# CENG 435 - Data Communications and Networking Term Project - Part I

Sina Sehlaver
*Department of Computer Engineering*
*Middle East Technical University*
2099729

Beyazit Yalcinkaya
*Department of Computer Engineering*
*Middle East Technical University*
2172138

***Abstract*—In this paper, we report our experiment results for discovery and experimental evaluation on a given network topology and discuss the results to understand the relation between network emulation delay and end-to-end delay. First, we setup a network of computers by using a given topology. Second, we explore this topology by measuring the link costs between nodes. Third, we find the shortest path on the explored topology. Finally, we setup an experiment on the shortest path to investigate the relationship between the network emulation delay and end-to-end delay. We discuss the results and give graphs.**

## I. Introduction

In this project, we set up virtual networks and analyzed network delays by doing experiments for sending/receiving packets between the networks on GENI which is a platform that provides infrastructure for education-purpose networking research. The report consists of explanations, results and insights on the work we have done.

The main motivation in these experiments were to examine the relation between network emulation delay and end-to-end delay of two nodes in a given topology. Network emulation delay in the context of this report is defined as the delay emulated in the outgoing queue of kernels in each node. In other words, it is the delay between each consecutive packet the node will send. End-to-end delay, on the other hand, is the time required for a packet to travel from one "end" to the other "end".

The importance of the relation between network emulation delay and end-to-end delay is immense in a real-life networking scenario. The aforementioned system will have limitations or requirements on both the delay between each consecutive packet during transmission and the delay perceived on the edge-device. Hence, we need to know the network emulation delay and end-to-end delay so that we can adjust one or both of them accordingly to meet the requirements of our system.

In accordance to our motivation, we have reserved resources from GENI, discovered and configured the given topology. From the given topology, we calculated the shortest path from our sender node to the receiver node. We ran experiments on the nodes on the path that we found by changing the network emulated delay and observing its impact on the end-to-end delay. We plotted the gathered data from the mentioned experiments and analyzed the relation.

## II. Methodology

In this part of the report, we describe the design and the logical flow behind the execution of the experiments. The descriptions are in the same order as the executions. In accordance with that, we started with GENI and its configurations. In order to reserve resources, we specified a topology file to the GENI platform in XML file format. We tried to reserve servers several times since the process of reserving resources was more cumbersome than thought and it failed a couple of times. In the end, we reserved servers according to the topology description.

The servers were configured so that each one had specific IP addresses for the connection to every other server. We used those IP addresses, so to say "gateways", to connect each node to one another.

The purpose of the configuration scripts provided for $r1$ and $r2$ (configureR1.sh and configureR2.sh) is to introduce an initially random but fixed network delay on each link the nodes have between other nodes. Those configurations were given to us in order to discover link costs. For the experiment part, we needed to write our own configuration scripts so that we set custom delays for nodes $s/r3/d$ for every experiment individually. We wrote scripts for every node-experiment combination and also wrote "resetting" configuration scripts since netem delays need resetting before setting them to a new delay value. In order to create a "real-life" emulation of the queue delays, we used "normally distributed" values.

The metric for the link costs is the round-trip-time defined as the time required for the packet to leave the sender node and arrive back to the same node again. Ideally, this time should be double the time required for the packet to arrive from the sender to the receiver (end to end time). In order to discover the link costs of each link in our topology, we specified "link cost calculator" nodes $r1/r2/r3$ that:

- send 1000 consecutive packets
- calculate the mean link costs
- save them into a file

$r1$ measures links:

- $r1$-$s$
- $r1$-$d$

$r2$ measures links:

- $r2$-$r1$

- $r2$-$r3$
- $r2$-$s$
- $r2$-$d$

$r3$ measures links:

- $r3$-$s$
- $r3$-$d$

On the other hand, each node except $r2$ is designed to send back the packets they receive to their senders, making $s$ and $d$ the nodes that only do send back and $r2$ the only node sending but not receiving. Each node handles the transmission with separate threads for each connection-direction combination so that we can transmit and retrieve packets concurrently. We designed a flow logic for the initialization and finalization of the link cost calculation process. The flow is conducted by a "master" node (which is $r2$) that commands all other nodes to begin the calculation and end it when all other nodes are done with their calculations.

