Network Term Project
Part 1

1st Ilker SIĞIRCI
Computer Engineering
2171940

METU
Ankara, Turkey
sigirci.ilker@gmail.com

2nd Yasin UYSAL
Computer Engineering
2172096

METU
Ankara, Turkey
yasinu97@gmail.com

I. Design and Implementation Part

A. Project Overview

This project aim to design a network, consisting 5 nodes, which are one source, one destination and three arbitrary routers. UDP is employed as protocol on every link between nodes. First part of the project focuses on calculating an average cost for each link. By making use of these data, we consider best path to route our messages, by manually using Dijkstra's Shortest Path Algorithm. On the second part of the project, after finding our route from source to destination, we conducted 3 experiments and observed the effect of delay configurations on that specified path. These experiments differ in terms of amount of delay applied to links. These results are then are plotted to a network emulation delay vs end-to-end delay histogram with a % 95 confidence interval.

B. Design Approach

Fundamentally, we had 5 identical nodes, only 2 of them are assigned special roles as source and destination. For the first part, it was expected from us to calculate average Round Trip Time(RTT) values so all nodes are treated equally. However, there was a specification stating that calculated values need to be stored in arbitrary nodes which are r1, r2 and r3 routers. To be able to achieve this goal, a design choice was made stating r1 and r3 should be used as clients which will be sending packets, while s and d nodes are assigned to server role which will be listening for packets at specifically picked ports. For r2, it needs to be working as both server and client. At this point, we realized another specification must be met which states communications must continue simultaneously. This is achieved by using threads. To be more specific, every node runs as many threads as its connections. r1 and r3 runs three distinct client threads while s and d are running three distinct server threads. As for r2, it is running two server and two client threads. Since while calculating round trip time, time spent on incoming and outgoing packages is equal so every link can be calculated independently.

C. Implementation Approach

Firstly, we chose to implement the project with python3. Reasons that why we chose Python3 are:
It is the most used scripting languages among other languages. It is easy to write and it handles the network operations very well. Creating sockets and threads are intuitive and because of that we could give our focus to code writing.
Before switching to actual project implementation part, we decided to implement a simple server and client application over localhost of the same computer. Our mentality to do that was to get familiar with socket programming concepts. Since the project is about User Datagram Protocol (UDP)-based socket application, our test application should also UDP. To do that, we needed to create a socket with the parameters:AF_INET, SOCK_DGRAM.
AF_INET: AF_INET is an address family that is used to designate the type of addresses that your socket can communicate with (in this case, Internet Protocol v4 addresses) [1]
SOCK_DGRAM: Provides datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times. This type is supported in the AF_INET, AF_INET6, and AF_UNIX domains. [2]
When client-server application with UDP was done, we decided to implement another test application which is that consist of 3 nodes "server/ client-server / client". We needed to implement this because the project has some internal nodes and we needed to understand how they worked. After these test applications, we switched back to implementation of project. By reading GENI tutorials to create a slice with using an XML file. Then, we downloaded the xml file and created the slices. Hence, our topology was created with manually given IP addresses.
Then, before implementing codes, we needed to choose one scenario out of two scenarios.We chose the first scenario which is: "The costs of the links (s-r1, r1-r2, r1-d) will be estimated and saved by r1, the costs of the links (s-r3, r2-r3, r3-d) will be estimated and saved by r3, and finally, the costs of the links (s-r2, r2-d) will be estimated and saved by r2" Actually,

there was no specific reason of choosing this scenario. But the second scenario calculates most of the time delays on the r2 and we chose to balance the delay calculations on the nodes by choosing the first scenario.

II. Methodology and Motivation

A. Methodology

We made use of "socket" library to be able to send packets to or receive feedback from ports. We also used "threading" library so we can simultaneously run tasks for every link of each node. Another important library used is "time" which is used to find end-to-end delays and emulation delays.

B. Motivation

We had the chance to experience socket programming for the first time in this assignment. While doing so, we also had experience to organize our implementation with threads such that a device can work on multiple sockets. Furthermore, we discussed several ways to manipulate the streams, in terms of accuracy and effectiveness.

