Assignment 2 (60 points)

Anirudh Sivaraman

2021/10/09

Before you start

This assignment is almost entirely programming with a few written questions that you will answer based on the code you write. Hence, most of the instructions are in the starter code for this assignment. We have also included an autogenerated PDF that provides you a code walkthrough; it contains information about the important Python classes in the starter code and is generated by combining together comments in the starter code. Please read the code walkthrough PDF in addition to this one.

Here, we'll summarize information that is not in the starter code. In addition to python3, you'll should install the python libraries for matplotlib, a graph plotting library. You should be able to install matplotlib by following the instructions here: `https://matplotlib.org/users/installing.html`. If you're using the VM image, which we recommend you do, matplotlib is already installed on it.

This assignment will take more time than assignment 1. Please start early and ask for help if you're stuck. Use Gradescope to upload your final assignment submission. When writing the code for the tasks, provide as much detail and comments as you think is appropriate, but err on the side of providing more rather than less detail for a question. In the case of a regrade request, we'll read and consider everything you have turned in.

The assignment will also teach you how to develop protocols using a network *simulator*, a computer program that imitates the behavior of the essential components of a network. For this assignment, we'll be using a custom simulator developed solely for assignment 2. I'll list some of the non-obvious details of the simulator here because they might help you when debugging your solutions to assignment 2.

The multiple choice questions on Gradescope strongly correlate to the individual tasks you will be implementing in this assignment. It is recommended to answer them after you have implemented the respective task. The questions posed in the tasks **here** are meant to make you think about the subject material. **They do not need to be answered and do not count for points.**

The assignment grading will largely be automated. So please **only edit** the code that is part of the TODO's. In particular, make use of all the attributes listed in the classes and do not remove them. Also avoid using custom imports that are not part of the standard Python distribution. You should not need them.

Randomness in the simulator. The simulator uses a random number generator to generate independent and identically distributed (IID) packet losses[1] when packets are dequeued. This is in addition to packet drops if a queue overflows when packets are enqueued. You do not need to understand probability for this assignment, but you should be aware of the fact that randomness causes non-determinism in simulations. For instance, if you run two simulations with identical settings and the same loss rate, you might see different outputs because the sequence of packet drops will be randomly generated. To make such random simulations deterministic, you can use a *seed* to initialize a random number generator so that it generates the same sequence of random numbers. If you pass the same seed to the simulator in two different simulation runs with the same settings, the output will be the same. Also, if you don't use random losses in your simulation (i.e., you set the loss ratio to 0), then your output should be deterministic because no other simulator component uses a random number generator.

[1] In other words, every packet is dropped independent of every other packet, but we use the same probability to decide whether to drop a packet or not.

Sequence numbers. As a matter of convention, sequence numbers for packets start from 0 (i.e., the first packet has sequence number 0). Also, we'll be dealing with providing reliability at the level of packets, not bytes.

Link capacity in the simulator. The link capacity in all our simulations is fixed to 1 packet per simulation tick. This seems like an arbitrary restriction, but it simplifies the implementation of the simulation and the assignment, without losing anything in terms of what you will learn from the assignment. All time-based quantities (RTT, timeout, RTT_{min}, queuing delay, etc.) in this assignment are measured in units of simulation ticks, an arbitrary unit of time we use in our simulations.

RTT_{min} in the simulator. Because time is quantized to ticks in our simulation, an RTT_{min} of 10 ticks will result in a delay of 11 ticks between when the first packet is sent out in StopAndWait and when the second packet is sent out. This is because the first packet will be sent out in tick 0, its ACK will be received 10 ticks, later, and then only in the next tick (tick 11) can the second packet be sent. For this reason, we subtract one from the user-specified RTT_{min} before feeding it into our simulator. This is not particularly important, but it's good to know in case you wonder why there is some odd-looking code in `simulator.py`.

Senders and receivers. Because this is a simulator, we get to simplify as much as we want, while still imitating network components relevant to what we're trying to study. Hence, the sender and the receiver are really the same Python object called host. This host could belong to one of three classes: StopAndWaitHost, SlidingWindowHost, or AimdHost. All three classes implement two methods `send()` and `recv()` corresponding to the sender and receiver respectively.

Running the simulator. Running `python3 simulator.py -h` or `python3 simulator.py --help` should give you the usage for the simulator. For instance, the simulator lets you set RTT_{min}, the limit on the number of packets in the link's queue, the loss ratio, the window size for the sliding window protocol and so on. The simulator reports the max sequence number that has been received *in order* at the end of the simulation period. Adding 1 to this gives you the number of packets that have been received in order. Dividing the number of packets by the simulation period will give you the transport-layer throughput.

