Comparing Jellyfish Topologies

Paul Spooren University of Hawai'i
Honolulu, Hawai'i
spooren@hawaii.edu

***Abstract*—The rise in Internet focused applications has caused the need of optimal and thereby cost-efficient network topologies. Data centers are becoming bigger and more complex, requiring more planning of the flexible network setup. Established approaches like *Fat Tree* topologies although performant have scalability constraint due to hierarchical structure. It may have found a successor based on entirely randomized connections. Instead of planning a complex switch network with multiple stages the *Jellyfish* approach connects switches hierarchy free and archives great performances. This report looks at different Jellyfish topology generation algorithms and compares them.**

***Index Terms*—Topologies; ECMP; Network Algorithms; Jellyfish**

I. Introduction

This report looks at the creation and simulation of Jellyfish topologies. It will describe the implementation of three different algorithms for the topology generation and simulate two different traffic scenarios. To understand the scalability six different graph constellation with varying number of nodes and degree per node are generated.

In the second section related work is mentioned. Followed by the third section covering the differences between the three Jellyfish topologies and covers details of the code implementation. In the fourth section the simulation results are described, followed by a comparison of results. Lastly a short conclusion summarizes the work.

The simulation is implemented via the NetworkX framework[1] for graph generation and all code runs within a Jupyter Notebook[2]. This approach allows easy extension and manipulation of current code as Python is widely established and understood, and by the nature of Jupyter Notebooks easy to reproduce.

II. Related Work

The report uses the initial Jellyfish paper[1] as the main source of information. It describes the three topology construction algorithms for *random*, *incremental* and *bipartite* which is further illustrated in the next section.

Over the course of multiple years the *Stanford CS244 Reproducing Network Research*[3] already showed different approaches to measure performance of Jellyfish topologies[2][3]. They used the Mininet framework[4] to generate topologies and simulate actual throughput. While Mininet allows the simulation with actually running virtual machines and not only mathematical formulas, it also requires much more computation power which limits the scale. Also the setup is more complex as it requires an actual installation of Mininet, while the Jupyter Setup is possible to run entirely in the browser without any dependencies.

For the analysis itself they concluded that Jellyfish topologies are equal and partly archive even higher performance than Fat Tree topologies. This finding allows to look deeper into Jellyfish topology generation algorithms instead of creating another comparison.

[1] https://networkx.github.io/
[2] https://jupyter.org/
[3] https://reproducingnetworkresearch.wordpress.com/tag/jellyfish/
[4] https://mininet.org/

III. Jellyfish Topologies

As described in the introduction this report does not focus on comparing Jellyfish to other topologies but compares three different generator algorithms to one another. In the following descriptions N refers to the number of desired nodes in the graph, E to the number of edges and d to the desired degree of each node.

The following list will explain the generator idea in short sentences. In the Appendix are all three Python functions to generate the topologies.

- **Random Jellyfish topology:** Add N unconnected nodes and randomly connect two nodes which are not connected yet until no longer possible. If there is any node x left with a degree of $d-2$, split up an existing edge between two other nodes by adding node x between them. This increases x degree by one, repeat this with other nodes with a degree equal or lower to $d-2$. The Python function is `generate_jellyfish_random(nodes, degree)`.
- **Incremental Jellyfish topology:** Create a fully connected graph with $d+1$ nodes, resulting in all nodes having a degree of d. Add a new node x and split an existing edge between to other nodes, connect x to both other nodes. Repeat this step until x has a degree of d, now add another loop. Repeat until the number of nodes in the graph equal N. The Python function is `generate_jellyfish_incremental(nodes, degree)`.
- **Bipartite-Jellyfish Topology:** Create a fully connected bipartite graph with 2 *timesd* nodes. Add two new nodes x and $x+1$, split up d existing edges and connect x and $x+1$ to respectively other side of the bipartite graph. Repeat that step until

the graph contains N nodes. The Python function is `generate_jellyfish_bipartite(nodes, degree)`.

The generation is based on the NetworkX framework which allows easy manipulation of the graph and automatically calculating the shortest path between nodes.

