Ceng435 Term Project Report Part1

Bilge Eren
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
2171585, Group80

Gurkan Kisaoglu
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
2171726, Group80

***Abstract*—CENG435 is a course offered by METU that teaches undergraduate students the fundemantals of Data Communication and Networking. In this project we wanted to implement a simple User Datagram Protocol *(UDP)* in addition to one experiment. This document consist of details of our implementation and results of the experiment.**

***Index Terms*—UDP, socket, thread, python**

I. Introduction

We have implemented UDP from given topology. We simply designed a system that has 5 nodes which are *source, destination, router1, router2 and router3*. All nodes implemented as a client/server application that sends and receives multiple messages at the same time.

We used the first one of the given scenarios to calculate the link costs (*rtt* values) between each node.

Experiment result and the shortest path between s and d will be determined by *Dijkstra Algorithm* in the third and fourth sections with the help of the calculated *rtt* values in first section.

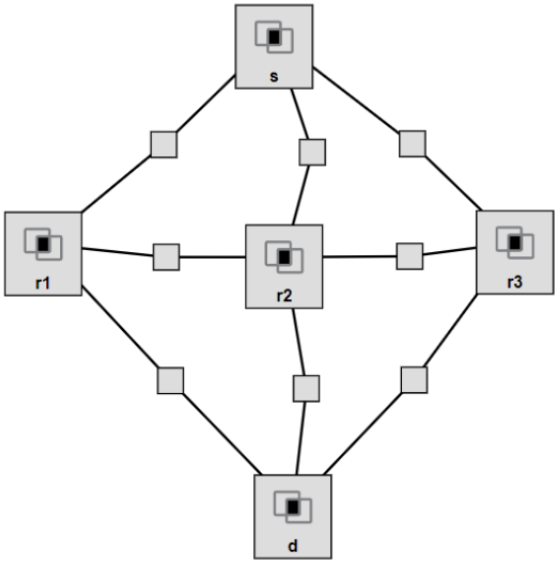

Figure1: Topology of nodes

II. Design and Implementation Approach

A. Link Cost Calculation

As we mentionen before we used first scenario to calculate link cost between nodes. Which is;

1. To be estimated by r1: s-r1 and r1-r2 and r1-d
2. To be estimated by r2: s-r2 and r2-d
3. To be estimated by r3: s-r3 and r2-r3 and r3-d

To understand the concept easier we have decided to start the implementation with only two nodes. So we choose *source* and *router1* nodes to create one *client-server* application. In the initial configuration, *router1* was sending messages to *source* and at the same time *source* was sending messages to *router1*. To accomplish this, we have created two thread in each node(*router1*, *source*), one for listening the incoming message and one for sending message over the UDP connection. We have seen that the nodes are sending messages and listening incoming messages at the same time. After this step, we have extended our implementation to 5 nodes.

To find the RTT values, we have used the *source* and *destination* nodes as listener nodes, *router1 router3* nodes as client nodes and *router2* node as both client and listener node, ie. it sends messages to *source* and *destination*, and listens incoming messages from *router1* and *router3*. For each listen and send operation to a specific ip and port, we have created threads. Each send thread is sending messages to other nodes, and each listen thread listens the specified ip/port and sends an acknowledge message to the incoming message's address.

After this step we have found the following link costs:

1. r1-s = 0.067s
2. r1-d = 0.107s
3. r1-r2 = 0.169s

4. r2-s = 0.110s
5. r2-d = 0.081s

6. r3-s = 0.000821s

7. r3-d = 0.000827s
8. r3-r2 = 0.107s

We have wrote these results into txt files on the nodes and copy to our local machine. The trimmed version of the txt files can be seen below as screenshots:

```
router2-source
0.11009311676
0.110353946686
 ...........
0.110256195068
0.110216856003
avg = 0.110142734051


router2-destination
0.0811150074005
0.0807909965515
 ............
0.0812950134277
0.0809669494629
avg = 0.0811762237549
```

Figure2: r2CostLinks.txt

```
router1-source
0.0678210258484
0.0673291683197
0.0674140453339
 ...........
0.0672681331635
0.0672149658203
avg = 0.067171933651


router1-destination
0.106920957565
0.10687494278
 ...........
0.106868028641
0.10696387291
0.107077121735
avg = 0.107126326561


router1-router2
0.168830871582
0.168668031693
0.168678998947
 ...........
0.168673992157
0.168642997742
0.168715953827
0.168873071671
avg = 0.168792519569
```

```
router3-source
0.00109720230103
0.000702142715454
 ............
0.000863075256348
0.00074291229248
avg = 0.000821840763092


router3-destination
0.000944852828979
0.00103998184204
0.000994920730591
 ............
0.000770092010498
0.00070595741272
avg = 0.000827085971832


router3-router2
0.10594701767
0.105752229691
 ............
0.106179952621
0.10573387146
avg = 0.105703525543
```

Figure3: r1CostLinks.txt r3CostLinks.txt

This result was not suprising for us. Since we have run configureR1.sh and configureR2.sh bash scripts on *router1* and *router2*, the links that contain r1 or r2 took more time to sent the packets(due to delay that comes from bash scripts).

