Throughput Analysis of Jellyfish Network Variations

Cristina McLaughlin
Department of Electrical Engineering
University of Hawaii at Manoa
Honolulu, United States
cemclaug@hawaii.edu

***Abstract*—Recent challenges facing modern data centers indicates a need for new network designs. The architectures must be flexible, incrementally scalable, fault tolerant, and capable of supporting high bandwidth applications. Jellyfish is a new high-capacity network that fulfills these design goals. Variations on the topology—random, incremental, and bipartite—were tested with different traffic loads using equal-cost multipath routing to observe the throughputs. Incremental expansion resulted in the same throughput as a Jellyfish network created from scratch. In addition, the bipartite Jellyfish outperformed the original Jellyfish in lower load situations, while underperforming in denser traffic. This paper describes the methodology of generating the Jellyfish variations, generating different traffic loads, simulating each network to find the average throughput, and finally reports and evaluates the results.**

Keywords—network architecture, throughput, ECMP routing

I. Introduction

A data center is a pool of computing resources that includes core design components such as routers, switches, firewalls, and servers. These centers are the foundation for important business activities like communication and collaboration, applications like big data, artificial intelligence, or machine learning, and infrastructure services like distributed files systems. Modern centers are also being built to provide popular online application services like web searching, gaming, and social media sites.

Conventional data centers are modeled as a multi-layer hierarchical network with servers as network nodes. For example, current data center topologies include fat-trees and folded-Clos. A fat-tree is a three-layer graph (edge, aggregation, and core) that has identical bandwidth at any bisection and each layer has the same aggregated bandwidth [1]. A folded-Clos network is a one-sided version of a fat-tree where it is essentially a bipartite graph. These current topologies are facing the challenges of rising traffic, bandwidth bottlenecks, and incremental network expansion, therefore it has become important to consider different network architectures [2]. For instance, in 2008 Facebook was running 10,000 servers in its data centers, by 2009 it was 30,000, and by 2010 it was 60,000 [3]. In 2019, the company also announced it would build four additional data centers to house more servers and storage [4].

These challenges have led to several major design goals in data center networking [5]. First, an idealized network should be scalable and flexible, allowing for incremental network expansion by adding servers and network capacity gradually. Second, the data center must be fault tolerant against server and link failures. Third, the data center must provide network capacity to support high-bandwidth services. Lastly, the network should be manageable both digitally and physically [2]. For example, developers should be able to manage or configure the network easily and the architecture should minimize cabling complexity.

This paper focuses on a new top-of-rack (ToR) switch architecture by Ankit Singla et al. [6] called **Jellyfish**. It is a degree-bounded random graph topology that handles the rising challenges of data centers and fulfills the design goals stated above. The random nature of the network makes it significantly more flexible and fault tolerant. On average, path lengths are shorter in Jellyfish compared to other topologies on all scales. Lastly, Jellyfish improves on the throughput of most conventional networks; it can support 27% more servers than a fat-tree topology [6]. Fig. 1 shows a diagram of a traditional fat-tree topology versus a random Jellyfish network [6].

The goal is to construct three variations of Jellyfish: **Random Jellyfish** (R-Jellyfish), **Incremental Jellyfish** (I-Jelly), and **Bipartite Jellyfish** (B-Jellyfish), then measure the capabilities and throughput of each topology in different traffic situations.

The outline of this report is as follows: Section 2 will further discuss the Jellyfish topology, it's variations, and how to construct the graphs, Section 3 will describe the throughput performance, Section 4 will be a comparison of results, and Section 5 will conclude the paper.

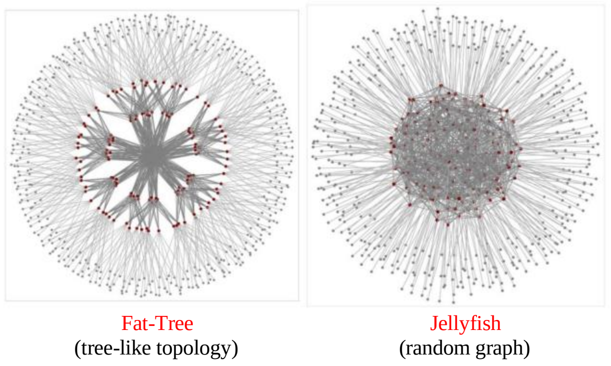

Fig. 1. *Traditional fat-tree network versus a random graph Jellyfish network. Random graphs have high throughput due to low average path length, resulting in less work to deliver packets.*

II. Jellyfish Topologies

This section further discusses the Jellyfish topology and ways to construct its variations. The Jellyfish network dictates the interconnection of ToR switches, which will be referred to as *nodes*, and the connections between switches as *links*. The *degree* of a node is the number of links connected to the node.

