CommNet Routing Project

Laurin Goeller Lukas Meier Marco Hassan

2020-05-16

Contents

1 Go Back N

To include in your report: Write a short paragraph for each of the three missing parts which explains how you implemented the required functionality. Explain in detail, how you handle a sequence number overflow.

1.1 Send State

We want to send a "data" packet, header length is from definition 6, num is the current number we are in, encoded in self.n_{bits} number of bits. The window size is determined by self.win and the legth depends on the length of the payload, so we calculate the length of it. (At most 64 bytes, last packet usually smaller) Then we send the packet via the command "send()", as attributes we add the sender/receiver IP addresses, the defined header and the payload.

```
header_GBN = GBN(type='data', hlen= 6, num=self.current,
                 win=self.win,len = len(payload))

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

Next we calculate the new packet number. This is done by adding 1 to the old packet number, but then we have to calculate the modulo of the number as we have limited bits to encode the sequence number. So if an overflow happens e.g. the next current number is bigger than $2^{n}{}_{bits}$ we calculate the modulo of this and get the new sequence number. Then we go the send state to send the defined packets.

```
self.current = int((self.current + 1) % 2**self.n_bits)

raise self.SEND()
```

1.2 Ack-IN State

In this state we first copy the received ack to prevent errors as we want to modify it. Then we test if an overflow happened at the receiver side. If yes then we add $2^{self.n}{}_{bits}$ to make sure the low numbers are included in the next function. Then we get the numbers of all items that are in the corresponding window size from the received ack and the local unack. This function also handles overflows as it calculates automaticaly the correspond overflow numbers. E.g. local unack is 13 and ack is 2 we get `window_items = [13, 14, 15, 0, 1]` if $n_{bits} = 4$.

```
ack_local = ack
if (ack < self.unack):
    ack_local += 2 ** self.n_bits
# items in window
window_items = [i % 2**self.n_bits for i in range(self.unack, ack_local)]
```

Then we delete all the items from $windows_{items}$ list that are in the self.buffer via the function "del". If this is done we set the unack = ack as now the expected packet is the last not acknowlegded packet. After that we go back to the send state.

```
    for item in window_items:
        if item in self.buffer:
            del self.buffer[item]
            log.debug("Sender unbuffered: %s", item)


    self.unack = ack

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

1.3 Retransmit

After a timeout occured we have to retransmit all not acknowledged packets. This is done by creating a new packet header with the data stored in the buffer. The header and payload and receiver/transmiter addresses are merged together to one packet which is then transmited again. Usually no error should occur but for safety all errors are catched. After we retransmitted every packet in the buffer we go back to the transmit state and wait for the ACK's.

```
for item in self.buffer.keys():
    try:
        header_GBN = GBN(type='data', hlen= 6, num=item, win=self.win, len = len(self.buffer[item])) # item = self.current line 142
        send(IP(src=self.sender, dst=self.receiver) / header_GBN / self.buffer[item])
        log.debug("Retransmitting %s after the timeout", item)

    except:
        log.debug("Retransmission Error at number %s", item)

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

1.4 Theoretical Question

> In addition, answer the following theoretical question: Assume the sender just transmitted data segment 3, 4 and 5 and received as response two times an ACK number 3. You conclude that data segment 3 was lost and 4 and 5 reached the receiver. Describe e.g., a network condition under which this conclusion is not true.

A possible condition could be that packet number 4 was duplicated somehow in the network (e.g. two different valid paths) and packet number 5 was lost. So the receiver got packet number 4 two times and the ACK's are the responses to them. The receiver doesn't know about packet number 5 and saves #4. It expects packet number 3 as next packet that's why it sended ACK 3 two times.

2 Selective Repeat

2.1 Receiver

> **To include in your report:** Explain your implementation, in particular how you decide if an out-of-order segment will be buffered.

The implementation for solving selective repeat on the receiver side was the one of buffering all of the out-of-order packages received that lies within the receiver window.

In order to do that we decided to set the following piece of code for buffering out of order packages that arrive, i.e. we are in the `else` of the *if num == self.next:* condition.

```
## create an array containing the packages within the receiver window.
ack_local = (self.next + self.win) % 2 ** self.n_bits
if (ack_local < self.next):
    ack_local += 2 ** self.n_bits           ## works with not overflow corrected numbers
# items in window
window_items = [i % 2**self.n_bits for i in range(self.next, ack_local)]  ## correct back for overflow
```

Then in buffer the out of order received packages that lies within the receiver window.