To generate this result, we take the startTime before sending message and sent it to another node. Once an acknowledge message comes back to the sender, we have take the difference of the endTime and startTime and pushed the result into a list. That operation is for just one packet only. We sent 100 messages over each connection and get 8 lists that each contains 100 rtt value. As a last step, we have calculated the average of the rtt values on each link.

B. Dijkstra Shortest Path

We know that we have the below graph from the link cost data.

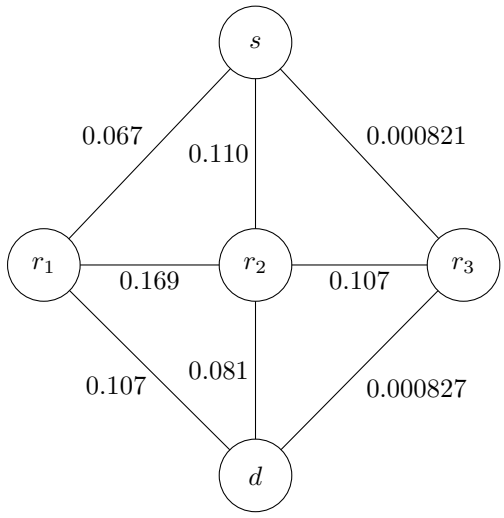

Figure4: Weighted graph of the linkings

The table given below includes *Dijkstra Algoritm* manuel implementation.

Algorithm:

If: Node is not linked with current node take the value from previous row.
Else:

If: Link cost from currentNode to target node + currentNode's cost smaller than previous row value change the value.

Else: Take the value from previous row.

We first start with node **s** and signed s as visited. In the first row we found cost of going each of the router from the selected node **s**.

After that we took r_3 since it's cost from **s** is less than others. From r_3 we calculated the costs of the all connected and not visited nodes which are r_2**, d** since r_1 is not linked with current node r_3 we took the previous rows result.

After that we took **d** since it's cost from r_3 is less than others. From **d** we calculated the costs of the all connected and not visited nodes which are r_2**,** r_1.

Lastly same process was applied for r_1.

currentNode	s	r_1	r_2	r_3	d
$initially$	∞	∞	∞	∞	∞
s	0_s	0.067_s	0.110_s	0.000821_s	∞s
r_3	0_s	0.067_s	$0,107821_{r3}$	0.000821_s	$0,009091_{r3}$
d	0_s	0.067_s	$0,09001_d$	0.000821_s	$0,009091_{r3}$
r_1	0_s	0.067_s	$0,09001_d$	0.000821_s	$0,009091_{r3}$
r_2	0_s	0.067_s	$0,09001_d$	0.000821_s	$0,009091_{r3}$

Figure4: Implementation of Dijkstra Algorithm

From the table above we found the shortest path from s to d is:

s - r_3 - d

C. Experiments

In this part we are expected to plot a figure which provides the relation of network emulation delay and end-to-end delay with a 95 percentage of confidence interval for each communication. We wanted to use the shortest path that we found in previous section which was $s - r_3 - d$. From this experiment we will provide informations about how end-end delay changes in the network.

To begin with the experiment part we first implemented *configure.sh* files for each **s, d,** r_3. These configuration files adds delay into the nodes at a desired amount (20ms, 40ms, 50ms), and we have synchronised the times of the nodes. To accomplish this, we have use NTP and NETEM.

NTP: Network Time Protocol (NTP) is a protocol used to synchronize computer clock times in a network. We used NTP for synchronize the clocks of the nodes to calculate end to end delay.

NETEM: NetEm is an enhancement of the Linux traffic control facilities that allow to add delay, packet loss, duplication and more other characteristics to packets outgoing from a selected network interface. We have used this to configure s, d, and r_3 nodes.

After that we write three different files names as *source.py, destination.py, router3.py* into as separate experiment folder. In this files we have implemented the shortest path that we mentioned above.
To be more specific in *source.py* we have a sender function that sends packets form **s** to r_3 and in *destination.py* we have a listener function that receives packets from r_3. These two nodes have only one duty sending or receiving packets. On the other hand *router3.py* have both sender and listener functionalities. r_3 takes packets from **s** and send them to the **d**.
In our first implementation, we didnt sent any **ACK** messages to the source and router3, but in this case we had packet loss. We were sending 100 packets from source but destination could receive only 80 packets. Hence, we have synchronised the messages with **ACK** messages. We changed the code so that after source send the message, it will wait until the acknowledge message came. In this way, we had successfully calculated the end-to-end delay between source and destination 100 times.