The original Jellyfish approach results in a *random regular graph* which has several desirable properties for data center architecture. Typically, random graphs have high throughput due to the low average path length. This results in less work delivering each packet. The random nature also allows for incremental expansion with just a few link swaps. Expansion through this method also produces a network that is identical in throughput and path length to one generated from scratch [7]. These properties will be tested using the graph variations below.

A. Random Jellyfish Topology (R-Jellyfish)

The original Jellyfish approach is to construct a random graph to connect ToR switches. To generate the topology first designate the number of nodes N and maximum degree of each node d. Randomly select two nodes, u and v, from the pool of nodes. Check if they are connected; if not, connect them with a link (u,v) and increment the current degree d for u and v. Repeat this process until it cannot be completed anymore.

Next, check the finished graph for nodes with degree d-2, which will be called node x. If x exists, select a random link (u, v). Remove the link between nodes u and v, then reassign links (u, x) and (v, x). Now x will have degree d. Repeat this process until it also cannot be completed anymore.

The final R-Jellyfish graph should contain N nodes with degree d, or possibly N-1 nodes with degree d and one node with degree d-1.

B. Incremental Jellyfish Topology (I-Jellyfish)

The I-Jellyfish approach is used to test if incremental expansion will result in a graph with the same throughput as R-Jellyfish. To generate the graph, start with a pool of $d+1$ nodes and link all nodes to each other to create a complete graph G. Add node x to G. Randomly select a link (u, v) and check that nodes u and v are not connected to x. Remove the link (u, v) then reassign links (u, x) and (v, x). Repeat this process until x has degree d. Continue to add links this way until the G contains N nodes.

As was the case with R-Jellyfish, the final graph should contain N nodes with degree d, or possibly N-1 nodes with degree d and one node with degree d-1.

C. Bipartite Jellyfish Topology (B-Jellyfish)

The B-Jellyfish approach randomly constructs a bipartite graph with an even number of nodes on the left and right sides. This variation is used to test traffic load-balancing against the R-Jellyfish. To construct the bipartite graph B, start with $2d$ nodes, with d nodes on the left side and d nodes on the right. Link all nodes on the left to all nodes on the right to create a complete bipartite graph.

Add a new node l to the left side of B and add new node r to the right side. Connect l and r with a link (l, r). Choose d-1 random links from B that do not share nodes with one another; the random links are (u_1, v_1), (u_2, v_2), …, (u_{d-1}, v_{d-1}). For each link (u_k, v_k), delete the link between nodes u_k and v_k, then add links (l, v_k) and (r, u_k). Repeat this process by adding pairs of nodes to B until there are N nodes.

III. Jellyfish Testing and Performance

To make comparisons between R-Jellyfish, I-Jellyfish, and B-Jellyfish simulations were conducted to measure the throughput r of each topology. Throughput is a common measure of network capacity because it indicates how much data is transferred at any given time [7]. Three assumptions were made for the simulations. First, the traffic matrix for each topology contains entries of 0 or r-value. Second, the traffic on each graph follows the shortest paths with equal-cost multi-path (ECMP) routing. ECMP is a mechanism to route packets along paths of equal cost to achieve an equal and distributed link load sharing []. The third assumption is that all links have capacity 1.

Throughput is defined as the maximum value r such that the traffic load on the links does not exceed the capacity, in this case 1. Find the highest traffic load L among all links. The resulting throughput equation is (1):

$$r = 1/L \tag{1}$$

Suppose an R-Jellyfish graph has a maximum link load $4r$. To ensure all the traffic loads on the graph are at most 1 due to the designated capacity, set $4r \leq 1$. The resulting throughput becomes $r = 0.25$.

A. Generating Traffic Matrices

The throughput for each Jellyfish topology was tested under two different traffic loads: **all-to-all** and **random permutation**. In all-to-all, each node sends r amount of traffic to each of the other nodes. This results in heavier loads across all connections; for example, a of graph of 300 nodes results in $\binom{300}{2} = 44{,}580$ directions for end-to-end traffic. To construct the all-to-all traffic matrix, create an $N \times N$ matrix where N is the number of nodes in the network. Fill each matrix entry with flow equal to 1. Both halves of the matrix must be filled in because all-to-all considers bidirectional traffic between nodes, e.g. node 1 sends traffic to node 300, but node 300 also sends traffic to node 1, effectively doubling the original load on all incident links.

