

# Daily Report File for the Semester Project

Furkan Karakaş

April 10, 2020

## 9 February 2020

I did the first installation. I also forked the tendermint repository to my GitHub repository <https://github.com/FurkanKarakas/tendermint>. We can run a single node tendermint at the moment but to run local testnets, we can utilize *docker*. To start a 4 node testnet, run:

```
make localnet-start
```

After executing multiple nodes, a new folder **build** is created. In this folder, each node has a log file, which is called **tendermint.log**. These log files contain the commit and execute steps of the program, written with their corresponding timestamps.

The nodes bind their RPC servers to ports 26657, 26660, 26662, and 26664 on the host. This file creates a 4-node network using the localnode image. The nodes of the network expose their P2P and RPC endpoints to the host machine on ports 26656-26657, 26659-26660, 26661-26662, and 26663-26664 respectively.

To stop and restart the environment, one can use the following commands:

```
make build-linux
make localnet-stop
make localnet-start
```

## 20 February 2020

We need to simulate network delays and packet losses in the Docker containers. There is a very good article here about it: [https://alexei-led.github.io/post/pumba\\_docker\\_netem/](https://alexei-led.github.io/post/pumba_docker_netem/). Linux allows us to manipulate the traffic flow using a tool called **tc**, which is available in **iproute2**. Another tool called **netem** is an extension of it. It allows us to emulate network failures such as delay, packet loss, packer reorder, duplication, corruption, and bandwidth rate.

For controlling traffic flow in the Docker containers, we will be using a software called *pumba*, which utilizes Linux traffic control and network emulation softwares *tc* and *netem*, respectively. A good documentation of it could be found at the website <https://github.com/alexei-led/pumba>.

Figure 1 illustrates the basic topology of a Docker host. Basically, **veth** interfaces are used in order to create a communication between each container and the docker itself. As I type the **ifconfig** command in the command line, I am able to see the aforementioned virtual interfaces, each corresponding to  $node_i$  where  $i \in \{0, 1, 2, 3\}$ :

```
veth282987a: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::dc1b:3cff:fe06:7484 prefixlen 64 scopeid 0x20<link>
```

