Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Spring 2020 Prof. L. Vanbever/ T. Bühler, R. Birkner, C. Busse-Grawitz, T. Holterbach

Communication Networks

Project 2: Reliable Transport

Deadline: May 22 2020 at 11.59pm

Now that you are all experts in routing, it is time to attack the remaining pillar of networking: reliable transport. Together with your group, your objective is to implement a TCP-like reliable transport protocol capable of sending binary data over unreliable IP networks (like the Internet).

As a starting point, we provide you with a Python-based skeleton of a Go-Back-N (GBN) sender and receiver. The receiver is basic (*e.g.*, it only accepts in-sequence segments), but *fully* functional, while the sender is *almost fully* functional. In addition, the receiver can also be used to simulate an unreliable network by (pseudo-)randomly dropping either data segments or acknowledgments.

The assignment starts with completing the basic GBN sender, and continues with improving the protocol by implementing first *Selective Repeat* and second *Selective Acknowledgments*. A bonus question further asks you to implement a *Congestion Control* mechanism at the sender.

As in the routing project, each group has to solve the task on a VM. All VMs are connected and you can test your sender/receiver implementation with the receiver/sender of another group.

The rest of this document is organized as follow: Section 1 provides general information about the project, including **submission instructions**. Section 2 introduces the GBN protocol while Section 3 contains programming instructions and explains how to access and test your implementation. Finally, Section 4 describes the tasks you have to solve.

1 General Information

This section tells you what to do if you have questions, how to submit your work and how it will be graded. Furthermore, it explains our policies on academic integrity and misuse of the resources.

1.1 If you have questions: use Slack or send us an email

Ask your questions on the `#transport_project` channel available at `comm-net20.slack.com`. Please do not ask questions in the `#general` channel. You can also ask questions by email.

1.2 Online Q&A sessions on Slack

During the normal exercise timeslots (Thursdays from 10:15AM to 11:55AM), we will be available on Slack to answer questions and help you with the project. We will offer additional Q&A sessions to assist you as needed. The times of these additional sessions will be announced through Slack and our website in a timely manner. During all the official sessions, we will also be available via voice chat in case of bigger problems. Outside of these timeslots, you can always post your questions on Slack, but you might not get an immediate answer.

1.3 Submit your work by e-mail

Create a folder which contains your PDF report and your final sender and receiver implementation (the files `sender.py` and `receiver.py`). Make sure that your report includes your group number as well as the name of all members composing your group. The maximum length for your PDF report is 10 A4 pages (including screenshots, code snippets, and plots) but can also be much shorter. Send a zip or tar.gz archive of the folder by email to Laurent Vanbever (`lvanbever@ethz.ch`), Rüdiger Birkner (`rbirkner@ethz.ch`) and Tobias Bühler (`buehlert@ethz.ch`).

Important The subject of your email must follow this format:
[comm_net] groupX project 2, where X is your group number.

1.4 Grading

This assignment will be graded and counts as 10% towards your final Communication Networks grade. There are a maximum of 10 points (plus one bonus point). Each group member will receive the same grade: $\min\{1 + \frac{\sum pts}{2}, 6\}$. We will test a part of the implementation with automatic test sequences. It is therefore important that you exactly follow the instructions and specifications in the questions.

1.5 Academic integrity

We adopt a *strict* zero tolerance policy when it comes to cheating. Cheating will immediately translate to the entire group failing the assignment and being reported to the ETH administration. You can only do your assignment with your teammates. Do not look at other groups' code and do not copy code from anywhere. It is OK to discuss things or find help online but you *must* do the assignment by yourself.

Your code and report may be checked with automated tools so as to discover plagiarism. Again, **do not copy-and-paste** code, text, etc.

2 The Go-Back-N Protocol

As seen in the exercises, your basic GBN protocol uses a buffer and timeout at the sender, and the receiver uses *cumulative acknowledgments*. The flow of data is uni-directional. Similarly to TCP, the protocol identifies DATA segments using sequence numbers. The sequence number is initialized to 0 and is incremented by one for each new segment. The receiver acknowledges correctly delivered segments by sending an ACK segment. Concretely, as we are using cumulative ACKs, the ACK contains the sequence number of the next expected data segment (and consequently acknowledges all segments up the next expected segment). In order to provide actual reliability, the sender uses a *timeout*: If nothing happens for a given amount of time (and the transmission is not yet complete), the sender retransmits all segments in the current window (Figure 1). Finally, the sender must also implement flow control. The receiver communicates its window size, and the sending window must be strictly smaller or equal to the receiving window.

