Project #2: QUIC-like reliable streaming transport protocol over UDP

Northwestern CS340, Fall 2022

1 Before starting

- Project #2 should be done in groups of two or individually.
- Please carefully read the instructions including all references and footnotes.

2 Overview

In this assignment, you will design a reliable streaming transport protocol on top of UDP. Essentially, you are building a simplified version of TCP.

- Part 1 will deal with chunking (breaking the stream into packets).
- Part 2 will deal with packet reordering.
- Part 3 will implement a simplified solution to deal with packet loss (with stop-and-wait ACKnowledgements and retransmissions).
- Part 4 will deal with data corruption errors (with hashes).
- Part 5 will introduce pipelining to improve performance.
- The Extra Credit will let you compete to pass the tests as efficiently as possible.

For all this class' projects, We strongly recommend that you write and run your code using PyCharm[1] (or another Python IDE). This will give you powerful debugging tools, like breakpoints and expression evaluation. After the code works on your machine, copy it to `moore.wot.eecs.northwestern.edu` for the final test (there should be no surprises there).

Run `git commit -a` when you are done with each part of the assignment. Committing your working intermediate code will allow you to experiment freely with drastic changes while being able to examine the differences you've introduced (`git diff`) and to roll back to the previous version (using `git reset --hard`). If you're new to git, then it would also be a good idea to save a backup copy of your `Streamer.py` to another folder each time you complete a part.

3 Part 0: The Skeleton Code

Clone the following git repository, which will give you the framework for this assignment:

`https://github.com/northwestern-cs340/reliable-transport-sim`

[1] JetBrain's PyCharm: `https://www.jetbrains.com/pycharm/`

This is written in Python 3 and includes just three simple files:

1. **lossy_socket.py** provides the class LossyUDP, which is a subclass of Python's BSD socket class. This class gives us a UDP socket with simulated packet loss, corruption, and out-of-order delivery. You **should NOT** modify this file, but it may be helpful to read it to understand how it works. Your code will call four functions in this file: the LossyUDP constructor, LossyUDP#**sendto**, LossyUDP#**recvfrom**, and in Part 3 it will also call LossyUDP#**stoprecv**.
2. **streamer.py** provides a skeleton for the Streamer class which you will implement for this project. It will implement a reliable streaming transport protocol (similar to TCP). Like TCP, it provides a simple interface to applications, but internally it must manage several complex details. You must add your implementation code to **streamer.py**, however you may **not** change the existing function signatures (declarations) for the Streamer constructor, Streamer#**send**, Streamer#**recv**, and Streamer#**close**. The grading scripts (and the provided test.py) will expect to interact with Streamer using the functions already defined.
3. **test.py** is a simple tester. This is an example of an application using our new streaming transport protocol. As written, it will test Part 1. After you have cloned the repository, the first part of the test will pass, but it will fail when host2 tries to send a very large chunk of data. Notice the line with `loss_rate=0.0, corruption_rate=0.0, max_delivery_delay=0.0`. You may change these values to test later parts of the project. This tester is just meant to get you started. You should write more tests.

We suggest you start by running the tester. Open two terminals. In the first terminal run:

```
$ python3 test.py 8000 8001 1
```

In the second terminal run:

```
$ python3 test.py 8000 8001 2
```

You should see that the two instances of the test program communicate the numbers counting from 0 to 999 by sending a series of small packets. These small messages flow from Instance 2 to Instance 1. After this finishes, the program will try to communicate the same numbers in the reverse direction (from Instance 1 to Instance 2) but it will try to send all the numbers in one big chunk (with one call to Streamer#send). At this point Instance 1 will fail and Instance 2 will be stuck wait for data from Instance 1.

Let's explore why it failed. Have a look at the **streamer.py** file that implements the streaming transport protocol. It defines the Streamer class, which has three parts:

1. The **constructor** (`__init__`) takes as parameters the source ip address and port and the destination ip address and port. It also lets you specify parameters for the simulator to control packet loss, packet corruption, and packet delay (for reordering).
2. The **send** function takes a parameter called "data_bytes" (which is of type "bytes") to send the next segment of the data stream. Notice that we are using Python's type hints[2] to document the function signature. The initial implementation of Streamer#**send** just passes the raw message to the LossyUDP socket. That works fine in the first half of the test because messages are small, but once messages exceeded 1472 bytes this caused an error. In

[2]hints: https://docs.python.org/3/library/typing.html

Part 1 you must modify this function to break the data_bytes into chunks small enough to be handled by LossyUDP#**sendto**. You will also change this function in the later parts.