IV. Performance

To measure the performance of the topologies we introduce concept *throughput* r. The value is calculated after applying the ECMP algorithm and finding the most used link called m. The throughput of the graph is $1/m$. The higher the throughput the higher is the performance of the network as it can transport more data given the requested structure of traffic.

For this report were two different traffic structures generated. Both have a different total traffic and are therefore not comparable with one another, however allow to compare the different topology algorithms.

- **All-to-All:** Every node within the network sends the same amount of data to every other node in the network excluding itself. The traffic generation is implemented in Python via `create_all2all_traffic(G)`.
- **Random permutation:** Every node sends traffic to a random node and receives traffic from a random node. As all created graphs contain a even number of nodes, every node receives and sends traffic. The traffic generation is implemented in Python via `create_random_traffic(G)`.

The simulation iterates over a variation of different graph constellations, varying in number of nodes and degree per node. The following six constellations where simulated:

Nodes	Degree(s)
64	8
100	8, 12
200	8, 12
200	12
300	12

Within each simulation the implemented *ECMP* algorithm distributed the traffic along the shortest path between two nodes exchanging traffic. The implementation would sum up all traffic run over each edge, allowing to simply search for the max value to find the previously mentioned m value required for the throughput calculation.

V. Comparison of Results

As the main objective of this report is the comparison of the three different topologies two graphs were created showing the *throughput performance* of *all to all* and *random* traffic.

The simulation for all graph constellations with different topology and traffic generators were run five times. In the table below is are the average results show the used type of generator, number of nodes N with degree d, the average shortest path ASP and the throughput r for all-to-all and random traffic.

Generator	N, d	ASP	r all-to-all	r random
Random	64, 8	2.194692	0.034795	0.633333
Incremental	64, 8	2.183894	0.032374	0.456495
Bipartite	64, 8	2.359623	0.030251	0.665657
Random	100, 8	2.422222	0.018021	0.445055
Incremental	100, 8	2.413663	0.018796	0.516340
Bipartite	100, 8	2.597576	0.017283	0.594049
Random	100, 12	2.067475	0.035258	0.576531
Incremental	100, 12	2.071881	0.035025	0.619898
Bipartite	100, 12	2.291717	0.033780	0.705169
Random	200, 8	2.765704	0.007635	0.380261
Incremental	200, 8	2.761219	0.007673	0.363707
Bipartite	200, 8	2.978543	0.007046	0.458042
Random	200, 12	2.403593	0.015214	0.472222
Incremental	200, 12	2.400672	0.015061	0.580579
Bipartite	200, 12	2.604221	0.014137	0.461082
Random	300, 12	2.570847	0.009322	0.475977
Incremental	300, 12	2.568195	0.009579	0.507085
Bipartite	300, 12	2.804326	0.008273	0.541816

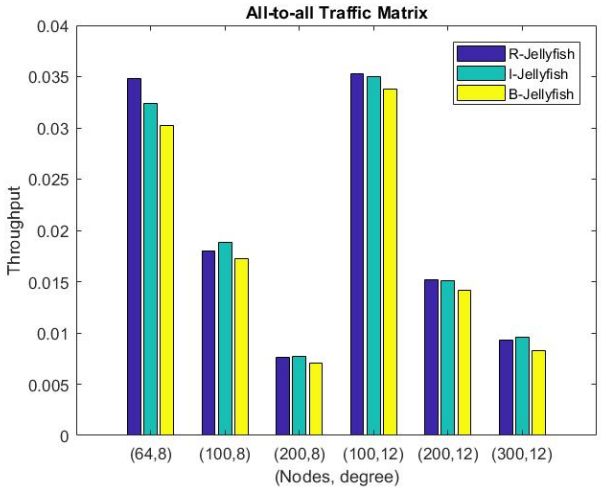

Figure 1. Throughput of all to all traffic

In all-to-all traffic matrix, the highest throughput is seen alternately for either random Jellyfish or for incremental Jellyfish. The bipartite has minimum throughput in all of the scenarios. It is due to the structure of bipartite and the reason that two nodes are added to the graph at once, increasing the distance between split up links by at least 3. As for the comparison between random and incremental jellyfish, the incremental jellyfish shows better throughput for higher number of nodes when the degree has not changed. So for the all-to-all network that need to be expanded constantly but with limited free ports, incremental jellyfish should be implemented.

