Programming Assignment 3: Reliable Communication over an Unreliable Network

CS 352 Internet Technology, Fall 2025

Released: October 28, 2025 | **Due:** November 14, 2025

Introduction

In project 3, you will implement a reliable sender using an unreliable UDP socket. There are two programs provided in the project 3 archive: `sender.py` and `receiver.py`. **You will only modify `sender.py`.** The `sender.py` program is a UDP sender that must implement the techniques of reliable delivery that we discussed during lecture to upload a file to the receiver. Specifically, you will implement reliability based on stop and wait and cumulative-ACK-based selective repeat. The `receiver.py` program is a UDP receiver that is attempting to download a file transmitted by the sender over a lossy channel that may drop packets, ACKs, or both. We will test reliability by checking that the receiver's version of the file matches exactly with the sender's version of the file (there are a few samples provided in the project archive for your testing and reference).

Instructions

Please read these instructions carefully before you begin.

- This is an individual project (no partners or groups).
- You must use Python 3 (not Python 2!).
- You are free to discuss the project on Piazza or through other means with your peers and the instructors. You may refer to the course materials, textbook, and resources on the Internet for a deeper understanding of the topics. However, you cannot lift solutions from other students or from the web including GitHub, Stack Overflow, generative AI (e.g. ChatGPT), or other resources. Do not post this project or parts of it to question-answering services like Chegg, academic hosting sites such as CourseHero, or generative AI. All written and programmed solutions must be your original work.
- For each question in the project report, please be clear and concise. Vague and rambling answers will receive zero credit.
- We encourage you to *start early* and get the bulk of the work for this project done the week(s) before it is due, rather than holding back your most significant efforts until the submission date is too close.

Step 1: Take it for a spin

The sender and the receiver are equipped with command line flags to customize their operation.

```
$ python3 sender.py --help
usage: sender.py [-h] [--port PORT] [--infile INFILE] [--winsize WINSIZE]

optional arguments:
  -h, --help         show this help message and exit
  --port PORT        receiver port to connect to (default 50007)
  --infile INFILE    name of input file (default test-input.txt)
  --winsize WINSIZE  Window size to use in pipelined reliability
```

You can specify which file you wish to upload through the `-infile` flag, the localhost port to which the file must be uploaded (i.e. where the receiver is listening), and other flags. By default, the file `test-input.txt` will be uploaded to localhost port 50007.

The receiver is more complex, with command line flags to control the packet and ACK losses that are simulated in the communication.

```
$ python3 receiver.py --help
usage: receiver.py [-h]
         [--pktloss {noloss,everyn,alteveryn,iid}]
         [--ackloss {noloss,everyn,alteveryn,iid}] [--pktlossN PKTLOSSN]
         [--acklossN ACKLOSSN] [--ooo_enabled]
         [--port PORT] [--outfile OUTFILE]

optional arguments:
  -h, --help            show this help message and exit
  --pktloss {noloss,everyn,alteveryn,iid}
                        Emulated loss behavior on packets (default
                        every n packets)
  --ackloss {noloss,everyn,alteveryn,iid}
                        Emulated loss behavior on ACKs (default noloss)
  --pktlossN PKTLOSSN   n for pkt loss behaviors (only if loss specified)
  --acklossN ACKLOSSN   n for ack loss behaviors (only if loss specified)
  --ooo_enabled         enable out of order data buffering (default false)
  --port PORT           receiver local port to bind to (default 50007)
  --outfile OUTFILE     name of output file (default test-output.txt)
```

By default, the channel between the sender and receiver drops every 3rd packet that is transmitted to it (i.e. `-pktloss everyn -pktlossN 3`) but does not drop ACKs (i.e. `-ackloss noloss`). By default, the receiver writes its downloaded output into the file `test-output.txt`.

