CENG435 Term Project: Part One

End-to-end delay calculation with network emulation delays

Narmin Aliyeva Berk Ozbalci

I. Introduction

The experiment consists of two parts. In the first part, we measured the RTT[1] between each neighboring node in a network. We determined the link costs of each link using the RTT measurements, and found the shortest path between the source node and the destination node using Dijkstra's algorithm. In the second part, we applied different levels of delays on these nodes for three separate experiments, and measured the end-to-end delay between the source node and destination node by comparing timestamps. Time synchronization was accomplished using NTP (RFC 5905).

II. Design and Implementation

We created two sets of scripts, one for network discovery and one for running the end-to-end delay experiment. Both are written in Python 3.6.5, which were the version running on the GENI-allocated machines at the time of starting the assignment. In addition, both scripts are heavily supported by shell scripts in order to facilitate setting up nodes, links and retrieving the experiment results.

A. Discovery Scripts

The discovery script creates a *messaging pair* at each node, for each of its neighboring nodes. The messaging pair is a pair of UDP server and UDP client, and each of them run in a separate thread. This means that for a node having three neighbors, it will have six threads, three of them which are UDP servers, and the remaining UDP clients. This allows the nodes to send and receive multiple messages concurrently, thereby inducing simultaneous messaging between all neighboring pairs.

The messaging protocol is very straightforward: each message contains a *sender ID*, which is a unique identifier of the node that the message originates from; and an *index*, which is the sequence index at which the UDP client has sent the message.

The sender ID is set by the UDP client upon generation of a message and used by the UDP server to decide whether an incoming message originates from the client of the same messaging pair, i.e. if the sender ID is same as the current node's sender ID, then it's a round-trip and we must calculate RTT based on this message; otherwise, it originates from another node and therefore the server echoes it back (without making any changes to the message content).

When the server has to echo these messages to the neighboring node, it does not communicate back to the client, instead it sends them to the server of the same messaging pair at that node. In this way, we ensure that the client is able to generate and send messages back-to-back, without having to wait for the previously sent message to return from the server.

In our scheme, every node constructs a messaging pair with all of its neighbors. Therefore, a total of 16 link costs have been measured. For each measurement, a total of 1,000 messages were sent, and the results were averaged.

B. Experiment Scripts

The experiment script takes a routing information file as input, which is a simple linked list of nodes that the end-to-end delay experiment will be conducted over. For example,

[1] round-trip time

```
["s", "r1", "r2", "r3"]
```

will set up sockets at each node in the following fashion:

- **s** will send messages to **r1**
- **r1** echoes all messages to **r2**
- **r2** echoes all messages to **r3**
- **r3** calculates the end-to-end delay.

When the experiment script is run with the above routing input at node **d**, it will find that its name does not appear on the routing list, and therefore exit immediately.

The experiment scripts use a different message structure from the measurement scripts. The only content is the UNIX timestamp with millisecond precision, represented as a floating point number. When this message is received at the final node, the receiver node obtains the current timestamp and computes the difference between this timestamp and the received message's timestamp. This difference is the *end-to-end delay*.

For this scheme to work, both the initial node and the final node must be time-synchronized. We accomplished this by using an external NTP server:

```
# ntpdate 0.tr.pool.ntp.org
```

C. Implementation Details Common to Both Parts

- Network emulation delay is achieved by using the **tc** commands.
- All nodes receive the network topology in an adjacency list format, and compute their neighbors dynamically.
- In the discovery messages, both **sender_id** and **index** are represented as integers.
- In the experiment messages, **timestamp** is IEEE 754 double-precision floating number.
- For both measurement and experimentation, we picked a base port (10000) and applied the following convention:
  - Each server host listens to packages from a client host on port $B + C$, where B is the base port, and C is the sender ID of the client host.

Note: We kept the routing information stored in the input file simple, since we are only interested in a single path from **s** to **d** in the first part of this project. However, if multiple nodes had to decide which route to take on message arrival, there would be a need for multiple files containing routing tables for each node. Additionally, we would have to include source and destination addresses as parts of a message.

D. Miscellenea

This section contains more specific implementation details on various parts of the project.

1) The Nature of UDP, and Ensuring the Right Number of Messages: Since UDP does not guarantee that all packets that are sent will arrive at the destination host, we didn't use a simple counter that attempts to send a message exactly N times, where N is the size of the experiment currently running.

In the discovery part, each messaging pair maintains the number of packets that have successfully returned from a round-trip. The clients keep on sending new messages until this number reaches N, the experiment size. Since both the server and the client threads are reading and writing to this variable, we used mutexes to prevent race conditions and ensure that no lack or excess of packets occur while calculating the average RTT.

In the experiment part, we apply a similar logic to the final node, which causes all the packets sent from the penultimate node to be dropped, thereby restricting the data set to the exact count of samples requested.

2) End-to-End Delay vs. Round-Trip Time: Our computation of the end-to-end delay and round-trip time differs in their implementation detail.

RTTs are computed by taking the difference of the timestamps at each packet event (being sent or received) at the same node. That is, if node A sends a packet to node B at time t_0, and node B sends back the same message to node A at time t_1, the round-trip time will be $t_1 - t_0$.