After we receive all the link costs, we merge and transfer them to a graph structure in order to calculate the shortest path. We use Dijkstra's Shortest Path Algorithm so that we are certain of finding the shortest path from $s$ to $d$ since the graph structure that we are creating is guaranteed to have positive edge weights. Since the graph is relatively not complex, we manually performed the algorithm on our graph. We found that the shortest path from $s$ to $d$ is through the path $s$-$r3$-$d$.

For the experiment, we designed a transmission logic so that we can measure the end-to-end delay as accurately as possible. Since we discovered the shortest path in discover part, it is redundant to implement a routing logic; hence, we manually arranged the routing.

- $s$ sends packets
- $r3$ transits the packets to $d$ and $d$ sends back packets to $s$ through $r3$ immediately as it receives them
- $s$ sends and wait for the packets sequentially
- $r3$ handles the transmission with separate threads for each direction and $d$ sequentially receives and send back packets
- $s$ saves a timestamp right before sending the packet
- $d$ saves a timestamp right after receiving the packet
- $d$ sends this timestamp as the packet to $s$
- $s$ calculates the difference between them and saves this value as the end-to-end delay

This explained process is the main element of the experiment and in order to have a confident result for the "real" end-to-day, the mentioned process is executed for $1000$ consecutive packets. However, there is the problem of initial timestamp difference of $s$ and $d$ for the same moment in time. To overcome this problem, we use the NTP (Network Time Protocol) to synchronize both nodes. $s$ commands $d$ to synchronize over NTP by sending a packet and waiting until $d$ is done with synchronizing. This synchronization is repeated in every $1000$ packets. Just like the discovery part, $s$ notifies the end of the experiment to $r3$ and $d$ by sending a termination packet.

## III. Implementation

In this part of the report, we explain what we have mentioned in methodology part for the scripts and their functionality in detail. The project is implemented with Python3 and Bash scripts. Each Python3 script "main.py" is run with a corresponding Bash script "start.sh" within the node.

We used several modules in the project:

- socket module for the transmission
- threading module for concurrency
- time module for timestamps
- subprocess module for executing NTP synchronization commands
- numpy module to efficiently observe data
- scipy module to calculate margin of error
- sys module to get input arguments
- ntpdate command to synchronize clocks of nodes
- tc/netem command to emulate network delays

We have created a "README.md" file that explains the project structure and the procedures to recreate the experiments. There are also comments in each script that explain the logic and implementation details.

For both discovery and experiment parts, we needed to kill the threads after the script that created them was finished with its execution. In each script we set the "daemon" variable provided by the thread objects to true so that when the script that created the threads was finished, the operating system would kill all the threads with "daemon" variable set to true. In order to let each script know when to stop their execution we specified a "master" node, made sure that the master node is always ran after all "non-master" nodes and all non-master nodes send an "I am done" packet to the master before they wait for a "finish" command from the master. While the normal packets are sent with port $8080$ the mentioned termination packets are sent with termination sockets bind to port $7070$ for clarity.

For the connection between each node, we used the socket module in Python. We specified the address type to be IPv4 and data type to be datagrams in our socket initializations. For each connection-direction pair that we had in the topology, we have created a socket and used "bind" function to make it listen to a specific port and a specific IP address. For sending packets, we used "sendto" function which accepts an IP address and a port as the input arguments. All of the mentioned connection function calls were made using "hard-coded" IP addresses and port bindings with the node information provided by GENI within each node.

