95705-Z Telecommunications Management Term Paper

Using Linux Traffic Control and mocha to test Node.js Applications Under Different Network Conditions: Project Faucet

August 2, 2018

Dario Alessandro Lecina-Talarico

Motivation

Developers usually design Software considering ideal network conditions, that is, infinite bandwidth and zero latency.

Sometimes we add retry strategies for handling errors and recovering from network failures but we rarely test our systems under “slow” network conditions as opposed to total failure.

As part of the integration tests, it would be very beneficial to submit the system to different network conditions like low egress bitrate and high latency to ensure that the system is able to operate or that it fails in a predictable way.

Objective and Scope of the Paper

In this paper, I present a strategy to test a web application under different network conditions using the Linux Traffic Control commands and Docker.

We also present a proof of concept client-server application written in nodejs that uses the concepts presented in this paper so that the user can control the egress and ingress bitrates and round trip time.

Glossary

Bitrate

Number of bits that are processed by unit of time.

Ingress Traffic

All the network traffic that it is directed to a particular node in a network from that node standpoint.

Egress Traffic

All the traffic directed towards an external network originated in from inside a particular node, from that node standpoint.

Bandwidth

In the context of network engineering is the maximum rate of data can be delivered through a given channel over time. It is measured in bit per second.

Throughput

In the context of network engineering is the actual rate of data was delivered through a given channel over time. It is measured in bits per second or bps.

Round-Trip Delay Time (RTD) or Round-Trip Time (RTT)

The length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgement of that signal to be received. This time delay includes the propagation times for the paths between the two communication endpoints. [7]

Introduction to the Used Technologies

Linux Traffic Control

“The tc command is used to configure Traffic Control in the Linux kernel. Traffic Control consists of the following:

SHAPING

When traffic is shaped, its rate of transmission is under control. Shaping may be more than lowering the available bandwidth - it is also used to smooth out bursts in traffic for better network behaviour. Shaping occurs on egress.

SCHEDULING

By scheduling the transmission of packets it is possible to improve interactivity for traffic that needs it while still guaranteeing bandwidth to bulk transfers. Reordering is also called prioritizing, and happens only on egress.

POLICING

Where shaping deals with transmission of traffic, policing pertains to traffic arriving. Policing thus occurs on ingress.

DROPPING

Traffic exceeding a set bandwidth may also be dropped forthwith, both on ingress and on egress."

Processing of traffic is controlled by three kinds of objects: qdiscs, classes and filters that will be further discussed in this paper.

A more comprehensive explanation of the Linux Traffic Control is available at [1]

Mocha

Mocha is a JavaScript test framework for Node.js programs, featuring browser support, asynchronous testing, test coverage reports, and use of any assertion library.

We will use Mocha to test our client application.

Docker Application Containers

"The first thing to know about Docker is that it is an application-container platform.

An application container has these characteristics:

- It is a software-defined environment. The container can be created and run entirely through software processes; it does not require any special hardware.

- The application that runs inside the container must be designed to run on the operating system that hosts the container. For example, an application running inside a container on a Linux server needs to be Linux-compatible.

- It is abstracted from the host system via controls that limit the ability of processes inside the container to interact with processes on the host. This isolation provides some security benefits, while also simplifying administration.

- In most cases, the container is also subject to limits on the amount of computing, storage, and networking resources that the containerized application can access." [4]

docker-compose

“docker-Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.” [5]

