

Assignment #3: Congestion Control

Due on June 9, 11:59 pm

1 Introduction

In this assignment, you will extend your code from the previous assignment to implement a congestion control algorithm (CCA). With this, your transport layer implementation will be complete and can be used by applications! “Real” implementations are more efficient and use a more efficient serialization mechanism than our JSON-based header format, but your implementation will have all the important features.

Recall that in the previous assignment, we picked a constant receive window size and timeout value. In this assignment, you will implement two additional functions, `get_cwnd` and `get_rto`, that return the congestion window (cwnd) and retransmission timeout (rto) that the sender should use. The units are in bytes and seconds, respectively. You will have to modify your implementations of the other functions to enable the computation of these quantities. Note that the congestion control algorithm we discussed in the class modifies only the sender and not the receiver.

Recall that the RTO is set as $\text{EstimatedRTT} + 4 * \text{DevRTT}$. Both are computed as Exponentially Weighted Moving Averages (EWMAs) of the respective quantities computed for each packet: `SampleRTT` and $|\text{SampleRTT} - \text{RTTAVG}|^1$. You will have to calculate the RTT for each packet as the difference in the time at which it was sent and when it was received. Having a unique `packet_id` is particularly useful here, since you know exactly which packet is being acknowledged. Further, at first you will not have any good RTT estimates. During this period, return a large conservative value like 1 second.

EWMA. For any sequence of numbers x_1, x_2, \dots , the EWMA at time t is computed recursively as $e_t = \alpha x_t + (1 - \alpha)e_{t-1}$, where $\alpha \in (0, 1)$ is a constant (say, we pick $\alpha = 1/64$). This has two benefits. First, it gives less weight to older data points. Expanding the formula yields $e_t = \alpha x_t + \alpha(1 - \alpha)x_{t-1} + \alpha(1 - \alpha)^2 x_{t-2} + \dots$. Thus, the weight of a data point Δt time steps ago decays exponentially with Δt . Second, it is extremely simple to compute without maintaining any additional state. In contrast, a simple moving average requires $O(w)$ memory, where w is the window size.

The congestion window should be computed using the AIMD algorithm discussed in class.

2 The MahiMahi Emulator

Congestion control only makes sense when the link has finite bandwidth. In the previous assignment, we ran the sender and receiver on the same machine while testing. Within a single machine, we get extremely high link rates. To properly test CCAs, we need a way to slow the packets down. We need to slow them down along two dimensions: throughput and delay. Throughput controls the maximum number of bytes the link will let pass every second. Delay controls how long the link takes to transmit any packet from one side to another.

We will use the MahiMahi network emulator for this purpose. You could have optionally used it in the previous assignment. This is the only assignment in this course that *requires* Linux since Mahimahi requires Linux’s ability to create isolated network namespaces. You should follow the steps outlined in <http://mahimahi.mit.edu>. For example, you may use the following steps on Ubuntu to install from source.

Install the pre-requisites:

```
sudo apt install autotools-dev protobuf-compiler \
libprotobuf-dev dh-autoreconf iptables pkg-config \
dnsmasq-base apache2-bin apache2-dev debhelper libssl-dev \
ssl-cert libxcb-present-dev libcairo2-dev libpango1.0-dev
```

¹We use the absolute difference instead of variance because it is less computationally expensive to compute. Whether this optimization matters today is debatable.

Download the code:

```
git clone https://github.com/ravinet/mahimahi
```

Next, we need to deal with the fact that C++ is a wonderfully stable language that is always backwards compatible and its compilers never change their rules (note that the preceding sentence was sarcasm). The problem is that the creator of mahimahi added a flag to convert all warnings to errors² and GCC added some extra warnings that were not there before. Modify line 15 of `configure.ac` to remove the C++ compiler flags `-Wall -Wextra` from it. These were the flags forcing the compiler to treat all warnings as errors. Removing it should make the build go smoothly. The final line should be:

```
PICKY CXXFLAGS="-pedantic -Wall -Werror"
```

Finally, run the following commands:

```
./autogen.sh ./configure make sudo make install
```

