Advanced Networked Systems SS24

Lab2: Network Topologies Are Fun

Maximum points:	10
Submission:	zipped source code on PANDA
Deadline:	15.05.2024 23:59
Contact:	lin.wang@upb.de

1 Introduction

In the lecture, we learned how to build a data center network by interconnecting data center servers with regular network topologies like fat-trees. There are also other approaches for constructing topologies for data center networks as we will detail soon in this lab. Different topologies offer different properties such as bandwidth and latency, and with different costs such as the number of switches needed to interconnect a given number of servers. The goal of this lab is to implement some of these topologies and compare their properties. Most of these topologies are documented in research papers where analysis is also provided. You are asked to reproduce some of the figures in these research papers.

Please run `git pull` in the labs codebase to retrieve the code template for this lab. You will see a new folder `lab2` and you are supposed to work in that directory for this lab. You are requested to use **Python 3** in this lab since in the next lab, the topologies you generate now will be used in Mininet with the Ryu controller. Using the same language across the two labs will provide you with some convenience. You are provided a code template. Please do not change the high-level structure in the template (e.g., class names) and keep the existing variables and functions. You are free to add extra variables and functions in the classes as you deem necessary.

2 The Beauty of (Ir)Regularity

In this task, you are supposed to generate two classic topology designs for data center networks. Both topologies exploit somehow the (ir)regularity to offer benefits.

The first topology to construct is fat-tree, which we have discussed in depth in the lecture. The original paper describing the topology is the following one.

> Mohammad Al-Fares, Alexander Loukissas, Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. ACM SIGCOMM 2008. [pdf]

Please follow what we discussed in the lecture and details in the paper to construct the fat-tree topology.

Fat-tree is an example of leveraging the regularity of the topology to manage the cabling complexity. Yet, there is also a way of interconnecting servers in a data center in an opposite direction—randomly generating links between switches to form a network to connect servers. The main motivation is to exploit the *irregularity* to improve connectivity. One such example is Jellyfish described in the following paper.

> Ankit Singla, Chi-Yao Hong, Lucian Popa, P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. USENIX NSDI 2012. [pdf]

You are asked to generate the Jellyfish topology. Please read the paper thoroughly and make sure you understand the main ideas before you start coding.

⚠ Please make sure your code is general so topologies with different sizes can be generated by changing some parameters (e.g., number of connected servers or number of ports on the switch).

3 Who Is Winning?

The Jellyfish paper claims that Jellyfish outperforms fat-tree in various aspects. The paper verifies these claims by showing a variety of interesting figures. In this task, we are going to reproduce two of these figures. This also allows us to verify whether the results in the published paper are reproducible or not. Reproducibility is a very important step for open science and is especially valued in computer systems and networking research since the experiment setup is usually complex and reproducibility is hard to verify.

3.1 Reproducing Figure 1(c)

The first figure to reproduce is Figure 1(c) in the Jellyfish paper. This figure compares the shortest path length distribution for server pairs in fat-tree and Jellyfish. For each pair of servers, find out the length of the shortest path between the servers. Once you do this for all possible server pairs, you can calculate the fraction of server pairs (y-axis in the figure) that use the shortest path of a specific length (x-axis in the figure). Note that you are supposed to implement all algorithms from scratch, e.g., you need to compute the shortest path by implementing the Dijkstra algorithm (or other algorithms if you prefer). You should not use any external Python libraries for this purpose. The algorithm may take some minutes to run. If your implementation appears to be too slow to finish, think about if you can optimize it.

Using the code you wrote in the first part, you can generate the two types of topologies, i.e., fat-tree and Jellyfish, in expected sizes. In particular, Figure 1(c) assumes interconnecting 686 servers, which can be achieved by using 14-port switches in a fat-tree network. For Jellyfish, you should also use the same number of 14-port switches and distribute the servers evenly to these switches. Since Jellyfish is a random topology, you should perform multiple (10 in the paper) trials and average the results. Finally, please use bar plots to depict the distribution as done in the paper and compare it with the original figure in the paper.

Does your figure match that in the paper? If yes, great. If not, what are the differences and why are there such differences? Think about it.