3. The **recv** function takes no parameters and it returns an object of type "bytes" when the next segment of the data stream has arrived. The initial implementation of Streamer#**recv** just calls LossyUDP#**recvfrom** and it passes the data received from the UDP socket directly to the caller (the application). In Part 2 you must modify this function to check the sequence number of the segment before delivering it, and later parts will add more complexities.
4. The **close** function is called when the application is finished sending data. This allows you to do any cleanup that may be necessary. Please notice that LossyUDP#**stoprecv** can be called to cancel a blocking call to LossyUDP#**recvfrom**.

4 Part 1: Chunking

This part is worth 20% of the total grade, if completed correctly.

Allow Streamer#**send** to support data larger than 1472 bytes. Break the data_bytes into chunks and send the data in multiple packets. The provided tester.py should pass after you implement chunking.

Once you have a working solution to each part, we recommend that you do a `git commit -a` and then develop the next part on top of the previous solution. Each part builds on the prior part, and you will submit one final solution implementing all parts.

5 Part 2: Reordering

This part is worth 20% of the total grade, if completed correctly.

Start by changing test.py to set `max_delivery_delay=0.1`. Packets will be delayed a random amount (in this case, up to 0.1 seconds) before delivery. This will cause packets to be re-ordered.

Your task in Part 2 is to make Streamer tolerate this reordering by adding **sequence numbers** and a **receive buffer**. You will have to add a segment **header** to store the sequence numbers. You may find it helpful to use the struct[3] library for headers.

6 Part 3: Packet Loss (with stop and wait)

This part is worth 30% of the total grade, if completed correctly.

Your task in Part 3 is to make Streamer tolerate this packet loss by adding **ACKs**, **timeouts**, and **retransmissions**. However, we will make your job easier by allowing you to limit your Streamer to sending one packet at a time (your code will stop and wait for an ACK after sending each packet). This will lead to very slow performance, which you will fix later in Part 5.

[3]struct library: https://docs.python.org/3/library/struct.html

Even with the stop-and-wait simplification, this part of the project will involve many changes. We suggest that you break your implementation into three stages:

- **Stage A**: Listen in a background thread. Verify that you still pass the tests from Part 2 (`loss_rate=0` and `max_delivery_delay=0.1`).
- **Stage B**: Add ACKs (although we don't need them yet because packets are not being dropped). Again, verify that you still pass the tests from Part 2.
- **Stage C**: Add timeouts and retransmissions. Set `loss_rate=0.1` so that 10% of sent packets (whether data or ACKs) will be discarded instead of delivered.

6.1 Stage A (background listener)

In this stage, you should create a background **thread** that listens continuously for new packets in an infinite loop. This will prepare us for Stage B.

About blocking and threads

In an Operating Systems class you would learn that "blocking" is when the OS pauses a program's execution because it called an OS function (a "system call") that must wait for some external operation to finish before the program can proceed. For example, reading data from disk or waiting for data from a network connection can be blocking operations.

In the first project, we used the select function to avoid blocking on one socket when another might be ready to provide data. In this project, we are instead going to use threads to prevent blocking operations from blocking our entire application. A thread is just an execution path in a program. Every program starts with one thread, but you can add additional threads if you want multiple things to happen concurrently.

To complete Stage A

- Move your receive buffer into a "data attribute" (a.k.a. instance variable) (*e.g.*, `self.receive_buffer`) so it can be shared by both threads.
- Define a function for your background thread.

  Hint: if a background thread experiences an exception it will die silently. To prevent this, you should wrap your listening in a try/catch statement like

```
def listener(self):
  while not self.closed: # a later hint will explain self.closed
    try:
      data, addr = self.socket.recvfrom()
      # store the data in the receive buffer
      # ...
    except Exception as e:
      print("listener died!")
      print(e)
```

- At the end of Streamer#__init__ (the constructor), start the listener function in a background thread using concurrent futures with ThreadPoolExecutor (see https://docs.python.org/dev/library/concurrent.futures.html).

```
executor = ThreadPoolExecutor(max_workers=1)
executor.submit(self.listener)
```

- As in Part 2, received data packets should be buffered. Calls to Streamer#recv can just pull data from the receive buffer if the next segment is available (otherwise wait for the buffer to be filled by the background thread).
- You'll need to add some code to Streamer#close to stop the listener thread so the program can quit. Do this by calling:

```
self.closed = True
self.socket.stoprecv()
```

Do not proceed to Stage B until you can pass the tests with reordering but no packet loss.

6.2 Stage B (Add ACKs)

1. Add a header to your packet format to indicate whether a packet is data or an ACK.
2. In your background listener, check whether the received packet was an ACK or data. If the received packet was an ACK, then store something that can be checked by the main thread (the next step).
3. At the end of Streamer#send, add code that waits until an ACK has been received by the background listener. In other words, do not consider the send complete until after an ACK has been received. We suggest implementing a loop with a short sleep in between each test (the sleep reduces the CPU burden of "busy waiting")

```
while not self.ack: time.sleep(0.01)
```

Do not proceed to Stage C until you can pass the tests with reordering but no packet loss.