III. Methodology and Motivation

A. Methodology

We used **Python** for implementing all the phases of this project.
With the help of **time.time()** we could determine the end to end delay between nodes. We sent the timestamp from *source* node to the *router3*, and routed it to the *destination* node . When *destination* node receives the data from *router3* it calculates the time difference between current time and the timestamp data that comes from the network.
Also with pyhton **socket** library we determined the specific **port**'s and the **Ip**'s for each linking.
Last of all with **threading** we calculated the **rtt** values at the same time. In overall design Python is used for implementation of **UDP** server application.

B. Motivation

This project was helpful for us to learn and gain hands-on experience on UDP socket programming and multithreading. We have learnt different tools like ntp, netem in this project. In addition to this we have become more familiar to the routing in sender and receiver nodes.

IV. Experiment Results

We have run the scripts with 20+-5ms, 40ms+-5ms, 50ms+-5ms configurations. And sent packets from s to d 100 times.

A. Experiment 1: 20ms+-5ms

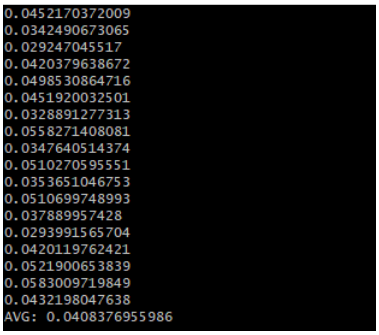

Figure5: Experiment results for 20 ms delay

In experiment 1, our mean value is 0.0408seconds for end-to-end delay.

B. Experiment 2: 40ms+-5ms

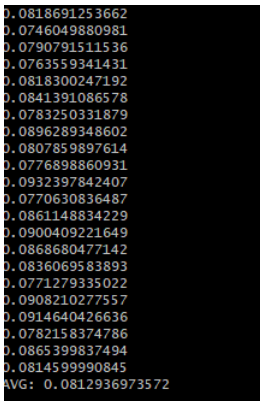

Figure5: Experiment results for 20 ms delay

In experiment 2, our mean value is 0.0812seconds for end-to-end delay.

C. Experiment 3: 50ms+-5ms

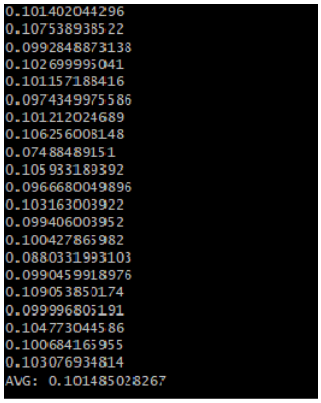

Figure5: Experiment results for 20 ms delay

In experiment 3, our mean value is 0.101seconds for end-to-end delay.

To get more accurate results we repeated our experiment 100 times for each. From the results of the three experiments given above .
To generate a confidence interval, we need to apply below formula in all configurations.
We have NumberOfExperiment = 100, and calculated StandardDeviationOfDelays in destination.py to find error.

$$e = z * \frac{StandardDeviationOfDelays}{\sqrt{NumberOfExperiment}}$$

$$e = 1.96 * \frac{StandardDeviationOfDelays}{\sqrt{NumberOfExperiment}}$$

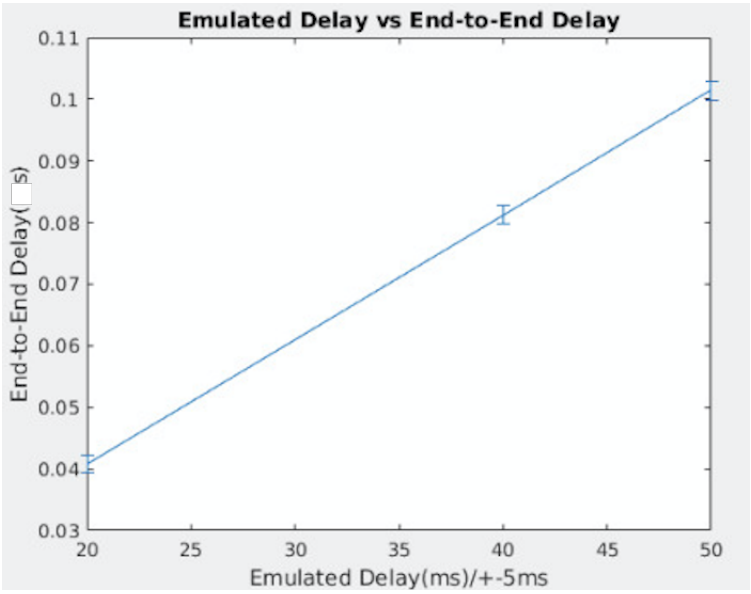

Figure6: Emulation vs End to End Delay Graph

From the error rates that we take, we construct the above chart. Which represents the confidence interval with a %95 confidence interval value. (z = 1.96)

In the shortest path we have 2 hop (one in between **s-**r_3 and one in between **d-**r_3.
As we see from the graph in every experiment emulated delay causes 2 times more end-to-end delay. This is because of we have 2 hops which are mentioned before and we apply the emulation delay in every node.