III. Experimental Results

A. Discovery Part

On the first part of the project, we are expected to calculate average round trip times for every node. Round trip time represent total time spent on a link, beginning from packet sending time of client until receiving feedback from server. In order to be able to calculate an average reliably, many packages must be sent. Using 1000 packages were suggested so we sticked with it. Our design is based on using 3 threads as client on r1 and r3 nodes, 3 threads as server on s and d nodes, 2 threads as client and 2 threads as server in addition to 2 threads as client. Namely, r1 and r3 are sending 1000 packets to r2, s, d and calculating feedback returning times. This happens at the same time as r2 is responding to packets sent from r1 and r3 while also sending its own 1000 messages to both s and d servers. s and d nodes are only responding to incoming packets. Since time spent on a link does not depend on the directions, this measurement fits perfectly with our purpose. Another reason for this design choice is the requirement to store average RTT values in the routers. The point we used about it is that round trip is completed when sent packet returns back to the packet's initial sender, which is namely client. So we made our choice in favor of selecting routers as clients. RTT values of each link can be seen in the Figure1.

After finding average RTT values for each link, next task was calculating a best path according to Dijkstra's Shortest Path Algorithm. This algorithms iterates as many times as node count. Initial nodes cost is equal to zero while other nodes' cost are equal to infinity. At each iteration, node with lowest cost is picked as current and for every node it connects, path costs of that node forwarding through current node is calculated. If that value is lower than picked node's current cost, it is replaced with the lower cost. That means if that node needs to be reached, best path to that node is passing through

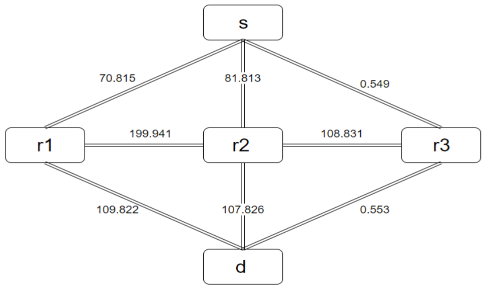
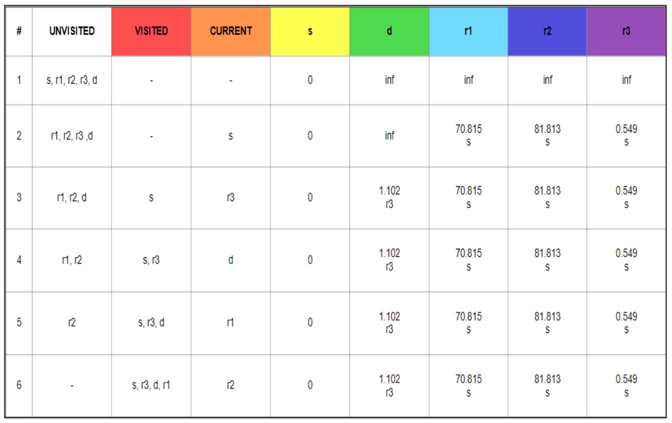


Fig. 1. RTT values of each node

current node. Since this is recursive, by following this rule, full shortest path to desired node can be found. Found path should be the lowest cost path from source to destination node. Judging by the RTT values we obtained, Dijkstra's algorithm is applied and shortest path is found to be s - r3 - d. When we observed the configurations which we were instructed to apply at the first place, we realized these configurations put some amount of delay to the links. Only links which are not affected by these configurations were s - r3 and r3 - d. Since delay on these links are way too low compared to other paths and these two links are completing the path from source to destination, we considered this path to be shortest. Each iteration of Dijkstra's Shortest Path Algorithm as been seen in the Figure2.