```
if num in window_items:
    self.buffer[num] = payload
    log.debug("Buffered package: %s", num)
```

Notice finally that in our implementation we do not write directly the buffered payloads to the file. We rather wait for that until the correct expected payload arrives.

In such a case in the *if num == self.next* condition we added the following piece of code at the end of the existing code.

```
while self.next in self.buffer.keys():
    log.debug("Writing buffered package: %s", self.next)

    # append payload (as binary data) to output file
    with open(self.out_file, 'ab') as file:
        file.write(self.buffer[self.next])

    del self.buffer[self.next]

    log.debug("Unbuffered: %s", self.next)

    self.next = int((self.next + 1) % 2**self.n_bits)
```

This will allow us to iteratively write to the output file and remove from the buffer the buffered packages that lie now in the expected sequence.

> In addition, answer the following 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?

The solution to this theoretical question relies on understanding that while it might well be beneficial to always store out of order segments in the case of no overflow, in the presence of such it might become indistinguishable for the receiver to recognize whether two buffered segment with the same number are in fact equal or different.

Respecting the receiver window size when buffering segments this difficult to handle situation is avoided as you would just buffer segments in the window that would ultimately be unique given the imposed condition

```
assert self.win <= 2**self.n_bits
```

Moreover given the window it is clear that as soon as the expected segments would arrive you would unbuffer the stored segment freeing up the slot for the next **different** segment carrying the same number due to overflow.

In summary, when respecting the receiver and sender window size you do not only save valuable resources but you rather also avoid the issue of overwriting important buffered payloads with out-of date payloads that were already properly handled.

2.2 Sender

To include in your report: Explain your implementation, especially under which conditions you increase the duplicate ACK counter and when you reset it.

First we want to make sure selective repeat ist only active when the flag Q_4_2 is enabled and if self.SACK is disabled (SACK and selective repeat cannot run simultaneously). These conditions are only checked for valid ACKs. For keeping track of the dublicated ACKs, we introduce the counter "ack_duplicate_number". When a valid ACK is received, "ack_duplicate_number" is set to 0 (which means we have zero duplicated ACKs). The counter is incremented when new ACKs with equal sequence number arrive (i.e ack equals self.unack). If the counter reaches "2", the buffered data package is resent immediatly. Also the counter is set to "-1" to make sure the package is resent only once for each queue of three identical ACKs. Upon receiving a new ACK, the counter is set to 0 again.

A possible problem in this implementation is that we don't keep track of the actual sequence number of the ACKs we are counting. A practical example: ACK 3, 3, 2, 3 does not trigger a fast retransmit for package 3, but a good implementation should do so. (I.e if ACK 2 got stuck in a congested network and was slowed down).

3 Selective Acknowledgment

3.1 Receiver

To include in your report: Explain your algorithm to find the SACK blocks from the data in the out-of-order buffer, especially if a block spans over the sequence number overflow. Furthermore, describe all the conditions that must hold for your receiver to add each of the optional fields to the header.

The algorithm leverages on the work of the selective repeat receiver implementation and the buffer of out of order packages stored there.

The basic idea is to take the buffered packages that did not match the expected packages to be received from the previous section.

Then you would simply order such packages and analyze the series of consecutive packages. Given a series of consecutive packages we have to store for each two piece of information:

(i) The length of the sequence

(ii) The start of the sequence

Such information is saved in two arrays in our implementation **blen** storing the length of the sequence, and **left** storing the beginning of the sequence.

The pseudo-code can therefore be expressed as follows

```
1  if receiver_buffer > 0:
2      left0 = sorted(receiver_buffer)[0]          # initializer left edge
3      expected = left0 + 1                  # initializer expected next value in the series
4      reference = 1                          # initializer helper variable for block length
5
6      left.append(left0)                          # append the first item as
7                                            # the first starting value
8                                            # for the first sequence
9
10     for idx in range(1,length_of_receiver_buffer):
11
12         package_number = sorted_receiver_buffer[index]
13
14         if package_number == expected_number:
15             increase_expected_number_by_one
16             increase_sequence_length_by_one
17
18         else if_package_number_not_the_expected:
19             blen.append(reference)
20             left.append(key)
21             expected = key + 1
22             reference = 1         ## the length starts at 1.
```

Important is that the pseudo-code above misses two important components that is necessary to deal with:

(i) when you reach the end of the sorted buffer and the package-number is the one expected the block length is not added to the block-length array as the else condition was not triggered.

(ii) overflow breaks the sorted component of the algorithm above. Moreover, it was not addressed in the expected variable above.

We decided to address the above two points in the following way

As for (i) we decided to append at the tail of the pseudo-code above outside of the `for` loop but inside the `if` conditional the following condition

