the header has at least one SACK block. If header length > 9, at least two SACK blocks. If header length > 12, all three SACK blocks exist. As long as we set the expected header length, the SACK fields should be correctly added.

```python
name = 'GBN'
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("block_len",0), lambda pkt:pkt.hlen>6),
               ConditionalField(ByteField("left_1",0), lambda pkt:pkt.hlen>6),
               ConditionalField(ByteField("len_1",0), lambda pkt:pkt.hlen>6),
               ConditionalField(ByteField("pad1",0), lambda pkt:pkt.hlen>9),
               ConditionalField(ByteField("left_2",0), lambda pkt:pkt.hlen>9),
               ConditionalField(ByteField("len_2",0), lambda pkt:pkt.hlen>9),
               ConditionalField(ByteField("pad2",0), lambda pkt:pkt.hlen>12),
               ConditionalField(ByteField("left_3",0), lambda pkt:pkt.hlen>12),
               ConditionalField(ByteField("len_3",0), lambda pkt:pkt.hlen>12)]
```

3.1.2 Generation of SACK header

The `self.support_SACK` is for convenience of development. It's set to 1 after we have implemented SACK. First we loop through index in desired range $[self.next+1, self.next+possible\_win)$, (modulo to prevent overflow), if the index is in the buffer, then:

- If num in current block, that is, previous num in the block = num -1 (overflow case considered): Update the previous num in the block to the num. Update the according block length.
- Else if number of blocks < 3, meaning there's space for new header: Number of blocks increase by 1, Update the left edge and length of corresponding block.

At last, calculate the header length as $6 + 3 \times$number of blocks and send the ack.

```
# default header length
header_length = 6
# initialize some parameters needed to construct header.
num_blocks = 0
left_edge_arr = [0,0,0] # the starting element of blocks
len_block_arr = [0,0,0] # the length of blocks
use_Sack = 0 # determine the option field in header
sender_SACK = pkt.getlayer(GBN).options
# use sack if both sender and receiver support
if sender_SACK == 1 and self.support_SACK == 1:
    use_Sack = 1
    send_win = pkt.getlayer(GBN).win
    # set to -5 to make it small enough. No special meaning for -5.
    prev_in_block = -5
    for i in range(self.next + 1, self.next + possible_win):
        # care of overflow
        packet_num = i % 2**self.n_bits
        if packet_num in self.buffer:
            # consecutive if num is 1 larger than previous one or num=0 and previous one =
                2**self.n_bits -1
            if prev_in_block == packet_num -1 or (packet_num == 0 and prev_in_block ==
                2**self.n_bits -1):
                # update the previous element number in the same block
                prev_in_block = packet_num
                # length of the block++ (-1 because array index start from 0)
                len_block_arr[num_blocks-1] += 1
            elif num_blocks < 3:
                # new block, so increase the number of blocks
                num_blocks += 1
                left_edge_arr[num_blocks-1] = packet_num
                len_block_arr[num_blocks-1] += 1
                prev_in_block = packet_num
    # calculate the header length based on number of blocks
    header_length += 3 * num_blocks
# set header
header_GBN = GBN(type="ack",
                 options=use_Sack,
                 len=0,
                 hlen=header_length,
                num=self.next,
                win=self.win,
                block_len=num_blocks if num_blocks >=1 else None,
                left_1=left_edge_arr[0] if num_blocks >=1 else None,
                len_1=len_block_arr[0] if num_blocks >=1 else None,
                pad1=0 if num_blocks >=2 else None,
                left_2=left_edge_arr[1] if num_blocks >=2 else None,
                len_2=len_block_arr[1] if num_blocks >=2 else None,
                pad2=0 if num_blocks >=3 else None,
                left_3=left_edge_arr[2] if num_blocks >=3 else None,
                len_3=len_block_arr[2] if num_blocks >=3 else None)
send(IP(src=self.receiver, dst=self.sender) / header_GBN,
    verbose=0)
```

Theoretical question
Describe two other optional header designs that the receiver could use to inform the sender of its current buffer state
Answer:

1. We could indicate the left edge (sequence number of the first segment of the block) and the right edge (sequence number of the last segment of the block) for each block. This design is quite similar to our implementation. The main difference is that the sender need to check if the right edge is after or before the left edge (because of wrapping).

