Assignment 2

Anirudh Sivaraman

2024/02/16

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.

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.

1 Moving Averages and Retransmission Timeouts

1.1 Overview of the Code

For this first part, we'll start with the `TimeoutCalculator` found in `util/timeout_calculator.py`. This class will be used to determine how long each packet we create should wait before our host retrires sending it. To compute the timeout, we'll estimate the mean RTT and it's standard deviation using the EWMA-based algorithm we discussed in lecture 4.

We set `alpha`, `beta`, and `k` (which have the same function as α, β, and k from the lecture) during the construction of the calculator. Remeber, EWMA is a moving average, it changes over time as additional RTT data is added to the system. In our implementation, this is done in the `add_data_point()` function. When a host receives an acknowledgement, it can update our calculator with the latest RTT data point. This data point is then used to update our estimate of the mean RTT and RTT variance before updating the timestamp recommendation. One the `add_data_point()` function returns, the host can `timeout()` method to get the timeout recommendation based on the latest RTT data.

Notice that, when we update the timeout, we first make sure that the new timeout is within the timeout bounds we'll set during the beginning of the simluation. 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.

1.2 Tasks for the `TimeoutCalculator`

Your first task is to implement the helper functions used to compute the mean estimate, standard deviation estimate, and timeout.

- Open the `TimeoutCalculator` in `util/timeout_calculator.py`.
- First, implement the `__compute_new_mean_estimate()` function to estimate the mean using the previous estimate of the mean, the latest RTT datapoint, and α.
- Next, implement the `__compute_new_stddiv_estimate()` function to estimate the standard deviation using the previous estimate of the standard deviation, the mean, the latest RTT datapoint, and β.
- Finally, use your mean and standard deviation estimates, as well as k, to compute the new timeout recommendation in `__compute_timeout()`

1.3 Timeout Simulation

Once the `TimeoutCalculator` is completed, we can start running the timeout simulation. This simulation is run from `run_timeout_simulation.py`.

We have implemented a few different scenarios to test the `TimeoutCalculator` against. These scenarios produce a stream of mock RTT data that is added to your calculator over time. Each time a new data point is added, the simulator will fetch the updated mean estimate and timeout. Finally, it will plot these values so you can visualize the results.

Currently, we have four scenarios implemented in the `NetworkSimulator` class. Each scenario runs for 100 ticks and sends 1 new data point at each tick.

- `short-spike`: All packets have an RTT of 100, except for ticks 10 and 11, where it spikes up to 200.
- `long-spike`: Same as before, except the spike lasts from tick 10 to 29.
- `permanent-change`: The RTTs start our at 100, but at tick 20, they jump up to 200 where they remain.
- `high-variance`: The RTTs vary widly between 10 and 200.

To run the simulation, you must specify α, β, and the scenario. You can also specify k. For example, if we want to see how $\alpha = 0.125$, $\beta = 0.25$, and $k = 1.5$ fair against a permanent change in latency, we can run the simulation using

```
python3 run_timeout_simulation.py -a 0.125 -b 0.25 -k 1.5 permanent-change
```

If your timeout calculator was implemented correctly, you should see an image in `timeout_simulation.png` that looks like 1.

