Jellyfish: Networking Data Centers Randomly

Ankit Singla[†][*], Chi-Yao Hong[†][*], Lucian Popa[♯], P. Brighten Godfrey[†]
[†] University of Illinois at Urbana–Champaign
[♯] HP Labs

Abstract

Industry experience indicates that the ability to incrementally expand data centers is essential. However, existing high-bandwidth network designs have rigid structure that interferes with incremental expansion. We present Jellyfish, a high-capacity network interconnect which, by adopting a random graph topology, yields itself naturally to incremental expansion. Somewhat surprisingly, Jellyfish is more cost-efficient than a fat-tree, supporting as many as 25% more servers at full capacity using the same equipment at the scale of a few thousand nodes, and this advantage improves with scale. Jellyfish also allows great flexibility in building networks with different degrees of oversubscription. However, Jellyfish's unstructured design brings new challenges in routing, physical layout, and wiring. We describe approaches to resolve these challenges, and our evaluation suggests that Jellyfish could be deployed in today's data centers.

1 Introduction

A well provisioned data center network is critical to ensure that servers do not face bandwidth bottlenecks to utilization; to help isolate services from each other; and to gain more freedom in workload placement, rather than having to tailor placement of workloads to where bandwidth is available [22]. As a result, a significant body of work has tackled the problem of building high capacity network interconnects [6, 17–20, 38, 42, 43].

One crucial problem that these designs encounter is incremental network expansion, *i.e.*, adding servers and network capacity incrementally to the data center. Expansion may be necessitated by growth of the user base, which requires more servers, or by the deployment of more bandwidth-hungry applications. Expansion within a data center is possible through either planned over-provisioning of space and power, or by upgrading old servers to a larger number of more powerful but energy-efficient new servers. Planned expansion is a practical strategy to reduce up-front capital expenditure [28].

Industry experience indicates that incremental expansion is important. Consider the growth of Facebook's data center server population from roughly 30,000 in Nov. 2009 to >60,000 by June 2010 [34]. While Facebook has added entirely new data center facilities, much of this growth involves incrementally expanding existing facilities by "adding capacity on a daily basis" [33]. For instance, Facebook announced that it would double the size of its facility at Prineville, Oregon by early 2012 [16]. A 2011 survey [15] of 300 enterprises that run data centers of a variety of sizes found that 84% of firms would probably or definitely expand their data centers in 2012. Several industry products advertise incremental expandability of the server pool, including SGI's IceCube (marketed as "The Expandable Modular Data Center" [5]; expands 4 racks at a time) and HP's EcoPod [24] (a "pay-as-you-grow" enabling technology [23]).

Do current high-bandwidth data center network proposals allow incremental growth? Consider the fat-tree interconnect, as proposed in [6], as an illustrative example. The entire structure is completely determined by the port-count of the switches available. This is limiting in at least two ways. First, it makes the design space very coarse: full bisection bandwidth fat-trees can only be built at sizes 3456, 8192, 27648, and 65536 corresponding to the commonly available port counts of 24, 32, 48, and 64[1]. Second, even if (for example) 50-port switches were available, the smallest "incremental" upgrade from the 48-port switch fat-tree would add 3,602 servers and would require replacing every switch.

There are, of course, some workarounds. One can replace a switch with one of larger port count or oversubscribe certain switches, but this makes capacity distribution constrained and uneven across the servers. One could leave free ports for future network connections [14, 20] but this wastes investment until actual expansion. Thus, without compromises on bandwidth or cost, such topologies are not amenable to incremental growth.

Since it seems that *structure* hinders incremental expansion, we propose the opposite: a random network in-

[*]A coin toss decided the order of the first two authors.

[1]Other topologies have similar problems: a hypercube [7] allows only power-of-2 sizes, a de Bruijn-like construction [39] allows only power-of-3 sizes, etc.

terconnect. The proposed interconnect, which we call **Jellyfish**, is a *degree-bounded*[2] *random graph* topology among top-of-rack (ToR) switches. The inherently sloppy nature of this design has the potential to be significantly more flexible than past designs. Additional components—racks of servers or switches to improve capacity—can be incorporated with a few random edge swaps. The design naturally supports heterogeneity, allowing the addition of newer network elements with higher port-counts as they become available, unlike past proposals which depend on certain regular port-counts [6, 18–20, 38, 42]. Jellyfish also allows construction of arbitrary-size networks, unlike topologies discussed above which limit the network to very coarse design points dictated by their structure.