6.3 Stage C (Timeouts and retransmissions)

Set `loss_rate=0.1`. Your tests should now hang because it waits for an ACK that won't ever arrive. To fix it:

- Add a timeout when waiting for ACKs. To simplify your implementation, you may hard-code an ACK timeout of 0.25 seconds and assume that we will test with `max_delivery_delay` ≤`0.1`. If you still have no ACK after 0.25 seconds, then repeat (resend the data and again wait for an ACK).
- In Streamer#close, you will have to implement some kind of connection teardown request (analogous to a TCP FIN request). This is necessary to confirm with the other host that both

parties have received all the ACKs they might be waiting for. Roughly, these are the steps I would follow in Streamer#close:

1. Wait for any sent data packets to be ACKed. Actually, if you're doing stop-and-wait then you know all of your sent data has been ACKed. However, in Part 5 you'll add code to maybe wait here.
2. Send a FIN packet.
3. Wait for an ACK of the FIN packet. Go back to Step 2 if a timer expires.
4. Wait until the listener records that a FIN packet was received from the other side.
5. Wait two seconds.
6. Stop the listener thread with

```
self.closed = True and self.socket.stoprecv()
```

7. Finally, return from Streamer#close

- Notice that your listener thread must be updated to deal with the arrival of a FIN packet.

Additional Explanation and Tips:

1. ACKs may be lost, so we send a FIN packet when we are done sending data and we have received ACKs for all previously-sent data.
2. The FIN message must be ACKed and either the FIN message or its ACK may be lost! The way TCP deals with this is for the socket to remain alive for a certain amount of time even after the final ACK has been sent. This allows the FIN message to be ACKed again if it is retransmitted. For this assignment we suggest you continue listening and responding for a two second "grace period" after sending your last ACK.
3. It can be tricky to implement a connection teardown handshake (like TCP FIN) if you allow for the possibility of packet loss. There is a simulation parameter `become_reliable_after` which tells the simulator to stop dropping packets after a certain number of seconds. You might want to set this to a value like 10.0 when you start Stage C. This will allow you to focus first on getting the basic retransmissions working. After that's working then set `become_reliable_after` back to the default of a very large number and focus on getting the FIN handshake working.
4. You may find it helpful to set `max_delivery_delay=0` during your initial testing, but your final solution should work with **both** reordering and packet loss.
5. When debugging your code, you may find it helpful to reduce the NUMS variable in test.py to the smallest possible value that still exhibits your bug. This will let you focus on just the first appearance of the bug.
6. The default implementation of `lossy_socket.py` (see line 8 of the code in this link!) seeds the random number generator with a constant. This forces the exact same packets to be dropped every time you run the tests. After you've passed your tests, try removing this line and run the tests a few more times. This will simulate different patterns of packet losses.

7 Part 4: Corruption

This part is worth 15% of the total grade, if completed correctly.

Start by changing test.py to set `corruption_rate=0.1`. 10% of sent packets will have **one or more** bit flipped (zero to one or one to zero).

Your task in Part 4 is to make Streamer tolerate these bit flips by adding a **hash** of the packet data to the segment header, and discarding any received segments which fail the hash test. (TCP and UDP use a checksum instead of a hash, but that only works for a single bit flip.) The code you implemented in Part 3 will cause the corrupted packet to be retransmitted.

After Part 4, your tests should pass even when the tester is simultaneously allowing packet loss, and corruption. However, performance will be slow because of the "stop and wait" design.

When we say "compute a hash of the packet data" we don't mean calling `hash(my_message)`*. This will call the* `__hash__` *method of the* `my_message` *object which is not guaranteed to be deterministic; it will not necessarily give the same answer on both the sender and receiver. Instead use a standard (deterministic) hash function like* **md5***. You can use the* **hashlib** *library as shown in the first code example on this page:* *https://docs.python.org/2/library/hashlib.html*

8 Part 5: Pipelined ACKs

This part is worth 15% of the total grade, if completed correctly.

Now we will allow multiple packets to be "in flight" at once ("pipelining"), which greatly increases throughput. You may choose either the "Go Back N" or "Selective Repeat" style. Set `max_delivery_delay=0.1` to allow for in-flight packets to be **reordered** before delivery.