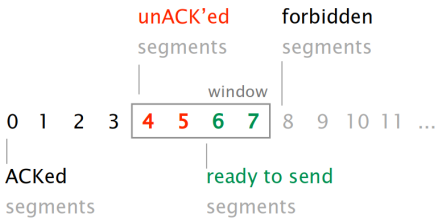

Figure 1: GBN sender window. On timeout, all segments in the window are retransmitted.

Figure 2 describes the header of your protocol. It is composed of 6 mandatory fields along with up to 9 optional fields. Table 2 explains the semantics of the mandatory fields. The optional header is relevant for Section 4.3 and is introduced and explained as soon as it is required.

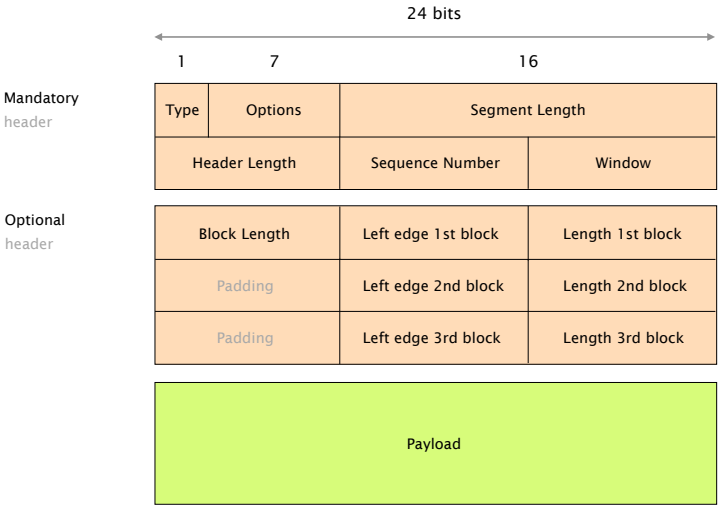

Figure 2: Header Format

Field	Length	Description
Type	1 bit	Encodes the segment type. 0 indicates a DATA segment, while 1 indicates an ACK segment.
Options	7 bit	Indicates the support of the optional SACK feature (see §4.3). 0 indicates no support, while 1 indicates SACK support.
Segment Length	16 bit	Encodes the length of the payload in bytes. All DATA segments contain a payload with 64 bytes of data, *except* for the last DATA segment of a transfer which can be shorter. The reception of a DATA segment whose length is different than 64 indicates the end of the data transfer.
Header Length	8 bit	Indicates the total size of the header (including the optional part, if present) in bytes.
Sequence Number	8 bit	Indicates the segment sequence number. The sequence number always starts at 0 and is incremented by 1 for each new DATA segment sent by the sender. Inside an ACK segment, the sequence field carries the sequence number of the next in-sequence segment that is expected by the receiver.
Window	8 bit	Encodes the size of the current window. In DATA segments, this field indicates the size of the sender's sending window expressed as a number of segments. In ACK segments, this field indicates the current value of the receiving window, also expressed as a number of segments.

Table 1: Mandatory fields in the GBN header.

3 Programming instructions

3.1 VM

Similar to the routing project, each group has a VM available on our server. To access your VM, you can use the password received via email. Use the login `root`, and the port $3000 + X$, X being your group number. For example for group 27, here is how you can access the VM with ssh.

```
> ssh -p 3027 root@snowball.ethz.ch
```

On your VM you find multiple files (use e.g., `ls` to get an overview). `sender.py` and `receiver.py` are the main files and already contain a skeleton implementation of the GBN protocol. You will work on these files to solve the questions in Section 4. The three starting scripts `start_local.sh`, `start_global.sh` and `start_test.sh` (as well as `client.py`) are used to start and test your sender and receiver implementation in various ways. Section 3.6 and 3.7 explain these files in more detail. Finally, the files `sample_text.txt`, `to_send_test.txt` and `ETH_logo.png` will be used as dummy data that you can transmit from your sender to your receiver.
Backup: Make sure that you back up your work (sender and receiver files) regularly. You can either save them on your local machine e.g., by copying them using `scp` or (preferred) frequently commit them to a private repository on the ETH GitLab platform as introduced during the exercise session.