```
    # check if the last was expected. if yes close
    # the block length by appending its length.
    if len(blen) < len(left):
        blen.append(reference)
        assert len(blen) == len(left)
```

With respect of (ii) we decided to keep the pseudo-code structure of above and deal with the overflow component by working directly with an overflowed array. The idea is the following.

Given a receiver buffer, for instance

```
buffer = {}

buffer[0] = "a"
buffer[31] = "a"
buffer[28] = "a"
buffer[27] = "a"
buffer[22] = "a"
buffer[21] = "a"
buffer[20] = "a"
```

You would get the overflowed array

```
ack_local = (self.next + window) % 2 ** n_bits            ## corrected overflowed window end
if (ack_local < self.next):
    ack_local += 2 ** n_bits                              ## not corrected overflowed window end

overflow_items = [i for i in range(self.next, ack_local)] ## not corrected overflowed window
```

Then you can intersect the above array containing the not for overflow corrected numbers with the payload numbers you have saved in your buffer.

```
overflowed = []
for i in receicer_buffer_keys:
    if (i + 2**n_bits) in overflow_items:
        overflowed.append(i + 2**n_bits)
    elif i in overflow_items:
        overflowed.append(i)
```

A final array of the following form would result:

`[32, 31, 28, 27, 22, 21, 20]`

It is now clear that the complexity of dealing with the overflow in the expected number when using the standard payload numbers in our receiver buffer was solved by creating a new array containing the not for overflow corrected receiver buffered numbers.

When looking at the sorted not for overflow corrected array, i.e.

`[20, 21, 22, 27, 28, 31, 32]`

It is then straightforward to see that, when inserting this sorted array, in place of the vanilla receiver-buffered-numbers, in the pseudo code introduced at the beginning of the section, we arrived to the desired solution. There is just an issue with the left-edges saved which are not corrected for overflow. However, this is immediately corrected by replacing the `left.append(key)` piece of code with `left.append(key% 2**n_bits)` in the pseudo code above.

You might refer to our python script for looking at the not-pseudo-code leveraging the sorted not for overflow corrected buffer.

> In addition, answer the following theoretical question: describe two other optional header designs that the receiver could use to inform the sender of its current buffer state.

For answering this question, we assume that the optional header size must be of 24-bits multiples due to the logic described here, padding makes sense and we therefore propose the following two designs.

One possible different header design is the one of omitting the block length. This is a redundant information given that we already give the length of the entire header.

Instead of sending dead end bits that carries no information, you might replace the optional header as follows

12-bits	12-bits
Left edge 1st block	Length 1st block
Left edge 2nd block	Length 2nd block
Left edge 3rd block	Length 3rd block

This might be useful when the number of bits used to encode sequence number and the window size are especially high.

A second design might be the one of working with simply the left edges in the optional header arguments and a first argument providing the length of the last block in the series. These carry enough information for the length of the blocks to be inferred.

8-bits	16-bits
$\text{Length}_{\text{last block}}$	Left edge 1nd block
Padding	Left edge 2nd block
Padding	Left edge 3nd block

Both of the methods actually reduce the amount of passed information by one column for which the bits might be distributed to carry the most relevant information.

Important is finally to see that reducing the passed information by some header row might be more beneficial in terms of saving the amount of bits exchanged. However, we could not come up with a simple design where the information carried by each cell would carry the same type of information. Such implementation would not be possible without a rearrangement of the sender receiver for dealing case by case with the different information content.

3.2 Sender

> **To include in your report:** Explain how the sender finds the missing blocks from the received SACK header. In addition, describe the SACK negotiation part of your implementation (options field).

For finding the missing blocks from the received SACK header, a list collecting all sequence numbers of unacknowledged packets is created. This is achieved by considering all packet numbers starting from first not yet acknowledged packet until the left edge of the first block. If the block length is larger or equal to 2, the list ist extended by adding the next missing elements (from the end of the first block to the left edge of the second block). If the block length is larger or equal to 3, we repeat the same procedure but with the next block. All these packages are put together in the "package_to_retransmit" list which are then resend. A SACK supporting Sender sets the options field in the GBN header to "1" in for all data packets. The receiver then checks for the option field and also returns the acknowledgements with the options field set to "1". If the value in the options field is not equal to 1, the sender/receiver is assumed to not support SACK.

> **To include in your report:** Finally, answer the following 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.

To reduce the number of resent packages, we could use a timer that prevents the retransmission of a segment. This timer has to be shorted than the trigger timeout but as long as possible to prevent unnecessary retransmissions.

4 Congestion Control

> **To include in your report:** Explain how your CWND interacts with the already existing sender and receiver windows. Next, justify your choice of mechanisms to increase and decrease the CWND. Finally, show us a graph of the CWND evolution during a transmission with failures.

To keep track on the congstion window we added four new global parameters by adding:

