Instituto Tecnológico y de Estudios Superiores de Monterrey

Santa Fe

Probabilistic TCP/IP Congestion Control Algorithm

Final Project

Quantitative Methods and Simulation

Salvador E. Venegas-Andraca

Daniela Vignau León	A01021698
Héctor Alexis Reyes Manrique	A01339607
René García Avilés	A01654359
Roberto Gervacio Guendulay	A01025780

2022-06-13

Table of Contents

1. Introduction

1.1. Congestion Control Algorithms on the Internet

The idea of congestion control arises from the need to manage the rate at which data is sent by any source on the Internet at a given time. Today, most network devices communicate by means of the Transmission Control Protocol (TCP). Among other tasks, TCP controls network capacity, and in doing so, its underlying implementation will invariably affect network performance. Because of this, it is critical to leverage an efficient TCP congestion control strategy that accommodates different network conditions. Just as vehicle traffic cannot be predicted, segment traffic cannot be predicted in computer networks.

Network congestion refers to a reduction in quality of service (QoS), causing what is commonly experienced as slow connection speeds. On a technical level, this refers to packet loss, delays, or blocking of new connections, all of which can occur when a node or link receives data traffic that exceeds its capacity, essentially falling victim to a DoS (Denial of Service) over the node.

1.1.1. Network Congestion Concepts and Terminology

1.1.1.1. Bandwidth

Temporary situations, such as excessive traffic, or intentional attacks, misconfiguration, and other factors can cause network traffic overload. However, bandwidth use is one of the primary causes of network congestion. The maximum rate at which data may move along a path is referred to as bandwidth. Congestion, like traffic on a highway, happens when there is more traffic than expected or than the bandwidth is built to manage..

1.1.1.2. Latency

Latency is the time a packet needs to travel from point A to point B. Latency is directly related to bandwidth—a slower latency rate can be a sign that congestion is taking place.

1.1.1.3. Jitter

Jitter means “delay variability”. It occurs when conditions are uncertain, such that a system cannot calculate the needed latency for a certain packet. When a system detects possible slow latency, it tries to adjust by holding packets before sending them; but at the same time this causes other systems to do the same, causing a cascade effect on a network.

1.1.1.4. Packet retransmission

Packets that are lost or arrive damaged must be resent, resulting in increased “traffic” and congestion. When the number of packets sent to the network is greater than its capacity, a blockage is created in the network: congestion occurs.

1.1.1.5. Collisions

Collisions can also cause a back-off process. They can be due to poor cabling or bad equipment, and might force all packets to be held before they are retransmitted.

1.1.1.6 Throughput

The throughput of a system refers to how many units of data it can handle in a given length of time. Throughput, bandwidth, and latency are all terms that are occasionally used interchangeably. In a nutshell, network bandwidth relates to the network's capacity for moving data at a single moment; throughput indicates the amount of information; and latency refers to the speed at which information is transmitted, as explained below. The combination of network throughput and latency reflects a network's performance.

1.1.1.7 Round-Trip Time (RTT)

“It is the length time it takes for a data packet to be sent to a destination plus the time it takes for an acknowledgment of that packet to be received back at the origin. The RTT between a network and server can be determined by using the ping command” (Mozilla).

Sometimes the propagation delays between the two communication endpoints are included in this time delay. RTT and network delay are connected, but they are not the same. The time it takes for a data packet to travel from the sending endpoint to the receiving endpoint is known as latency (only one trip). This course could be influenced by a number of variables. Because the latency between any two given endpoints may be asymmetric, the latency is not always equal to half of the RTT. The processing delay at the echo endpoint is included in the RTT.

1.2. Need to Develop Algorithms to Control Congestion

Congestion control is really important for one simple reason: WiFi speed. Millions of operations are carried out daily on the Internet; these can range from contracting services, purchasing products, transferring money, paying hospital bills, etc. Every day, the amount of Internet users and intelligent systems connected to the Internet also increases. Many of these operations are of vital importance, and the speed with which they are carried out is crucial.

For example, investors rely heavily on near real-time trading to keep Wall Street running smoothly. If network speed rates were to fail as a result of congestion, the global economy would take a huge hit. Another example that highlights the importance of congestion control (taking inspiration from the Stuxnet Virus) would be urban infrastructure. In the event of persistent congestion, many services would be lost, generating an unexpected situation in a big city with probably lots of fatalities.

Congestion control can help avoid these situations by mitigating network failures. Contemporary society depends heavily on the Internet and its benefits, so its proper functioning must be ensured.

2. Congestion Control Algorithm: TCP Reno

2.1. TCP Reno and Congestion Control

2.1.1. Original TCP Drawbacks

Because TCP is a connection-oriented protocol, it already contains mechanisms to ensure reliability and retransmission of packets when necessary. It is safe to assume that on every TCP connection, there will be losses due to packet overflows at the router, causing packets to be dropped, or simply because there was a network error. And because TCP should be completely reliable, it resends packets, worsening the congestion in the network that caused the packet to drop in the first place.

2.1.1. How TCP Reno Works

Various algorithms have been developed in an attempt to overcome network congestion in different ways, one of which is TCP Reno. To understand TCP Reno, one must first understand the congestion control strategy from which it draws the most: TCP Tahoe (Nishitha, 2002).