2. We could indicate the sequence numbers of all buffered segments. This design is better suited for really small blocks (less or equal to 2 segments). Otherwise, it consumes a lot of space in the header.

3.2 Sender

The sender has the same SACK header design as stated in section 3.1.1. Sender knows number of blocks in SACK header by interpreting the header length field. Then:

1. If header length > 6, meaning first block exists.
   Add segment in the range [self.unack , left edge of 1st block) to the list to send. The unack segment is inside because receiving segment of larger num indicates the unack one might be lost.

2. If header length > 9, meaning second block exists.
   Add segment in range [left edge of 1st block + length of 1st block , left edge of 2nd block) to the list.

3. If header length > 12, meaning thirid block exists.
   Add segment in range [left edge of 2nd block + length of 2nd block , left edge of 3rd block) to the list.

At last, send all segments with num in the list.

```
if self.SACK == 1:
    header_len = pkt.getlayer(GBN).hlen
    send_list = []
    if header_len > 6:
        # resend packets in range [self.unack , left edge of 1st block)
        first_elem = self.unack
        if pkt.getlayer(GBN).left_1 < first_elem:
            # overflow
            send_list = list(range(first_elem, 2**self.n_bits))
            send_list.extend(range(0, pkt.getlayer(GBN).left_1))
        else:
            # no overflow
            send_list = list(range(first_elem, pkt.getlayer(GBN).left_1))

    # between first and second block
    # [left edge of 1st block + length of 1st block , left edge of 2nd block)
    if header_len > 9:
        first_elem = (pkt.getlayer(GBN).left_1 + pkt.getlayer(GBN).len_1) % 2**self.n_bits
        if pkt.getlayer(GBN).left_2 < first_elem:
            # overflow
            send_list.extend(range(first_elem, 2**self.n_bits))
            send_list.extend(range(0, pkt.getlayer(GBN).left_2))
        else:
            # no overflow
            send_list.extend(range(first_elem, pkt.getlayer(GBN).left_2))

    # between second and third block
    # [left edge of 2nd block + length of 2nd block , left edge of 3rd block)
    if header_len > 12:
        first_elem = (pkt.getlayer(GBN).left_2 + pkt.getlayer(GBN).len_2) % 2**self.n_bits
        if pkt.getlayer(GBN).left_3 < first_elem:
            # overflow
            send_list.extend(range(first_elem, 2**self.n_bits))
            send_list.extend(range(0, pkt.getlayer(GBN).left_3))
        else:
            # no overflow
            send_list.extend(range(first_elem, pkt.getlayer(GBN).left_3))

    for idx in send_list:
        payload = self.buffer[idx]
        header_GBN = GBN(type="data",
                        options=self.SACK,
                        len=len(payload),
                        hlen=6,
                        num=idx,
                        win=min(self.win,self.receiver_win))
        send(IP(src=self.sender,dst=self.receiver) / header_GBN / payload)
```

Theoretical question
As you probably realized our SACK implementation generates many (unnecessary) packets as the sender is retransmitting unacknowledged packets for each received SACK header. Explain a possible implementation which uses the information from the SACK header, but reduces the number of retransmitted segments.
Answer MAYBE
We should add a timeout to all the segments that we are sending, and not resend them until the end of the timeout.
To be more specific, we maintain a dict to memory the previous SACK information. The dictionary is initialized empty. Set a threshold as the interval to send same segment. When we receive a new SACK header, we generate a list of segments numbers that need to send
Loop through the list:

1. If number is already in dict:
   Increase the value of this number by 1. If the value reaches threshold, resend, and set the value to 0.