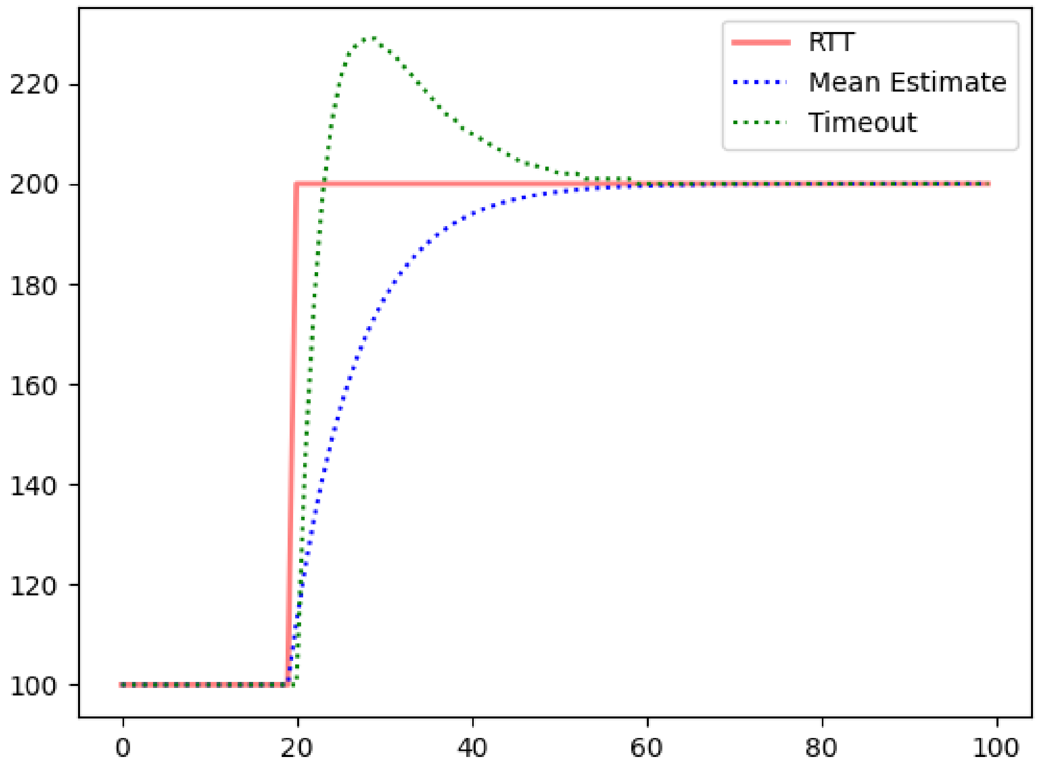

Figure 1: The result of a timeout simulation with $\alpha = 0.125$, $\beta = 0.25$, and $k = 1.5$ with a `permanent-change` scenario.

1.4 Tasks for the the Timeout Simulation

- Answer the conceptual questions on grade scope.

The timeout simulation can be run using

```
python3 run_timeout_simulation.py  -a ALPHA  -b BETA  -k K  SCENARIO
```

2 Reliability

This next part of the assignment will have you implement a few of the reliability protocols we learned about in class. Specifically, we will implement the Stop-And-Wait and Sliding Window protocols to track outbound messages and retries, and we'll implement AIMD to prevent congesion collapse.

These will all run on the network simulator we mentioned at the beginning. We'll reveiw some of the details of that simulator before diving into the tasks.

2.1 Overview of the Simulator

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.

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 the packet being in flight for 11 ticks. For example, if we send a packet from the host during tick 0, we'd expect to see that packet returned by `receive_all()` during tick 10.

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. Therefore, we do not actually implement a separate sender and receiver.

During this simulation, the host we implement will send a stream of data out to the network. In reality, this stream of data is just being looped back to the host (after being processed by the mock devices we'll talk about shortly). That said, when implementing the host, you should do so from the perspective of an engineer implementing a light-weight version of TCP streaming data to a remote server.

Treat all the data sent as messages, and all the packets received as acknowledgements from the remote server.

Devices. There are a few physical devices represented by different classes in the simulator. The most important of these for this assignment are the *hosts*, since those are your job To This is where we'll do the bulk of our work implementing the protocols.

The hardware device on most computers responsible for interacting with the network is the *network card*, or *network interface*. Each host has a network interface attached to it. The network interface contains egress (transmission) and ingress (reception) buffers. While the host is running, it should check the ingress buffers to see if any packets have been sent to this host. It should then figure out what data it wants to transmit, and add those packets to the egress buffer.

Next is the *clock*. To simplify things, our simulation has a global clock that tracks the current tick value. Any hardware device is able to read from this clock at any time to see what tick the simulation is currently on.

[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.

Then the *link*. When data is sent from the host over the network card, that packet's first stop will be the link. The link will add that packet to it's buffer. The link will then transmit the packet on the front of the buffer queue when there's available capacity (i.e., once per tick).