But end-to-end delays are computed by taking the difference between timestamps at different nodes. This means that, if node A sends a packet at its local time t_0, and node B receives that packet at its local time t_1', the end-to-end delay will be measured as

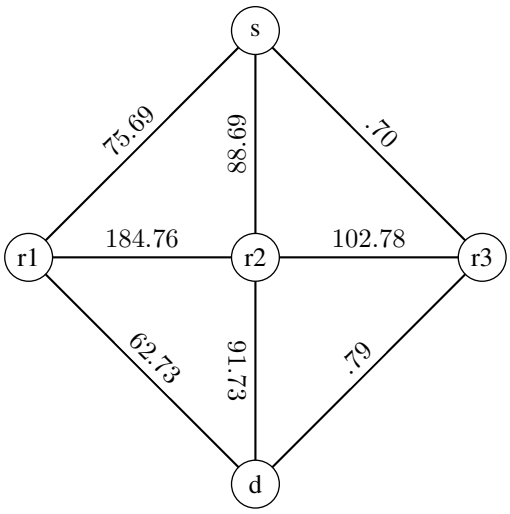

Fig. 1: Network topology and link costs (ms)

$$t_1' - t_0 + \delta_s$$

where δ_s is the synchronization offset, which is the time difference between nodes A and B. To produce a more accurate end-to-end delay measurement, δ_s must be as small as possible. We used NTP as a synchronization protocol to reduce the time difference between nodes.

III. Network Topology and RTTs

The measurement script was ran on all nodes, but we have obtained the RTTs shown in Fig. 1 from the inner nodes only (**r1**, **r2** and **r3**).

IV. Determining the Shortest Path using Dijkstra's Algorithm

For the purpose of the experiment to be carried out in the next step, we were only interested in the shortest path between **s** and **d**. However, for the sake of completeness, we constructed the table for finding the shortest path from **s** to any other node in the network. The costs are expressed as RTTs in milliseconds.

First, we start by exploring node **s**. It has three direct neighbors, and we add the costs of those links to the first row. Since there is no direct link to **d**, the weight is set to infinity for this node. We then proceed to the next step by choosing to follow the link with the lowest cost, arriving at **r3**. For any node reachable from **r3** that is not marked as explored (boxed), we calculate the cost of reaching that node through **r3**, by adding the cost of arriving at **r3** from **s** to the cost of the link with that node. If this sum is smaller than the minimum cost that is already found (if it exists) for that node, we take that step, updating the node's minimum cost and subscript. Otherwise, we keep the previous result as is.

node	s	$r1$	$r2$	$r3$	d
s	$\boxed{0_s}$	75.69_s	88.69_s	$.70_s$	∞
$r3$		75.69_s	88.69_s	$\boxed{.70_s}$	1.49_{r3}
d		64.23_d	88.69_s		$\boxed{1.49_{r3}}$
$r1$		$\boxed{64.23_d}$	88.69_s		
$r2$			$\boxed{88.69_s}$		

The subscripts at each table cell indicate the node that was used in order to reach at the number shown in the table cell. That is, $.70_s$ indicates that the node in question can be reached from **s**, and the total cost of doing so is 0.70.

To find the shortest path for a node using the constructed table, we trace the subscripts starting from that node up to node **s**. The shortest path between **s** and **d** is then found to be

[" s ", " r3 ", " d "]

We believe that this is a natural result, since there is no preconfigured network emulation delay on the links from **s** to **r3**, and **r3** to **d**.[2]

V. The Effect of Network Emulation Delay on End-to-End Delay

We conducted three experiments with varying levels of network emulation delay.

- **Experiment I.** Delay of 20ms $\pm$ 5ms
- **Experiment II.** Delay of 40ms $\pm$ 5ms
- **Experiment III.** Delay of 50ms $\pm$ 5ms

[2]Unlike the nodes **r1** and **r2**, which have network emulation delays associated with all of their outgoing links, which are defined in the assignment bundled scripts, **configureR1.sh**, **configureR2.sh**.

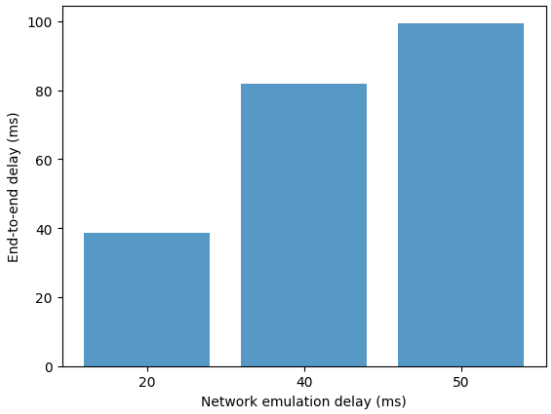

Fig. 2: Network emulation delay vs. end-to-end delay

All of the above experiments also use normal distribution to vary the amount of delay impacted to the adapters.

To obtain a more accurate result, we've conducted the end-to-end delay transmission for 100 messages at each experiment. In order to construct a 95% confidence interval, we needed at least 30 samples, but since we've automated most of the script fairly well, we confidently increased the test size.

The experimental data is provided in the below table. CI denotes "Confidence Interval".

	Exp. I	Exp. II	Exp. III
Mean	38.74	81.95	99.45
CI min	37.44	80.57	97.95
CI max	40.02	83.31	100.95

TABLE I: End-to-end delay mean and confidence intervals

Initially we expected the mean end-to-end delay to be approximately twice of the network emulation delay applied to each node, because there are only two links in the path and they both have the same amount of delay associated with them.

Upon running the experiments, we have met our initial expectations with very close results.