Step 1.1: Test without any drops

To test the file upload/download without any packet drop, run the following commands on separate terminals:

```
python3 receiver.py --pktloss noloss
```

and

```
python3 sender.py
```

By doing this, we are configuring the receiver to not drop any data (packets or ACKs); instructing `sender.py` to simply push the bytes in the input file `test-input.txt` through the UDP socket to the receiver, and the receiver to write the downloaded outputs into `test-output.txt`. You will see many helpful prints on both the sender and the receiver. At the sender:

```
[S] Transmitting file test-input.txt
[S]: Sender socket created
Transmitted Seq: 5   ACK: 2345367   Len: 2   Msg: 88
Transmitted Seq: 7   ACK: 2345367   Len: 8   Msg: msg   1
Transmitted Seq: 15   ACK: 2345367   Len: 8   Msg: msg   2
Transmitted Seq: 23   ACK: 2345367   Len: 8   Msg: msg   3
Transmitted Seq: 31   ACK: 2345367   Len: 8   Msg: msg   4
Transmitted Seq: 39   ACK: 2345367   Len: 8   Msg: msg   5
Transmitted Seq: 47   ACK: 2345367   Len: 8   Msg: msg   6
Transmitted Seq: 55   ACK: 2345367   Len: 8   Msg: msg   7
Transmitted Seq: 63   ACK: 2345367   Len: 8   Msg: msg   8
Transmitted Seq: 71   ACK: 2345367   Len: 8   Msg: msg   9
Transmitted Seq: 79   ACK: 2345367   Len: 8   Msg: msg  10
Transmitted Seq: 87   ACK: 2345367   Len: 8   Msg: msg  11
[S] Sender finished all transmissions.
```

The input file is broken up into packets each of which contains 8 bytes. Each message is transmitted with a protocol header that includes a sequence number, ACK number, message length, and the actual application bytes. At the receiver, you will see:

```
[R]: Receiver socket created
Received    Seq: 5   ACK: 2345367   Len: 2   Msg: 88
Transmitted Seq: 235347   ACK: 7   Len: 0   Msg:
Received    Seq: 7   ACK: 2345367   Len: 8   Msg: msg   1
Transmitted Seq: 235347   ACK: 15   Len: 0   Msg:
Received    Seq: 15   ACK: 2345367   Len: 8   Msg: msg   2
Transmitted Seq: 235347   ACK: 23   Len: 0   Msg:
Received    Seq: 23   ACK: 2345367   Len: 8   Msg: msg   3
Transmitted Seq: 235347   ACK: 31   Len: 0   Msg:
Received    Seq: 31   ACK: 2345367   Len: 8   Msg: msg   4
Transmitted Seq: 235347   ACK: 39   Len: 0   Msg:
Received    Seq: 39   ACK: 2345367   Len: 8   Msg: msg   5
Transmitted Seq: 235347   ACK: 47   Len: 0   Msg:
Received    Seq: 47   ACK: 2345367   Len: 8   Msg: msg   6
Transmitted Seq: 235347   ACK: 55   Len: 0   Msg:
Received    Seq: 55   ACK: 2345367   Len: 8   Msg: msg   7
Transmitted Seq: 235347   ACK: 63   Len: 0   Msg:
Received    Seq: 63   ACK: 2345367   Len: 8   Msg: msg   8
Transmitted Seq: 235347   ACK: 71   Len: 0   Msg:
Received    Seq: 71   ACK: 2345367   Len: 8   Msg: msg   9
Transmitted Seq: 235347   ACK: 79   Len: 0   Msg:
Received    Seq: 79   ACK: 2345367   Len: 8   Msg: msg  10
Transmitted Seq: 235347   ACK: 87   Len: 0   Msg:
Received    Seq: 87   ACK: 2345367   Len: 8   Msg: msg  11
[R] Writing results into test-output.txt
[R] Receiver finished downloading file data.
```

You can check that the file was received intact by comparing the input and output files. In this project, we can use the `diff` command to do this. Run

```
diff test-output.txt test-input.txt
```

If you see a blank output, that means the files are exactly the same. The entire file was reliably and correctly received!