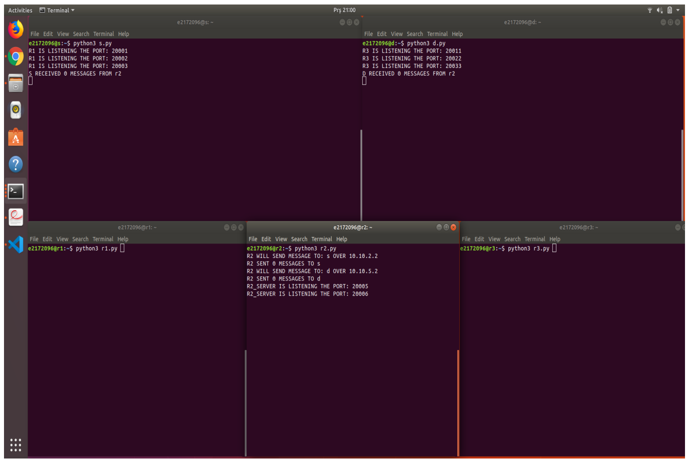


#	UNVISITED	VISITED	CURRENT	s	d	r1	r2	r3
1	s, r1, r2, r3, d	-	-	0	inf	inf	inf	inf
2	r1, r2, r3 ,d	-	s	0	inf	70.815 s	81.813 s	0.549 s
3	r1, r2, d	s	r3	0	1.102 r3	70.815 s	81.813 s	0.549 s
4	r1, r2	s, r3	d	0	1.102 r3	70.815 s	81.813 s	0.549 s
5	r2	s, r3, d	r1	0	1.102 r3	70.815 s	81.813 s	0.549 s
6	-	s, r3, d, r1	r2	0	1.102 r3	70.815 s	81.813 s	0.549 s

Fig. 2. Iteration Table of Dijkstra's Shortest Path

B. Experiment Part

After finding the best path, at the last part of the project, we are required to modify costs of the links on that best path. For this part of the project, communication will only be done through the best path. The point we want to observe is that when we configure these links, different experiments will be conducted, by using different delay values and difference between them will be observed. These experiments are conducted by using delays of 20 ms, 40 ms and 50 ms on both s - r3



```
e2172096@s:~$ python3 s.py
R1 IS LISTENING THE PORT: 20001
R1 IS LISTENING THE PORT: 20002
R1 IS LISTENING THE PORT: 20003
S RECEIVED 0 MESSAGES FROM r2
```

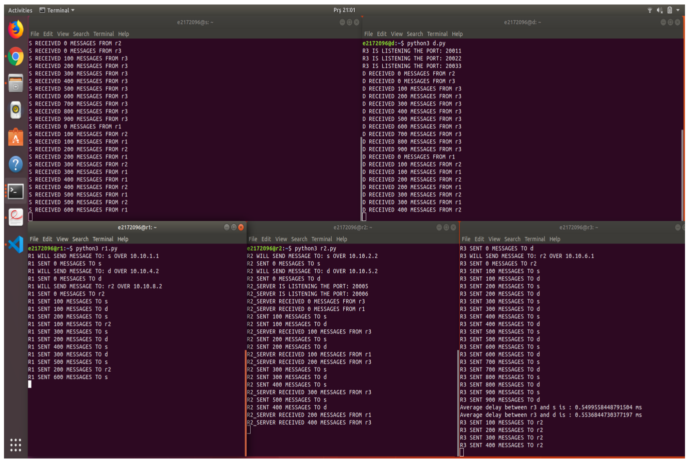
```
e2172096@d:~$ python3 d.py
R3 IS LISTENING THE PORT: 20011
R3 IS LISTENING THE PORT: 20022
R3 IS LISTENING THE PORT: 20033
D RECEIVED 0 MESSAGES FROM r2
```

```
e2172096@r1:~$ python3 r1.py
```

```
e2172096@r2:~$ python3 r2.py
R2 WILL SEND MESSAGE TO: s OVER 10.10.2.2
R2 SENT 0 MESSAGES TO s
R2 WILL SEND MESSAGE TO: d OVER 10.10.5.2
R2 SENT 0 MESSAGES TO d
R2_SERVER IS LISTENING THE PORT: 20005
R2_SERVER IS LISTENING THE PORT: 20006
```

```
e2172096@r3:~$ python3 r3.py
```