Somewhat surprisingly, Jellyfish supports *more* servers than a fat-tree [6] built using the same network equipment while providing at least as high per-server bandwidth, measured either via bisection bandwidth or in throughput under a random-permutation traffic pattern. In addition, Jellyfish has lower mean path length, and is resilient to failures and miswirings.

But a data center network that lacks regular structure is a somewhat radical departure from traditional designs, and this presents several important challenges that must be addressed for Jellyfish to be viable. Among these are routing (schemes depending on a structured topology are not applicable), physical construction, and cabling layout. We describe simple approaches to these problems which suggest that Jellyfish could be effectively deployed in today's data centers.

Our key contributions and conclusions are as follows:

- We propose Jellyfish, an incrementally-expandable, high-bandwidth data center interconnect based on a random graph.

- We show that Jellyfish provides quantitatively easier incremental expansion than prior work on incremental expansion in Clos networks [14], growing incrementally at only 40% of the expense of [14].

- We conduct a comparative study of the bandwidth of several proposed data center network topologies. Jellyfish can support 25% more servers than a fat-tree while using the same switch equipment and providing at least as high bandwidth. This advantage increases with network size and switch port-count. Moreover, we propose *degree-diameter optimal graphs* [12] as benchmark topologies for high capacity at low cost, and show that Jellyfish remains within 10% of these carefully-optimized networks.

- Despite its lack of regular structure, packet-level simulations show that Jellyfish's bandwidth can be effectively utilized via existing forwarding technologies that provide high path diversity.

- We discuss effective techniques to realize physical layout and cabling of Jellyfish. Jellyfish may have higher cabling cost than other topologies, since its cables can be longer; but when we restrict Jellyfish to use cables of length similar to the fat-tree, it still improves on the fat-tree's throughput.

Outline: Next, we discuss related work (§2), followed by a description of the Jellyfish topology (§3), and an evaluation of the topology's properties, unhindered by routing and congestion control (§4). We then evaluate the topology's performance with routing and congestion control mechanisms (§5). We discuss effective cabling schemes and physical construction of Jellyfish in various deployment scenarios (§6), and conclude (§7).

2 Related Work

Several recent proposals for high-capacity networks exploit special structure for topology and routing. These include folded-Clos (or fat-tree) designs [6, 18, 38], several designs that use servers for forwarding [19, 20, 45], and designs using optical networking technology [17, 43]. High performance computing literature has also studied carefully-structured expander graphs [27].

However, none of these architectures address the *incremental expansion* problem. For some (including the fat-tree), adding servers while preserving the structural properties would require replacing a large number of network elements and extensive rewiring. MDCube [45] allows expansion at a very coarse rate (several thousand servers). DCell and BCube [19, 20] allow expansion to an *a priori* known target size, but require servers with free ports reserved for planned future expansion.

Two recent proposals, Scafida [21] (based on scale-free graphs) and Small-World Datacenters (SWDC) [41], are similar to Jellyfish in that they employ randomness, but are significantly different than our design because they require correlation (i.e., structure) among edges. This structured design makes it unclear whether the topology retains its characteristics upon incremental expansion; neither proposal investigates this issue. Further, in SWDC, the use of a regular lattice underlying the topology creates familiar problems with incremental expansion.[3] Jellyfish also has a capacity advantage over

[2] Degree-bounded means that the number of connections per node is limited, in this case by switch port-counts.

[3] For instance, using a 2D torus as the lattice implies that maintaining the network structure when expanding an n node network requires adding $\Theta(\sqrt{n})$ new nodes. The higher the dimensionality of the lattice, the more complicated expansion becomes.

both proposals: Scafida has marginally worse bisection bandwidth and diameter than a fat-tree, while Jellyfish improves on fat-trees on both metrics. We show in §4.1 that Jellyfish has higher bandwidth than SWDC topologies built using the same equipment.