Tips:

1. Your implementation of Streamer#close should wait for any in-flight packets to be ACKed before sending a FIN packet.
2. If you are using Selective Repeat, you may want to use Python's `threading.Timer` (check out this link!) to handle the scheduling of future retransmission checks. Timer lets you run a function (or a lambda) after a timeout. The function will be run in another thread.
3. However, if you create too many Timers (and thus threads) you'll get weird runtime errors. To avoid this, we suggest you use just one timer in your implementation. In other words, do something like Go-Back-N or TCP instead of selective repeat.
4. Don't worry about **window** size; let the window be infinite. There is a 10ms pause (take a look at the code) built into my implementation of LossyUDP#sendto. One of the reasons I did this was to simplify the problem and prevent you from worrying about either the receiver or the network from being overwhelmed by too many packets. This delay limits the total connection throughput to just 147kbyte/s (100 UDP packets per second 1472 bytes/packet).
5. The concurrency introduced by both the background listener and the multiple timers have the potential to introduce some tricky race condition bugs. This topic is covered in an OS class. The Python `threading.Lock` objects provides a way to prevent two threads from accessing the same shared data simultaneously. For example, the code below uses a lock to ensure that `get_sum` always returns zero. Only one thread may be running code inside the `with`

`self.lock` (only one thread may "hold the lock"). Without the lock, it might be possible for `get_sum` to return "1" because it accessed a and b in the middle of the update function's work.

```
__init__(self):
  self.lock = Lock()
  self.a = 0
  self.b = 0

def update(self):
  with self.lock:
    self.a += 1
    self.b -= 1

def get_sum(self):
  with self.lock:
    return self.a + self.b
```

In the end, your tests should pass even when the tester is simultaneously allowing packet loss, corruption, and reorderings. Your solution should be much faster than the code you wrote for Part 4, so try setting the NUMS variable in test.py to a much larger value than you used in Part 3 and ensure that your code still works.

9 Extra Credit: More Performance Optimizations

This part is worth an extra 10% of the total grade, if completed correctly.

In the extra credit you should implement **piggy-backed ACKs**, delayed ACKs[4] and Nagle's Algorithm[5] to minimize the total number of bytes transmitted during testing.

Notice that LossyUDP tracks the total number of packets and bytes sent and received. At the end of execution it prints the values like the following example:

```
PACKETS_SENT=1
UDP_BYTES_SENT=3890
ETH_BYTES_SENT=3936
PACKETS_RECV=1000
UDP_BYTES_RECV=3890
ETH_BYTES_RECV=49890
```

Every UDP packet has

- 18 bytes for the Ethernet header
- 20 bytes for the IPv4 header

[4]Delayed ACK: https://en.wikipedia.org/wiki/TCP_delayed_acknowledgment
[5]Nagle's Algorithm: https://en.wikipedia.org/wiki/Nagle%27s_algorithm

- 8 bytes for the UDP header

We'll discuss IP and Ethernet later in the quarter, but you have probably noticed these headers when using Wireshark. ETH_BYTES_SENT and ETH_BYTES_RECV includes this overhead in the calculation. Sending lots of little UDP packets is really inefficient! Your goal in the extra credit is to minimize ETH_BYTES_SENT and ETH_BYTES_RECV.

There is a tradeoff between latency and efficiency in some protocol design decisions. In implementing the extra credit, the total time needed to complete the tests may increase **slightly**, but that is OK.

Notice also that the extra credit gives you an incentive to make your reliable transport header as small as possible. In particular, you don't want to waste space with large sequence numbers. If you choose a small maximum sequence number (that loops around to zero) then you will have to limit your window size to prevent old packets from being confused for new ones.

10 Submission guidelines

- You should work in pairs. List the participants (names and net IDs) in a README.txt file.
- Please make just one submission for the pair. The parts build on each other, so submit the solution for the last part that you are able to complete.
- Your code must compile and run on `moore.wot.eecs.northwestern.edu`.
- You should include all python source file, most importantly `streamer.py`.
- You may not make any modifications to `lossy_socket.py`.
- You may not use any outside libraries to in your code (do not `pip install` anything).
- Your submission should be a .tgz file (a gzipped tarball), including a README.txt, and the python sources. To create the archive, use a command like the following

```
tar -czvf project2_NETID1_NETID2.tgz README.txt *.py
```

- Please make sure to include both partners' net IDs in the filename.
- Remember that you should leave a comment explaining any bits of code that you copied from the Internet (just in case two groups copy the same code from the Internet, we won't think that you copied each other directly).
- After you make your tarball, copy it to a temporary folder, extract it, copy over the original test.py and lossy_socket.py, and test it again to make sure you included everything.

11 Sharing files with your partner using git

We recommend you to create a private repository on GitHub to share your code with your partner, to allow better collaboration. **Please make sure to set the repositories as private to avoid violating the cheating policy**.