Fig. 3. Start of the Discovery Part



```
S RECEIVED 0 MESSAGES FROM r2
S RECEIVED 0 MESSAGES FROM r3
S RECEIVED 100 MESSAGES FROM r3
S RECEIVED 200 MESSAGES FROM r3
S RECEIVED 300 MESSAGES FROM r3
S RECEIVED 400 MESSAGES FROM r3
S RECEIVED 500 MESSAGES FROM r3
S RECEIVED 600 MESSAGES FROM r3
S RECEIVED 700 MESSAGES FROM r3
S RECEIVED 800 MESSAGES FROM r3
S RECEIVED 900 MESSAGES FROM r3
S RECEIVED 0 MESSAGES FROM r1
S RECEIVED 100 MESSAGES FROM r2
S RECEIVED 100 MESSAGES FROM r1
S RECEIVED 200 MESSAGES FROM r2
S RECEIVED 200 MESSAGES FROM r1
S RECEIVED 300 MESSAGES FROM r2
S RECEIVED 300 MESSAGES FROM r1
S RECEIVED 400 MESSAGES FROM r1
S RECEIVED 400 MESSAGES FROM r2
S RECEIVED 500 MESSAGES FROM r1
S RECEIVED 500 MESSAGES FROM r2
S RECEIVED 600 MESSAGES FROM r1
```