2. If number is not in dict, meaning the segment is newly added:
   Add an entry {number:0} to the dict.

Loop through the dict:

1. If number is not in the list, meaning the segment is received:
   Delete the corresponding entry in dict.

4 Congestion Control

Our congestion control only interact with sender. If congestion control is turned on, the sender window is set to min(self.cwnd, self.win, self.receiver_win).
We designed the congestion control to work with any possible setups.(4.2 on & 4.3 off, 4.2 off & 4.3 on, 4.2 off & 4.3 off).
The sacrifice is some readability because Q4.4 share the same duplicate ack counter with Q4.2.

4.1 Mechanism

We adapted the congestion control algorithm from class and did two modifications. When receiving duplicate acks ≥ 3 times, we set the cwnd = $\max(1, ssthresh)$, so that cwnd is always ≥ 1, and we set dup_ack to 1 so that if we receive the same ack many times, the cwnd won't decrease too fast.

```
Initially:
cwnd = 1.0
ssthresh = inf
New ACK received:
if $cwnd < ssthresh$ then
    cwnd = cwnd + 1
else
    cwnd = cwnd + 1.0/cwnd
end
dup_ack = 1
Timeout:
ssthresh = cwnd / 2.0
cwnd = 1
Duplicate ACKs received:
dup_ack ++
if $dup\_ack \geq 3$ then
    ssthresh = cwnd / 2.0
    cwnd = $\max(1, ssthresh)$
    dup_ack = 1
end
```

Algorithm 1: Congestion control

4.2 Experiment

4.2.1 Setup:

We record CWND size and corresponding time.
Test with `start_local.sh`. Parameters: NBITS=8, IN_FILE=ETH_logo.png, DATA_L=0.02, ACK_L=0.
We test congestion control under original GBN, selective repeat and SACK.