```
#how big the linear increase should be
self.congestion_add = 1
#how big the multiplicative increase or decrease should be
self.congestion_mult = 2
#current window threshold
self.congestion_threshold = 32
#what is the biggest window size possible
self.congestion_window_max = win
```

We implemented the congestion window very similar to the TCP Congestion control according to this website.

It is very stable and not much further things have to be changed. The max window size can easily be adapted. We chose this as we increment continously, and have a hard reset only in few edge cases if a timeout occurs. In the tests we only had 1 hard resets per run which is very good.

In the "BEGIN" state we added the following construction to save the max window size and to set the current window size to 1 => slow start.

```
#Set initial congestion window to 1 if its enabled and save max window size
if self.Q_4_4:
    log.debug("Congestion Control enabled")
    self.congestion_window_max = self.win
    self.win = 1
```

Next we enable the slow start phase and the additive increment of the congestion

```
#Increase congestion control Window
if self.Q_4_4:
    # slow start
    if self.win < self.congestion_threshold:
        self.win = self.win * self.congestion_mult

    # additive increment
    else:
        self.win = self.win + self.congestion_add

    # Reset window if the increase was too big.
    if self.win > self.congestion_window_max:
        self.win = self.congestion_window_max
    log.debug("Congestion window now: %s", self.win)
#if congestion control not enabled maximize the window if it has changed during runtime
else:
    self.win = self.congestion_window_max
```

To decrease the window size we added the following structure to the one already added in task 4.3.1 where we test for three ACK's which means a soft reset.

```
#soft reset congestion window
if self.Q_4_4:
    self.congestion_threshold = int(self.win / self.congestion_mult)
    self.win = self.congestion_threshold
    log.debug("(soft reset) Congestion window now: %s", self.win)
```

If a timeout happens a hard reset should be made. So we added to the $\text{timeout}_{\text{reached}}$ state

```
#decrease congestion window, hard reset
if self.Q_4_4:
    self.congestion_threshold = int(self.win/self.congestion_mult)
    self.win = 1
    log.debug("(hard reset) Congestion window now: %s", self.win)
```

In the following pictures you can see two tests with congestion control. They are form $\text{start}_{\text{local.sh}}$, with th following parameters:

```
# Parameters for sender and receiver
NBITS=5                    # Number of bits used to encode the sequence number

# Parameters for sender
IN_FILE=sample_text.txt # Data to send [e.g. sample_text.txt or ETH_logo.png]
SENDER_WIN_SIZE=31          # Window size of the sender
Q_4_2=1                    # Use Selective Repeat (Q4.2) [0 or 1]
Q_4_3=0                    # Use Selective Acknowledgments (Q4.3) [0 or 1]
Q_4_4=1                    # Use Congestion Control (Q4.4/Bonus) [0 or 1]

# Parameters for receiver
OUT_FILE=out_temp.txt      # Output file for the received data from the sender
RECEIVER_WIN_SIZE=31        # Window size of the receiver
DATA_L=0.1                   # Loss probability for data once done with 0.1 and once with 0.2
ACK_L=0.1                   # Loss probability for ACKs once done with 0.1 and once with 0.2
```

0.0 Image 2020-05-21 at 23.17.16.bb

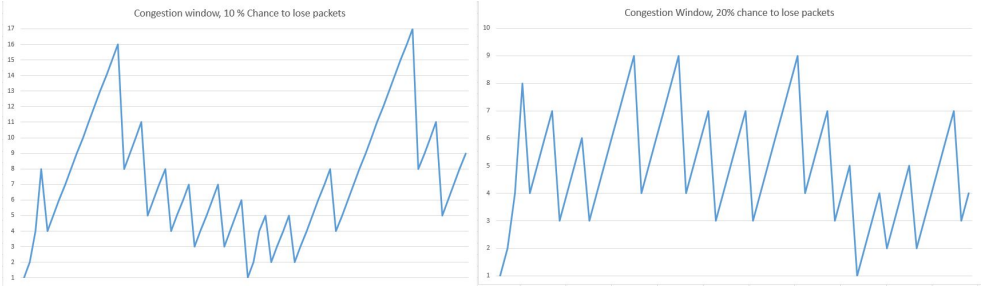

Figure 1: Congestion Control Time Series