

Communication Networks

Project 2: Reliable Transport

Deadline: June 1 2018 at 11.59pm

1 Introduction

Now that you are all experts in routing, it is time to attack the last remaining networking pillar: reliable transport. Together with your teammates, your objective will be to implement a TCP-like, interoperable reliable transport protocol with which you will be able to exchange binary files over unreliable IP networks (including the Internet).

As a starting point, we provide you with a Python-based skeleton of a Go-Back-N 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 randomly dropping segments or acknowledgments.

Most of the assignment (§3) consists in improving the performance of the protocol by switching the recovery mechanism from Go-Back-N to Selective Repeat and finally enabling the support for Selective Acknowledgments. For the “warriors” amongst you, a bonus question further asks you to implement a simple Congestion Control mechanisms at the sender level.

Similar to the Internet project, each group will have a VM to solve the task. All VMs are interconnected and you can test your sender or receiver implementation with the receiver or sender of another group.

1.1 Protocol description

Similarly to TCP, your 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. The sequence number field in the ACK segment always contains the sequence number of the *next* expected segment number. The flow of data is unidirectional. The sender only sends DATA segments, while the receiver only sends ACK segments. The sender must also comply with the flow control information provided by the receiver by making sure its sending window is strictly less than or equal to the receiving window.

Figure 1 describes the header of your protocol. It is composed of 6 mandatory fields along with up to 9 optional fields. The semantic of the mandatory fields is as follow:

- a) Type (1 bit): Encodes the segment type. 0 indicates a DATA segment, while 1 indicates an ACK segment.
- b) Options (7 bits): Indicates the support of the optional SACK feature (see §3.3). 0 indicates no support, while 1 indicates SACK support.
- c) Segment Length (16 bits): 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.

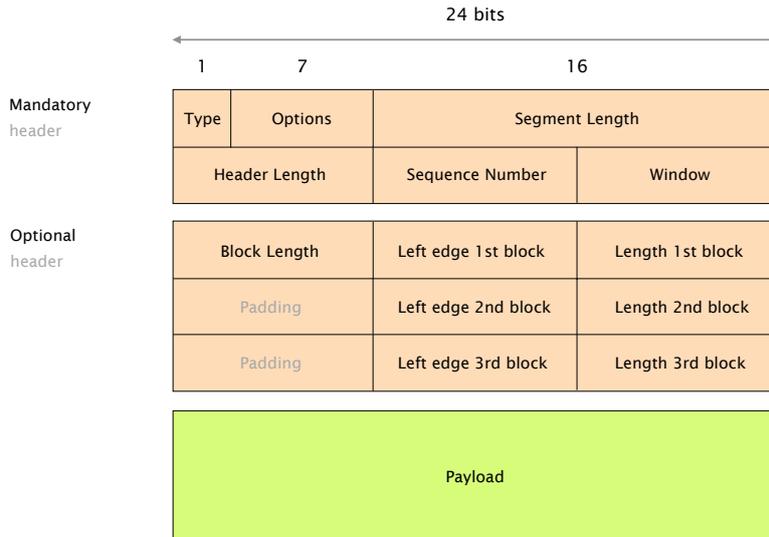


Figure 1: Header Format

- d) Header Length (8 bits): Indicates the total size of the header (including the optional part, if present) in bytes.
- e) Sequence Number (8 bits): 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.
- f) Window (8 bits): 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.

See Section 3.3 for the description of the optional header.

2 Programming instructions

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

We use Scapy [1] to implement the Go-Back-N (GBN) protocol, packet sending and receiving as well as the state machine for the GBN protocol. Scapy implements a lot of the low-level details and makes packet creation and transmission particularly easy.