TCP Tahoe consists of three phases:

a) *Slow Start*. Each time an acknowledgement (ACK) is received, the congestion window (*cwnd*) increases by one. Once the *cwnd* reaches a certain threshold (called the “slow start threshold,” or *ssthresh*), *Slow Start* ends, with TCP Tahoe entering *AMID* (Nishitha, 2002).

b) *Additive Increase and Multiplicative Decrease (AMID)*. In this phase, the *cwnd* is increased by one, and the *ssthresh* is reduced to 50% of the *cwnd* (Nishitha, 2002).

c) *Fast Retransmit*. Whenever three duplicate ACKs are received, TCP Tahoe assumes that there is a packet loss, therefore working as a loss detection algorithm. In this phase, the *ssthresh* is reduced to 50% of the *cwnd* and the *cwnd* is reset to 1 (Nishitha, 2002).

Under TCP Tahoe, the *cwnd* is reset to 1 during *Fast Retransmit* every time a packet is lost, making it a very slow algorithm in terms of *cwnd* growth rate. This is where the more efficient TCP Reno comes into play. The main differences between both strategies is that *a)* TCP Reno skips *Slow Start* when a fast retransmit occurs during *Fast Retransmit*, and *b)* it introduces a new algorithm called *Fast Recovery*, where instead of resetting the *cwnd* to 1, it simply reduces it to half and sets the *ssthresh* to the new *cwnd* (Nishitha, 2002).

For comparison, below is a graph detailing the *cwnd* growth rate in TCP Tahoe vs. TCP Reno (*Fig. 1*). Under TCP Reno, instead of the *cwnd* starting over when a loss is detected, it simply reduces its size to 50%, resulting in a *cwnd* that is much larger than 1.

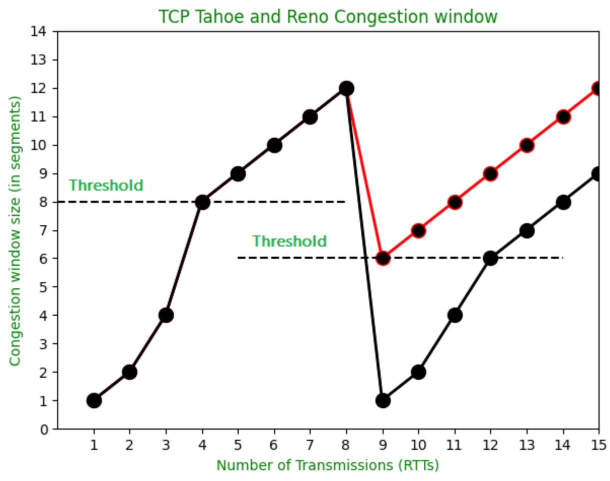


Fig. 1. TCP Tahoe vs. TCP Reno[1]

2.1.2. Limitations of TCP Reno

Multiple packet losses in the same congestion window require a long time to discover. Also when there are numerous packet losses in the same window, it decreases the congestion window many times where one reduction was adequate.

When packet losses are low, Reno performs admirably over TCP. However, when several packet losses occur in a single window, Reno underperforms, and its performance is nearly identical to Tahoe under these situations. Because it can only identify single packet losses. If there are numerous packet drops, the first indication of packet loss occurs in the form of double ACKs.

TCP Reno has been slightly modified by New Reno. It is substantially more efficient than Reno when multiple packet losses occur. When it gets several duplicate packets, New Reno, like Reno, enters fast-retransmit mode. However, unlike Reno, it does not depart fast-recovery mode until all data that was outstanding at the time it entered fast-recovery mode is acknowledged. As a result, it solves Reno's challenge of decreasing the *cwnd* many times.

In New Reno, the difference in the fast recovery phase allows for many retransmissions. When New Reno enters fast recovery, it makes a note of the outstanding maximums segment. As in Reno, the fast-recovery phase continues.

[1] From *TCP Tahoe and Reno*, by GeeksForGeeks, n.d. (https://www.geeksforgeeks.org/tcp-tahoe-and-tcp-reno/)

3. TCP Reno Implementation and Simulation

3.1. Methodology

The development team decided to implement and simulate congestion control with TCP Reno using a fork of an existing project as scaffolding. The project, available on GitHub, leverages *mahimahi*, a network toolkit from MIT that provides command-line utilities to simulate different network conditions on the same computer.

On top of *mahimahi*, the original project features a thread-based TCP-like connection pipeline between ports using Python, as well as two example sender strategies: one in which there is no congestion control (i.e., the *cwnd* remains fixed) and another based on TCP Tahoe.

By treating the basic TCP infrastructure as a black box, the development team was able to focus exclusively on the implementation and simulation of TCP Reno.

3.2. User Guide

3.2.1. Project Setup

Our fork containing an original implementation of TCP Reno can be found here. To install the project, first clone the repository.

```
git clone https://github.com/hreyesm/demonstrating-congestion-control.git
```