Random permutation is traffic generated by randomly selecting sending and receiving node pairs. The receiving node is only considered if it does not already have incoming traffic which results in much lighter loads compared to all-to-all. A graph of 300 nodes results in 300 directions for end-to-end traffic—compared to the 44,580 generated by all-to-all—because each node can only have incoming traffic once. To construct the random permutation matrix, create an $N \times N$ matrix. Next, randomly select a sending node and a receiving node, check if the receiving node already has incoming traffic, and if not, update the matrix entry with 1. Repeat the process until all nodes in the graph receive traffic once from a source.

B. Simulation Methodology

An ECMP routing program was designed to calculate the throughput for the Jellyfish topologies. It reads in text files of a graph topology and traffic matrix, then outputs the traffic loads on each link. The program also reports the maximum load among links and uses (1) to calculate the throughput.

The procedure for routing follows. The program reads the sending node *u* and receiving node *v* from the traffic matrix along with the traffic load—in all cases it is 1. Starting at *v*, the *preferred neighbor* nodes are computed and saved. A preferred neighbor is a neighboring node that is along a shortest path from *u* to *v*. The traffic load is split evenly between all preferred neighbors and forwarded. The process repeats for all preferred neighbors until the program reaches the original sending node *u*. Finally, each link used in the forwarding will have its final flow saved to a traffic load matrix. The process repeats for all inputs of sending and receiving nodes, while updating all link usage. The simulation keeps track of which link has the highest flow, and reports the throughput using that value.

The ECMP simulation was tested 15 times with each Jellyfish topology, traffic style, and node-degree combination. R-Jellyfish, I-Jellyfish, and B-Jellyfish files were generated where $N = 64$ $d = 8$, $N = 100$ $d = 8$, $N = 200$ $d = 8$, $N = 100$ $d = 12$, $N = 200$ $d = 12$, and $N = 300$ $d = 12$. For example, to simulate an R-Jellyfish network containing 64 nodes of degree 8 using random permutation traffic, 15 topology files and 15 traffic matrices are generated. The files are inputted into the ECMP simulation, which outputs 15 throughput values. The data points are then averaged and presented in the results section. The testing was limited to 15 times per combination due to constraints and long execution time on larger node sets. The ECMP simulation takes upward of one minute to complete when $N = 300$ $d = 12$ with all-to-all traffic; this does not account for time necessary to generate the files.

C. Throughput Results

The throughput simulation results are shown in Table 1 and Table 2 below. The data is split between traffic load types: all-to-all and random permutation. The results show that without normalization, the average throughput in all-to-all traffic is significantly lower than random permutation due to the much heavier link loads. For example, with parameters $N = 300$ and $d = 12$ in the R-Jellyfish network, the average maximum load was 90 using all-to-all, compared to 3 using random permutation.

Overall, the distribution of the simulation data was much tighter than anticipated. High variance was expected due to the random nature of the topologies. However, all standard deviations calculated were >1 indicating the spread was low. The highest variance was $\sigma^2 = 0.07$, which occurred with parameters $N = 64$ $d = 8$ in the I-Jellyfish network using random permutation traffic. The maximum load on the network ranged from 2 to 4 with an average of 2.73. There were also multiple scenarios with $\sigma^2 = 0$; this occurred at $N = 300$ $d = 12$ in all graph topologies with random permutation. In every simulation the maximum load recorded was 3, resulting in an average throughput of 0.33 shown in Table 2. In addition, there were no significant outliers in any simulation. All throughputs calculated remained within one standard deviation of their respective means.

TABLE I. AVERAGE THROUGHPUT OF JELLYFISH TOPOLOGIES USING ALL-TO-ALL TRAFFIC LOAD

Nodes (*N*)	**Average Throughput (*r*)**					
	R-Jellyfish		***I-Jellyfish***		***B-Jellyfish***	
	d = 8	***d = 12***	***d = 8***	***d = 12***	***d = 8***	***d = 12***
64	0.0420		0.0440		0.0416	
100	0.0206	0.0349	0.0216	0.0368	0.0187	0.0319
200	0.0097	0.0167	0.0098	0.0165	0.0086	0.0145
300		0.0100		0.0106		0.0096

TABLE II. AVERAGE THROUGHPUT OF JELLYFISH TOPOLOGIES USING RANDOM PERMUTATION TRAFFIC LOAD

Nodes (*N*)	**Average Throughput (*r*), Random Permutation Traffic**					
	R-Jellyfish		***I-Jellyfish***		***B-Jellyfish***	
	d = 8	***d = 12***	***d = 8***	***d = 12***	***d = 8***	***d = 12***
64	0.3722		0.3833		0.4556	
100	0.3167	0.3278	0.3167	0.3389	0.3602	0.4524
200	0.2333	0.3055	0.2488	0.3194	0.2855	0.3333
300		0.3333		0.3330		0.3333

IV. EVALUATION OF RESULTS

This section evaluates the throughput results gathered in Section 3 and discusses the capabilities of each topology. The goal of Part A is to evaluate the performance of Jellyfish created from scratch versus one that has been incrementally expanded. The original Jellyfish research states that the random nature of the topology implies both graphs will be near identical.