Finally, we have the *delay box*, which simulates the packet as it travels through the wide-area network (WAN). Specifically, this device holds the packet and releases it only after the specified RTT has passed. While the delay box is processing the packet, it also marks it as an ACK packet. This is also where packets are randomly dropped to represent packet loss. Luckily, since this is a simulation, we know with certainty when a packet was dropped and log its sequence number, making debugging easier.

Ticks. It might be helpful to review what exactly happens during each tick of the simulation. During each tick, the simulator first updates the clock. Next, it hands control of the simulation over to the host. During this time, the host should read all packets that are currently on the ingress buffer of the network interface and process the newly received messages. The host will then place any new outbound packets on the egress buffer of the network interface. Once the host finishes processing the messages, the simulator flushes the network interface's egress buffer to the link. It then flushes the link's buffer out to the WAN (the delay box). Finally, it any packets arriving from the WAN are placed onto the network interface's ingress buffer to be processed during the next tick.

2.2 Overview of the Code

Entry Point: `run_reliability_simulation.py`

This file is the script that starts the simulation. It just acts as an entry point. It parses command line arguments to determine how to configure the network. You must specify the RTT_{min} and the duration of the simulation (in number of ticks). Additionally, you can configure the simulation's loss rate, the maximum number of packets in the link's queue, and the timeout bounds. Finally, you can manually specify the seed for the random number generator, if you want to work with deterministic output.

Once you've passed in the arguments to configure the simulation, you can configure the host. Start by specifying which protocol the host will implement. This can be `stop-and-wait`, `sliding-window`, or `aimd`. Finally, on a host that implements the sliding window protocol, you must specify the window size. Example commands can be found in §5.

The script then build the network devices, before handing control over to the simulator itself. It prints the results of the simulation to the terminal. 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.

Network Card: `network/network_interface.py`

Since this simulation isn't interested in the actual data being transmitted (just the packets themselves), the packets only contain metadata. Specifically, each packet contains the sequence number of that message, the send timestamp, and the timeout.

Remeber, all network interactions from the host should happen through the network interface. To send a `packet` from the host, use the `network_interface.transmit(packet)` method.

Similarly, you can use `network_interface.receive_all()` to get a list of all packets that were sent to this host.

Hosts: `host/host.py`

This file contains the interface for the hosts. The interface represents the contract between the hosts and the simulator. Specifically, each host must implement the `run_one_tick()` method, which reads data off the network card, writes new data to the network card, and returns the latest in-order sequence number that's been acknowledged.

All the different host types that implement this interface are also defined in the `host/` directory.

2.3 Tasks for the Stop-and-Wait Protocol

- Fill in the TODOs in `host/stop_and_wait_host.py`. Specifically, complete the `run_one_tick()` function. You'll notice this function is broken into three stages
  1. Read all packets received on the network card. Process the acknowledgements.
  2. Determine which packets haven't been acknowledged, but have timeout. Then, retransmit those messages.
  3. Determine if any new packets can be trasmitted. If they can be, transmit them.

  Note: All hosts will implement the same three stages during their execution of each tick.
- Complete the conceptual questions on gradescope.

Testing the Stop-and-Wait Protocol Implementation

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.
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?

2.4 Tasks for the Sliding Window Protocol

- Finish the TODOs in `host/sliding_window_host.py`.
- Complete the conceptual questions on gradescope.

Testing the Sliding Window Protocol Implementation

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.

3 Congestion collapse

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. 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.

4 AIMD

In the final part of this assignment, you will implement the AIMD algorithm. As previously mentioned, this algorithm uses the sliding window protocol we've already implemented. The only major new parts of AIMD are implementing the Additive Increase and Multiplicative Decrease rules.

4.1 Tasks for the AIMD Protocol

- Finish the TODOs in the `set_window_size()` and `run_one_tick()` functions of `host/aimd_host.py`
- Finish the TODO in the `shudown_hook()` function so the AIMD host will print a graph after the simulation ends.
- Complete the conceptual questions on gradescope.

Testing the AIMD Implementation

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. 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?
2. Set the queue size limit to something small, like about half the BDP. Use matplotlib to plot the evolution of the window size over time. 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?