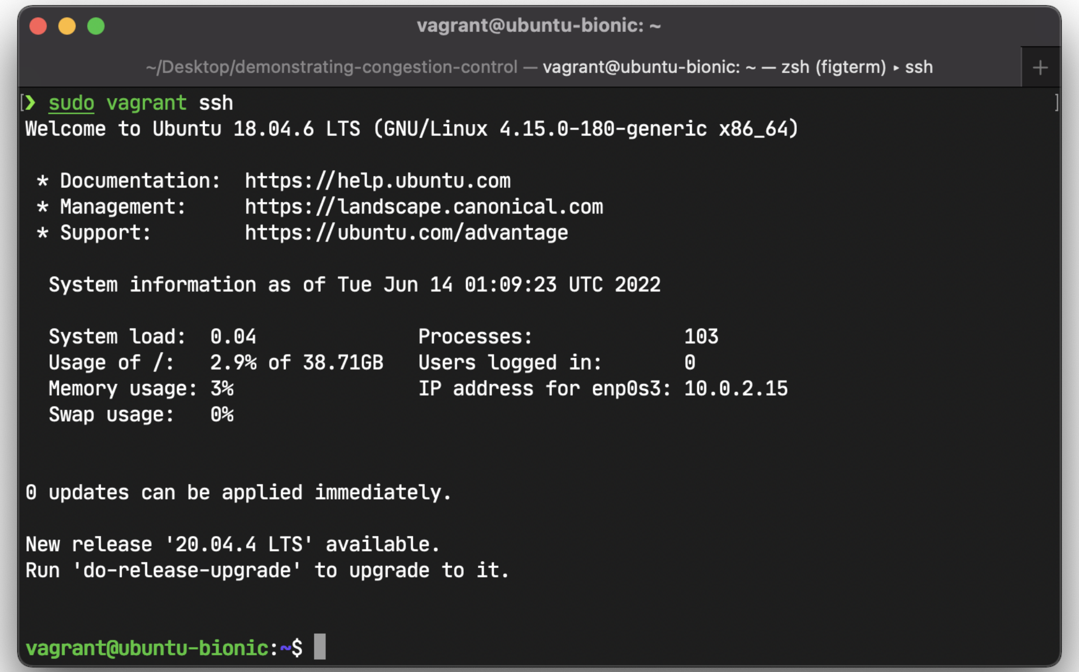
Since the project setup requires a few dependencies that are only available on Linux, a Vagrantfile was provided in the original repository through which an Ubuntu virtual machine can be created. For that, Vagrant must be installed on the host machine.

With Vagrant installed, move to the project directory, then start and provision the Vagrant environment. This will create a new Ubuntu virtual machine.

```
cd demonstrating-congestion-control
sudo vagrant up
```

After the virtual machine has been successfully created, login to the environment via SSH.

```
sudo vagrant ssh
```

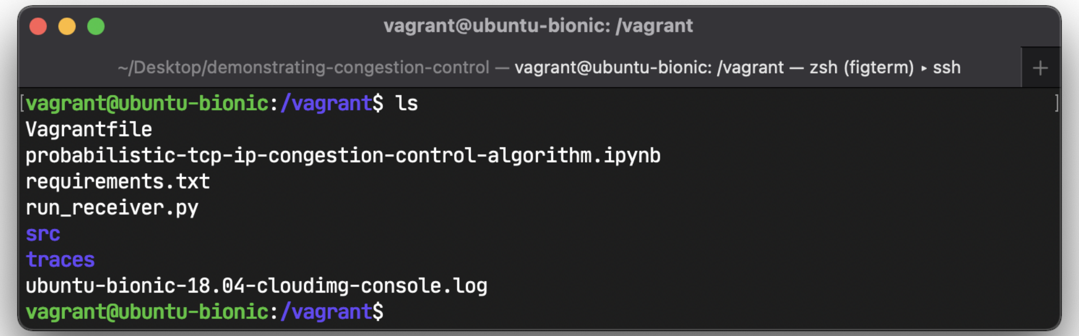

vagrant@ubuntu-bionic: ~
~/Desktop/demonstrating-congestion-control — vagrant@ubuntu-bionic: ~ — zsh (figterm) ▸ ssh

```
❯ sudo vagrant ssh
Welcome to Ubuntu 18.04.6 LTS (GNU/Linux 4.15.0-180-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Tue Jun 14 01:09:23 UTC 2022

  System load:  0.04              Processes:             103
  Usage of /:   2.9% of 38.71GB   Users logged in:       0
  Memory usage: 3%                IP address for enp0s3: 10.0.2.15
  Swap usage:   0%


0 updates can be applied immediately.

New release '20.04.4 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


vagrant@ubuntu-bionic:~$
```

Fig. 2. Ubuntu virtual machine login

Once logged in, move to the `/vagrant` directory which acts as a link between the host and the virtual machine. All repository files will be there.

```
cd /vagrant
```


vagrant@ubuntu-bionic: /vagrant
~/Desktop/demonstrating-congestion-control — vagrant@ubuntu-bionic: /vagrant — zsh (figterm) ▸ ssh

```
vagrant@ubuntu-bionic:/vagrant$ ls
Vagrantfile
probabilistic-tcp-ip-congestion-control-algorithm.ipynb
requirements.txt
run_receiver.py
src
traces
ubuntu-bionic-18.04-cloudimg-console.log
vagrant@ubuntu-bionic:/vagrant$
```

Fig. 3. Link between host and virtual machine

Run the following commands to install Python, *pip*, *jupyter-notebook*, and *mahimahi*, as well as the required Python dependencies to run the project:

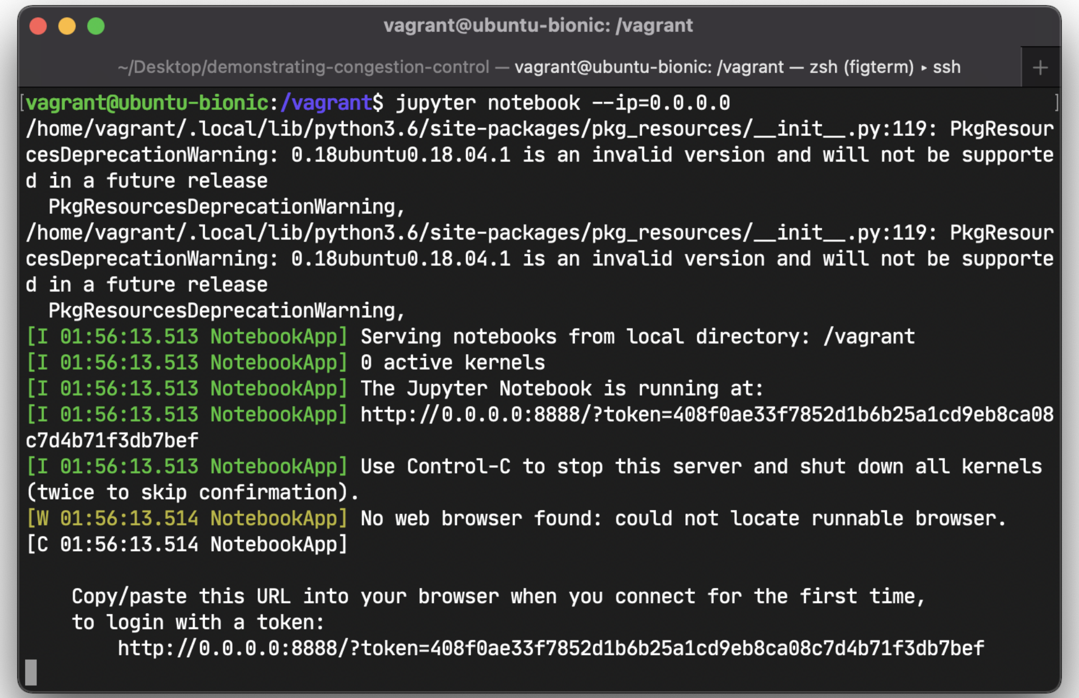
```
sudo apt-get update
sudo apt-get install mahimahi python-pip -y
sudo apt-get install python3-pip
sudo apt-get install jupyter-notebook
sudo apt-get install libjpeg-dev zlib1g-dev
pip3 install -r requirements.txt
```

Now, enable IP forwarding on the virtual machine. This command must be executed every time the computer is restarted.

```
sudo sysctl -w net.ipv4.ip_forward=1
```

Finally, open the provided Jupyter Notebook in the host browser by adding the `--ip=0.0.0.0` flag.

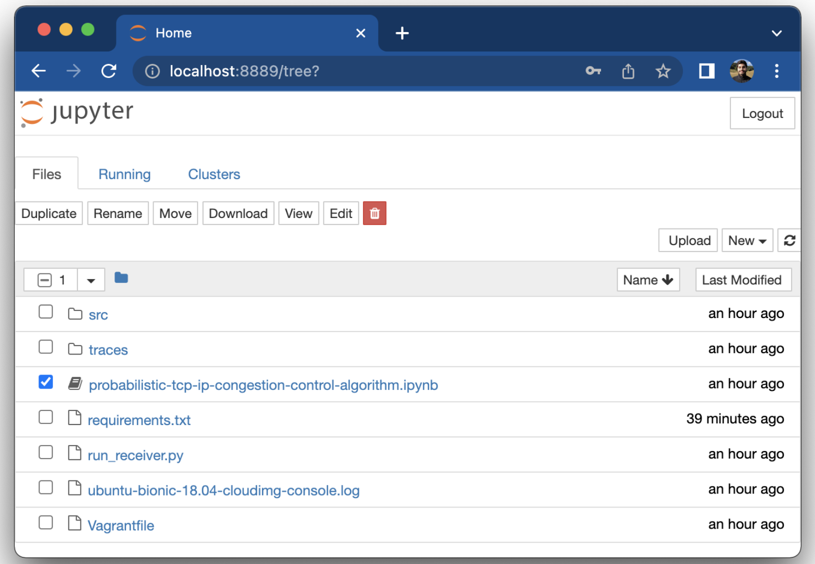
```
jupyter notebook --ip=0.0.0.0
```


vagrant@ubuntu-bionic: /vagrant
~/Desktop/demonstrating-congestion-control — vagrant@ubuntu-bionic: /vagrant — zsh (figterm) ▸ ssh

```
vagrant@ubuntu-bionic:/vagrant$ jupyter notebook --ip=0.0.0.0
/home/vagrant/.local/lib/python3.6/site-packages/pkg_resources/__init__.py:119: PkgResour
cesDeprecationWarning: 0.18ubuntu0.18.04.1 is an invalid version and will not be supporte
d in a future release
  PkgResourcesDeprecationWarning,
/home/vagrant/.local/lib/python3.6/site-packages/pkg_resources/__init__.py:119: PkgResour
cesDeprecationWarning: 0.18ubuntu0.18.04.1 is an invalid version and will not be supporte
d in a future release
  PkgResourcesDeprecationWarning,
[I 01:56:13.513 NotebookApp] Serving notebooks from local directory: /vagrant
[I 01:56:13.513 NotebookApp] 0 active kernels
[I 01:56:13.513 NotebookApp] The Jupyter Notebook is running at:
[I 01:56:13.513 NotebookApp] http://0.0.0.0:8888/?token=408f0ae33f7852d1b6b25a1cd9eb8ca08
c7d4b71f3db7bef
[I 01:56:13.513 NotebookApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
[W 01:56:13.514 NotebookApp] No web browser found: could not locate runnable browser.
[C 01:56:13.514 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://0.0.0.0:8888/?token=408f0ae33f7852d1b6b25a1cd9eb8ca08c7d4b71f3db7bef
```

Fig. 4. Running the Jupyter Notebook

To access the Jupyter Notebook, enter the URL http://localhost:8889/ in the browser. When presented with a "Password or token" input, copy/paste the token displayed in the terminal window and then click "Log in". The Jupyter Notebook can be found in the *probabilistic-tcp-ip-congestion-control-algorithm.ipynb* file.