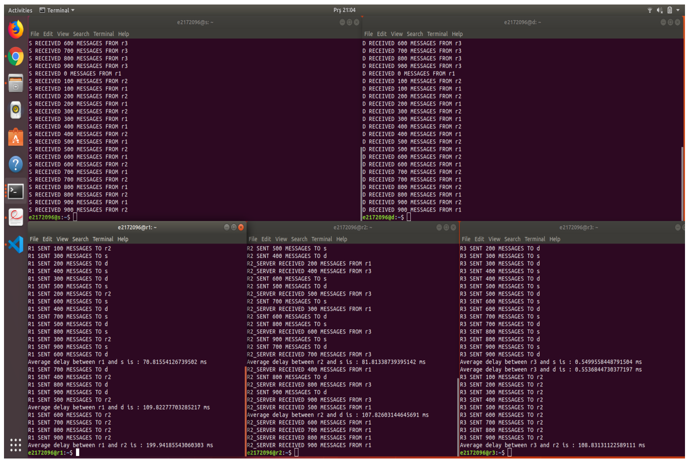
```
e2172096@d:~$ python3 d.py
R3 IS LISTENING THE PORT: 20011
R3 IS LISTENING THE PORT: 20022
R3 IS LISTENING THE PORT: 20033
D RECEIVED 0 MESSAGES FROM r2
D RECEIVED 0 MESSAGES FROM r3
D RECEIVED 100 MESSAGES FROM r3
D RECEIVED 200 MESSAGES FROM r3
D RECEIVED 300 MESSAGES FROM r3
D RECEIVED 400 MESSAGES FROM r3
D RECEIVED 500 MESSAGES FROM r3
D RECEIVED 600 MESSAGES FROM r3
D RECEIVED 700 MESSAGES FROM r3
D RECEIVED 800 MESSAGES FROM r3
D RECEIVED 900 MESSAGES FROM r3
D RECEIVED 0 MESSAGES FROM r1
D RECEIVED 100 MESSAGES FROM r2
D RECEIVED 100 MESSAGES FROM r1
D RECEIVED 200 MESSAGES FROM r2
D RECEIVED 200 MESSAGES FROM r1
D RECEIVED 300 MESSAGES FROM r2
D RECEIVED 300 MESSAGES FROM r1
D RECEIVED 400 MESSAGES FROM r2
```

```
e2172096@r1:~$ python3 r1.py
R1 WILL SEND MESSAGE TO: s OVER 10.10.1.1
R1 SENT 0 MESSAGES TO s
R1 WILL SEND MESSAGE TO: d OVER 10.10.4.2
R1 SENT 0 MESSAGES TO d
R1 WILL SEND MESSAGE TO: r2 OVER 10.10.8.2
R1 SENT 0 MESSAGES TO r2
R1 SENT 100 MESSAGES TO s
R1 SENT 100 MESSAGES TO d
R1 SENT 200 MESSAGES TO s
R1 SENT 100 MESSAGES TO r2
R1 SENT 300 MESSAGES TO s
R1 SENT 200 MESSAGES TO d
R1 SENT 400 MESSAGES TO s
R1 SENT 300 MESSAGES TO d
R1 SENT 500 MESSAGES TO s
R1 SENT 200 MESSAGES TO r2
R1 SENT 600 MESSAGES TO s
```

```
e2172096@r2:~$ python3 r2.py
R2 WILL SEND MESSAGE TO: s OVER 10.10.2.2
R2 SENT 0 MESSAGES TO s
R2 WILL SEND MESSAGE TO: d OVER 10.10.5.2
R2 SENT 0 MESSAGES TO d
R2_SERVER IS LISTENING THE PORT: 20005
R2_SERVER IS LISTENING THE PORT: 20006
R2_SERVER RECEIVED 0 MESSAGES FROM r3
R2_SERVER RECEIVED 0 MESSAGES FROM r1
R2 SENT 100 MESSAGES TO s
R2 SENT 100 MESSAGES TO d
R2_SERVER RECEIVED 100 MESSAGES FROM r3
R2 SENT 200 MESSAGES TO s
R2 SENT 200 MESSAGES TO d
R2_SERVER RECEIVED 100 MESSAGES FROM r1
R2_SERVER RECEIVED 200 MESSAGES FROM r3
R2 SENT 300 MESSAGES TO s
R2 SENT 300 MESSAGES TO d
R2 SENT 400 MESSAGES TO s
R2_SERVER RECEIVED 300 MESSAGES FROM r3
R2 SENT 500 MESSAGES TO s
R2 SENT 400 MESSAGES TO d
R2_SERVER RECEIVED 200 MESSAGES FROM r1
R2_SERVER RECEIVED 400 MESSAGES FROM r3
```

```
R3 SENT 0 MESSAGES TO d
R3 WILL SEND MESSAGE TO: r2 OVER 10.10.6.1
R3 SENT 0 MESSAGES TO r2
R3 SENT 100 MESSAGES TO s
R3 SENT 100 MESSAGES TO d
R3 SENT 200 MESSAGES TO s
R3 SENT 200 MESSAGES TO d
R3 SENT 300 MESSAGES TO s
R3 SENT 300 MESSAGES TO d
R3 SENT 400 MESSAGES TO s
R3 SENT 400 MESSAGES TO d
R3 SENT 500 MESSAGES TO s
R3 SENT 500 MESSAGES TO d
R3 SENT 600 MESSAGES TO s
R3 SENT 600 MESSAGES TO d
R3 SENT 700 MESSAGES TO s
R3 SENT 700 MESSAGES TO d
R3 SENT 800 MESSAGES TO s
R3 SENT 800 MESSAGES TO d
R3 SENT 900 MESSAGES TO s
R3 SENT 900 MESSAGES TO d
Average delay between r3 and s is : 0.5499558448791504 ms
Average delay between r3 and d is : 0.5536844730377197 ms
R3 SENT 100 MESSAGES TO r2
R3 SENT 200 MESSAGES TO r2
R3 SENT 300 MESSAGES TO r2
R3 SENT 400 MESSAGES TO r2
```