LEGUP [14] attacks the expansion problem by trying to find optimal upgrades for Clos networks. However, such an approach is fundamentally limited by having to start from a rigid structure, and adhering to it during the upgrade process. Unless free ports are preserved for such expansion (which is part of LEGUP's approach), this can cause significant overhauls of the topology even when adding just a few new servers. In this paper, we show that Jellyfish provides a simple method to expand the network to almost any desirable scale. Further, our comparison with LEGUP (§4.2) over a sequence of network expansions illustrates that Jellyfish provides significant cost-efficiency gains in incremental expansion.

REWIRE [13] is a heuristic optimization method to find high capacity topologies with a given cost budget, taking into account length-varying cable cost. While [13] compares with random graphs, the results are inconclusive.[4] Due to the recency of [13], we have left a direct quantitative comparison to future work.

Random graphs have been examined in the context of communication networks [31] previously. Our contribution lies in applying random graphs to allow incremental expansion and in quantifying the efficiency gains such graphs bring over traditional data center topologies.

3 Jellyfish Topology

Construction: The Jellyfish approach is to construct a random graph at the top-of-rack (ToR) switch layer. Each ToR switch i has some number k_i of ports, of which it uses r_i to connect to other ToR switches, and uses the remaining $k_i - r_i$ ports for servers. In the simplest case, which we consider by default throughout this paper, every switch has the same number of ports and servers: $\forall i$, $k = k_i$ and $r = r_i$. With N racks, the network supports $N(k-r)$ servers. In this case, the network is a *random regular graph*, which we denote as RRG(N, k, r). This is a well known construct in graph theory and has several desirable properties as we shall discuss later.

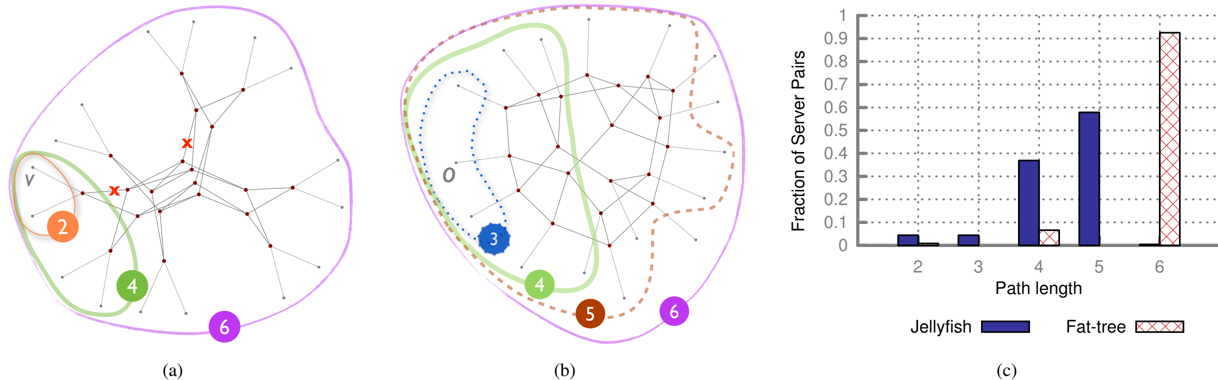
Formally, RRGs are sampled uniformly from the space of all r-regular graphs. This is a complex problem in graph theory [29]; however, a simple procedure produces "sufficiently uniform" random graphs which empirically have the desired performance properties. One can simply pick a random pair of switches with free ports (for the switch-pairs are not already neighbors), join them with a link, and repeat until no further links can be added. If a switch remains with ≥ 2 free ports (p_1, p_2) — which includes the case of incremental expansion by adding a new switch — these can be incorporated by removing a uniform-random existing link (x, y), and adding links (p_1, x) and (p_2, y). Thus only a single unmatched port might remain across the whole network.

Using the above idea, we generate a blueprint for the physical interconnection. (Allowing human operators to "wire at will" may result in poor topologies due to human biases – for instance, favoring shorter cables over longer ones.) We discuss cabling later in §6.

Intuition: Our two key goals are high bandwidth and flexibility. The intuition for the latter property is simple: lacking structure, the RRG's network capacity becomes "fluid", easily wiring up any number of switches, heterogeneous degree distributions, and newly added switches with a few random link swaps.