Step 1.2: Test with packet loss

Unfortunately, real networks can experience loss of packet data, and UDP sockets don't provide a reliable interface over lossy networks. Let's try our file upload/download over a lossy network. On separate terminals, run

```
python3 receiver.py
```

and

```
python3 sender.py
```

By default (i.e. with no options to `receiver.py`), the communication channel deterministically drops every 3rd packet that is transmitted by the sender. The `sender.py` program happily thinks it has sent all the data correctly. However, there are mistakes in the downloaded file. The content corresponding to messages 1, 4, 7, and 10 are only in the input, but not in the output!

```
$ diff test-output.txt test-input.txt
0a1
> msg   1
2a4
> msg   4
4a7
> msg   7
6a10
> msg  10
```

You will see from the `receiver.py` output that the receiver indeed detects these losses through the gaps in the sequence numbers. Specifically, there are messages that say "fresh data creating seq space hole", corresponding to data received after dropped sequence numbers:

```
[R]: Receiver socket created
Received    Seq: 5   ACK: 2345367   Len: 2   Msg: 88
Transmitted Seq: 235347   ACK: 7   Len: 0   Msg:
Received    Seq: 15   ACK: 2345367   Len: 8   Msg: msg   2
[R] Fresh data creating seq space hole
Transmitted Seq: 235347   ACK: 23   Len: 0   Msg:
Received    Seq: 23   ACK: 2345367   Len: 8   Msg: msg   3
Transmitted Seq: 235347   ACK: 31   Len: 0   Msg:
Received    Seq: 39   ACK: 2345367   Len: 8   Msg: msg   5
[R] Fresh data creating seq space hole
Transmitted Seq: 235347   ACK: 47   Len: 0   Msg:
Received    Seq: 47   ACK: 2345367   Len: 8   Msg: msg   6
Transmitted Seq: 235347   ACK: 55   Len: 0   Msg:
```

```
Received    Seq: 63   ACK: 2345367   Len: 8   Msg: msg   8
[R] Fresh data creating seq space hole
Transmitted Seq: 235347   ACK: 71   Len: 0   Msg:
Received    Seq: 71   ACK: 2345367   Len: 8   Msg: msg   9
Transmitted Seq: 235347   ACK: 79   Len: 0   Msg:
Received    Seq: 87   ACK: 2345367   Len: 8   Msg: msg  11
[R] Fresh data creating seq space hole
[R] Writing results into test-output.txt
[R] Receiver finished downloading file data.
```

Step 1.3: Understand different modes of loss

You can experiment with many different kinds of loss behaviors for this unreliable network channel. You can configure the loss behavior through command line flags at the receiver. Here is a description of these flags you can supply to `receiver.py`.

- `-pktloss everyn -pktlossN N`: The network is configured to deterministically drop every `N`th packet transmitted to the receiver, starting from packet number `N - 1`. Similar loss behavior can be effected on ACKs by using the flag `-ackloss everyn -acklossN N`.
- `-pktloss alteveryn -pktlossN N`: For every `N` packets transmitted by the sender, the network deterministically drops packets `N-3` and `N-1`. This creates not just one hole, but two holes in the sequence space. Similar loss behavior can be effected on ACKs by using the flag `-ackloss alteveryn -acklossN N`.
- `-pktloss iid -pktlossN N`: The network drops each packet probabilistically with an independent identical probability (I.I.D) of `1/N`. Similar loss behavior can be effected on ACKs by using the flag `-ackloss iid -acklossN N`.
- `-pktloss noloss` ensures that no packets are dropped. The flag `-ackloss noloss` ensures that no ACKs are dropped. This is helpful to debug your program.

Packet loss and ACK loss are orthogonal behaviors that can be configured independently. For example, you could ask that every 5th data packet is dropped and every 8th and 10th ACK is dropped. This is achieved by invoking the receiver script with the flags `-pktloss everyn -pktlossN 5 -ackloss alteveryn -acklossN 10`.

Your task in this project is to design a sender that is reliable against all of these modes of packet and ACK loss. You will achieve this by implementing a stop-and-wait reliable sender. Then, you will also improve the file download time by implementing a pipelined reliable sender.