This should create binaries called `mm-delay` and `mm-link` among others in your path. For more information, you can read the linked website and their “man pages” using `man mm-delay`. To emulate a 12 Mbit/s link, you need to create a trace file containing just a single line with the number “1”. For instance, you can create it using `echo 1 > 12mbps`. Now, use the following command:

```
mm-delay 10 mm-link --meter-uplink --meter-uplink-delay \
--downlink-queue=infinite --uplink-queue=droptail \
--uplink-queue-args=bytes=30000 12mbps 12mbps
```

This will create a new shell (well, it creates two new shells; first with `mm-delay` and then `mm-link`). Any programs executed after running the above command will be “inside” the shell. Any packets sent by such programs will have to pass through a network emulator before they go out to programs that started “outside” the shell. Packets going from inside to outside are said to go on the “uplink”. The other direction is called “downlink”. You should start the sender inside and the receiver outside.

Note that you can run mahimahi shells inside other mahimahi shells. For instance, if you run the above command twice, you will get a total of 40ms of RTT. Make sure that you are inside only one pair of shells (one `mm-delay` and one `mm-link` shell). When you run “exit” inside a mahimahi shell, you will exit to the outer shell you were originally in before running the command.

The parameters in the command above make Mahimahi emulate a 12 Mbit/s link with a 20 ms RTT (10 ms each way) and 30000 bytes (1 BDP) of buffer in the uplink direction and an infinitely large downlink buffer. You can make the downlink buffer finite too, but will not matter since ACK packets are much smaller than data packets anyway. It will also display two graphs, one to show the number of bytes going from inside to outside (in bytes/s) and another to show the amount of time the packets waited in the queue.

You should also test your algorithm on other configurations. You can create files with different link rates. To create a 6 Mbit/s link, create a file with a single line containing “2”. To create a 24 Mbit/s link, create a file with two lines each containing “1”. You can find other, more interesting, traces with time-varying link rates here: <https://github.com/ravinet/mahimahi/tree/master/traces>. While you can run your algorithm over those traces, and it will likely not perform terribly, we won’t concern ourselves with those in this assignment.

3 Deliverables

Submission Please upload the following files to the CU:

- `transport.py`: This file should contain your implementation of the AIMD algorithm.
- `report.pdf`: A report that includes items from the grading rubric.

Grading rubric The grading is based on the correctness of your implementation and the quality of your report. The points are distributed as follows:

- **(5 points) Name & ID:** Your name and student ID are included in `transport.py`.
- **(25 points) What happens when cwnd is too large?** As a first step, instead of implementing a CCA, just return a constant every time `get_cwnd` is called. For each constant you return, calculate the *goodput* when you run it over the emulated link described above. Goodput is defined as the number of *unique* bytes transmitted. It is the effective throughput you get when you do not count re-transmissions. Plot this value for cwnd values ranging from 1 packet to 10 BDP (product of link rate and base delay). Include this plot in the PDF and explain your observations.
- **(45 points) AIMD.** Run it on various choices of parameters and explain how it works. You can use the live graphs that Mahimahi creates to get an intuition for what is happening. Each time, make sure you transmit a large enough file that it takes 100 seconds to finish. In particular, run it on an emulated link with an infinitely large buffer.
- **(25 points) Fairness.** Although we discussed in class, AIMD is also fair (well... sort of). If you run two independent pairs of senders/receivers over the same emulated link and plot the cwnd that each picks, on average they will pick the same value. You can try running this experiment and understand why it is happening. Surprisingly, even though AIMD is an ancient algorithm and is far from being perfectly fair. Finding algorithms that get high throughput, maintain small queues and are fair to each other over all the possible internet paths remains an open problem in spite of decades of intense research. Run some experiments and include the plots.

4 Things to be careful about

Here are a few implementation-level details you will want to be careful about:

- Make sure the congestion window you return is never smaller than `packet_size`, since otherwise the sender will not be able to transmit any packets.
- Make sure `rto` is never smaller than your machine's ability to measure time, say 1 ms (or 0.001 seconds).
- It may be useful to liberally add print statements to understand how your algorithm is behaving.
- If your implementation is inefficient, it may slow down when transmitting a large file. For instance, if you are not careful, you might accidentally write an algorithm that has complexity that is quadratic in the number of packets sent.
- Do not use a VPN or a similar software while running the Mahimahi emulator. Both of them create software interfaces (which you can see using `ifconfig`) and sometimes conflict with each other.