The following Python commands define the mandatory header (see Figure 1) using Scapy. It is already implemented in the template files for the sender and the receiver.

```
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 the Packet class. The list called `fields_desc` defines the different header fields. Every field has at least a name and a default value. The

different field types specify the size of the field. For the two bit fields, we also specify the number of bits (1 and 7). Remember that a byte is 8 bits long, while a short is 16 bits long.

2.2 Create and send a packet

Now that we have defined the GBN header, we can use it to create a packet. 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 our GBN header right after the IP header. In Scapy, we can use the following command to define this relationship¹:

```
bind.layers(IP, GBN, frag=0, proto=222)
```

Now it is quite easy to build a packet containing multiple headers and a payload. We first 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`, is 4. All the fields for which we did not define a specific value will be set to their default values. This is just an example! In your implementation, you should always set all header fields (e.g. the `hlen` field should contain the correct size of the header). The whole packet can now be defined and sent using only one line of code:

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

Scapy uses the `/` operator to glue different layers together. The arguments in the IP header define the source and destination IP address for the packet. The packet is automatically transmitted.

Finally, we will have a quick 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 (Scapy calls the headers usually 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 will print the packet in a nice human readable form. You can (and probably, should) use this to debug your code.

2.3 The GBN automaton

To implement the functionality of our Go-Back-N protocol we use the Scapy automaton class. In the sender and receiver template the automaton skeleton is already implemented. It is possible to solve all the questions with the given automaton structure, but you can also add additional states. Figure 2 shows the automaton of the sender. We have five states (`BEGIN`, `SEND`, `RETRANSMIT`, `ACK_IN` and `END`). The `SEND` state is the main state. It will send all the data packets in a given window and then wait. Different transitions can 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 given timeout value is reached for a transmitted packet (we did not get an `ACK` from the receiver). The automaton then moves from the `SEND` state to the `RETRANSMIT` state. There, the packets are re-sent. Afterwards, the automaton transitions back 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:

¹We pick the protocol number (222) arbitrarily.

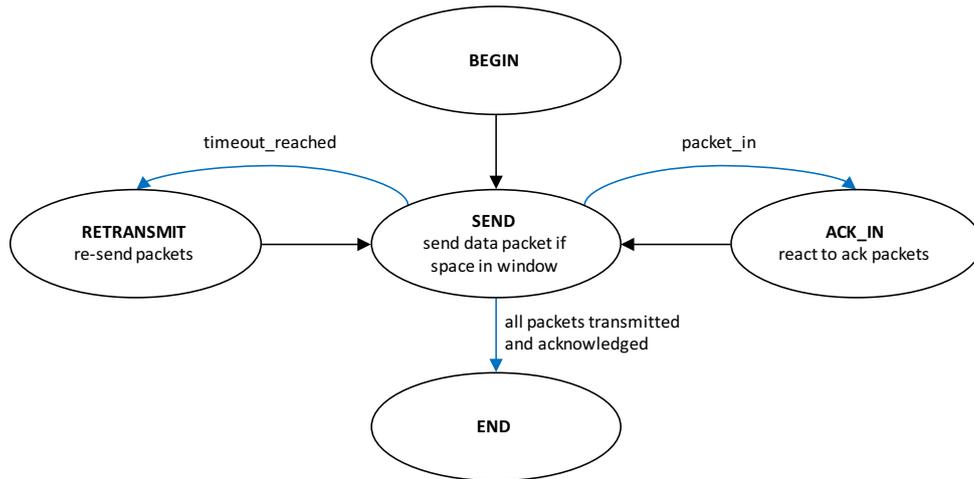


Figure 2: Sender automaton

```

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 automatically uses this method every time it receives a packet. The sender automaton will therefore only receive packets of interest. You can always assume that a received packet contains a GBN header and is coming from the receiver.

Scapy uses exceptions to go from one state to another one. With the following code you can raise an exception to change to the SEND state:

```

raise self.SEND()
  
```

2.4 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 a number of different 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.3.2, Q.3.3, Q.3.4). 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 used random number generator is initialized by a seed value (beginning of the receiver file). Therefore, the same packets should be lost if you execute the scripts multiple times. You can also set the seed to a different value.