Fig. 4. Middle of the Discovery Part



```
S RECEIVED 600 MESSAGES FROM r3
S RECEIVED 700 MESSAGES FROM r3
S RECEIVED 800 MESSAGES FROM r3
S RECEIVED 900 MESSAGES FROM r3
S RECEIVED 0 MESSAGES FROM r1
S RECEIVED 100 MESSAGES FROM r2
S RECEIVED 100 MESSAGES FROM r1
S RECEIVED 200 MESSAGES FROM r2
S RECEIVED 200 MESSAGES FROM r1
S RECEIVED 300 MESSAGES FROM r2
S RECEIVED 300 MESSAGES FROM r1
S RECEIVED 400 MESSAGES FROM r1
S RECEIVED 400 MESSAGES FROM r2
S RECEIVED 500 MESSAGES FROM r1
S RECEIVED 500 MESSAGES FROM r2
S RECEIVED 600 MESSAGES FROM r1
S RECEIVED 600 MESSAGES FROM r2
S RECEIVED 700 MESSAGES FROM r1
S RECEIVED 700 MESSAGES FROM r2
S RECEIVED 800 MESSAGES FROM r1
S RECEIVED 800 MESSAGES FROM r2
S RECEIVED 900 MESSAGES FROM r1
S RECEIVED 900 MESSAGES FROM r2
e2172096@s:~$
```

```
D RECEIVED 600 MESSAGES FROM r3
D RECEIVED 700 MESSAGES FROM r3
D RECEIVED 800 MESSAGES FROM r3
D RECEIVED 900 MESSAGES FROM r3
D RECEIVED 0 MESSAGES FROM r1
D RECEIVED 100 MESSAGES FROM r2
D RECEIVED 100 MESSAGES FROM r1
D RECEIVED 200 MESSAGES FROM r2
D RECEIVED 200 MESSAGES FROM r1
D RECEIVED 300 MESSAGES FROM r2
D RECEIVED 300 MESSAGES FROM r1
D RECEIVED 400 MESSAGES FROM r2
D RECEIVED 400 MESSAGES FROM r1
D RECEIVED 500 MESSAGES FROM r2
D RECEIVED 500 MESSAGES FROM r1
D RECEIVED 600 MESSAGES FROM r2
D RECEIVED 600 MESSAGES FROM r1
D RECEIVED 700 MESSAGES FROM r2
D RECEIVED 700 MESSAGES FROM r1
D RECEIVED 800 MESSAGES FROM r2
D RECEIVED 800 MESSAGES FROM r1
D RECEIVED 900 MESSAGES FROM r2
D RECEIVED 900 MESSAGES FROM r1
e2172096@d:~$
```

```
R1 SENT 100 MESSAGES TO r2
R1 SENT 300 MESSAGES TO s
R1 SENT 200 MESSAGES TO d
R1 SENT 400 MESSAGES TO s
R1 SENT 300 MESSAGES TO d
R1 SENT 500 MESSAGES TO s
R1 SENT 200 MESSAGES TO r2
R1 SENT 600 MESSAGES TO s
R1 SENT 400 MESSAGES TO d
R1 SENT 700 MESSAGES TO s
R1 SENT 500 MESSAGES TO d
R1 SENT 800 MESSAGES TO s
R1 SENT 300 MESSAGES TO r2
R1 SENT 900 MESSAGES TO s
R1 SENT 600 MESSAGES TO d
Average delay between r1 and s is : 70.81554126739502 ms
R1 SENT 700 MESSAGES TO d
R1 SENT 400 MESSAGES TO r2
R1 SENT 800 MESSAGES TO d
R1 SENT 900 MESSAGES TO d
R1 SENT 500 MESSAGES TO r2
Average delay between r1 and d is : 109.82277703285217 ms
R1 SENT 600 MESSAGES TO r2
R1 SENT 700 MESSAGES TO r2
R1 SENT 800 MESSAGES TO r2
R1 SENT 900 MESSAGES TO r2
Average delay between r1 and r2 is : 199.94185543060303 ms
e2172096@r1:~$
```

```
R2 SENT 500 MESSAGES TO s
R2 SENT 400 MESSAGES TO d
R2_SERVER RECEIVED 200 MESSAGES FROM r1
R2_SERVER RECEIVED 400 MESSAGES FROM r3
R2 SENT 600 MESSAGES TO s
R2 SENT 500 MESSAGES TO d
R2_SERVER RECEIVED 500 MESSAGES FROM r3
R2 SENT 700 MESSAGES TO s
R2_SERVER RECEIVED 300 MESSAGES FROM r1
R2 SENT 600 MESSAGES TO d
R2 SENT 800 MESSAGES TO s
R2_SERVER RECEIVED 600 MESSAGES FROM r3
R2 SENT 900 MESSAGES TO s
R2 SENT 700 MESSAGES TO d
R2_SERVER RECEIVED 700 MESSAGES FROM r3
Average delay between r2 and s is : 81.81338739395142 ms
R2_SERVER RECEIVED 400 MESSAGES FROM r1
R2 SENT 800 MESSAGES TO d
R2_SERVER RECEIVED 800 MESSAGES FROM r3
R2 SENT 900 MESSAGES TO d
R2_SERVER RECEIVED 900 MESSAGES FROM r3
R2_SERVER RECEIVED 500 MESSAGES FROM r1
Average delay between r2 and d is : 107.82603144645691 ms
R2_SERVER RECEIVED 600 MESSAGES FROM r1
R2_SERVER RECEIVED 700 MESSAGES FROM r1
R2_SERVER RECEIVED 800 MESSAGES FROM r1
R2_SERVER RECEIVED 900 MESSAGES FROM r1
e2172096@r2:~$
```

```
R3 SENT 200 MESSAGES TO d
R3 SENT 300 MESSAGES TO s
R3 SENT 300 MESSAGES TO d
R3 SENT 400 MESSAGES TO s
R3 SENT 400 MESSAGES TO d
R3 SENT 500 MESSAGES TO s
R3 SENT 500 MESSAGES TO d
R3 SENT 600 MESSAGES TO s
R3 SENT 600 MESSAGES TO d
R3 SENT 700 MESSAGES TO s
R3 SENT 700 MESSAGES TO d
R3 SENT 800 MESSAGES TO s
R3 SENT 800 MESSAGES TO d
R3 SENT 900 MESSAGES TO s
R3 SENT 900 MESSAGES TO d
Average delay between r3 and s is : 0.5499558448791504 ms
Average delay between r3 and d is : 0.5536844730377197 ms
R3 SENT 100 MESSAGES TO r2
R3 SENT 200 MESSAGES TO r2
R3 SENT 300 MESSAGES TO r2
R3 SENT 400 MESSAGES TO r2
R3 SENT 500 MESSAGES TO r2
R3 SENT 600 MESSAGES TO r2
R3 SENT 700 MESSAGES TO r2
R3 SENT 800 MESSAGES TO r2
R3 SENT 900 MESSAGES TO r2
Average delay between r3 and r2 is : 108.8313122589111 ms
e2172096@r3:~$
```