Our sample applications: Client and Server implementation using Node.js

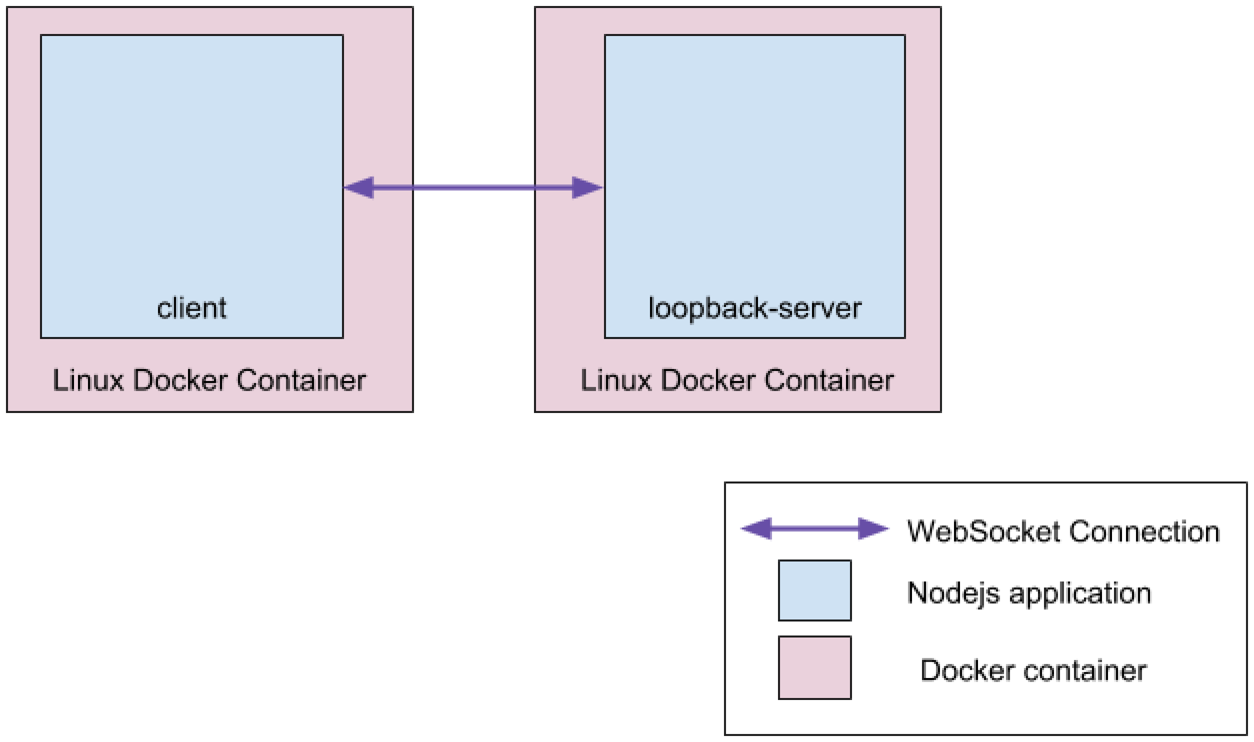


Figure 1. Components and Connections diagram

Figure 1 presents the Components and Diagram of the node.js proof of concept application.

Client

Our Client application does the following:

1. Configures the network settings using the specified parameters for outbound and inbound bitrates and round trip delay settings.

2. Connects to the Server.
3. Creates a payload with the specified configurable size.
4. Sends the payload and measures the time that it takes for the message to be returned by the server.
5. Closes the Server connection.

We decided to wrap our client in a **mocha** test case and we automated the process of setting up the client different network conditions using the faucet function.

```
describe('server-client', function() {
   const networkConditions = [
       { up:100, rtt:10 },
       { up:1000, rtt:10 },
       { up:10000, rtt:200 },
       { up:10, rtt:10 },
   ];

   networkConditions.forEach(async (networkConditions) => {
       it('Send and receive a 50kb payload in less than 1 second', async function() {
           this.timeout(500000);
           await faucet.start(networkConditions);
           const serverClient = new ServerClient({serverUrl: SERVER_URL});
           await serverClient.connect({retryCount2: 10});
           const payload= createFakePayload(50);
           const requestSent = Date.now();
           await serverClient.sendAndWaitForResponse(payload);
           const responseReceived = Date.now();
           const roundTripDelay = responseReceived - requestSent;
           console.log('roundTripDelay: ', roundTripDelay);
           assert(roundTripDelay < 1000, `Request must take less than\
            1 second, but it took ${roundTripDelay}mS`);
           await serverClient.disconnect();
           await faucet.stop();
       });
   });
});
```

Figure 2: client test case for sending a payload and receiving it in less than 1 second

The Faucet function

Faucet is a small library that I created to set the network conditions.

It provides a start and stop methods.

```
await faucet.start(networkConditions);
```

sets the network parameters, it interacts with the Linux OS to do that, specifically the tc or Traffic Control command, further details are explained in the “Using Linux Traffic Control for Traffic Shaping”

```
await faucet.stop();
```

Used to remove all the Traffic Shaping parameters from the OS.

Loopback-server

Our sample application is very simple, the Server exposes a WebSocket Server endpoint at “ws://loopback-server:3201/ws” it serves as an echo server, meaning, when a client sends a message, the server logs it and sends it back to the client.

Using Linux Traffic Control for Traffic Shaping

As described earlier in this paper, the tc or traffic control command will be used to configure the traffic shaping parameters.

This is the command that was used to configure the RTT and egress bitrate settings:

```
tc qdisc add dev i${iFace} root handle 1:0 netem delay
${halfWayRTT}ms rate ${egress}kbit
```

qdisc is the network scheduler used by the Linux Kernel.

