Advanced Computer Networks Report

Lab2: Data Center Network Topologies

1. Team members (Group 10)

- Corneliu Soficu [2675454]
- Andrei-Ioan Bolos [2701860]

2. Generation of fat-tree and jellyfish topologies

- *How code should be run?*
  First of all, make sure you create a Python Virtual Environment while being in the lab2 folder using: **python3 -m venv venv.** After this you should activate the environment using: **source venv/bin/activate**
  Secondly, you will need to install all the required dependencies using:
  pip install -r requirements.txt
  After all of these steps you will be able to run all the code. To generate the topologies, you should run the following commands:

  >***python fat_tree.py***

  >***python jellyfish.py***

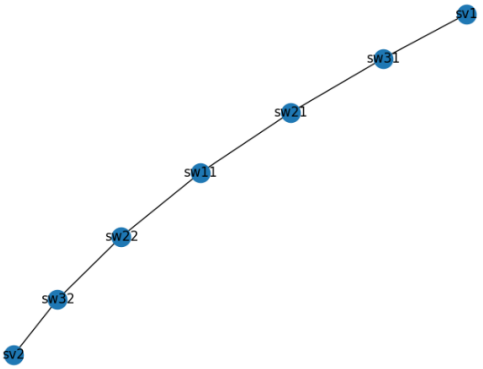
  By running the above commands, a fat-tree topology with 8 switch ports and a jellyfish topology with 80 servers, 20 switches and 8 ports are being created, the results also being plotted on the screen. Other small-scale and large-scale topologies can be tested by changing the invocation parameters of the class in each script.

- *What is the general methodology used to generate the topologies?*
  [Fat-tree] We created the class Fattree, which builds the desired topology relying on nodes and edges. Based on the number of switch ports, we were able to compute the number of each switch layer (core, aggregation, edge), the density, pod number and server count. Separate lists for each type of switch and for servers were created, and they were populated using addSwitch () method and the structure Node defined in topo.py script. In order to link the layers, we defined a method called generateLinks () and

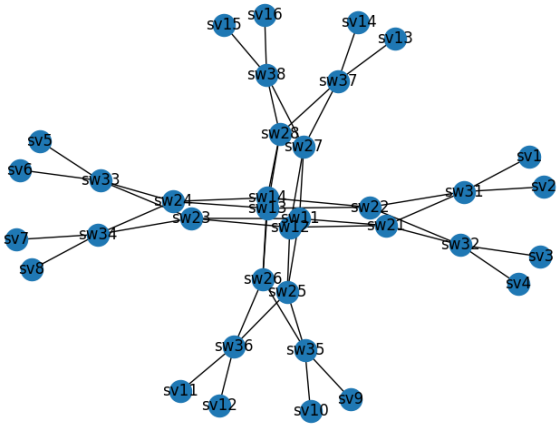
we used the Edge structure defined in the topo.py script, following closely the fat-tree paper. Bandwidth is also added for each edge and gets lower the more it goes deeper into the topology. The members and the links are displayed in the console using the functions printTopo () and printLinks (). Moreover, the plot is being displayed on the screen after calling the function draw_topology (), defined in the topo.py script.

[JellyFish] To create the Jellyfish topology, we are using the same underlining topology classes as for the FatTree: Node and Edge. When generating the topology, we use a seed value besides the other topology parameters in order to allow for a randomly generated graph. The first step of generating the topology is to generate the list of switches by instantiating a list of Nodes that are of type: switch. After this we use our own function to generate a random regular graph that have as nodes all the switches that we previously created. After we created the random regular graph that has a node degree equal to half of the total number of ports, we proceed to connect the servers to the switches. When connecting the servers to switches, we iterate through the entire list of switches, trying to find the switches that have open ports and that don't have any other server connected to it. In case there are no more switches that don't have any servers connected to them, we proceed to connect the remaining servers to these switches. We do this in order to make sure we spread out all the servers to all the switches and to not have situations where all servers are connected to a few switches and the other switches remain empty of servers.

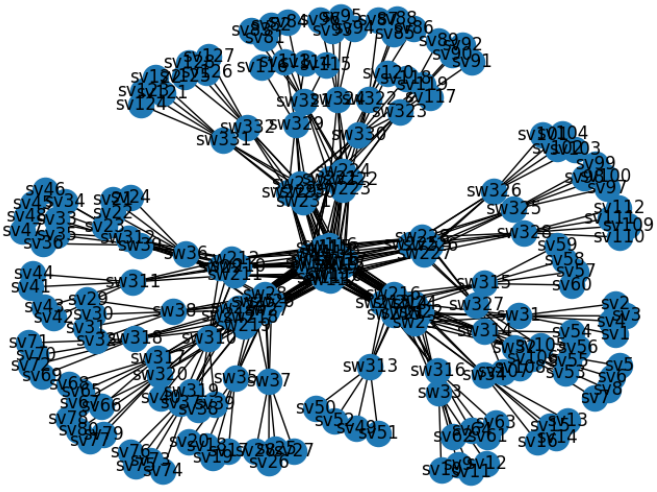
- *Some small-scale example topologies*
  [Fat-tree] Using matplotlib, we were able to display some small-scale example topologies:
    - 2-port switch Fat-tree:



- 4-port switch Fat-tree:

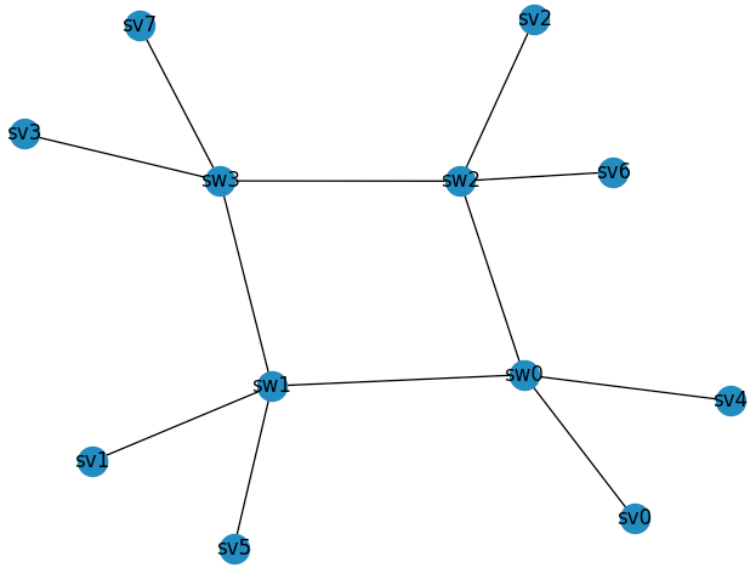


- 8-port switch Fat-tree:

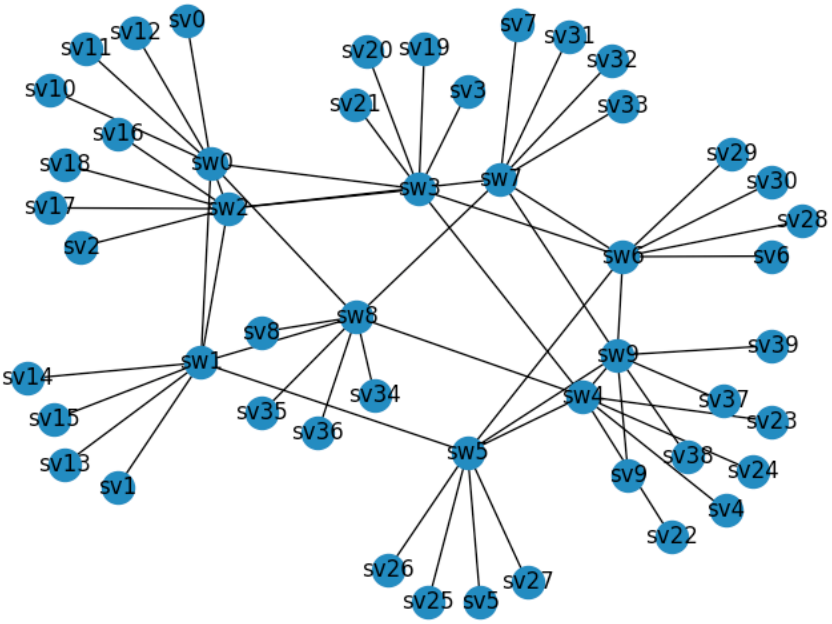


[Jellyfish] Using matplotlib we were able to generate the following Jellyfish small-scale topologies:

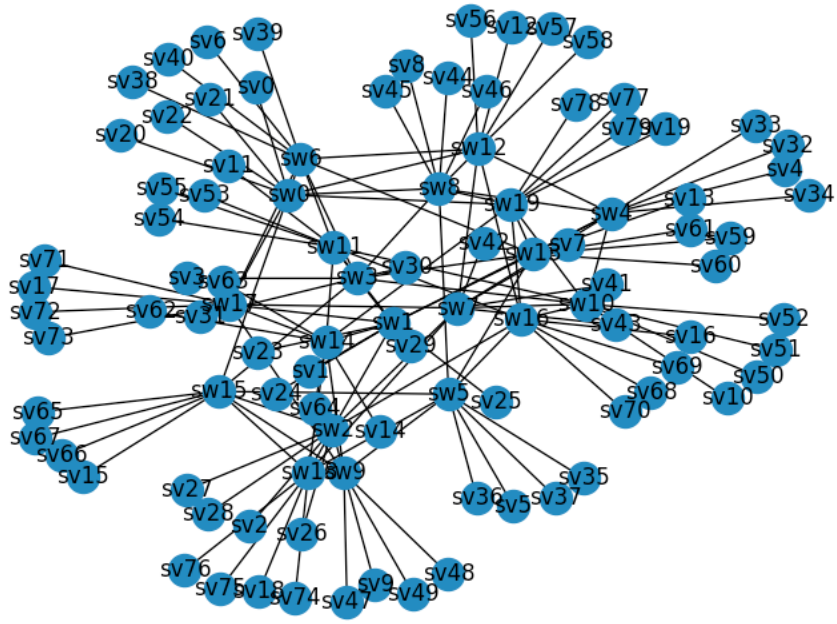
- 8 servers, 4 switches and 4 ports Jellyfish topology:



- 40 servers, 10 switches, 8 ports Jellyfish topology:



- 80 servers, 20 switches, 10 ports Jellyfish topology:



3. Reproduction of Figure 1c

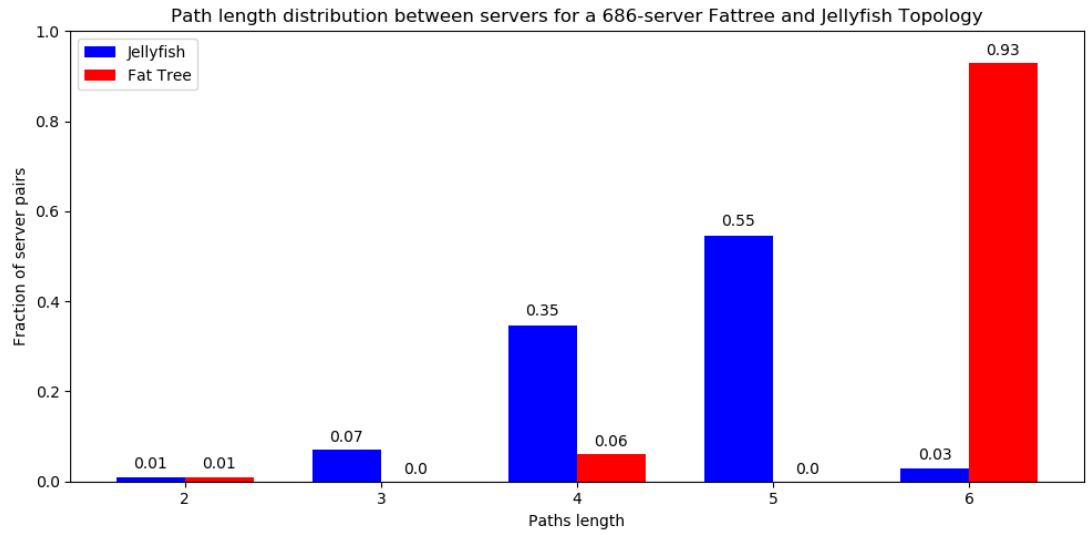
- *How code should be run?*
  In order to plot the representation for each topology, the following command should be run:
  >***python reproduce_1c.py --servers 686 --switches 245 --ports 14 --repetitions 10***

- *What is the general methodology used to reproduce the figure?*
  [Fat-tree] In order to reproduce the figure, we build several functions in the reproduce_1c.py script. Firstly, a fat-tree topology is created using 14-port switches. Then, a dictionary is created, where the key represents the node and the value represents the list of its neighbors. After computing the dictionary, the BFS algorithm is used on it in order to determine the minimum length for each path between each pair of any two servers. The fraction of server pairs with certain lengths are counted and displayed in the console and also plotted on the screen, using plot_results () function.