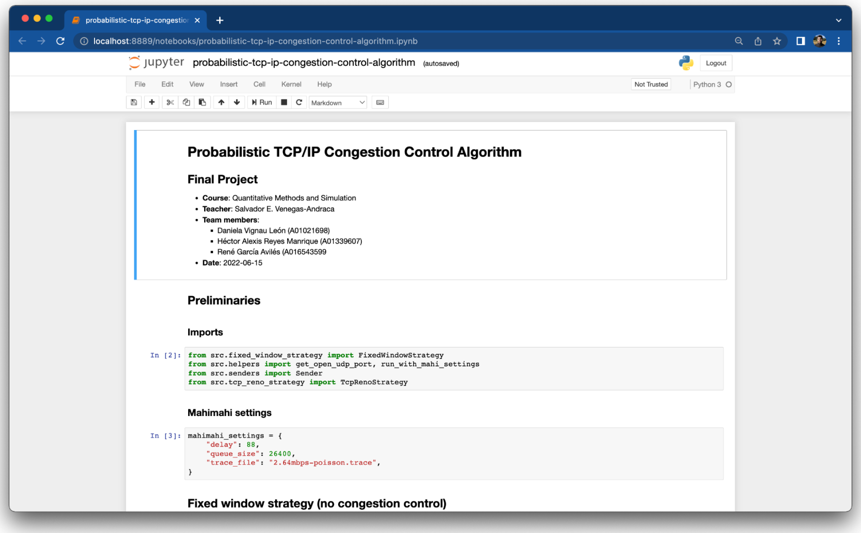


src	an hour ago
traces	an hour ago
probabilistic-tcp-ip-congestion-control-algorithm.ipynb	an hour ago
requirements.txt	39 minutes ago
run_receiver.py	an hour ago
ubuntu-bionic-18.04-cloudimg-console.log	an hour ago
Vagrantfile	an hour ago

Fig. 5. Jupyter file tree

3.2.2. Usage

Once the Jupyter Notebook is running and can be accessed through the browser, expect to see the following (*Fig. 6*):


probabilistic-tcp-ip-congestio
localhost:8889/notebooks/probabilistic-tcp-ip-congestion-control-algorithm.ipynb
jupyter probabilistic-tcp-ip-congestion-control-algorithm (autosaved)
Logout
File Edit View Insert Cell Kernel Help
Not Trusted
Python 3
Run
Markdown

Probabilistic TCP/IP Congestion Control Algorithm

Final Project

- **Course**: Quantitative Methods and Simulation
- **Teacher**: Salvador E. Venegas-Andraca
- **Team members**:
  - Daniela Vignau León (A01021698)
  - Héctor Alexis Reyes Manrique (A01339607)
  - René García Avilés (A016543599
- **Date**: 2022-06-15

Preliminaries

Imports

```
In [2]: from src.fixed_window_strategy import FixedWindowStrategy
        from src.helpers import get_open_udp_port, run_with_mahi_settings
        from src.senders import Sender
        from src.tcp_reno_strategy import TcpRenoStrategy
```

Mahimahi settings

```
In [3]: mahimahi_settings = {
            "delay": 88,
            "queue_size": 26400,
            "trace_file": "2.64mbps-poisson.trace",
        }
```

Fixed window strategy (no congestion control)

Fig. 6. Jupyter Notebook visualized on a browser

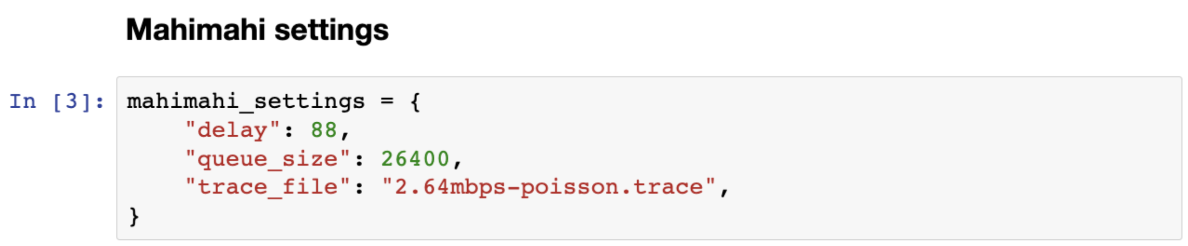
There are several cells in the Jupyter Notebook that contain the necessary code to execute the TCP Reno strategy. The first one contains the imports of the project. The second one, which is titled “Mahimahi settings”, groups default values that are used when *mahimahi* starts the network and are passed on as a dictionary to the `run_with_mahi_settings()` function below (*Fig. 7*).