3.2 Defining the Go-Back-N protocol header in Scapy

We use Scapy [1] to implement the GBN protocol with a state machine, as well as for receiving and sending packets. Scapy implements a lot of the low-level details of packet handling for you.

First of all, the GBN header needs to be defined, such that Scapy is able to parse and create packets using your GBN protocol. The following Python commands define the mandatory header (Figure 2) using Scapy and are already implemented in the project template files on your VM.

```
class GBN(Packet):
    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)]
```

As you can see, the `GBN` class inherits from Scapy's `Packet` class and has two properties: A protocol `name`, and `fields_desc`, a list that defines the different header fields. Every field has at least a name and a default value. The different field types (e.g. `ByteField` or `ShortField`) specify the size of the field. For the `BitFields` *type* and *options*, the size is explicitly set to 1 and 7 (second argument). Remember that a byte is 8 bits long, while a short is 16 bits long.

Additionally, we can instruct Scapy about its relationship with other protocols, such as IP. As discussed during the lecture, a packet normally consists of multiple headers from different protocols along with a payload. Being a transport protocol, we place the GBN header right after the IP header. In Scapy, we can use the following command to define this relationship:

```
bind_layers(IP, GBN, frag=0, proto=222)
```

As you can see, we have assigned the (unused) protocol number 222 to your GBN protocol. Scapy now automatically sets the *Protocol* field in the IP header to 222 whenever a GBN packet is sent, and similarly parses any IP packet with protocol number 222 as a GBN packet. Neat!

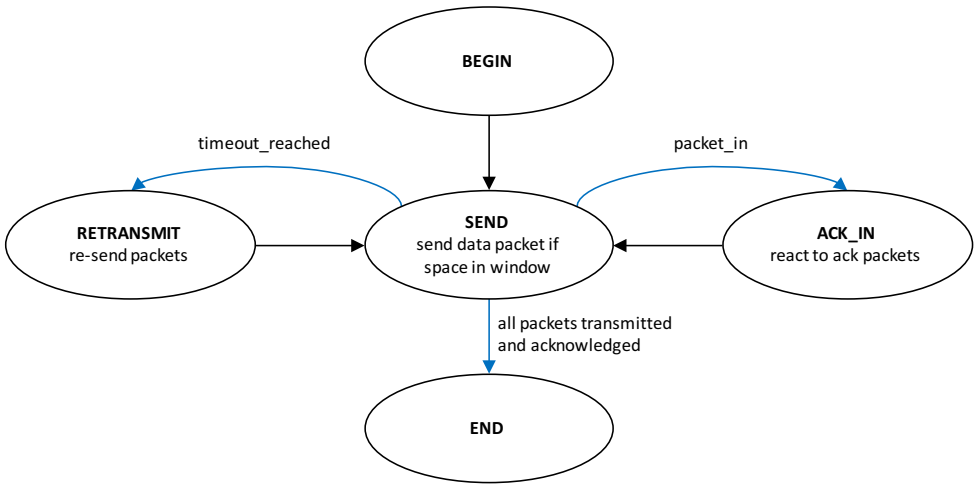

Figure 3: Sender automaton

3.3 Create and send a packet

Now that we have defined the GBN header, we can use it to create a packet. First, we build the GBN header.

```
header_GBN = GBN(type='data', num=6, win=4)
```

In this example the type is `data`, the sequence number, `num`, is 6 and the window size, `win`, 4. All the fields for which we did not define a specific value are set to their default values. In your implementation, you have to take care to set *all* header fields to their correct values (e.g. the `hlen` field should contain the correct size of the header). Finally, we define an IP header containing `sender` and `receiver` IPs, combine both headers with a payload, and send the packet:

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

For convenience of notation, Scapy overloads the `"/"` operator to glue different layers together.

3.4 Dissect a packet

Next, we take a look at how you can access different fields of a packet or print the packet content to the terminal. In the following code section we assume that we have a packet called `pkt`:

```
pkt_num = pkt.getlayer(GBN).num
payload = pkt.getlayer(GBN).payload
print(pkt.show())
```

With the first command, we get the value of the `num` field of the GBN header (In Scapy, headers are usually called 'layers'). The second command returns the payload of the packet (the transmitted message). Finally, the last command prints the whole packet to the terminal. In most cases Scapy prints the packet in human readable form, which can assist you in debugging.