3. 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?
4. 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?

5 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.

5.1 Stop-And-Wait

```
python3 run_reliability_simulation.py --seed 1 --rtt-min 10 --ticks 50 stop-and-wait

host_type: stop-and-wait
rtt_min: 10
ticks: 50
seed: 1
loss_ratio: 0.0
queue_limit: 1000000
min_timeout: 100
max_timeout: 10000

------------------------------------------
| Tick | Event Type | Event Description  |
|------|------------|--------------------|
|    0 |   Transmit | Sequence number: 0 |
|   10 |    Receive | Sequence number: 0 |
|   10 |   Transmit | Sequence number: 1 |
|   20 |    Receive | Sequence number: 1 |
|   20 |   Transmit | Sequence number: 2 |
|   30 |    Receive | Sequence number: 2 |
|   30 |   Transmit | Sequence number: 3 |
|   40 |    Receive | Sequence number: 3 |
|   40 |   Transmit | Sequence number: 4 |
------------------------------------------

Maximum in order received sequence number 3
```

5.2 Sliding Window

```
python3 run_reliability_simulation.py --seed 1 --rtt-min 2 --ticks 10 sliding-window
--window-size 5

host_type: sliding-window
rtt_min: 2
ticks: 10
seed: 1
loss_ratio: 0.0
queue_limit: 1000000
min_timeout: 100
max_timeout: 10000
window_size: 5

-------------------------------------------
| Tick | Event Type | Event Description   |
|------|------------|---------------------|
|    0 |   Transmit | Sequence number: 0  |
|    0 |   Transmit | Sequence number: 1  |
|    0 |   Transmit | Sequence number: 2  |
|    0 |   Transmit | Sequence number: 3  |
|    0 |   Transmit | Sequence number: 4  |
|    2 |    Receive | Sequence number: 0  |
|    2 |   Transmit | Sequence number: 5  |
|    3 |    Receive | Sequence number: 1  |
|    3 |   Transmit | Sequence number: 6  |
|    4 |    Receive | Sequence number: 2  |
|    4 |   Transmit | Sequence number: 7  |
|    5 |    Receive | Sequence number: 3  |
|    5 |   Transmit | Sequence number: 8  |
|    6 |    Receive | Sequence number: 4  |
|    6 |   Transmit | Sequence number: 9  |
|    7 |    Receive | Sequence number: 5  |
|    7 |   Transmit | Sequence number: 10 |
|    8 |    Receive | Sequence number: 6  |
|    8 |   Transmit | Sequence number: 11 |
|    9 |    Receive | Sequence number: 7  |
|    9 |   Transmit | Sequence number: 12 |
-------------------------------------------

Maximum in order received sequence number 7
```

5.3 AIMD

See Figure 2. The figure was generated using the command line:

```
python3 run_reliability_simulation.py --loss-ratio 0.0 --seed 1 --rtt-min 20 --ticks 10000
--queue-limit 10 aimd

host_type: aimd
rtt_min: 20
ticks: 10000
seed: 1
loss_ratio: 0.0
queue_limit: 10
min_timeout: 100
max_timeout: 10000

--------------------------------------------------------------------------------------
| Tick | Event Type               | Event Description                                |
|------|--------------------------|--------------------------------------------------|
|    0 |                 Transmit | Sequence number: 0                               |
|   20 |                  Receive | Sequence number: 0                               |
|   20 |         Expanding Window | Old: 1.0, New: 2.0                               |
|   20 |                 Transmit | Sequence number: 1                               |
|   20 |                 Transmit | Sequence number: 2                               |
|   40 |                  Receive | Sequence number: 1                               |
|   40 |         Expanding Window | Old: 2.0, New: 3.0                               |
|   40 |                 Transmit | Sequence number: 3                               |
|   40 |                 Transmit | Sequence number: 4                               |
|   41 |                  Receive | Sequence number: 2                               |
|   41 |         Expanding Window | Old: 3.0, New: 4.0                               |
|   41 |                 Transmit | Sequence number: 5                               |
|   41 |                 Transmit | Sequence number: 6                               |
|   60 |                  Receive | Sequence number: 3                               |
|   60 |         Expanding Window | Old: 4.0, New: 5.0                               |
|   60 |                 Transmit | Sequence number: 7                               |
|   60 |                 Transmit | Sequence number: 8                               |
|   61 |                  Receive | Sequence number: 4                               |
...
| 9997 |                 Transmit | Sequence number: 7516                            |
| 9998 |                  Receive | Sequence number: 7505                            |
| 9998 |         Expanding Window | Old: 12.611559591457588, New: 12.690851925823383 |
| 9998 |                 Transmit | Sequence number: 7517                            |
| 9999 |                  Receive | Sequence number: 7506                            |
| 9999 |         Expanding Window | Old: 12.690851925823383, New: 12.76964884236176  |
| 9999 |                 Transmit | Sequence number: 7518                            |
--------------------------------------------------------------------------------------

Maximum in order received sequence number 7506
```