Python Linters. Using a python linter is a good idea to help catch errors and generally improve your coding style. For a compiled language such as C or Java, the compiler will often catch errors for you, however in the case of interpreted languages (such as python), there is no compiler to do this. To help catch the errors that a compiler might normally catch, you can use a tool called "pylint", which more scrupulously reviews your code to help catch bugs. To install, use pip3: `pip3 install pylint`. It can be run in "full output mode" as simply `pylint` or with a suppression flag enabled to edit down the warnings that it gives: `pylint -E` for example. An example of a typical warning you might get if you ran pylint on the `simulator.py` class in the homework might be: `simulator.py:38:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)`. This would tell you that you have imported something without calling it in the file, and that you should make sure it gets used to make the program run properly.

Plotting graphs. For several questions in this assignment graphs can help you understand the problem. You can use the Python library matplotlib for this purpose, and the file `ewma.py` in the starter code contains an example of how to use matplotlib. Essentially, you can use the `plot()` (`https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html`) and show() (`https://matplotlib.org/api/_as_gen/matplotlib.pyplot.show.html?highlight=show#matplotlib.pyplot.show`) functions in matplotlib. You are free to use other graph plotting programs (e.g., Excel, gnuplot) if you find them more convenient than matplotlib.

1 Moving averages and retransmission timeouts (10 points)

1.1 Moving averages (5 points)

Implement a simple exponentially weighted moving average (EWMA) filter by translating the equations for the EWMA from lecture 4 into Python code. We have provided starter code in `ewma.py`. You'll use your implementation to understand how α (also called the gain of the EWMA) affects how quickly the mean value calculated by the EWMA converges to the true value.

For this, we'll use a synthetic set of 100 samples. The first 70 samples are 1, and the next 10 are 2. After, we return to 1. Essentially, we are modelling a sudden burst in latency. We'll feed this to an EWMA and see how the mean estimate (also called the smoothed estimate) tracks the actual samples. Run ewma.py using four different α values: one high (1.0), one medium (0.1), one low (0.05), and one very low (0.01).

1.2 Retransmission timeouts (5 points)

Complete the TODOs in `timeout_calculator.py`, which is a class used by both the Stop-And-Wait, sliding window and AIMD protocols for their timeout calculations. Make sure that any computed timeout values always fall between `min_timeout` and `max_timeout`. You'll be able to test out your retransmission logic as part of the Stop-And-Wait and sliding window implementations in the next two questions.

Why do we need a minimum value of the retransmission timeout? Why do we need a maximum value of the retransmission timeout? These questions might be easier to answer once you have incorporated the retransmission logic into both the Stop-And-Wait and sliding window protocols. Try disabling either the min- or the max timeout guards you have implemented and see (1) what the effect on throughput is and (2) how it affects the likelihood of congestion collapse.

2 The Stop-And-Wait protocol (10 points)

Implement the Stop-And-Wait protocol using the starter code provided in `stop_and_wait_host.py`. Run the protocol using `simulator.py`. Carry out and report on the results of the following experiments. Use the simulator's default large queue size limits (1M packets) for this experiment.

1. Make sure your implementation works. This means checking that a packet is being sent out every RTT_{min} ticks. Print out a line to the terminal every time you send out a packet. This should help you debugging your submission. You will receive 10 points if you implement the stop-and-wait protocol correctly according to the TODOs in the template file.

2. Run the Stop-And-Wait protocol for several different values of RTT_{min} . Check that the throughput tracks the $\frac{1}{RTT_{min}}$ equation we derived in class. Plot the throughput you get as a function of $\frac{1}{RTT_{min}}$ and compare it with the equation we derived in class. You can use matplotlib for this.

3. Introduce a small amount of IID loss (about 1%). Check (1) if the throughput continues to be close to the value predicted by the equation and (2) that the protocol continues to function correctly despite the loss of packets. If the throughput is much less than the value predicted by the equation, explain why, by looking at when packets are originally transmitted and when they are retransmitted. During the presence of a small amount of IID loss (about 1%), does the divergence between the simulation's throughput and the equation's predicted throughput ($\frac{1}{RTT_{min}}$) increase or decrease as RTT_{min} increases? Why?

You will receive 10 points if you implement the stop-and-wait protocol correctly according to the TODOs in the template file.

3 The sliding window protocol (10 points)

Implement the sliding window protocol using the starter code provided in `sliding_window_host.py`. Again, run the protocol using simulator.py. Use the simulator's default large queue size limits for this experiment. Answer the same three questions as the Stop-And-Wait protocol in the previous section, but remember to vary the window size as well in addition to RTT_{min}. When a small amount of loss (1%) is introduced, how does the divergence between the simulation's throughput and the equation's predicted throughput vary now as a function of *both* RTT_{min} and the window size. You will receive 10 points if you implement the window size algorithm correctly according to the TODOs in the template file.