3.5 The GBN automaton

We realize the functionality of the GBN protocol as a state machine using the Scapy automaton class, which is already implemented in the project template files on your VM. It is possible to solve all the questions with the given automaton structure, but you are free to add additional states. Figure 3 shows the sender automaton, which has five states (`BEGIN`, `SEND`, `RETRANSMIT`, `ACK_IN` and `END`). The `SEND` state is the main state. It sends all the data packets in a given window and then waits. Different transitions happen when specific conditions arise. For example, the

automaton could receive a packet from the receiver. It will then transit from the SEND state to the ACK_IN state. Another possibility is that a timeout is reached (this means we did not get an ACK from the receiver). The automaton then transitions from the SEND state to the RETRANSMIT state, in which packets are re-sent. Afterwards, the automaton returns to the SEND state.

The automaton is implemented in the class GBNSender which inherits from the Automaton class. First, we define a method called parse_args which handles the input arguments and initializes important variables. Then, we define the master_filter method:

```
def master_filter(self, pkt):
        return (IP in pkt and pkt[IP].src == self.receiver and GBN in pkt
                and ICMP not in pkt)
```

Although the return expression in the method looks a bit cryptic, it is quite handy. It returns only the packets which contain an IP and a GBN header and are coming from the receiver. Furthermore, we block ICMP (Internet Control Message Protocol e.g. used by traceroute) packets, which could confuse our GBN sender. Scapy applies this *master filter* every time it receives any packet. The sender automaton therefore receives only packets of interest and thus you can always assume that a received packet contains a GBN header and is coming from the receiver.

The Scapy automaton uses exceptions to go from one state to another. To go to a state, raise it as an exception. For example, the following code triggers a transition to the SEND state:

```
raise self.SEND()
```

3.6 Start sender and receiver

We created two scripts to correctly start (and terminate) the sender and receiver.

Use ./start_local.sh to test your sender with your receiver implementation. In the script you can change several parameters. The NBITS parameter defines the number of bits used to encode the sequence number. For the sender, you can define its window size (SENDER_WIN_SIZE), the file (IN_FILE) you want to transmit to the receiver as well as multiple variables to enable the different questions (Q_4_2, Q_4_3, Q_4_4). For the receiver, you can also define the window size (RECEIVER_WIN_SIZE) as well as the output file (OUT_FILE) to store the received data.

Finally, there are two special parameters for the receiver. DATA_L indicates the loss probability of a data packet and ACK_L is the loss probability of an ACK packet. You can use these two parameters to simulate packet losses between the sender and the receiver. The (pseudo-)random number generator is initialized by a seed value (at the beginning of the receiver file), such that the same packets get lost if you execute the script multiple times. You can also change the seed.

You can also execute ./start_global.sh to test your sender/receiver with the receiver/sender of another group. In addition to the parameters described above, you can set NEIGHBOR to the number of the other group. The group numbers are the same as in the Internet Routing project. With the parameter TEST_SENDER_OR_RECEIVER you can indicate if you want to test your sender (1) or receiver (2). As an example, if you are group 5 and you want to test your receiver with the sender of group 11, you should use: NEIGHBOR=11, TEST_SENDER_OR_RECEIVER=2. Group 11 uses: NEIGHBOR=5, TEST_SENDER_OR_RECEIVER=1.

3.7 Testing your code

Make sure that you test your code thoroughly. This not only means verifying that your implementation successfully transmits a file from your sender to your receiver, but also handles corner cases well (e.g., sequence number wrap around) and interoperates with the implementation of other groups. For this purpose, you can change various parameters within the start script (./start_local.sh) as explained in 3.6. To observe a wrap around faster, you can, for example, set the number of bits used for the sequence number (NBITS) to a lower value. Test your implementation with multiple other groups using the ./start_global.sh script.

Optional Tests We provide a very basic test framework which runs your sender respectively receiver implementation against test sequences and reports correct and unexpected behavior. Use the script `./start_test.sh` to start a test. There are parameters to set your sender and receiver files and the specific test you want to perform (`TEST_NUM`). Do not change the other parameters. **Important:** Even though passing all the tests is certainly a good sign that your implementation behaves as expected, it does not at all guarantee that you will receive a 6 in the end. When we grade your submission, we will use different test sequences and also take into account other parts of your work, for example your report.

3.8 Important points

Comment your code! As you will have to submit your code at the end of the assignment, it is important that you ensure we can understand what your code is about.

Window Size To make your life easier, always choose a maximal window size (sender and receiver) which is at most half of the maximal sequence number (2^{NBITS}). Otherwise it is possible to see two different packets with the same sequence number in the same window.