Step 2: Understand `sender.py`

It is worthwhile to spend a little time understanding the code in `sender.py`. Here are some of the salient pieces:

Message class `Msg`. Objects of this class represent our own custom protocol message at the application layer. The message contains a sequence number (for data packets), an ACK number (for ACKs), a payload length (corresponds to the length of application data in the message), and finally the actual application content. These fields are separated by the pipe (‘`|`‘) character if you inspect the packet. The `Msg` class has helper methods to construct application “packets” with specific sequence number, ACK, and application-layer messages, and also to construct a `Msg` object out of a message that is received on the wire.

Helper methods. There are several helper functions to initialize a socket, read an entire file and chunk this data into packets for transmission, and to parse command line arguments. You must implement `transmit_entire_window_from(x)`. Inside the while loop, you will find a `TODO` comment that indicates where you must add your code. You can use `seq_to_msgindex` to map sequence numbers to indices, and call `sendto()` on `cs` to send a message to the receiver. You may refer to `transmit_one()` for an example of how to send a single packet. You should implement this helper method before moving on to the next steps.

The reliable sending function `send_reliable`. This is where you’ll be spending most of your time working on this project. This function has two *internal* helper methods `transmit_one()` and `transmit_entire_window_from()` which are useful to transmit one and multiple packets respectively.

When you’re implementing stop-and-wait reliability, you’ll only need to use `transmit_one()`. Pay particular attention to how this function calls `sendto()` on the socket, rather than `send()` that we’re used to in TCP. This is a key difference between TCP and UDP; UDP senders (usually) state who the intended recipient of each message is because they are connectionless and intended for one-off packet transfers. A similar socket call for receiving data, `recvfrom()`, returns both the content of the data as well as the sender who sent it, unlike `recv()` which only returns the data.

When you implement pipelined reliability, you will call `transmit_entire_window_from(x)` to transmit all data with sequence numbers starting at `x`. For example, `x` could be the “left” edge of the window (`win_left_edge`) which is the first unACKed sequence number in order. The value `x` could also be somewhere further into the window ahead of `win_left_edge`. The function will transmit all data up to the “right” edge of the window, i.e. the latest sequence number that is allowed to be transmitted by the sender given where the window is.

The window size is configured through the `-winsize` flag and defaults to a value of 20 bytes. You only need to use the window size when you implement and test pipelined reliability.

At the end of this function comes the most interesting part of the function’s logic where you will make your changes. This region of code is marked by a `TODO` in the code. In the starter code, there is a `while` loop that transmits one packet each time, running until the left edge of the window exceeds the last valid sequence number of the data, i.e. until `win_left_edge >= INIT_SEQNO + content_len`. If you think about what this code is doing, this sender isn’t a reliable sender at all; the sender merely transmits the packets one by one without checking whether they were actually delivered to the receiver!

Constants. The initial sequence number of a stream in our protocol can be any arbitrary value. The starter code for `sender.py` uses the value `INIT_SEQNO = 5`. The size of the application payload in each packet is set in the variable `CHUNK_SIZE`, which is 8 bytes. The retransmission timeout is set to a constant `RTO = 500 ms`. **Please do not change these constants.**

The `main` thread. The code block under `if __name__ == "__main__"` shows the main thread of execution of the sender. The command line arguments are parsed into the Python dictionary `args`. The rest of this block simply constitutes calls to various functions including `send_reliable()`. The flow of logic here should be straightforward to understand.

Step 3: Stop-and-wait reliability

As discussed in lecture, stop and wait reliability requires a sender to (1) wait for an ACK before transmitting the next piece of data, (2) if ACKs don't return within a timeout (RTO), retransmit the data, (3) distinguish different pieces of data using sequence numbers. The existing starter code already simplifies your life quite a bit: `receiver.py` already sends ACKs, and `transmit_one()` already sends data with the appropriate sequence numbers to help the receiver disambiguate fresh transmissions from retransmissions.

Step 3.1: How to implement stop-and-wait

Your main task in this step reduces to waiting for an ACK of a transmitted packet, and either (i) transmitting the next piece of data if an ACK is successfully received, or (ii) retransmit the old packet if no ACK was received before a timeout (use the variable `RTO` for the timeout).