The values that *mahimahi* receives in order to simulate the network are the following:

a) *Delay value:* Indicates the specific *delay* (in milliseconds) that every packet undergoes when entering and leaving the container.

b) *Queue size:* Internally, the overall *mahimahi* command that is being executed, contains the command `mm-link` which is in charge of emulating the links using packet delivery trace files (more on this later). The `mm-link` can receive an optional flag that limits the queue size of the downlink, meaning all the packets that are coming from the Internet (or the servers, in this case) to the receiver.

c) *Trace File:* The name of the file located under the `/traces` directory that contains a list of timestamps where a packet of 1500 bytes can be sent. It is used to simulate the bandwidth of the network. There are many more trace files that can be found in this repository, as well as the one included in the project repository (*2.64mpbs-poisson.trace*). As the name suggests, the simulated network will have a bandwidth of 2.64 megabits per second—low enough to simulate congestion.



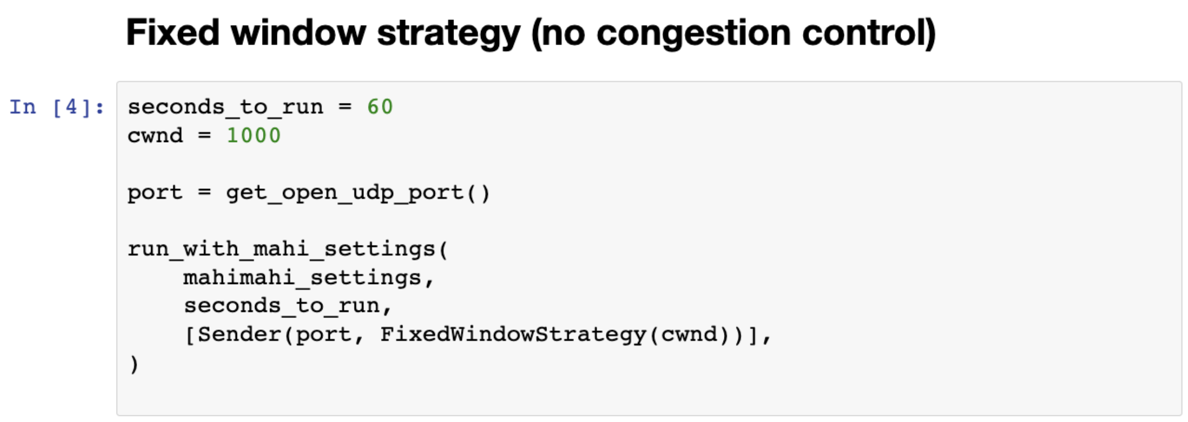
Mahimahi settings

```
In [3]: mahimahi_settings = {
            "delay": 88,
            "queue_size": 26400,
            "trace_file": "2.64mbps-poisson.trace",
        }
```

Fig. 7. Mahimahi settings

For the fixed window strategy execution (i.e., the one in which no congestion control is applied), the next cell in the Jupyter notebook contains the specific arguments needed to run the strategy. The **`seconds_to_run`** variable is quite simply an integer that indicates the servers how long they should continue to run. The **`cwnd`** variable is the whole size of the congestion window; meaning the number of packets that can be sent before receiving an ACK. Finally, the function **`run_with_mahi_settings()`** is called, and it receives the previously mentioned arguments in the following manner:

- The **`mahimahi_settings`** dictionary that was created in the previous cell
- The seconds to run the program
- A list of Sender instances, which in turn receives the port and the strategy to execute along with its necessary arguments. In this case, as it is the Fixed window, it only receives the **`cwnd`** variable.



Fixed window strategy (no congestion control)

```
In [4]: seconds_to_run = 60
        cwnd = 1000

        port = get_open_udp_port()

        run_with_mahi_settings(
            mahimahi_settings,
            seconds_to_run,
            [Sender(port, FixedWindowStrategy(cwnd))],
        )
```

Fig. 8. Fixed window strategy API

```
[Sender 100.64.0.2:46220] Connected to receiver 100.64.0.2:58356

Results for sender 46220:
Total Acks: 43139
Num Duplicate Acks: 41996
% duplicate acks: 97.350425
Throughput (bytes/s): 381.000000
Average RTT (ms): 2292.233256
```