3.2 Reproducing Figure 9

Now, let us reproduce another figure from the paper, i.e., Figure 9. This figure compares the performance of a Jellyfish network when different routing schemes are employed. The performance is measured by the number of distinct paths each link is on (similar to edge betweenness centrality[1] in graph theory).

Equal-cost multipath (ECMP) is a network routing strategy where traffic of the same flow (e.g., a TCP connection) is spread across multiple equal-length paths with equal priority. ECMP is a powerful idea for achieving better network utilization and load balancing in large-scale data center networks where redundant paths between server pairs are often available. Practically, ECMP applies a hash function on the five-tuple defining a network flow and decides the next hop (hence the path to take) based on the hash result. Note that packets from the same flow will be hashed to the same result and thus will always follow the same path so no extra packet reordering is needed as in packet-level multipath routing.

The routing schemes under comparison are:

- **8 shortest paths:** This scheme uses the 8 paths with the least lengths (these paths may have different lengths, but they are ranked the lowest in terms of the path length).
- **8-way ECMP:** This scheme uses 8 paths that have an equal length to the shortest path (note that the number of used paths could be less than 8 if the total number of paths that have an equal length to the shortest path is less than 8).
- **64-way ECMP:** This scheme uses 64 paths that have an equal length to the shortest path (note that the number of paths could be less than 64 if the total number of paths that have an equal length to the shortest path is less than 64).

[1]https://en.wikipedia.org/wiki/Betweenness_centrality

You can use the Yen's algorithm to calculate k shortest paths simultaneously. The Yen's algorithm is documented on this page. Please implement the algorithm by yourself. You are not allowed to use any external libraries for this algorithm.

The Jellyfish topology you use is the same one you have used when reproducing Figure 1(c): 686 servers interconnected with 14-port switches where the number of switches is equal to that used in a 14-port fat-tree topology. The traffic pattern we use in this experiment is called random permutation: each server sends traffic to exactly one other randomly chosen server. This can be achieved by performing a random permutation over the server index and mapping the original server indices to the permuted indices to form pairs. For example, given servers $[1, 2, 3]$ and a random permutation $[3, 1, 2]$, the expected traffic flows would be 1–3, 2–1, and 3–2.

Depending on your implementation, this may take up to tens of minutes. If your implementation is extremely slow, e.g., taking hours, please think about possible optimizations and improve your implementation.

4 Grading Criteria

The grading will be done based on an in-person interview-style oral examination. The detailed schedule for the interview will be announced when the submission deadline approaches. For fairness consideration, you must upload your code in a zip file. Please use the naming convention `Lab2_GroupX_LastName1_LastName2.zip` and rename the folder before you zip it. Here `X` is your group number and `LastName2` can be omitted if you are alone in the group. by the specified deadline and use the uploaded version for the interview. No interview will be scheduled for you if there is no code upload; this is a strict rule.

During the interview, you are supposed to show and explain the following:

- Your code can generate fat-tree and Jellyfish topologies in different sizes as required. Note that you need to find ways to demonstrate correctness. Some hints here: You can plot the topology you have built, but this only works well when the topology is small. You can also come up with some basic sanity checks like node degrees, number of links of certain types, etc. to see if they meet your expectations. (4 points)
- Reproduce Figure 1(c) and reason about the differences if any. (3 points)
- Reproduce Figure 9 and reason about the differences if any. (3 points)

5 You Can "Win" More (Optional)

If you want to challenge yourself further, the following bonus worth 5 points is available for you. You have to complete the two tasks to claim this bonus. No partial points will be given if only partial work is done.

5.1 More Topologies, More Fun

You are expected to generate other two (inspiring but not widely used) data center network topologies (i.e., BCube and DCell). The papers that describe the two topologies are listed below.

[BCube] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, Songwu Lu, Guohan Lv. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. ACM SIGCOMM 2009. [pdf]

[DCell] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, Songwu Lu. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. ACM SIGCOMM 2008. [pdf]

5.2 The Final Winner Is?

In reproducing Figure 9 of the Jellyfish paper, you have compared the performance of Jellyfish under different routing schemes. Now, try to carry out the same comparison for the other three topologies, i.e., fat-tree, BCube, and DCell. You should generate one plot for each of these topologies similar to that for Jellyfish and conduct a comparative analysis for the different topologies as to which routing scheme is more friendly to which topology.