End of Transmission The sender terminates once all the packets are sent and the last packet is acknowledged. Determining the end of the receiver is a bit harder, so we use the following fact. All the payloads of the packets have a fixed size of 64 bytes. The last payload is normally shorter. The receiver therefore looks for a shorter packet and terminates once an ACK for the last packet is sent (already implemented). Note that this termination mechanism will fail, should the data size be a multiple of 64 bytes. You then have to manually terminate the scripts.

Postel's Law A last word of advise. In your implementation, we recommend to follow Postel's law, named after the Internet pioneer Jon Postel [2], who drafted the early TCP specification:

> *Be conservative in what you do, be liberal in what you accept from others.*

Making sure your implementation can tolerate (*i.e.*, not crash) wrong inputs is typically particularly handy during tests with other groups.

4 Assignment

4.1 Complete the Go-Back-N sender (3 points)

As mentioned before, the Go-Back-N sender is almost functional. We ask you to complete it so as to make it both functional and compliant with the protocol specification.

There are three missing parts separating the skeleton from a complete implementation. These are clearly indicated with comments in the sender template. We advise you to complete them in the following order.

First, complete the implementation of the `SEND` state by crafting segments with the appropriate header and use the send command. Second, add support for dealing with acknowledgments by completing the `ACK_IN` state. In particular, whenever the receiver acknowledges the reception of a segment, you should remove it from your sending buffer and therefore open your window. Observe that opening up the window could then enable the sender to send more segments (if any). Third, add support for dealing with losses and timeouts by completing the `RETRANSMIT` state. In Go-Back-N, a sender sends back *all* the unacknowledged segments upon timeout.

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. 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.

Figure 4: At each red arrow the sender should re-send (Selective Repeat) data segment 4.

4.2 Implement Selective Repeat (3 points)

As discussed during the lecture, Go-Back-N is not the most efficient protocol when it comes to dealing with failures. To improve performance, we ask you to implement Selective Repeat as a recovery mechanism. The implementation will be split into two parts. In this question, we ask you to adapt the behavior of the receiver and the sender without changing anything to the headers of your protocol. In the next question (§4.3), you will boost the performance of your protocol once more by supporting Selective Acknowledgments (SACK).

4.2.1 Receiver (1 point)

The Go-Back-N receiver we provided you with only accepts segments in-sequence. To support Selective Repeat, modify your receiver to buffer out-of-order segments and deliver (here, write to the output file) more than one of them when missing segments are received.

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

4.2.2 Sender (2 points)

As of now, the sender doesn't do anything upon the reception of duplicated ACKs. As we have seen in the lecture, duplicate ACKs are a sign of isolated loss and can be used by the sender to prevent useless timeouts (similar to fast retransmit in TCP). We ask you to modify your sender to preventively re-send the next unacknowledged segment, and only this one, upon the reception of 3 duplicate acknowledgements.

As an example, look at Figure 4. Assume the sender receives the shown sequence of ACKs. A red arrow indicates when the sender should re-send data segment 4. If the re-sent packets are lost as well, the sender will eventually reach the timeout and retransmit all unacknowledged segments from its buffer (as in question §4.1).

Important Make sure that your sender is using Selective Repeat *only* if the parameter `Q_4_2` is set to 1 in the starting scripts.

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

4.3 Implement Selective Acknowledgment (4 points)

With Selective Repeat, your protocol is now much more efficient than before, yet it can still suffer from poor performance when multiple segments are lost. Indeed, with the limited information contained in a cumulative acknowledgement, the sender learns at most *one* lost packet per RTT.

To improve this situation, we ask you to implement a Selective Acknowledgment (SACK) mechanism similar to the one used by TCP [3]. With SACK, the receiver can inform the sender about blocks of consecutive packets that it received correctly.

The support for SACK is negotiated using the Options field. The sender and the receiver indicate that they support SACK by setting the field to 1 in all their packets. SACK should be used only if *both* the sender and the receiver support it.

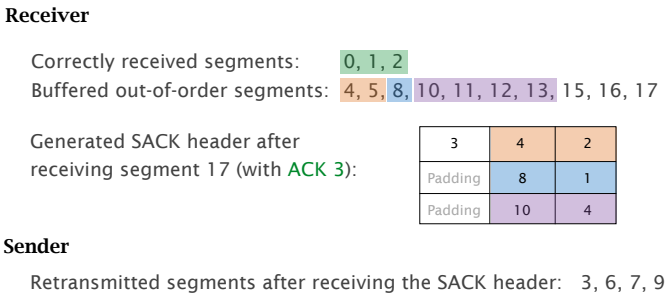

Figure 5: SACK example for receiver and sender.