But why should random graphs have high bandwidth? We show quantitative results later, but here we present the intuition. The end-to-end throughput of a topology depends not only on the capacity of the network, but is also inversely proportional to the *amount of network capacity consumed to deliver each byte* — that is, the average path length. Therefore, assuming that the routing protocol is able to utilize the network's full capacity, low average path length allows us to support more flows at high throughput. To see why Jellyfish has low path length, Fig. 1(a) and 1(b) visualize a fat-tree and a representative Jellyfish topology, respectively, with *identical* equipment. Both topologies have diameter 6, meaning that any server can reach all other servers in 6 hops. However, in the fat-tree, each server can only reach 3 others in ≤ 5 hops. In contrast, in the random graph, the typical origin server labeled o can reach 12 servers in ≤ 5 hops, and 6 servers in ≤ 4 hops. The reason for this is that many edges in the fat-tree are not useful from the perspective of their effect on path length; for example, deleting the two edges marked X in Fig. 1(a) does not increase the path length between any pair of servers. In contrast, the RRG's diverse random connections lead

[4]REWIRE attempts to improve a given "seed" graph. The seed could be a random graph, so in principle [13] should be able to obtain results at least as good as Jellyfish. In [13] the seed was an empty graph. The results show, in some cases, fat-trees obtaining more than an order of magnitude worse bisection bandwidth than random graphs, which in turn are more than an order of magnitude worse than REWIRE topologies, all at equal cost. In other cases, [13] shows random graphs that are disconnected. These significant discrepancies could arise from (a) separating network port costs from cable costs rather than optimizing over the total budget, causing the random graph to pay for more ports than it can afford cables to connect; (b) assuming linear physical placement of all racks, so cable costs for distant servers scale as $\Theta(n)$ rather than $\Theta(\sqrt{n})$ in a more typical two-dimensional layout; and (c) evaluating very low bisection bandwidths (**0.04** to 0.37) — in fact, at the highest bisection bandwidth evaluated, [13] indicates the random graph has higher throughput than REWIRE. The authors indicated to us that REWIRE has difficulty scaling beyond a few hundred nodes.



(a) (b) (c)

Figure 1: *Random graphs have high throughput because they have low average path length, and therefore do less work to deliver each packet.* ***(a):*** *Fat-tree with* 16 *servers and* 20 *four-port switches.* ***(b):*** *Jellyfish with identical equipment. The servers are leaf nodes; switches are interior nodes. Each 'concentric' ring contains servers reachable from any server v in the fat-tree, and an arbitrary server o in Jellyfish, within the number of hops in the marked label. Jellyfish can reach many more servers in few hops because in the fat tree, many edges (like those marked "X") are redundant from a path-length perspective.* ***(c):*** *Path length distribution between servers for a* 686*-server Jellyfish (drawn from* 10 *trials) and same-equipment fat-tree.*

to lower mean path length.[5] Figure 1(c) demonstrates these effects at larger scale. With 686 servers, >99.5% of source-destination pairs in Jellyfish can be reached in fewer than 6 hops, while the corresponding number is only 7.5% in the fat-tree.

4 Jellyfish Topology Properties

This section evaluates the efficiency, flexibility and resilience of Jellyfish and other topologies. Our goal is to measure the raw capabilities of the topologies, were they to be coupled with optimal routing and congestion control. We study how to perform routing and congestion control separately, in §5.

Our key findings from these experiments are:

- Jellyfish can support 27% more servers at full capacity than a (same-switching-equipment) fat-tree at a scale of <900 servers. The trend is for this advantage to *improve* with scale.
- Jellyfish's network capacity is >91% of the best-known degree-diameter graphs [12], which we propose as benchmark bandwidth-efficient graphs.
- Paths are shorter on average in Jellyfish than in a fat-tree, and the *maximum* shortest path length (diameter) is the same or lower for all scales we tested.
- Incremental expansion of Jellyfish produces topologies identical in throughput and path length Jellyfish topologies generated from scratch.
- Jellyfish provides a significant cost-efficiency advantage over prior work (LEGUP [14]) on incremental network expansion in Clos networks. In a network expansion scenario that was made available for us to test, Jellyfish builds a slightly higher-capacity expanded network at only 40% of LEGUP's expense.
- Jellyfish is highly failure resilient, even more so than the fat-tree. Failing a random 15% of all links results in a capacity decrease of <16%.