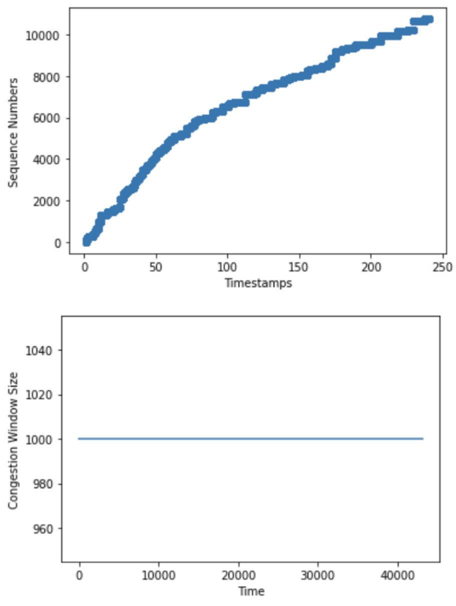


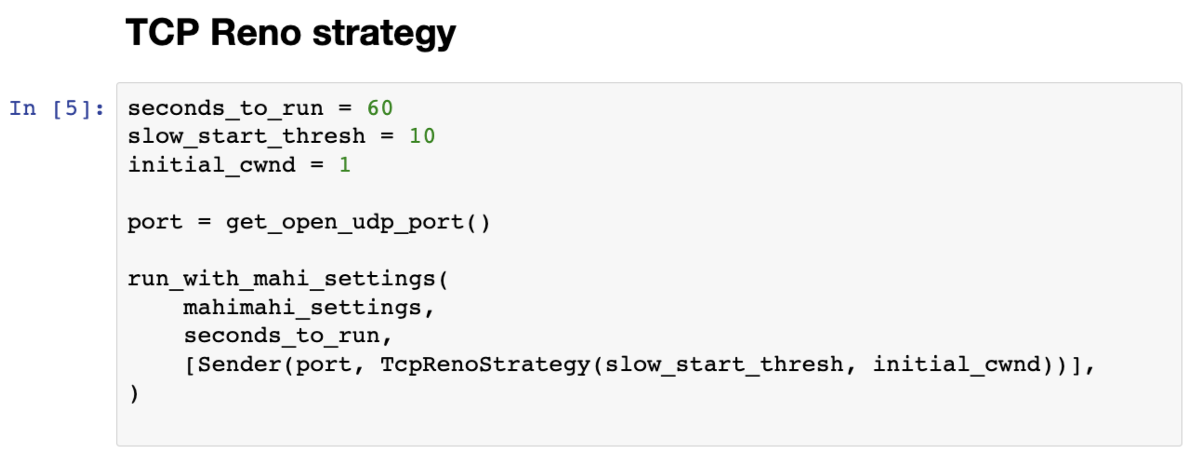
Fig. 9. Fixed window results

Once the previously mentioned cell is executed the results should appear below it. Refer to [interpreting the results](#)for more information on this.

The following cell has the other arguments specific to the TCP Reno Strategy. The only things that vary between this cell and the the previous one are:

- `slow_start_threshold` variable which is an integer that tells the program the total size of the Slow Start Phase.
- `intial_cwnd` variable typically set to 1, 5 or 10.
- A list of Sender instances with the `TcpRenoStrategy` instance with its corresponding arguments.

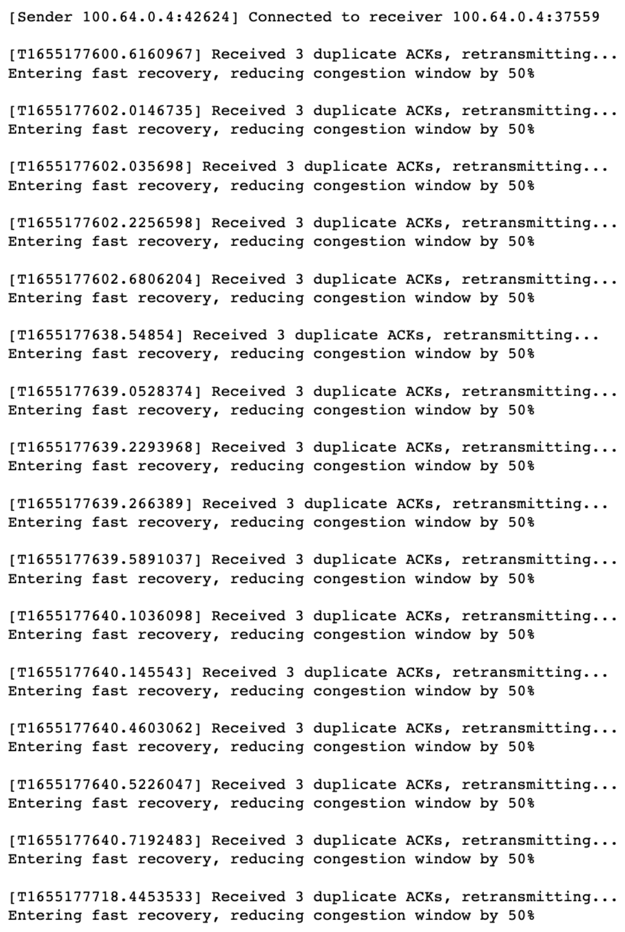
When executing the TCP Reno Strategy cell, it will be shown to the user a timestamp of the moment the server received three duplicate ACKs and when it is reducing the *cwnd*, along with the same results that are more thoroughly described in the following section (interpreting the results).



TCP Reno strategy

```
In [5]: seconds_to_run = 60
        slow_start_thresh = 10
        initial_cwnd = 1

        port = get_open_udp_port()

        run_with_mahi_settings(
            mahimahi_settings,
            seconds_to_run,
            [Sender(port, TcpRenoStrategy(slow_start_thresh, initial_cwnd))],
        )
```

Fig. 10. TCP Reno strategy API



```
[Sender 100.64.0.4:42624] Connected to receiver 100.64.0.4:37559

[T1655177600.6160967] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177602.0146735] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177602.035698] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177602.2256598] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177602.6806204] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177638.54854] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177639.0528374] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177639.2293968] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177639.266389] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177639.5891037] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177640.1036098] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177640.145543] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177640.4603062] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177640.5226047] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177640.7192483] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%

[T1655177718.4453533] Received 3 duplicate ACKs, retransmitting...
Entering fast recovery, reducing congestion window by 50%
```