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 the `NEIGHBOR_IP`. The VM of each group has its own IP. The syntax is `192.168.56.X` with `X` the group number. 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_IP="192.168.56.11"`, `TEST_SENDER_OR_RECEIVER=2`. Group 11 uses: `NEIGHBOR_IP="192.168.56.5"`, `TEST_SENDER_OR_RECEIVER=1`.

2.5 VM

As for the routing project, each group has a VM available on our server. To access your VM, you can use the same password as before. Use the login `root`, and the port `3000 + X`, `X` being your group number. For example for group 11, here is how you can access the VM with `ssh`.

```
> ssh -p 3011 root@samichlaus.ethz.ch
```

2.6 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 2.4. 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: 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.

2.7 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 it easily.

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 belonging to the same window.

The sender will terminate once all the packets are sent and the last packet is acknowledged by the receiver. Determining the end of the receiver is a bit harder. We will 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 total length of your transmitted message be a multiple of 64 bytes. You then have to manually terminate the scripts.

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

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 will turn out to be particularly handy during tests with other groups.

3 Assignment

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

As mentioned above, 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 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.

3.2 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 (§3.3), you will boost the performance of your protocol once more by supporting Selective Acknowledgments (SACK).

3.2.1 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 should be buffered or dropped.

3.2.2 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 3. 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 §3.1).

Important: Make sure that your sender is using Selective Repeat *only* if the parameter `Q_3_2` is set to 1.

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

3.3 Implement Selective Acknowledgment (4 points)

Thanks to 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

ACK: 4, 4, 4,  4, 4, 4,  4, 4, 4,  ... timeout

Figure 3: At each red arrow the sender should re-send (Selective Repeat) data segment 4.

Receiver

Correctly received segments: 0, 1, 2
 Buffered out-of-order segments: 4, 5, 8, 10, 11, 12, 13, 15, 16, 17

Generated SACK header after receiving segment 17 (with ACK 3):

3	4	2
Padding	8	1
Padding	10	4

Sender

Retransmitted segments after receiving the SACK header: 3, 6, 7, 9

Figure 4: SACK example for receiver and sender.

information contained in a cumulative acknowledgement, the sender can only learn about *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.

3.3.1 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 1. The semantic of the fields is as follow:

- 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 to the sender. 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 4 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 1). 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. Using multiple conditional header fields with different conditions, you can create the optional header from Figure 1.

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.

3.3.2 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 4 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_3_3` is set to 1. The sender then sets a 1 in the `options` field (compare Figure 1) to inform the receiver. Finally, the sender should either use SACK or Selective Repeat (§3.2) but not both at the same time. You can control that with the two parameters `Q_3_2` and `Q_3_3`.

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).

3.4 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 (duplicates 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_3_4` is set to 1.

To include in your report: Explain how your CWND interacts with the already existing sender and receiver windows. Furthermore, 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.

4 Organization

4.1 If you have questions: use Slack or come to an exercise session

Use the Slack channel available at `comm-net18.slack.com`. Please do not ask questions in the `#general` channel, but use the `#transport_project` channel instead. You can also send questions on Slack directly to Tobias Bühler (`@buehlert`), Maximilian Schütte (`@Maximilian (TA)`), Alexander Dietmüller (`@Alexander (TA)`), Rüdiger Birkner (`@rbirkner`) or Roland Meier (`@roland`).

4.2 Submit your work by e-mail

Create a folder which contains your PDF report and your final sender and receiver implementation. Make sure that your report includes your group number as well as the name of all members composing your group. 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.

4.3 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.

4.4 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.

References

- [1] Scapy. [Online]. Available: <http://www.secdev.org/projects/scapy/>
- [2] Wired. Remembering Jon Postel—And the Day He Hijacked the Internet. [Online]. Available: <http://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: <http://www.ietf.org/rfc/rfc2018.txt>