The configuration scripts "configureR1.sh" and "configureR2.sh" execute the "tc" command together with "netem" command to set run-time-calculated fix queueing delay for each neighbor connection of $r1$ and $r2$. The configuration scripts "configExp*.sh" set the delays for the experiment phase connections of $s$, $r3$ and $d$. The difference is that this time we generate delays actively changing with time according to the normal distribution around a mean value with a specified fluctuation rate. We specify the mean value, fluctuation rate

TABLE I: Initial

| Node | Distance from s | Previous node |
|---|---|---|
| s | 0 | None |
| r1 | $\infty$ | None |
| r2 | $\infty$ | None |
| r3 | $\infty$ | None |
| d | $\infty$ | None |

TABLE II: After First Iteration

| Node | Distance from s | Previous node |
|---|---|---|
| s | 0 | None |
| r1 | 104.81 | s |
| r2 | 108.81 | s |
| r3 | 0.50 | s |
| d | $\infty$ | None |

TABLE III: After Second Iteration

| Node | Distance from s | Previous node |
|---|---|---|
| s | 0 | None |
| r1 | 104.81 | s |
| r2 | 104.31 | r3 |
| r3 | 0.50 | s |
| d | 1.00 | r3 |

TABLE IV: After Third Iteration

| Node | Distance from s | Previous node |
|---|---|---|
| s | 0 | None |
| r1 | 80.81 | d |
| r2 | 88.81 | d |
| r3 | 0.50 | s |
| d | 1.00 | r3 |

and distribution type as arguments of the "netem" command. Since the delay is different for each experiment, we also have "resetConfigure*.sh" scripts that uses "tc" command's "del" argument to reset the previous delay configuration from the network traffic.

The discovery scripts are implemented with server-client architecture. Each node has a different role in the flow as mentioned in the methodology part. The only common point is that, every node has to have at least one server thread. $r1/r2/r3$ are the nodes that do the calculations; hence, they save the calculated values into a file named "link_cost.txt".

$s$ and $d$ have identical scripts because they have the same role in the discovery part. These nodes only have the responsibility to immediately send back the packet they receive. They create server threads for their neighbour connections so that each server thread handles the packets coming from that specific connection, enabling the node to handle as many packets as its connection concurrently. Each thread I/O blocked for a packet in an infinite loop.

$r1$ and $r3$ have identical scripts because they also have the role of sending packets and calculating link costs in the discovery part. They create client threads for each of their neighbours. In order to start sending packets from clients of the nodes, we need to be sure that the servers of the nodes are ready to receive. Thus, the "non-master" nodes that have client threads namely $r1/r3$, have a locking mechanism in their main thread so that they can listen to the master node. The mechanism is as follows:

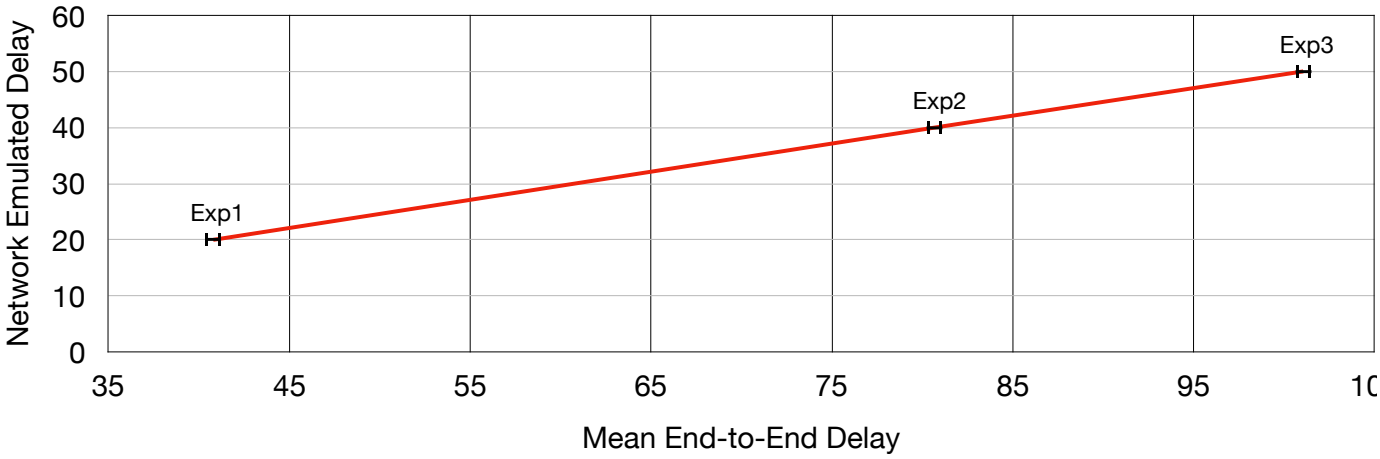
- main thread holds the lock until the server thread is initialized
- when the server thread is started main thread gives the lock to the server thread and begins to wait for the lock
- meanwhile the server thread is waiting for just one packet that it needs from the master
- when that packet is received the server thread releases the lock
- main thread having the lock releases it and starts its client thread