You must implement a receiver that times out waiting for data, in case the packet or an ACK is lost. We recommend learning and using the `select()` socket call, which allows you to be notified (among other things) whenever there is readable data on any socket among a list of sockets or if a timeout has elapsed. See:

https://docs.python.org/3/library/select.html

and

https://docs.python.org/3/library/select.html#select.select

If you have readable data returned after this call, you can use the socket `recvfrom()` method to obtain readable data. Here is some example code you could use to parse the data that you've read from `recvfrom()`:

```
# r is a readable socket
data_from_receiver, receiver_addr = r.recvfrom(100)
ack_msg = Msg.deserialize(data_from_receiver)
print("Received    {}".format(str(ack_msg)))
```

You can obtain the acknowledgment number on the received ACK (really, the only useful information in the ACK) by reading `ack_msg.ack`. Like TCP, the sender and receiver in this project use **byte-based sequence and ACK numbers**. Further, like TCP, the ACK number returned by `receiver.py` is the **next expected sequence number.**

Once you receive an ACK, you can advance the left edge of the window `win_left_edge` to the acknowledgment number in the ACK. Check if this window edge has advanced past the last sequence number you need to transmit, and terminate your sender appropriately. Otherwise, loop to transmit the next message using `transmit_one()`.

Step 3.2: Test your code

Use all the lossy scenarios described in step 1.3 above and transmit files `test-input.txt`, `medium-input.txt`, and `long-input.txt`. The difference between the uploaded and downloaded versions of the file (as checked with the `diff` command) should be empty. We recommend checking with the following cases:

1. no packet loss, no ACK loss
2. packet loss `everyn` with `N = 3`, no ACK loss
3. packet loss `everyn` with `N = 10`, no ACK loss
4. no packet loss, ACK loss `everyn` with `N = 3`
5. no packet loss, ACK loss `everyn` with `N = 10`
6. packet loss `alteveryn` with `N = 8`, no ACK loss
7. no packet loss, ACK loss `alteveryn` with `N = 8`
8. packet loss `iid` with `N = 5`, no ACK loss
9. no packet loss, ACK loss `iid` with `N = 5`
10. packet loss `everyn` with `N = 3`, ACK loss `alteveryn` with `N = 4`
11. start the sender first, give a few seconds, then start the receiver (with any of the loss behaviors mentioned above)

In all of the cases above, a correct sender will:

1. eventually *finish execution* by making progress through the sequence numbers corresponding to the application message. File upload must take just a few seconds for `test-input.txt`. Other files may take a bit longer, but the sequence numbers being transmitted (as printed) must clearly advance over time.
2. return an *empty* `diff` between the uploaded and downloaded files for *all loss behaviors mentioned above.* and *all input (upload) files* provided in the project archive.

If all of this is working as intended, freeze your code by copying `sender.py` into a separate file, `stopandwait.py`. You will submit `stopandwait.py` in addition to the final version of `sender.py` after step 4 (described below).

Step 4: Pipelined reliability

If you've made it this far, you've already built a reliable sender out of an unreliable network! Your task in this step is to make this sender more efficient while still being reliable. Further, you will implement selective repeat through cumulative acknowledgments.

The first step to implement pipelining with selective repeat is to ask the receiver to buffer out-of-order data so that the sender does not need to resend all data after a packet loss. The receiver-side functionality needed for this is already available: just invoke the receiver in the following way (you may include any additional flags you need for the intended loss behaviors):

```
python3 receiver.py --ooo_enabled
```

Proceed to implement the pipelined sender with the following additional steps.

(Step 4.1) Transmit a window of packets first. When transmission begins, use the function `transmit_entire_window_from()` to send a burst of packets starting from the current left edge of the window, `win_left_edge`.

(Step 4.2) Preliminaries It helps to initialize and maintain a variable `last_acked` that tracks the highest cumulative ACK number received so far. (Recall that you can retrieve the ACK number from an ACK through the `.ack` member of a `Msg` object.)