`add dev ${iFace} root handle 1:0` is used to attach a new handle to the network scheduler.

iFace is the name of the interface to attach the **qdisc** handle to, we used the **route** command to find its name dynamically:

```
"route | grep '^default' | grep -o '[^ ]*$'"
```

netem provides Network Emulation functionality for testing protocols by emulating the properties of wide area networks. The current version emulates variable delay, loss, duplication and re-ordering.

```
netem delay ${halfWayRTT}ms rate ${egress}kbit
```

delay is used to simulate RTT
rate is used to limit egress bitrate.

Using Docker and Docker compose to containerize and launch our Node.js Applications

Figure 3 presents the Dockerfile uses for both the server and the client Docker containers.
It is important to note that the iproute2 is required to access the Linux Traffic Control features.

Figure 4 contains the docker-compose file that is used to launch both the client and the server. It is important to notice that the client application is able to connect to the server because of the link and ports that are specified in the docker-compose file.

```
FROM node:8.10-alpine

ENV NPM_CONFIG_LOGLEVEL=http
ENV PATH /app/node_modules/.bin:$PATH
ENV NODE_PATH /app/node_modules

WORKDIR /app

COPY package.json /app/package.json

# Enable sudo
RUN apk update
## Install TC
RUN apk add --no-cache iproute2
RUN apk add --no-cache net-tools

WORKDIR /app
COPY . /app/


STOPSIGNAL SIGINT
```

Figure 3. Dockerfile used for both Server and Client containers

```
version: '3.3'
```

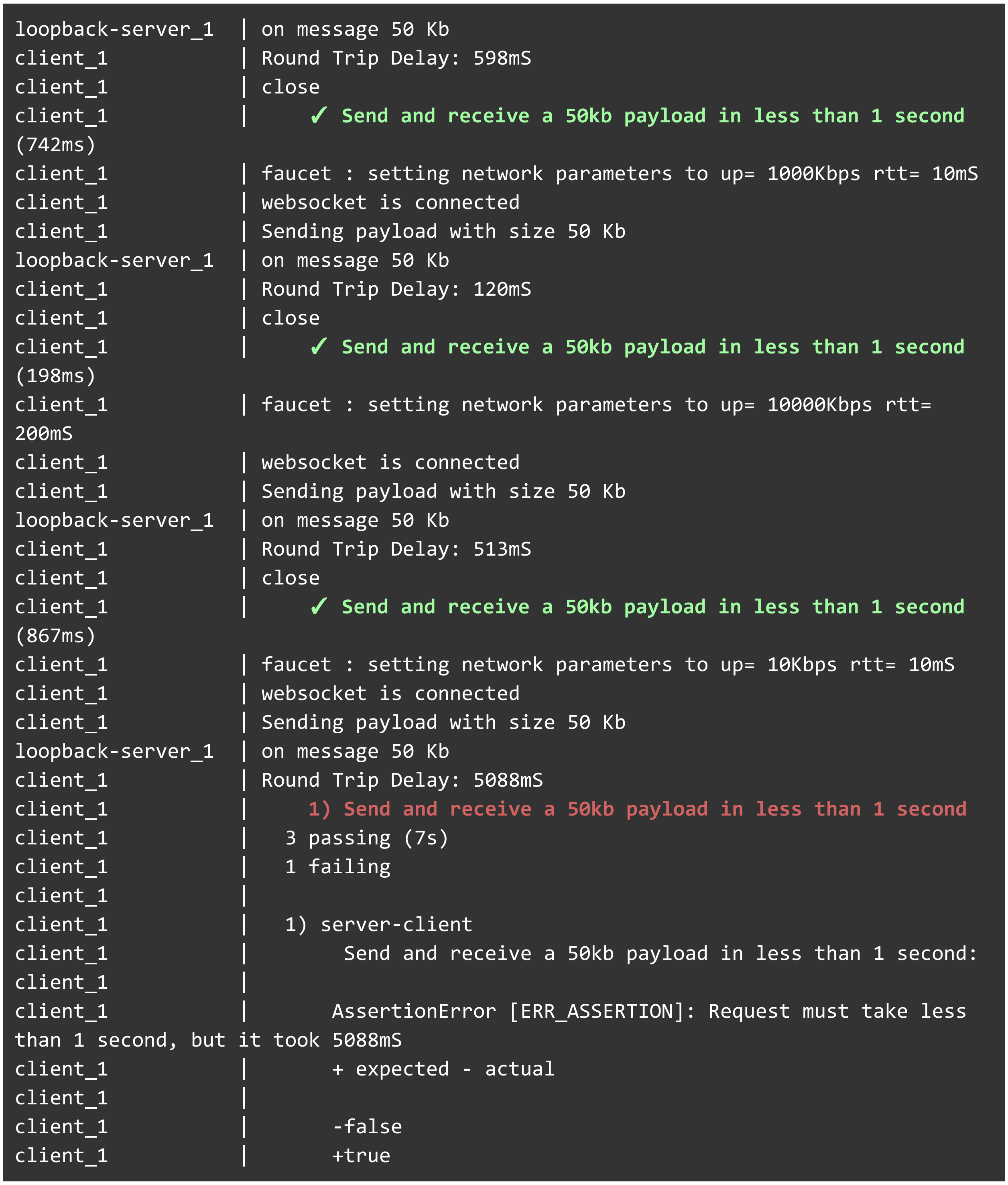
```
services:
 loopback-server:
   cap_add:
     - ALL
   privileged: true
   command: nodemon --config nodemon.json src/server/index.js
   env_file: .env
   build:
     context: .
     cache_from:
       - node:8.10-alpine
   volumes:
     - .:/app
   ports:
     - '3201:3201'
 client:
   cap_add:
     - ALL
   privileged: true
   command: mocha src/example/client/tests/*
   env_file: .env
   build:
     context: .
     cache_from:
       - node:8.10-alpine
   volumes:
     - .:/app
   links:
     - loopback-server
```