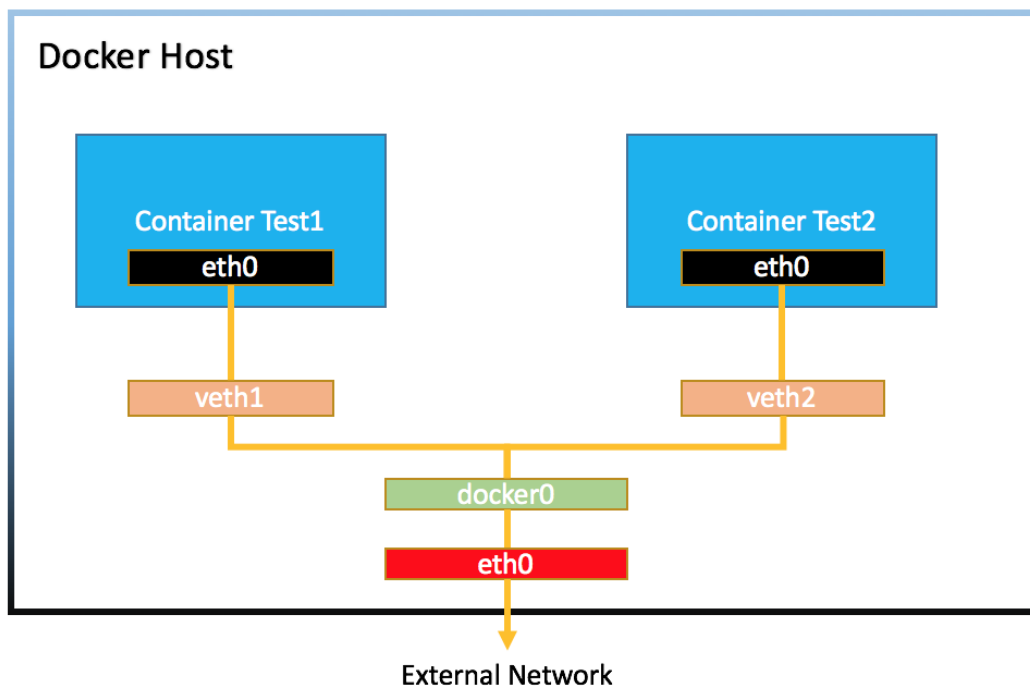


Figure 1: The basic network topology inside a Docker host

```

ether de:1b:3c:06:74:84 txqueuelen 0 (Ethernet)
RX packets 94167 bytes 57527418 (57.5 MB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 94874 bytes 58402540 (58.4 MB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth454c67a: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet6 fe80::24fd:25ff:fee5:45f prefixlen 64 scopeid 0x20<link>
ether 26:fd:25:e5:04:5f txqueuelen 0 (Ethernet)
RX packets 95109 bytes 58670384 (58.6 MB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 95470 bytes 58291946 (58.2 MB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

vethba77776: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet6 fe80::b0e9:54ff:fe9d:52cb prefixlen 64 scopeid 0x20<link>
ether b2:e9:54:9d:52:cb txqueuelen 0 (Ethernet)
RX packets 95284 bytes 58311392 (58.3 MB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 94967 bytes 58303854 (58.3 MB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

vethe97798a: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet6 fe80::4c55:84ff:fe93:4da8 prefixlen 64 scopeid 0x20<link>
ether 4e:55:84:93:4d:a8 txqueuelen 0 (Ethernet)
RX packets 94980 bytes 58487742 (58.4 MB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 95391 bytes 58199768 (58.1 MB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Following is a simple command to delay the traffic at `node0` 20 seconds with a duration of 30 seconds:

```

sudo pumba netem --duration 30s --interface eth0 --tc-image 'gaiadocker/
↪ iproute2' delay --time 20000 node0

```

## 27 February 2020

We need some network emulation techniques to test the tendermint under various conditions, including but not limited to:

- network delays by an arbitrary amount of time including jitter,
- network packet loss while sending packets from one node to another,
- packet duplication,
- corrupt packets,
- process crashes under different models, e.g., crash-recovery, crash-stop, etc.

We can emulate those behaviours by means of the **pumba** software, which is basically a toolbox for stress testing the docker containers.

In order to send transactions to the nodes, we can utilize the following command:

```
curl -s 'localhost:{NODE_PORT}/broadcast_tx_commit?tx="{TX_NAME}"'
```

where **NODE\_PORT** is the RPC port of the nodes that participate in the algorithm, and **TX\_NAME** is the name of the transaction that we would like to send. After sending a transaction, each participating node attests that this was a valid transaction in an *executed block*, then commits these changed in a *committed state*. After committing, the value of the **appHash** in the nodes also changes, where **appHash** is a hexadecimal number. For example, at start, the value of **appHash** was **appHash=0000000000000000**. After sending the transaction successfully, it became **appHash=0200000000000000**, after another transaction, **appHash=0400000000000000**, etc. Each participating node printed the following text after a successful transaction individually:

```
I[2020-02-27|10:08:48.317] Executed block module=state height=6183
    ↪ validTxs=1 invalidTxs=0
```

```
I[2020-02-27|10:08:48.319] Committed state module=state height=6183 txs
    ↪ =1 appHash=0400000000000000
```

Hence, this transaction is registered in the blockchain at height 6183.

Using network delays, I tried to send a transaction. With a network delay of 5000 ms, it took a while to validate the transaction, but at the end, the transaction is eventually validated by every participating node in the algorithm.

I created appropriate bash scripts for the stress testing of the Docker containers. From time to time, when I use the network emulation commands, e.g., when I try to simulate packet duplication, I get the following error from various nodes:

```
node2 | E[2020-02-27|13:35:05.265] dialing failed (attempts: 1): self
    ↪ ID<fa97dbb3d7afd62fa818f9a535fc88ed54fc283c> module=pex addr=
    ↪ fa97dbb3d7afd62fa818f9a535fc88ed54fc283c@192.167.10.4:26656
```

Currently, I created a bash script for sending successive transactions to a node.

## 5 March 2020

We can also send a key-value pair to the program. We observe that in the HTTP response, we have only the key in the JSON format:

```
{
  "key": "Y3JIYXRvcg==",
  "value": "Q29zbW9zaGkgTmV0b3dva28="
},
{
  "key": "a2V5",
  "value": "Y0FCWEw0UUVVNQ=="
}
```

Decoding base64 values, we see **creator** : **Cosmoshi Netowoko** and **key** : **KEY**. To get the value from the key, we need to send an HTTP request to the RPC port:

```
curl -s 'localhost:26657/abci_query?data="KEY" '
```

where **KEY** is the key that we have used initially. The response of this HTTP request returns the value that we have specified alongside with the key initially.

I moved the scripts from bash to Golang since working in Go will be much more convenient in the future.

Now, since I am working in Golang, concurrency is much easier to implement. I can send multiple transactions in a single block. I can go up to 14 transactions per height. The function to send a transaction is given below:

```
// Sends transactions
func sendTX(wg *sync.WaitGroup, portNr *string, TXSize *int) {
    defer wg.Done() // Decrement by 1 after function returns.
    resp, err := http.Get("http://127.0.0.1:" + *portNr + "/"
        ↪ broadcast_tx_commit?tx=\" + randSeq(*TXSize) + "\"")
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()

    if resp.StatusCode == http.StatusOK {
        bodyBytes, err := ioutil.ReadAll(resp.Body)
        if err != nil {
            panic(err)
        }
        bodyString := string(bodyBytes)
        log.Info(bodyString)
    }
}
```

We add the current goroutine to the work group **wg** so that the function does not return unless all goroutines finished executing.

My next goal is to find a metric where I can evaluate the performance of the system under network stress tests. A suitable metric would be “transactions per unit time”:

$$\text{Throughput} = \frac{\text{Total transactions}}{\text{Total elapsed time}}$$

## 12 March 2020

I noticed that sending transactions too fast causes problems. For example, I get the following error if I send a transaction every 10 ms:

```
{
  "jsonrpc": "2.0",
  "id": -1,
  "error": {
    "code": -32603,
    "message": "Internal error",
    "data": "max_subscription_clients 100 reached"
  }
}
```

```
}
}
```

**In this case, the transaction is lost** and it won't appear in one of the heights. Another error I get in the HTTP response body is the following:

```
{
  "jsonrpc": "2.0",
  "id": -1,
  "error": {
    "code": -32603,
    "message": "Internal error",
    "data": "timed out waiting for tx to be included in a block"
  }
}
```

In this case, the transactions are still applied in a future block even though the client gets an error message.

Table 1: Throughput values under various network delays

Network Delay (ms)	Valid TX Ratio	Throughput (TX/s)
100	100%	9.03
200	100%	8.71
500	100%	7.60
1000	100%	6.08
1500	100%	4.36
2000	100%	4.58
3000	99%	3.83

So, for measuring the throughput, I will consider that the error of 2nd kind still contributes to overall number of transactions whereas the error of 1st kind does not contribute to overall number of transactions. The results of the network delay are illustrated in Table 1.

## 21 March 2020

There was a concurrency issue in my code. Now, I use another goroutine to count the number of transactions. Now, the concurrency problem should be fixed. Furthermore, I opened two separate issues in my GitHub repository concerning the max client reached error and tx not included error.

## 2 April 2020

I found solutions for the error messages. There are configuration files where you can change preset variables for each node. The file is located at `build/nodeX/config/config.toml` where  $X \in \{0, 1, 2, 3\}$ . The variable names are `max_subscription_clients`, `timeout_broadcast_tx_commit`. I set these variables for each node to higher integer values so the problems disappeared for me. Also, the nodes were still broadcasting blocks even though the blocks did not contain any data. This setting can also be changed by modifying the variable `create_empty_blocks = false`. This way,

no blocks will be created unless one explicitly sends a transaction to one of the nodes. This way, we have much less traffic and the readability of the transactions is much better.

From time to time I get “connection reset” error:

```
panic: Get http://127.0.0.1:26657/broadcast_tx_commit?tx="g173jq6iKyiugzFSig79\
vgIam0Mz2qysKuV14h6hSaZANfo4f8iv0Y40g6JrHaYZu4ZsShXrD89mLZVJlZVcCvsv5AV05WbTGQ\
5R": read tcp 127.0.0.1:49034->127.0.0.1:26657: read: connection reset by peer
```

I coded my program in such a way that once I get an error, the program panics and stops execution. So once I get this error my program panics and gives me the error message. However, after I get this error message I start my program once again and I do not get this error message anymore. So it suffices to run the program again after getting this error. The throughput experimental results are shown in Table 2. Note that all transactions are valid in this scenario since we no longer get max client subscription limit reached error. The table was recorded with a total number of 200 transactions, each being sent every 1 milliseconds.

Table 2: Throughput values under various network delays after error fix

Network Delay (ms)	Valid TX Ratio	Throughput (TX/s)
100	100%	111.62
200	100%	71.83
500	100%	37.23
1000	100%	17.38
1500	100%	12.72
2000	100%	12.12
3000	100%	11.07
5000	100%	6.54

## 9 April 2020

I added two new scripts to find the abci query result and the size of a block. To find the size of a block, I query the block height and I find the length of the returned JSON object which is basically the size of that string **in bytes**. I push the changes to <https://github.com/FurkanKarakas/tendermint> in branch `release/v0.33.1`.

## 10 April 2020

In order to have more sound measurements, we need to have longer transactions. For example, it would be really important to have a minimum transaction time of 40 seconds. Assuming that we are sending a transaction every 50 milliseconds, we need  $\frac{1}{50ms} \cdot 40s = 800$  transactions approximately. Note that I had to increase the Ubuntu operating system limits for reading since socket accept method was giving “limit reached” error. In order to do that, I used the command `ulimit -n 4096` to increase the simultaneous connections to 4096. I also wanted to send transactions every 10 ms but it seems that it gets really difficult after some point. The problems I faced was like EOF error. I did some research in the Internet and I found that the server is terminating the connections.

The results of my findings are shown in Table 3.

Table 3: Throughput values under various network delays including zero delay throughput for a total number of transactions of 800

<b>Network Delay (ms)</b>	<b>Valid TX Ratio</b>	<b>Throughput (TX/s)</b>
0 (reference)	100%	19.47
100	100%	19.40
200	100%	19.10
500	100%	18.82
1000	100%	18.21
1500	100%	17.05
2000	100%	16.58
3000	100%	11.05
5000	100%	7.31

It is also important to use some other network stress tests such as packet loss. My findings are illustrated in Table 4. Note that after some particular packet loss probability (in my case: 40%), the throughput decreases significantly. It might be due to the fact that each concurrent HTTP request makes it harder to repair the packet loss over long periods of time.

Table 4: Throughput values under various network packet loss probabilities including zero loss probability throughput for a total number of transactions of 800

<b>Probability of packet loss (%)</b>	<b>Valid TX Ratio</b>	<b>Throughput (TX/s)</b>
0 (reference)	100%	19.58
5	100%	19.34
10	100%	19.28
15	100%	19.00
20	100%	17.69
25	100%	18.88
30	100%	18.78
40	100%	15.34
50	100%	9.42