4.3.1 Receiver (2 points)

A receiver supporting SACK informs the sender of contiguous, but isolated blocks of data that have been received and queued properly. For this, it uses the optional headers described in Figure 2. The semantic of the fields is as follows:

a) Block Length (8 bits): Indicates the number of blocks (between 1 and 3) included in the optional header.

b) Left edge (8 bits): Indicates the sequence number of the *first* segment of a contiguous block that has been correctly received.

c) Length of block (8 bits): Indicates the length of the contiguous block that has been correctly received.

To simplify the protocol, the receiver can only advertise up to the first 3 contiguous, but isolated blocks it has correctly received. Additional blocks are simply not advertised. Observe that a block can be of unary size, in this case, the length of the block will be 1. If the receiver has currently no blocks to report, the additional header should not be present at all.

When the missing segments are received, the receiver acknowledges the data normally by advancing its window. In particular, the SACK option does not change anything to the semantic and use of acknowledgments (encoded in the Sequence Number field). Figure 5 shows an example of an expected SACK header generated by the receiver.

Important The receiver should only generate SACK headers if the received data segments from the sender contain a 1 in the `options` field (compare Figure 2). The receiver will then also always set a 1 in the `options` field of the generated ACKs (even if a particular ACK contains no SACK header).

Hint To solve this questions, you have to modify the GBN header defined at the beginning of the sender and receiver template. We want to be as efficient as possible. Therefore, if the receiver is only acknowledging one block, the header fields for the second and third block should not be present in the final packet. You can achieve that with Scapy using a new field type called `ConditionalField`. The following code segment shows a simple example of a possible `fields_desc`:

```
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("test", 0), lambda pkt:pkt.hlen == 7) ]
```

In this case, Scapy will only add the additional header field, called `test`, if you give the `hlen` field a value of 7 during the packet creation. By using multiple conditional header fields with different conditions, you can create the optional header from Figure 2.

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. 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.

4.3.2 Sender (2 points)

When receiving a packet containing a SACK header, the data sender should retransmit all the unacknowledged packets that fall outside of the boundaries of contiguous blocks by using its buffer of transmitted but not yet acknowledged segments. See Figure 5 for an example. It is important to note that the sender is not retransmitting unacknowledged packets after the last SACK block (e.g. 15, 16 or 17 in the example). These packets may still be in transit and are not necessarily lost. The sending window of the sender does not move after these retransmissions as these do not constitute new transmissions.

Important Make sure that your sender is using SACK *only* if the corresponding parameter `Q_4_3` is set to 1. The sender then sets a 1 in the `options` field (compare Figure 2) to inform the receiver. Finally, the sender should either use SACK or Selective Repeat (§4.2) but not both at the same time. You can control that with the two parameters `Q_4_2` and `Q_4_3` in the scripts.

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). 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.

4.4 Bonus question: Implement congestion control (1 point)

The protocol described above looks a lot like the original TCP in the sense that rate limiting is only done by adapting the size of the receive window. Yet, not caring about network conditions basically leads to the big congestion collapse you saw in the lecture.

For this bonus question, we ask you to implement some basic congestion control at the sender side. To do so, you will now need to support a congestion window (CWND) at the sender side along with mechanisms to increase and decrease it. In the course, we saw that TCP uses loss signals (duplicated ACKs and timeouts) to detect congestion, a "Slow-Start" phase to quickly estimate the bandwidth, and an "Additive-Increase Multiplicative-Decrease" phase to track the bandwidth from thereon, in a fair way. You are welcome to implement the same kind of congestion control as TCP, or invent a new one!

Important Make sure that your sender is using the congestion control mechanism *only* if the corresponding parameter `Q_4_4` is set to 1.

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.

References

[1] Scapy. [Online]. Available: https://scapy.net/

[2] Wired. Remembering Jon Postel—And the Day He Hijacked the Internet. [Online]. Available: https://www.wired.com/2012/10/joe-postel/

[3] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018 (Proposed Standard), Oct. 1996. [Online]. Available: https://www.ietf.org/rfc/rfc2018.txt