It also helps to separately initialize and maintain the first sequence number that must be transmitted by the sender after a number of previous transmissions. Let's call this value `first_to_tx`. After each transmission of application data, you may use the value returned by the functions `transmit_one()` or `transmit_entire_window_from()` to set `first_to_tx`.

Finally, it is useful to determine what the final cumulative ACK from the receiver (i.e. after successful reception of all data) must indicate on its acknowledgment number field. The `final_ack` is just `INIT_SEQNO + content_len`. We will use `final_ack` to determine when to terminate the sender.

Next, enhance your stop-and-wait implementation with these additional checks whenever you have ACKs after a `recvfrom()` call.

(Step 4.3) If the ACK acknowledges fresh data, slide the window forward. The slide must reflect changes to the variables corresponding to the left window edge (`win_left_edge`), the right window edge (`win_right_edge`), and the cumulative ACK number (`last_acked`).

(Step 4.4) Is there more data to transmit? If the window has slid forward, the sender now becomes free to transmit more data, assuming that the right edge of the window hasn't gone beyond the set of available sequence numbers to transmit (bounded by `final_ack`). Call the `transmit_entire_window_from()` function supplied with the first sequence number that must be transmitted, `first_to_tx`.

(Step 4.5) If there is no more fresh data, have all required ACKs been received? This can be determined by checking whether `last_acked` is the same as `final_ack`. At this point, we can terminate all transmission and reception.

If there is either more fresh data to transmit, or not all ACKs have been received, the sender should wait for ACKs (same as it did with stop-and-wait) to help slide its window forward for the subsequent transmissions or finish receiving all ACKs.

(Step 4.6) Handling timeouts. Ensure that you only `transmit_one()` after a timeout, since we are using selective repeat, and the receiver is buffering out-of-order data to reduce unnecessary transmissions.

In a true pipelined reliable sender, there is a distinct timeout for each sequence number in flight. However, the `select()` call implements a timeout on the entire socket. The distinction between a timeout per socket and a timeout per sequence number did not matter in stop-and-wait, because all sequence numbers in flight will either be ACKed together or none of those sequence numbers will be ACKed at all. However, with pipelined reliability, if you use a `select()` call as you did in step 3.1, a single lost packet may not trigger a retransmission timeout on the socket if the sender is continuing to receive any data at all (usually duplicate ACKs) on the socket.

In this project, we will keep things simple and just implement a single RTO on the socket rather than one RTO per sequence number. So, you can keep the `select()` implementation from step 3.1.

(Step 4.7) Test your code. Your code should continue to run correctly with all the test cases in step 3.2. In addition, you can test multiple window sizes. We suggest testing with three values, the default (20 bytes), one medium (200 bytes), and one large (2000 bytes). You can change the sender's window size using the `-winsize` flag. A correct pipelined sender should clearly run faster than the corresponding stop-and-wait sender.

What you must submit

For your project submission on Canvas, please turn in `stopandwait.py` (unmodified after step 3), `sender.py` (after step 4), and your project report `report.pdf`. You cannot change the command line arguments for the programs relative to the starter code. The questions for the report are listed below. We will be running your code on the `ilab` machines with the default Python 3 version on those machines. **Please compress the files into a single Zip archive before uploading to Canvas.** We will not accept modified versions of `receiver.py`; your senders must work with the unmodified receiver script.

Testing your programs. We have already described how we will test your programs in steps 3.2 and 4.7 above.

Project report. Please answer the following questions for your project report.

1. Describe **two** technical observations or facts you learned while working on this project. Please answer in specific and precise terms. Your observations could relate to topics including reliable data delivery in general, your implementation of it in this project, reliable delivery in TCP, using UDP sockets, or other topics that are relevant to your software and implementation in this project. **Please ensure your responses are clear, specific, and technical.**
2. Is there any portion of your code that does not work as required in the description above? Please explain.
3. Did you encounter any difficulties? If so, explain.
4. What did you learn from working on this project? Add any interesting observations not otherwise covered in the questions above. Please be specific and technical in your response.