$r2$ is the master node and it creates client threads for each of its neighbours. Each client thread of $r2$ creates a socket for its specific connection and sends a dummy packet to its receiver so that the locking mechanism receives the "go" signal. Before any other action the master node starts its server threads so that when it sends the starting signal, its own servers are ready to receive. The big difference between the server threads of $r2$ and other nodes is that $r2$ creates its server threads only to listen for the "end-of-measurement" packets. The master node has the responsibility to wait for all other nodes in the topology. It can not simply send termination signals to every other node when its own calculations are done since other nodes may still be serving and doing calculations. Therefore, it waits for its server threads to finish and only after that it sends the termination packets to every other node.

The experiment scripts are different in the sense of initialization and termination since $s$ is an active node while $r3$ and $d$ are reactive nodes. In other words, when $s$ is done the experiment is done. Therefore, we did not need any additional mechanism other than just one termination packet sent from $s$. The only problem the experiment has is the synchronization of node clocks as mentioned in the methodology part. In accordance with our design, we have used "ntpdate" command called by the subprocess call in both nodes to update and synchronize their clocks.

For $r3$, the implementation is trivial since its only responsibility is to transit the packets it receives to their hard-coded destination. It has two server threads; one for $s$ to $d$ and one for $d$ to $s$ so that it can concurrently handle the transitions. After the creation of its two server threads it waits for the

Fig. 1: Mean End-to-End Delay vs Network Emulated Delay




termination packet from $s$ so that when it receives, it can also inform $d$ and finish its execution.

The implementation of $s$ and $d$ are intertwined. $s$ has one client thread that:

- for every 1000 packets $s$ sends it updates its clock
- sends command to $d$ saying that $d$ also should update its clock
- waits for $d$ to tell that it updated and then moves on with the experiment

Meanwhile, $d$ has one server thread which waits for a packet and updates its clock if the packet is a "synchronization packet". As for the end-to-end delay calculation part, according to the design:

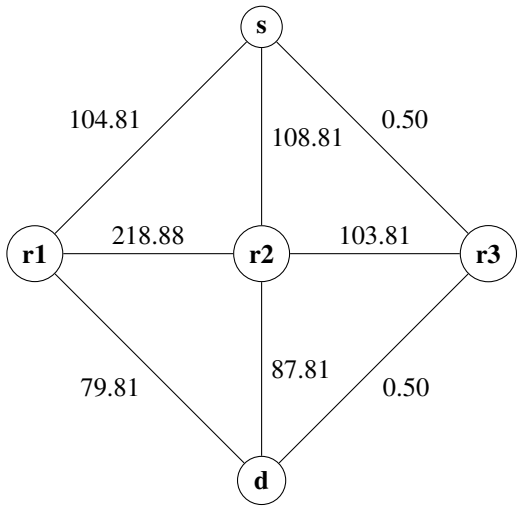
- $s$ saves the timestamp just before it sends a packet
- $d$ saves the timestamp right after it receives the packet
- $d$ encodes the timestamp as a string and sends the packet back to $s$
- $s$ receives the timestamp and with the assumption of being synchronized saves the difference between two timestamps as the end-to-end delay for each packet

When the client thread of $s$ is finished, the main thread writes out the data to a text file named "exp*.txt" and sends out the termination packet.