Fig. 5. End of the Discovery Part

and r3 - d links, respectively. To be able to get reliable results, each experiment is conducted 10 times and mean values are recorded for later comparison.

One important point in our implementation is Network Time Protocol(NTP). We used NTP in order to be able to sync the times of routers, namely when using time.time(). When NTP is used, regardless of their real life location, routers are returning a timestamp based on same server so synchronization is achieved that way. Also, end-to-end delays are calculated by the following logic in our implementation: Instead of sending a random message in packets, timestamp of source node is get and put into packet, in bytecode. This timestamp is calculated just before sending packet. On the destination node, immediately after getting the packet, another timestamp is taken. Then, message in packet is decoded to be able to used. When difference between these two timestamps are taken, end-to-end delay is found.

There were two design approaches which can be followed at this point. First one, which is the one we implemented in our design, is not using acknowledge feedbacks. In more detail, packets flow is only from s to r3, then redirected from r3 to d. This approach is different from the approach on the first part. Source nodes send packets consecutively in a for loop, without waiting a response from r3. When r3 receives packets from s, without sending a feedback to s, it directs this packet to the destination node. Destination node is only responsible for catching that packets. Other approach is implementing an ACK/NACK logic, similar to the one implemented in the first part. However, since aim of that part is not running several tasks on each node and in order to be able to direct packets to d, r3 needs to get that packet from s first, threading logic is not used here.