Next, Part B compares the performance of the normal Jellyfish to a bipartite version. This comparison was to observe if a bipartite configuration is more effective at load balancing using ECMP routing.

A. R-Jellyfish vs. I-Jellyfish

Fig. 1(a) shows that with all-to-all traffic, R-Jellyfish (in blue) and I-Jellyfish (in red) had approximately the same throughput under all node and degree conditions. At most there was a 5.16% variation of throughput between R-Jellyfish and I-Jellyfish when $N = 100$ $d = 12$. Across the all-to-all simulations the average variation was 3.38%.

Fig. 1(b) shows a comparison between the topologies using random permutation traffic instead. The throughput using this load is much higher than all-to-all when it is not normalized. However, the results are the same and verify that R-Jellyfish and I-Jellyfish again have approximately the same throughput. The average data variation was 2.79%, with an outlier of 6.23% when $N = 200$ an$d = 8$.

In addition, another interested trend occurred. Fig. 1 shows that in all instances, I-Jellyfish has an equal or slightly higher throughput. It is unclear if this trend is significant or coincidence due to the low sample size; running more simulations could verify it. The original research did not show the same trend of incrementally built graphs having a slightly higher throughput. However, an overall evaluation of these results corroborates statements made in the original report—the throughputs are close to identical in all cases.

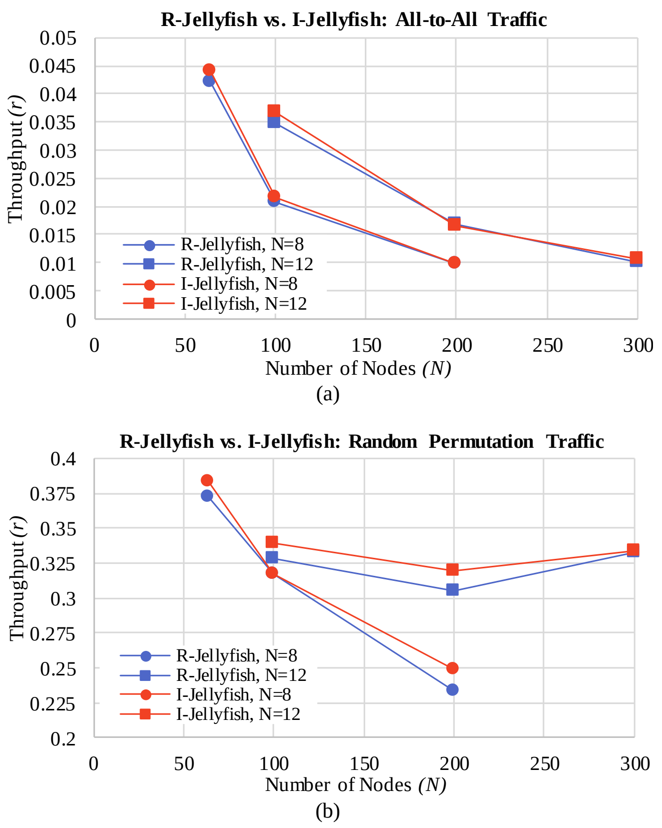

Fig. 3. *I-Jellyfish has the same thorughput capacity as R-Jellyfish under all node-degree paramaters and traffic conditions. The Jellyfish topology was expanded in increments of one node, then compared to the throughput of Jellyfish network built from scratch. The plot shows the average throughput over 15 runs for all-to-all traffic in (a) and random permutation traffic in (b).*

B. *R-Jellyfish vs. B-Jellyfish*

Since R-Jellyfish and I-Jellyfish were confirmed to have approximately the same throughputs, B-Jellyfish will only be compared to R-Jellyfish. The advantage of Jellyfish is that its average path lengths are shorter than topologies like the fat tree and folded Clos, however it becomes less effective in load balancing using ECMP routing because there is not enough path diversity. Fig. 2(a) shows the unnormalized throughput for all parameters under all-to-all traffic. In these simulations R-Jellyfish and B-Jellyfish performed similarly; R-Jellyfish had an average of 8.28% higher throughput. In comparison, Fig. 2(b) shows the topologies with random permutation traffic, and B-Jellyfish outperformed R-Jellyfish by an average of 15.37% more throughput, with a maximum of 20.4%. Fig. 2(b) also shows a trend of R-Jellyfish and B-Jellyfish converging when $N = 12$ but remaining relatively constant when $N = 8$.