[5] This is related to the fact that RRGs are *expander graphs* [11].

Evaluation methodology: Some of the results for network capacity in this section are based on explicit calculations of the theoretical bounds for bisection bandwidth for regular random graphs.

All throughput results presented in this section are based on calculations of throughput for a specific class of traffic demand matrices with optimal routing. The traffic matrices we use are *random permutation traffic*: each server sends at its full output link rate to a single other server, and receives from a single other server, and this permutation is chosen uniform-randomly. Intuitively, random permutation traffic represents the case of no locality in traffic, as might arise if VMs are placed without regard to what is convenient for the network[6]. Nevertheless, evaluating other traffic patterns is an important question that we leave for future work.

Given a traffic matrix, we characterize a topology's raw capacity with "ideal" load balancing by treating flows as splittable and fluid. This corresponds to solving a standard multi-commodity network flow problem. (We use the CPLEX linear program solver [1].)

For all throughput comparisons, we use the *same switching equipment* (in terms of both number of switches, and ports on each switch) for each set of

[6] Supporting such flexible network-oblivious VM placement without a performance penalty is highly desirable [22].

topologies compared. Throughput results are always normalized to $[0,1]$, and averaged over all flows.

For comparisons with the full bisection bandwidth fat-tree, we attempt to find, using a binary search procedure, the maximum number of servers Jellyfish can support using the same switching equipment as the fat-tree while satisfying the full traffic demands. Specifically, each step of the binary search checks a certain number of servers m by sampling three random permutation traffic matrices, and checking whether Jellyfish supports full capacity for *all* flows in *all* three matrices. If so, we say that Jellyfish supports m servers at full capacity. After our binary search terminates, we verify that the returned number of servers is able to get full capacity over each of 10 more samples of random permutation traffic matrices.

4.1 Efficiency

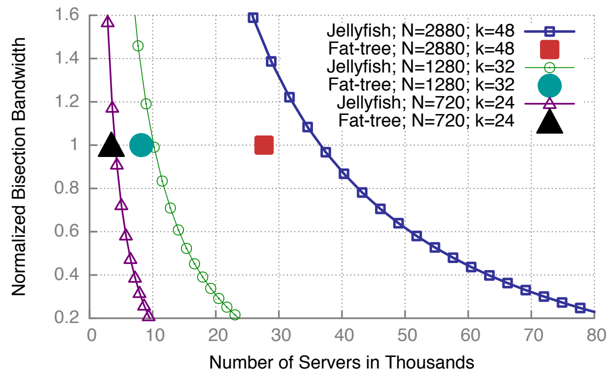
Bisection Bandwidth vs. Fat-Tree: Bisection bandwidth, a common measure of network capacity, is the *worst-case* bandwidth spanning any two equal-size partitions of a network. Here, we compute the fat-tree's bisection bandwidth directly from its parameters; for Jellyfish, we model the network as a RRG and apply a lower bound of Bollobás [8]. We normalize bisection bandwidth by dividing it by the total line-rate bandwidth of the servers in one partition[7].

Fig. 2(a) shows that at the same cost, Jellyfish supports a larger number of servers (x axis) at full bisection bandwidth (y axis $= 1$). For instance, at the same cost as a fat-tree with 16,000 servers, Jellyfish can support >20,000 servers at full bisection bandwidth. Also, Jellyfish allows the freedom to accept lower bisection bandwidth in exchange for supporting more servers or cutting costs by using fewer switches.