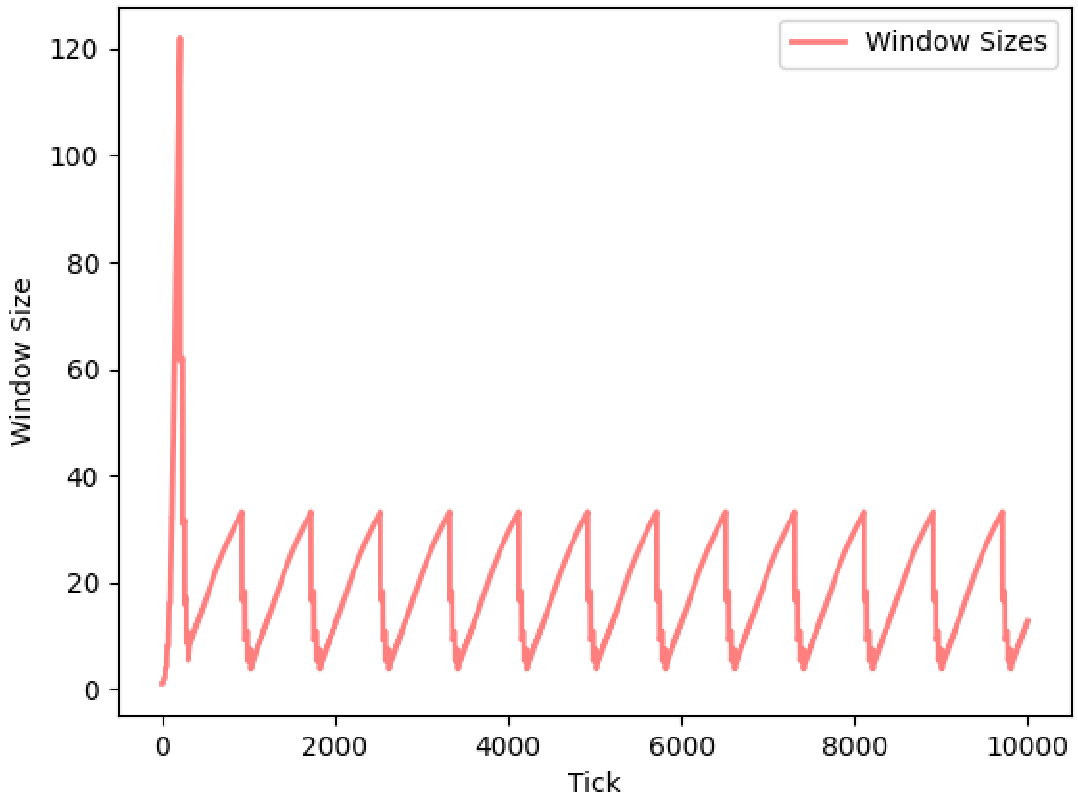

Figure 2: Evolution of window size when running the AIMD protocol. AIMD should set the window size W according to this pattern over time; the window size slowly ramps up (slow start) until reaching a threshold where it is larger than the network can allow, so AIMD quickly backs off to a much smaller baseline W.