Each approach has its drawbacks. ACK/NACK approach is better in the sense that for s to be able to send a new packet, it needs to get an acknowledgement packet from r3. This way, r3 is able to catch every packet sent from s. Packet loss is prevented by using this design. However, this requires extra lines to set up and control this communication. The approach we implemented is not using these acknowledgement packets. Source node continuously sends specified amount of packets to r3. Major drawback of this method is this point. r3 is not able to catch all of these packets. As a result, d receives way less packets. Statistically, when 1000 packets are sent from s, only about 250 packets reached d node. Also, because of that reason, to handle incoming packets, r3 and d needs to run a while loop since receiver does not know how many packets it would be able to catch from sender. However, ACK/NACK strategy is guaranteed to get every sent packet so for loop for a fixed amount of iterations would work well in that case. When we observed the mean values acquired, results were fluctuating around 43ms - 38ms, for the first experiment. However, theoretically it must be around 40ms. Why this is happening is losing many packets, in other words losing elements in normal distribution. To prevent this, a very small delay between packet sending operations are enforced, only on the source node. Fundamentally, this is a primitive version of ACK/NACK approach. Instead of waiting for a feedback, a foretold delay is used so packets can be received at r3. When this addition is implemented, as a result, d was able to catch almost 980 packet out of 1000 and mean end-to-end delays are more consistent, around 40ms for first experiment, around 80ms for second experiment, around 100ms for third experiment.

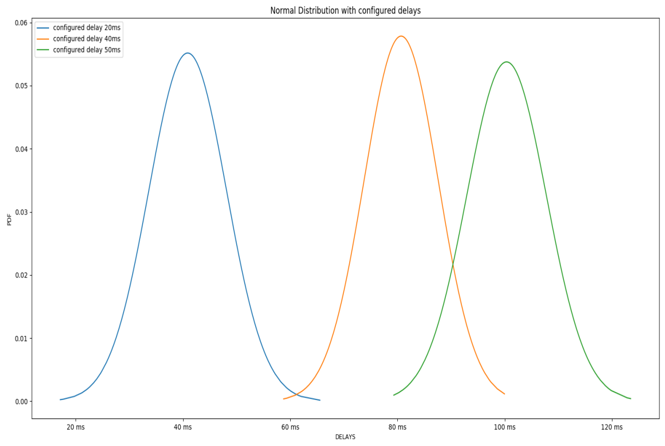


Fig. 6. Gaussian Distribution of Configured Delays

Above figure is a composite graph of the distribution of end-to-end delays. It proves that package delays indeed forms a Gaussian curve. Although we configured our 20 ms, 40 ms and 50 ms delay configuration files with ”distribution normal” keyword, we didn’t exactly sure that package delays conforms the normal distribution until we saw this graph.

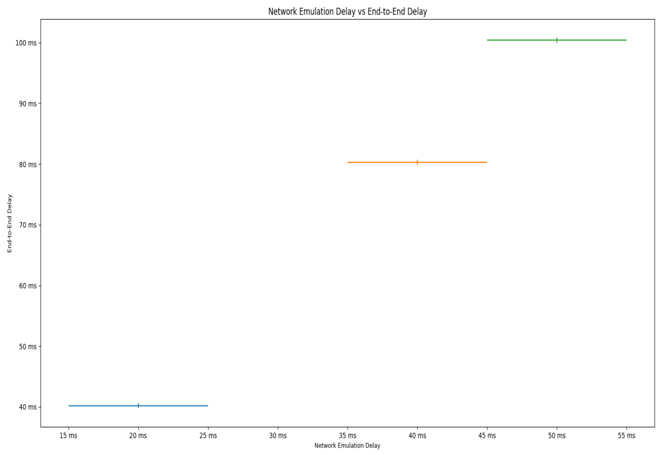


Fig. 7. Enumeration Delay vs End-to-End

Above figure is a graph of enumeration delay and end-to-end delay. When the enumeration delay is 20 ms with the error rate of 5 ms, corresponding end-to-end delay is 40.2 ms with the error rate of 0.4214 ms. When the enumeration delay is 40 ms with the error rate of 5 ms, corresponding end-to-end delay is 80.3 ms with the error rate of 0.4276 ms. When the enumeration delay is 80 ms with the error rate of 5 ms, corresponding end-to-end delay is 100.3 ms with the error rate of 0.4338 ms.

We can find the error rate with the following formula:

$$error = z^* \frac{\sigma}{\sqrt{n}} \tag{1}$$

Here, the value of $z*$ can be found with confidence interval value. $C = 95\%$ corresponds to $z^* = 1.96$

References

[1] https://stackoverflow.com/questions/1593946/what-is-af-inet-and-why-do-i-need-it
[2] https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0 /com.ibm.zos.v2r1.bpxbd00/socket.html