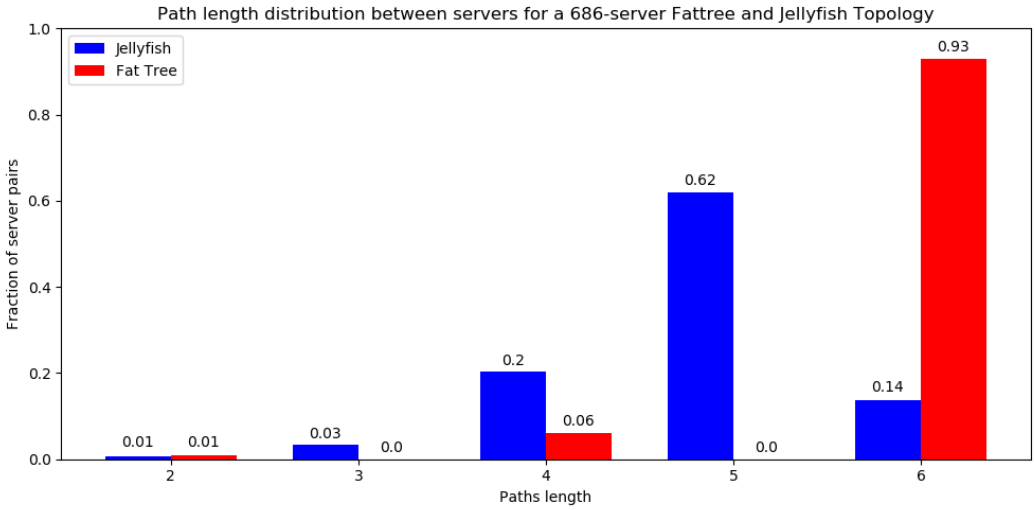
  [Jellyfish] In order to reproduce the figure for Jellyfish, we run through almost the same steps as for the Fat-Tree case. We generate a topology for Jellyfish, we convert the topology to a dictionary

format and then we run BFS algorithm for all pairs of servers. Because we have to run multiple repetitions for the Jellyfish situation, we used multiprocessing package to generate 10 parallel processes that execute the measurements. In the end we plot the results for Jellyfish together with the Fat Tree results.

- *Reproduced figure using a configuration of 686 servers, 98 switches and 14 ports for Jellyfish and 14 port switches for Fat-tree:*



- *Reproduced figure using a configuration of 686 servers, 245 switches and 14 ports for Jellyfish and 14 port switches for Fat-tree:*



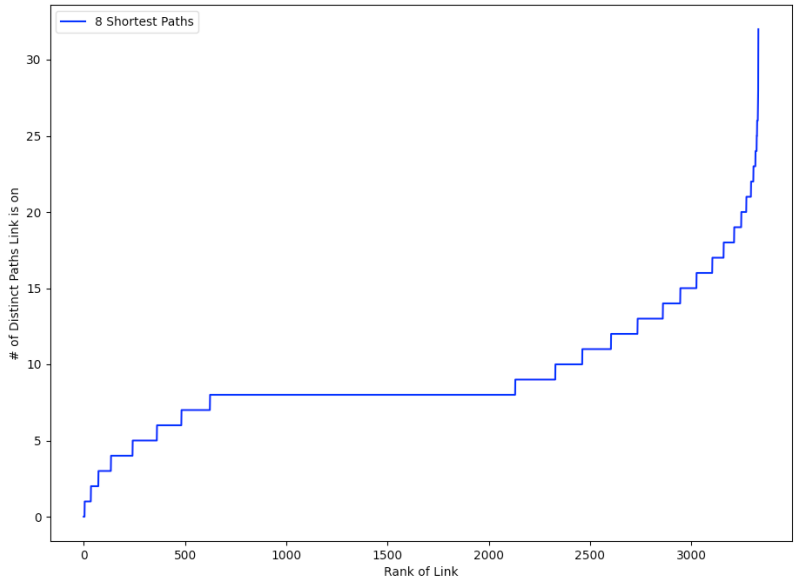
Both figures obtained after running the reproduce_1c script have a very close resemblance to the original figure. A difference can be identified in the 2nd figure, where 14% of paths for Jellyfish have a length of 6. The reason for this might be that the authors of the paper could have used a configuration with less switches as this is not specified in the paper. By having more switches, we expect to see more paths of length 6. The first reproduced diagram has a very close resemblance to the original diagram leading us to believe that the authors might have used a configuration with around 98 switches.

4. Reproduction of Figure 9

- *How code should be run?*
  In order to plot the representation for each topology, the following command should be run:
  > ***python3 reproduce_9.py --servers 686 -switches 244 -ports 14***

- *What is the general methodology used to reproduce the figure?*
  In order to reproduce the figure, we have implemented the Lee algorithm for k shortest paths. We initially generate the Jellyfish topology according to the provided parameters and then we generate a random permutation that includes unique links between servers to simulate the traffic. We use all the links in the permutation to generate all the k shortest paths using the implemented algorithm. To speed the computation, we split the array of permutations in equal chunks and run multiple parallel python processes to compute this list. In the end we compute the figure by iterating trough a list of all links in the topology and counting the ones that appear in the mapping with all the k shortest paths for the permutation.

- *Reproduced figure*



- *Explanation on the comparison*

We notice that in general the figure resembles the one in the original paper. However, the differences that can be identified can be attributed to a different configuration used by the authors of the paper. Because we only computed a small subset of all the possible links between all the 686 servers, due to performance considerations, we can expect a significant deviation from the original figure.