4 Congestion collapse (10 points)

We'll now simulate congestion collapse using our simulator. For this, use the sliding window protocol. Calculate the bandwidth-delay product (BDP) as the product of the link capacity and the minimum round-trip time. Vary the window size from 1 to the BDP. Check that the transport-layer throughput matches up with the $\frac{W}{RTT_{min}}$ equation, similar to the previous question. Now vary the window size beyond the BDP all the way until $100.BDP$. Plot the transport-layer throughput as a function of the window size. Can you see the congestion collapse pattern that we discussed in class (i.e., utility goes down as offered load increases)? What is the reason for this congestion collapse? Can you provide evidence for this by looking at the number of retransmissions and the number of original packets being delivered by the link?

For this assignment, to keep the simulations short, pick an RTT_{min} of around 10 so that the BDP is around 10 (recall link capacity is 1). This will allow you to vary window sizes from 1 to 1000 and still complete each simulation in 100000 simulation ticks or so. Large window sizes will need a longer simulation time before the simulation settles into steady state. Do not introduce any IID loss (i.e., set `loss_ratio` to 0) for this experiment. You could also try modifying `min_timeout` and `max_timeout` to observe a sharper version of the congestion collapse, which may also help you answer §1.2. Finally, you could also try reducing the queue size limit to observe a sharper version of the congestion collapse.

For this task, you get full marks if you can demonstrate one set of simulation settings that leads to a congestion collapse. Fill out the template `congestion_collapse.py` with these settings. Do **not** modify the maximum queue limit and loss of the simulator. All other configurations are permitted.

To verify a congestion collapse, you should plot the transport-layer throughput vs. window size that resembles the classic congestion collapse curve we discussed in class. Feel free to submit a plot of this with your assignment. This is **not** required, but it will help us judge your submission.

5 AIMD (20 points)

In the final part of this assignment, you will implement the AIMD algorithm that fixes congestion collapse. Use the file `aimd_host.py`, which shares a considerable amount of code with `sliding_window_host.py`. So if you have completed `sliding_window_host.py`, most of your work for AIMD is already done. The only major new parts of AIMD are implementing the Additive Increase and Multiplicative Decrease rules. Answer the following questions. If you need a concrete value of RTT_{min}, you can set it to 10 ticks for this experiment, but feel free to use your own value of RTT_{min} if you wish.

1. First make sure the AIMD algorithm is implemented as per the instructions in the TODOs.
2. Why do we wait for an RTT before we decrease the window a second time using multiplicative decrease? What would happen if we didn't wait?
3. Set the queue size limit to something small, like about half the BDP. Use matplotlib (or a program of your choice) to plot the evolution of the window size over time. You can plot the window size by printing out the window size and the tick number every time the send() function is called. Attach the plot with your submission. Do you see the additive increase, multiplicative decrease, sawtooth pattern that we discussed

in the lecture? What is the period of the sawtooth? You can measure the period from the window size plot that demonstrates the sawtooth pattern. What is the throughput of AIMD in this case?

4. Increase the queue size limit from half the BDP to 1 BDP and then 2 and 3 BDP. What happens to the throughput of AIMD? Why?

5. AIMD needs a certain amount of queue capacity so that it achieves throughput close to the link's capacity. What is the purpose of this queue?

6 Sample output

We have provided some sample output below for you to confirm if your implementations of the Stop-And-Wait, Sliding Window, and AIMD protocols are working correctly. Note that your output may not match up exactly with these outputs because there is some flexibility in how you implement each protocol and there are no simple "unit tests" for congestion-control protocols. If you're unsure if your protocol is working correctly, ask the course staff.

6.1 Stop-And-Wait

```
python3 simulator.py  --seed 1 --host_type StopAndWait --rtt_min 10 --ticks 50

seed: 1
host_type: stopandwait
rtt_min: 10
ticks: 50
loss_ratio: 0.0
queue_limit: 1000000
window_size: None
min_timeout: 100
max_timeout: 10000
sent packet @ 0 with sequence number 0
@ 9 timeout computed to be 100
rx packet @ 9 with sequence number 0
sent packet @ 10 with sequence number 1
@ 19 timeout computed to be 100
rx packet @ 19 with sequence number 1
sent packet @ 20 with sequence number 2
@ 29 timeout computed to be 100
rx packet @ 29 with sequence number 2
sent packet @ 30 with sequence number 3
@ 39 timeout computed to be 100
rx packet @ 39 with sequence number 3
sent packet @ 40 with sequence number 4
@ 49 timeout computed to be 100
rx packet @ 49 with sequence number 4
Maximum in order received sequence number 4
```

