ACN - Lab2 Report

Midas Amersfoort (2711832), Neeraj Sathyan (2660192)

Group 26

November 2020

1 Generate fat-tree and jellyfish topology

1.1 The jellyfish topology

Here, we describe our general approach for constructing the jellyfish topology.

1. Generate two lists containing objects of class Node. The first list is reserved for servers (length = num_servers, type = 'server'). And the second list is reserved for switches (length = num_switches, type = 'switch').
2. Random uniformly pick two switches and connect them with an Edge if they both have available ports and they are not neighbours (or the same switch). Keep doing this until no more switches can be connected
3. Connect switches that have two or more free ports with two random uniformly selected switches. This is done by disconnecting a randomly uniformly chosen edge from two random uniformly chosen switches, and laying an Edge between the switch with two or more free ports and the two random uniformly chosen other switches. Keep doing this until there are no more switches with two or more ports.
4. Connect the remaining empty ports of switches (which is max. 1 per switch) to random uniformly selected switches from the list with all the switches.

1.2 The fat-tree topology

Here, we describe our general approach for constructing the Fat-Tree topology.

1. There are four separate server/switch configurations, such as core switches, aggregate switches, edge switches and servers.
2. Based on the number of ports, the size of each configuration is limited.
3. $Pods = ports$

4. Servers per switch = $ports/2$
5. Core Switches = $(ports/2)^2$
6. Aggregate Switches = $(ports/2) * ports$
7. Edge Switches = $(ports/2) * ports$
8. Servers = $(ports^3)/4$
9. Create the nodes.
10. Create edges between servers and edge switches where each switch holds $ports/2$ servers.
11. Create edges between edge switches and aggregate switches within a pod.
12. Create edges between aggregate switches and core switches, where each aggregate switch is connected to $port/2$ core switches within a pod, in an incremental manner.

2 Compare the properties of fat-tree and jellyfish

2.1 Reproduce Figure 1c

2.1.1 Approach

1. Given two lists of servers and switches, returned by either the jellyfish or fat-tree topology, we run Dijkstra's shortest path algorithm to compute the path lengths between a server all server pairs.
2. As we need to compute the path lengths for all server pairs, and pairs work in two ways, we need to compute the path lengths of *'round(len(servers))'* starting point servers to all other servers (excluding the servers that were already used as a starting point).
3. Thus, for each of these starting point servers, we save the path length between the starting point server and all other servers (note the exceptions above, as these pairs are already accounted for), using the greedy path finding approach of Dijkstra's algorithm.
4. Then, if we have the computed the path length between all possible server pairs using the methodology above, we count the occurrences of each path length and the total number of server pairs found.
5. (Only for jellyfish topology) If we have multiple iterations of the jellyfish topology, we repeat from step 2 for each newly generated jellyfish topology and add to the count of occurrences of each path length and the total number of server pairs found.

6. Lastly, we want each path length's occurrence to be expressed as a fraction of server pairs. Thus, we divide the number of occurrences by the total number of server pairs found and plot the result.

2.1.2 Test Figure 1c

To test whether our implementation of the jellyfish and fat tree topology are correct, one can test whether they can faithfully reproduce Figure 1c. To run this test for the default configuration (686 servers, 245 switches and 14 ports), just run the python script in your terminal as described below:

```
$ python reproduce_1c.py
```

Running this script on our own produced Fig. 2.1.4. However, do note that, as the jellyfish topology is a random one, your mileage in reproducing the figure may vary.

2.1.3 Additional testing

Besides reproducing Figure 1c for the default configuration of 686 servers, 245 switches and 14 ports, we also let you alter the configuration manually. Likewise, we also give you the option to adjust the number of iterations (-i) the jellyfish topology's result will be averaged over. As an example, emulating the default configuration in command line translates to:

```
$ python reproduce_1c.py 686 245 14 -i 10
```

Additionally, we also give you the option to test the topologies individually. Testing 'Jellyfish' topology for the default configuration can be done as follows:

```
$ python reproduce_1c.py jellyfish 686 245 14 -i 10
```

To enable more in depth debugging, we also implemented several optional reports, which can displays information about the network in the terminal. These optional reports are: a network report (-nr), a Dijkstra shortest path report (-dr), and a full report (-fr) which displays both. Summoning the reports is done in the following manner:

```
$ python reproduce_1c.py jellyfish 686 245 14 -nr -i 10

$ python reproduce_1c.py jellyfish 686 245 14 -dr -i 10

$ python reproduce_1c.py jellyfish 686 245 14 -fr -i 10
```

In order to run the 'Fat tree' topology with another configuration, one can edit the default of configuration of 14 ports in the command line like so:

```
$ python reproduce_1c.py fattree 14
```

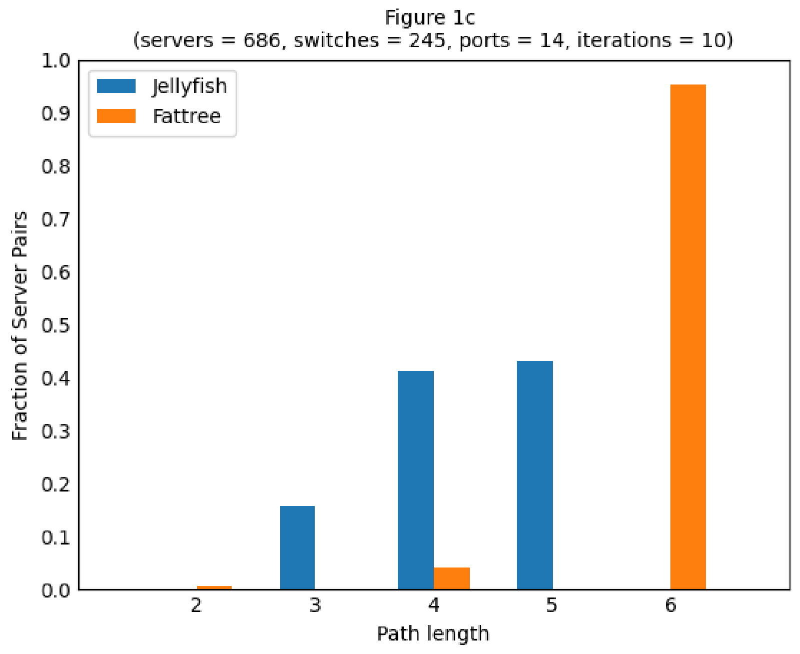
Also, as with the 'Jellyfish' topology, the same optional reports are also implemented for the 'Fat tree' topology:

```
$ python reproduce_1c.py fattree 14 -nr
```

```
$ python reproduce_1c.py fattree 14 -dr
```

```
$ python reproduce_1c.py fattree 14 -fr
```

2.1.4 Result



2.1.5 Comparison with Figure 1c in the paper

The result shown above closely resembles Figure 1c in the paper. However, as noted before, the jellyfish topology is constructed in a random procedural way, meaning that its topology is bound to be different every time time we construct one. This, as its links are prone to variation due to their randomness.

2.2 Reproduce Figure 9

2.2.1 Approach

Yen's algorithm is used to find the loopless k-shortest path between combinations of random server pairs for the jellyfish topology. The following are calculated based on program arguements, 8-shortes path, 8-shortest ECMP and 64-shortest ECMP. The number of distinct paths which for each link is calculated and shown in the figure.

2.2.2 Usage

```
$ python reproduce_9.py 686 858 14
```

where the first argument is the number of servers, second argument is number of switches and third is number of ports.

2.2.3 Result

...

2.2.4 Comparison with Figure 9 in the paper

...