4.2.2 Result:

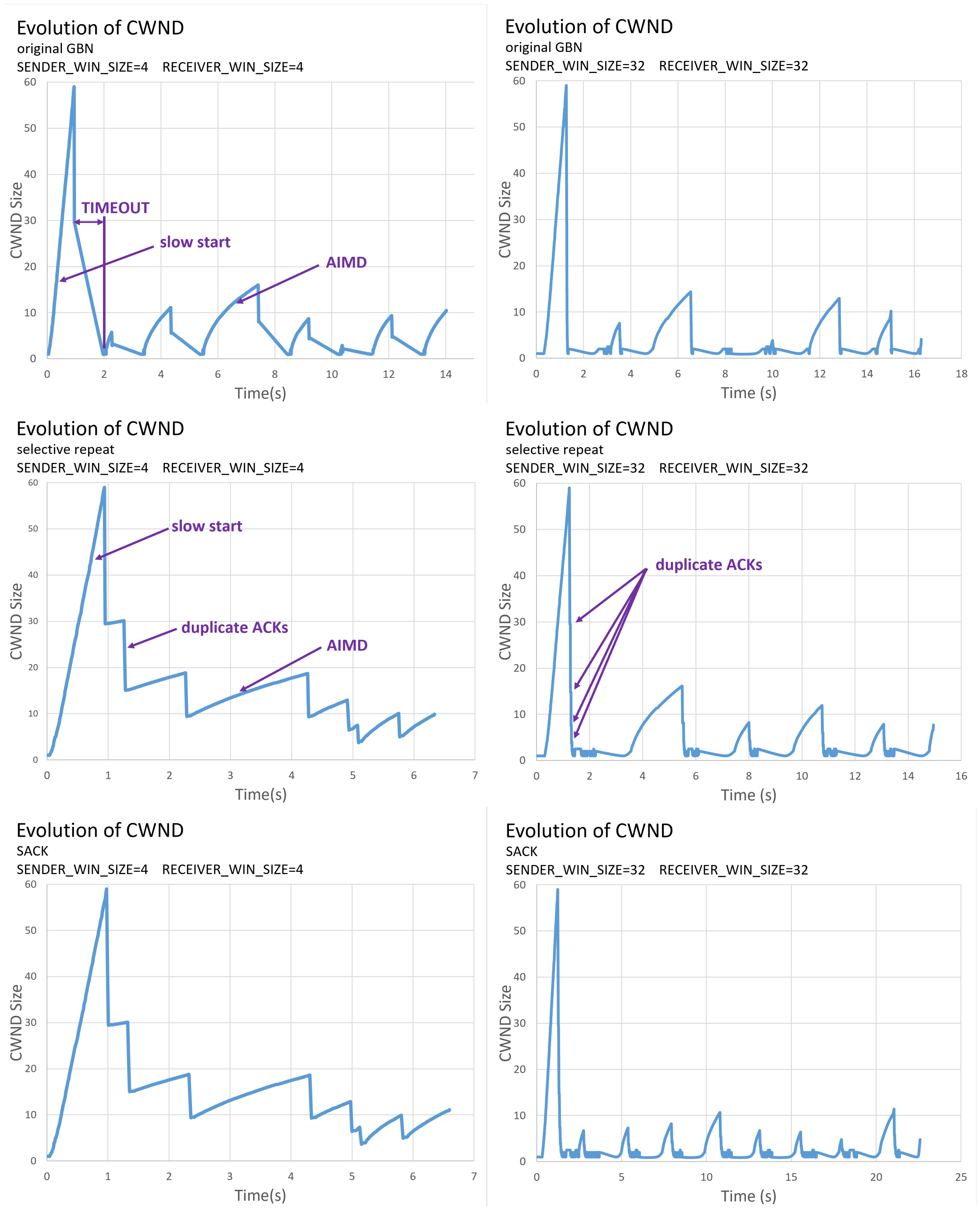

Figure 1: Left: small RWND. Right: large RWND.

4.2.3 Discussion:

- Small window size:
  When RWND is set to 4, the CWND does not affect actual sending window at most time.
  - Small slow start threshold in original GBN.
    In the original GBN, there's no resend mechanism. The sender will receive several duplicate acks (in our case, 1) and then reaches TIMEOUT. By examining the raw data, we found there's a TIMEOUT between two duplicate ack signals. According to our algorithm, the ssthresh will be 1 afterward. This pattern repeats till the end of transmission.
  - Growth rate of slow start and AIMD period.
    The growth rate is as expected. In theory, CWND grows exponentially in Slow start period and linear AIMD period regarding to transmission rounds. However, we record the data with axis time (second). It's obvious that CWND increase by 1 per received ACK in slow start. The log-shape in AIMD is because the update rule CWND = CWND + 1 / CWND. The CWND increases when new ACK is received. So 1/CWND, the added term, becomes smaller.
  - Shortened transmission time with selective repeat and SACK.
    With duplicate ack detection in selective repeat and additional header information in SACK, the sender resend the possibly lost segments. The transmission time is cut by half comparing to original GBN.
- Large window size:
  When RWND is set to 32, the CWND affects actual sending window at most time.
  - Resend mechanism loses effect.
    From raw data, we conclude that low transmission efficiency is because the the count limit of duplicate ack is too small for large window size. The automation sent 32 segments then waited for ACKs. Early received ACKs are stored in buffers not accessible to us. For example, if segment 5 is lost, then the sender will receive 20 or more ACK 5 until receiver respond to the resend. Our algorithm decrease CWND by half when the duplicate counter is 3, so the CWND would decrease to 1, which limits the efficiency of transmission.

4.2.4 Further improvements:

- The threshold for duplicate ACK counter (in our algorithm it is fixed to 3) should be in line with RWND.
  If RWND is large while the counter threshold is small, then a bunch of duplicate ACKs will decrease CWND to 1 as discussed above.