Figure 4. docker-compose file for this project

Analysis of Data Collected using different Linux Traffic Control parameters

Figure 5 is the output of running our test case presented in Figure 2, It is important to notice that the last test case failed because it took 5088 mS as opposed to the 1 second limit.

```
client_1            | faucet : setting network parameters to up= 100Kbps rtt= 10mS
client_1            | websocket is connected
client_1            | Sending payload with size 50 Kb
```



```
loopback-server_1  | on message 50 Kb
client_1           | Round Trip Delay: 598mS
client_1           | close
client_1           |     ✓ Send and receive a 50kb payload in less than 1 second
(742ms)
client_1           | faucet : setting network parameters to up= 1000Kbps rtt= 10mS
client_1           | websocket is connected
client_1           | Sending payload with size 50 Kb
loopback-server_1  | on message 50 Kb
client_1           | Round Trip Delay: 120mS
client_1           | close
client_1           |     ✓ Send and receive a 50kb payload in less than 1 second
(198ms)
client_1           | faucet : setting network parameters to up= 10000Kbps rtt=
200mS
client_1           | websocket is connected
client_1           | Sending payload with size 50 Kb
loopback-server_1  | on message 50 Kb
client_1           | Round Trip Delay: 513mS
client_1           | close
client_1           |     ✓ Send and receive a 50kb payload in less than 1 second
(867ms)
client_1           | faucet : setting network parameters to up= 10Kbps rtt= 10mS
client_1           | websocket is connected
client_1           | Sending payload with size 50 Kb
loopback-server_1  | on message 50 Kb
client_1           | Round Trip Delay: 5088mS
client_1           |     1) Send and receive a 50kb payload in less than 1 second
client_1           |   3 passing (7s)
client_1           |   1 failing
client_1           |
client_1           |   1) server-client
client_1           |        Send and receive a 50kb payload in less than 1 second:
client_1           |
client_1           |       AssertionError [ERR_ASSERTION]: Request must take less
than 1 second, but it took 5088mS
client_1           |       + expected - actual
client_1           |
client_1           |       -false
client_1           |       +true
```

Figure 5. Output of the provided test case

Recommendations

1. As presented in the mocha client test case in **Figure 2**, we introduced a proof of concept of how to incorporate network conditions in a test case without manipulating the network settings manually.
2. As it can be seen in **Figure 5** we proved that different network conditions can make a test case fail, we propose that developers incorporate network testing as part of the core development process so that their application is more robust to different network conditions.
3. We introduced the “faucet” function which can be used to configure network so that we can simulate different network conditions.
4. The Faucet source code is available on https://github.com/darioalessandro/faucet

Future work

It would be very important to control the egress traffic bitrate.

I found that in order to do that, it is necessary to create a virtual interface to proxy egress traffic through.

It is not trivial to create virtual network interfaces using docker because that requires the ifb or Intermediate Functional Block.

Bibliography

[1] https://linux.die.net/man/8/tc
[2] https://www.docker.com/what-docker
[3] Docker: Up & Running, 2nd Edition Karl Matthias; Sean P. Kane Published by O'Reilly Media, Inc., 2018
[4] Tozzi, C. (2017). Enterprise Docker, 1st Edition. O’Reilly Media, Inc.
[5] https://docs.docker.com/compose/overview/
[6] https://wiki.gentoo.org/wiki/Traffic_shaping
[7] Douglas E. Comer (2000). Internetworking with TCP/IP - Principles, Protocols and Architecture (4th ed.). Prentice Hall. p. 226. ISBN 0-13-018380-6.
[8] https://wiki.linuxfoundation.org/networking/netem