6.2 Sliding Window

```
python3 simulator.py  --seed 1 --host_type SlidingWindow --rtt_min 10 --ticks
50 --window_size 5
```

```
seed: 1
host_type: slidingwindow
rtt_min: 10
ticks: 50
loss_ratio: 0.0
queue_limit: 1000000
window_size: 5
min_timeout: 100
max_timeout: 10000
sent packet @ 0 with sequence number 0
sent packet @ 0 with sequence number 1
sent packet @ 0 with sequence number 2
sent packet @ 0 with sequence number 3
sent packet @ 0 with sequence number 4
rx packet @ 9 with sequence number 0
sent packet @ 10 with sequence number 5
rx packet @ 10 with sequence number 1
sent packet @ 11 with sequence number 6
rx packet @ 11 with sequence number 2
sent packet @ 12 with sequence number 7
rx packet @ 12 with sequence number 3
sent packet @ 13 with sequence number 8
rx packet @ 13 with sequence number 4
sent packet @ 14 with sequence number 9
rx packet @ 19 with sequence number 5
sent packet @ 20 with sequence number 10
rx packet @ 20 with sequence number 6
sent packet @ 21 with sequence number 11
rx packet @ 21 with sequence number 7
sent packet @ 22 with sequence number 12
rx packet @ 22 with sequence number 8
sent packet @ 23 with sequence number 13
rx packet @ 23 with sequence number 9
sent packet @ 24 with sequence number 14
rx packet @ 29 with sequence number 10
sent packet @ 30 with sequence number 15
rx packet @ 30 with sequence number 11
sent packet @ 31 with sequence number 16
rx packet @ 31 with sequence number 12
sent packet @ 32 with sequence number 17
rx packet @ 32 with sequence number 13
sent packet @ 33 with sequence number 18
rx packet @ 33 with sequence number 14
sent packet @ 34 with sequence number 19
rx packet @ 39 with sequence number 15
sent packet @ 40 with sequence number 20
rx packet @ 40 with sequence number 16
sent packet @ 41 with sequence number 21
rx packet @ 41 with sequence number 17
sent packet @ 42 with sequence number 22
rx packet @ 42 with sequence number 18
```

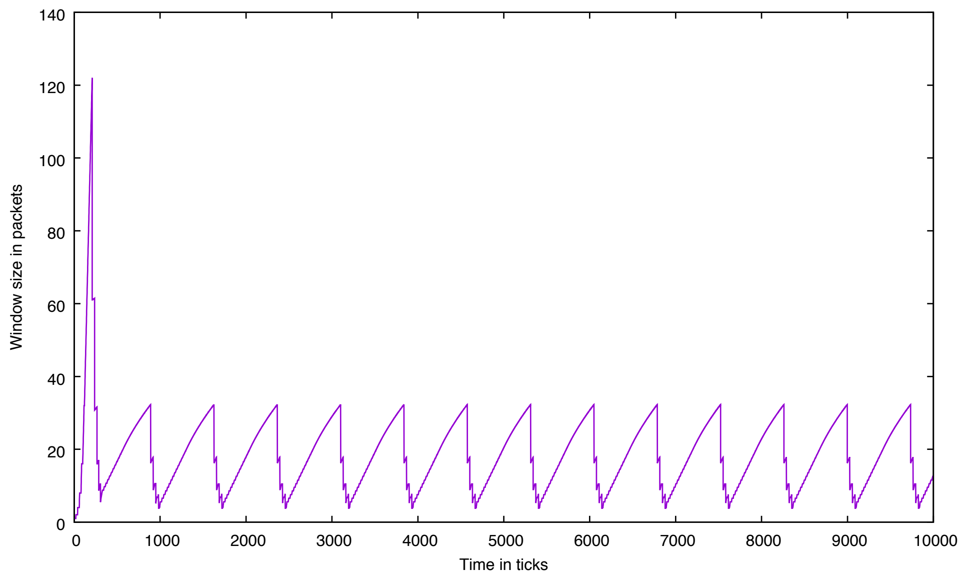

Figure 1: Evolution of window size when running the AIMD protocol. AIMD should set the window size W according to this pattern over time; the window size slowly ramps up (slow start) until reaching a threshold where it is larger than the network can allow, so AIMD quickly backs off to a much smaller baseline W.

```
sent packet @ 43 with sequence number 23
rx packet @ 43 with sequence number 19
sent packet @ 44 with sequence number 24
rx packet @ 49 with sequence number 20
Maximum in order received sequence number 20
```

6.3 AIMD

See Figure 1. The figure was generated using the command line:

```
python3 simulator.py  --loss_ratio 0.0 --seed 1 --host_type Aimd --rtt_min 20
--ticks 10000 --queue_limit 10

seed: 1
host_type: aimd
rtt_min: 20
ticks: 10000
loss_ratio: 0.0
queue_limit: 10
window_size: None
min_timeout: 100
max_timeout: 10000
...
Maximum in order received sequence number
7717
```