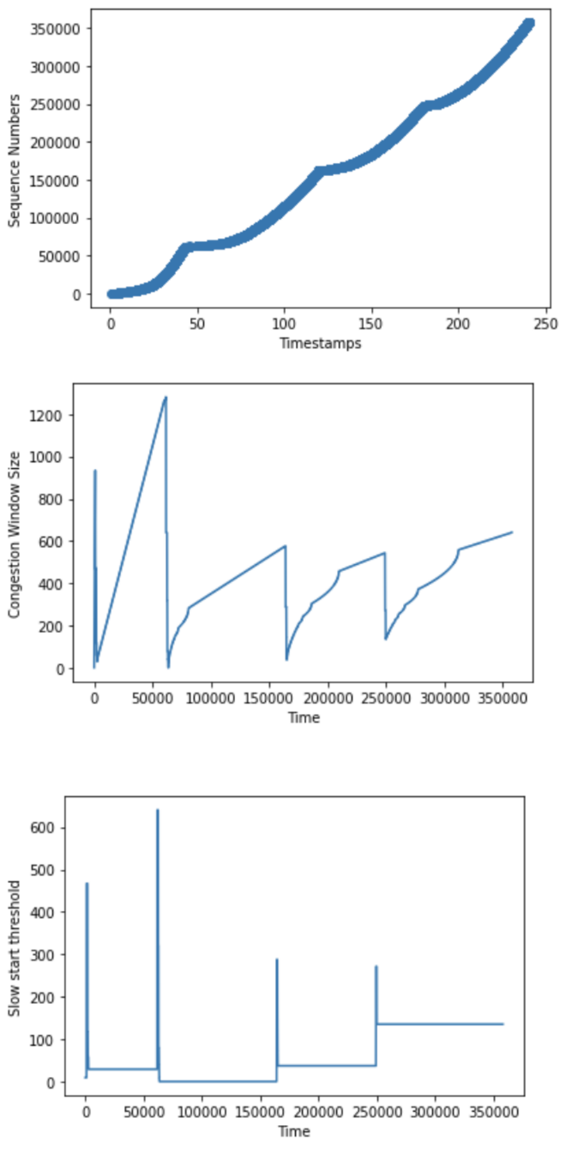


Fig. 11. TCP Reno results

3.2.2.1 Interpreting the Results

Once the corresponding cells are executed, and after waiting the necessary time, the output is shown. It is a brief description of the results obtained by the sender (or senders) along with a couple of graphs that are plotted using different variables.

In the summary of the results, the throughput is listed, which immediately tells us how much data was successfully transferred from the sender to the receiver. The higher the number, the better. Another interesting piece of information is the Round-Trip Time (RTT), which is the duration in milliseconds of the time it takes a request to leave the sender, to reach the receiver and return to the sender. This term tells us about the speed and reliability of the network. Therefore, the smaller, the better.

The first graph shows all the data transferred and the time it took the sender to transmit it. The Fixed Window graph shows a more linear output, whereas the TCP Reno strategy should show an exponential result.

Next, a graph plotting the size of the *cwnd* over time is shown. As expected, in the Fixed Window strategy, there is only a straight line, meaning that the *cwnd* remained the same during the program execution. On the contrary, in the TCP Reno graph the *cwnd* grows and shrinks in a saw-tooth manner; notice that the cwnd does not ever reach a value of 1 again.

Finally, a third graph is shown only after executing the TCP Reno strategy as it is the only one that contains the Slow Start Threshold variable. In this graph it can be seen how the threshold varies throughout time, therefore giving the reader an overview of when the packet losses occurred.

References

Anonymous. (2020). Network Congestion. 10/06/2022, de AVI Networks Sitio web: https://avinetworks.com/glossary/network-congestion/

Anonymous. (2019). What Is Network Congestion?. 10/06/2022, de Solarwinds Sitio web: https://www.solarwinds.com/es/resources/it-glossary/network-congestion

Barik, R., Welzl, M., Teymoori, P., Islam, S., & Gjessing, S. (2020). Performance Evaluation of In-network Packet Retransmissions using Markov Chains. 2020.

Bommisetty, L. (2021). Performance Analysis of TCP Queues: Effect of Buffer Size and Round Trip Time. 2021 6th International Conference on Signal Processing, Computing and Control (ISPCC), Signal Processing, Computing and Control (ISPCC), 2021 6th International Conference On, 631–635. https://0-doi-org.biblioteca-ils.tec.mx/10.1109/ISPCC53510.2021.9609523

Forouzan, B. A. (2010). TCP/IP protocol suite (4th ed.). McGraw-Hill/Higher Education.

Laxmi, V., Mehta, D., Gaur, M. S., Faruki, P., & lal, C. (2013). Impact analysis of JellyFish attack on TCP-based mobile ad-hoc networks

Nishitha, T. (2002). *A Comparative Analysis of TCP Tahoe, Reno, New- Reno, SACK and Vegas in Homogeneous Networks.* Semantic Scholar https://inst.eecs.berkeley.edu//~ee122/fa05/projects/Project2/SACKRENEVEGAS.pdf

Round Trip Time (RTT) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. (2022). Retrieved 14 June 2022, from https://developer.mozilla.org/en-US/docs/Glossary/Round_Trip_Time_(RTT)

Velte, T. J., & Velte, A. T. (2007). Cisco : a beginner's guide (4th ed.). McGraw-Hill.

Zhang, J., Zhao, C., Zheng, Z., & Cai, J. (2022). SR-WMN: Online Network Throughput Optimization in Wireless Mesh Networks With Segment Routing. IEEE Wireless Communications Letters, Wireless Communications Letters, IEEE, IEEE Wireless Commun. Lett, 11(2), 396–400. https://0-doi-org.biblioteca-ils.tec.mx/10.1109/LWC.2021.3129893