The load of random permutation traffic is much lighter than all-to-all. This suggests that B-Jellyfish can significantly outperform R-Jellyfish on light loads while still performing nearly as well with heavier loads. This trend might be useful to take advantage of in future network designs depending on the data center needs.

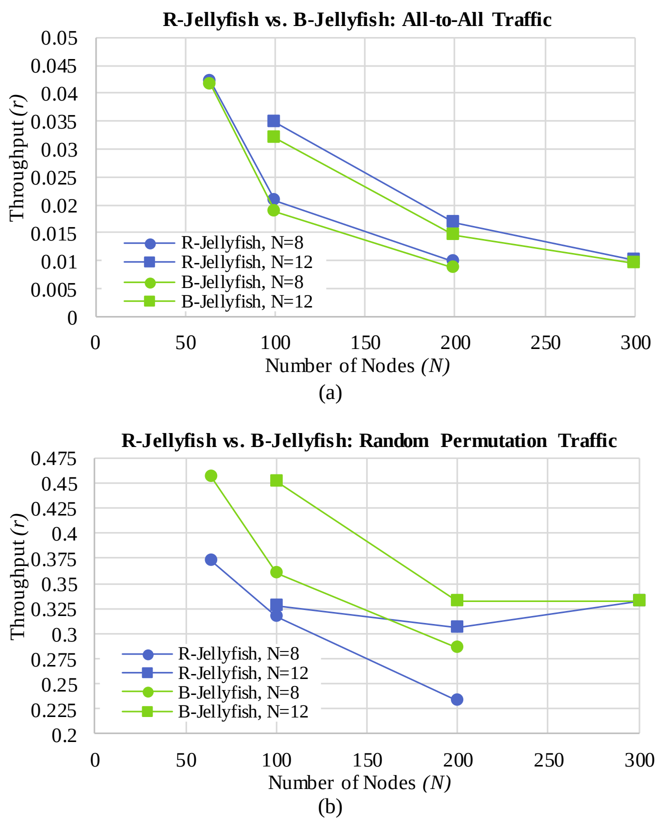

Fig. 2. *B-Jellyfish has nearly the same thorughput capacity as R-Jellyfish under all node-degree paramaters and traffic conditions. The bipartite topology was created by starting with a complete bipartite graph, then incrementally adding pairs of nodes and randomly reconnecting links. The plot shows the average throughput over 15 runs for all-to-all traffic in (a) and random permutation traffic in (b). Under light loads B-Jellyfish performs significantly better, then converges to the performance of R-Jellyfish as traffic increases.*

V. Conclusion

In conclusion, Jellyfish is a highly flexible network architecture for data centers. It is a new approach to solving the modern-day challenges of incremental expansion, rising traffic, and bandwidth bottlenecks, while maintaining short paths, high fault tolerance, and high throughput. After evaluating randomly generated Jellyfish, incrementally generated Jellyfish, and bipartite Jellyfish, results show minor differences between throughput in each, with the bipartite performing marginally better than others under lighter loads. The novel network performs well theoretically, however physical implementation will be the ultimate challenge in the long run.

References

[1] K. Solnushkin, “Fat-Tree Design,” ClusterDesign.org. [Online]. Available: https://clusterdesign.org/fat-trees/. [Accessed: 17-Mar-2020].

[2] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan, “Scale-Out Networking in the Data Center,” IEEE Micro, vol. 30, no. 4, pp. 29–41, 2010.

[3] “The Facebook Data Center FAQ,” *Data Center Knowledge*, 27-Sep-2010. [Online]. Available: https://www.datacenterknowledge.com/data-center-faqs/facebook-data-center-faq-page-2. [Accessed: 15-Mar-2020].

[4] R. Miller and C. Land, “Facebook Accelerates its Data Center Expansion,” Data Center Frontier, 19-Mar-2018. [Online]. Available: https://datacenterfrontier.com/facebooks-accelerates-data-center-expansion/. [Accessed: 18-Mar-2020].

[5] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, “Dcell,” *Proceedings of the ACM SIGCOMM 2008 conference on Data communication - SIGCOMM 08*, 2008.

[6] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, “Jellyfish: Networking Data Centers Randomly,” Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), pp. 225–238, 2012.

[7] “Bandwidth and Throughput in Networking: Guide and Tools,” DNSstuff, 19-Sep-2019. [Online]. Available: https://www.dnsstuff.com/network-throughput-bandwidth. [Accessed: 20-Mar-2020].