In random traffic permutation, the bipartite Jellyfish perform well with highest throughput and incremental Jellyfish holds better result than random Jellyfish. Hence, implementation of bipartite Jellyfish using random permutation yields higher performance and for all-to-all the choice lies between incremental and random Jellyfish depending upon the size of your network.

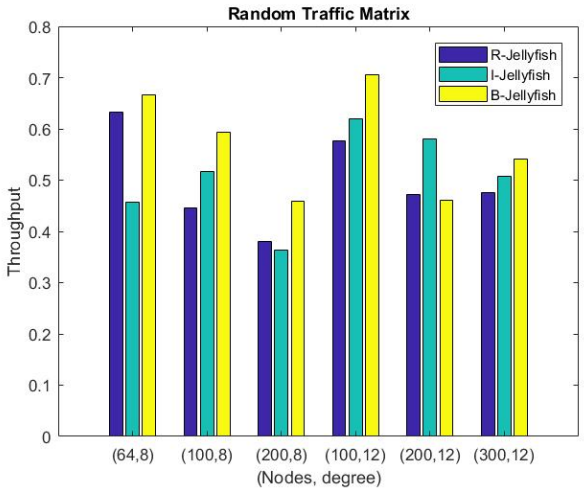

Figure 2. Throughput of random traffic

VI. Conclusion

The conclusion of this work is clearly that other than fully randomized graphs, it is more efficient to have specific topology generation algorithms per traffic setup.

As seen in Fig. 1 the fully random algorithm is outperformed by its competitor. While random Jellyfish works best on small graphs, on a scale the incremental Jellyfish generation results in a smaller average path length, which results in a higher throughput. Due to its generation algorithm it is trivial to extend the incremental approach which makes it therefore more flexible to scale over time. The random connection algorithm creates a graph which is not extendable once created as it is based on adding the total number of nodes N as a first step. Extending the randomly generated graph afterwards would follow the steps of the incremental algorithm, therefore the incremental algorithms should be used from the start.

Fig. 2 shows that for random traffic the bipartite Jellyfish performs better on nearly all graph constellations. Especially in *harder* constellations with a lower number of edges, which increases the average shortest path length, the bipartite algorithm still outperforms its competitors. The drawback of the bipartite approach is that it is only extendable by two new nodes at a time. In a concrete data center implementation this means two new *TOR* switches need to be added on each scale step.

All algorithms where successfully implemented in a reproducible way via NetworkX and Jupyter Notebooks. All used code can be found on GitHub[5] to verify this work. By using MyBinder[6] it is possible to run the setup within a browser with installing any dependencies.

[5]https://github.com/aparcar/ee607-project-jellyfish
[6]https://mybinder.org/

VII. Future Work

Based on the findings in the Conclusion section the two Jellyfish algorithms incremental and bipartite both seem as an appealing replacement for established *Fat Tree* setups. An additional factor would be the fault tolerance within the network. The current analysis takes not into account the performance while single nodes fail. This is an important field of research as in real setups node failures are highly likely and bypassed by redundant node setup. In a future analysis it should be explored how the random graphs create redundancy by its generation itself.

Further work should also be done with a more efficient implementation of the algorithms to allow bigger networks. This way it would be possible to see if the scalability assumption of incremental and bipartite Jellyfish proof to be right or is just a coincidence for smaller graphs.

References

[1] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *Presented as part of the 9th usenix symposium on networked systems design and implementation (nsdi 12)*, 2012, pp. 225–238.

[2] E. Conte and D. Franco, "Jellyfish vs. Fat tree," *Reproducing Network Research*. Aug-2012.

[3] C. Nguyen and C. Chong, "Fairness of jellyfish vs. Fat-tree," *Reproducing Network Research*. Aug-2012.