After the experiment is done, there is a "merge_and_calculate.py" script that calculates; the mean, the margin of error, lower and upper bounds according to the confidence interval we hard-coded. Then, we save these results to a "exps.csv" file and manually transfer them into the "Numbers" app on macOs. The aim of using numbers was to be able to adequately show the margin of error in a way that is visually sufficient for the report since they were very small.

## IV. Discovery Results

Fig. 2: Topology with Link Costs




The purpose of discovery part was to examine the topology and calculate the link costs in order to find the shortest path. We have gathered the link costs into a graph structure and manually followed the Dijkstra's Shortest Path Algorithm for the shortest path (Fig. 2).

Initially, (Table I) we see that the distance to every other node is infinite and we are at the node $s$. After discovering the link costs of neighbours of $s$, we update the distances of $r1, r2, r3$ (Table II). It is clear that we have to choose node $r3$ to continue. After discovering the link costs of neighbors of $r3$, we realize that there is a path shorter than $s$-$r2$ which is $s$-$r3$-$r2$, hence we update the distance to $r2$ and $d$ (Table III). After second iteration, when we pick $d$ we realize that we have reached the destination and save the necessary info. Due to termination condition of the algorithm, we process left nodes in the queue and terminate the algorithm (Table IV). Now we are aware that the shortest path from $s$ to $d$ is through the path $s$-$r3$-$d$.

Because of the fact that we executed configuration scripts that add emulated delays on $r1$ and $r2$, it is clear that $r3$ (the node that has no emulated delay) would be on the shortest path.

The valuable information that we were after was the shortest path of the topology. In a real-life scenario, we would always prefer sending our data through the shortest path. In order to conduct the experiment we needed the path we have found so that we can route the nodes accordingly. In this part of the project we have experienced how to analyze one of the essential properties of a network system which is the round-trip-time.

## V. Experimental Results

The aim of this experiment was to understand the relation between the network emulated delay and the end-to-end delay. We conducted $1000$ iterations for each experiment and gathered the data into a bar graph where we show the relation between the mentioned two delays (Fig. 1). We calculated the margin of error with $95\%$ confidence interval and $1000$ iterations for each experiment. Our margin of error:

- for experiment 1 $\longrightarrow 0.4349$ ms
- for experiment 2 $\longrightarrow 0.4471$ ms
- for experiment 3 $\longrightarrow 0.4444$ ms

We observed that our results pointed a direct relation between the delays. With further analysis, we noticed that the reason was the fact that the propagation delay between the links on our path was in the order of our margin of error. In other words, the impact of the propagation delay was so little that we were observing a direct relation. Thus, the data expressed only the impact of "queuing" delay for our experiments. Combining the information we gathered we agreed that the relation is in fact

$$d_{e2e} = \sum_{l \in L} d^l_{prop} + \sum_{n \in N-1} d^n_{netem}, \text{ where}$$

- $d_{e2e}$ is the end-to-end delay
- $L$ is all the links on the path
- $d^l_{prop}$ is the propagation delay on the link $l$
- $N-1$ is all the nodes on the path except the receiver node
- $d^n_{netem}$ is the emulated delay in the node $n$

For this topology what we observe was that we almost eliminated the $d^l_{prop}$ and then left with a direct relation between $d^n_{netem}$ and $d_{e2e}$.

## VI. Conclusion

In this project, we examined the relationship between delays. We gained experience with GENI platform, discovered the properties of network topologies and extracted information from the data we got from our experiments. In accordance with our main goal, we grasped the fact that the relation between the network emulated delay and the end-to-end delay is linear.