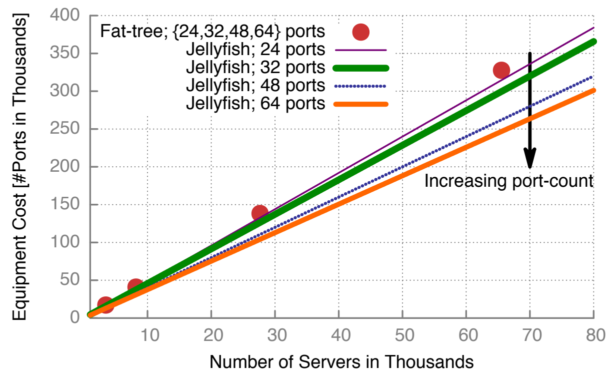
Fig. 2(b) shows that the cost of building a full bisection-bandwidth network increases more slowly with the number of servers for Jellyfish than for the fat-tree, especially for high port-counts. Also, the design choices for Jellyfish are essentially continuous, while the fat-tree (following the design of [6]) allows only certain discrete jumps in size which are further restricted by the port-counts of available switches. (Note that this observation would hold even for over-subscribed fat-trees.)

Jellyfish's advantage increases with port-count, approaching twice the fat-tree's bisection bandwidth. To see this, note that the fat-tree built using k-port switches has $k^3/4$ servers, and being a full-bisection interconnect, it has $k^3/8$ edges crossing each bisection. The fat-tree has $k^3/2$ switch-switch links, implying that its bisection bandwidth represents $\frac{1}{4}$ of its switch-switch links. For Jellyfish, in expectation, $\frac{1}{2}$ of its switch-switch links

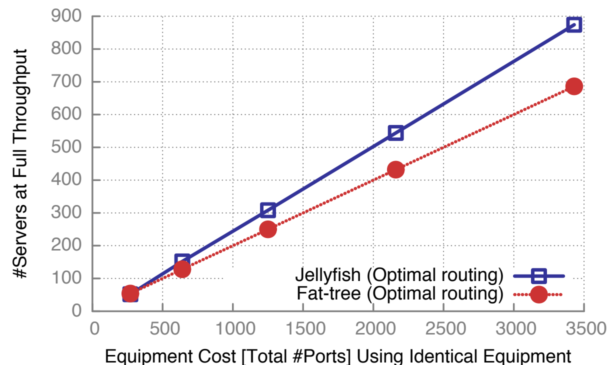
[7] Values larger than 1 indicate overprovisioning.



(a)



(b)



(c)

Figure 2: *Jellyfish offers virtually continuous design space, and packs in more servers at high network capacity at the same expense as a fat-tree. From theoretical bounds: (a) Normalized bisection bandwidth versus the number of servers supported; equal-cost curves, and (b) Equipment cost versus the number of servers for commodity-switch port-counts (*24, 32, 48*) at full bisection bandwidth. Under optimal routing, with random-permutation traffic: (c) Servers supported at full capacity with the same switching equipment, for* 6, 8, 10, 12 *and* 14*-port switches. Results for (c) are averaged over* 8 *runs.*

cross any given bisection of the switches, which is *twice* that of the fat-tree assuming they are built with the same number of switches and servers. Intuitively, Jellyfish's worst-case bisection should be slightly worse than this average bisection. The bound of [8] bears this out: in

almost every r-regular graph with N nodes, every set of $N/2$ nodes is joined by at least $N(\frac{r}{4} - \frac{\sqrt{r \ln 2}}{2})$ edges to the rest of the graph. As the number of network ports $r \to \infty$ this quantity approaches $Nr/4$, i.e., $\frac{1}{2}$ of the $Nr/2$ links.

Throughput vs. Fat Tree: Fig. 2(c) uses the random-permutation traffic model to find the number of servers Jellyfish can support at full capacity, matching the fat-tree in capacity and switching equipment. The improvement is as much as 27% more servers than the fat-tree at the largest size (874 servers) we can use CPLEX to evaluate. As with results from Bollobás' theoretical lower bounds on bisection bandwidth (Fig. 2(a), 2(b)), the trend indicates that this improvement increases with scale.

Throughput vs. Degree-Diameter Graphs: We compare Jellyfish's capacity with that of the best known degree-diameter graphs. Below, we briefly explain what these graphs are, and why this comparison makes sense.

There is a fundamental trade-off between the degree and diameter of a graph of a fixed vertex-set (say of size N). At one extreme is a clique — maximum possible degree ($N-1$), and minimum possible diameter (1). At the other extreme is a disconnected graph with degree 0 and diameter ∞. The problem of constructing a graph with maximum possible number N of nodes while preserving given diameter and degree bounds is known as the *degree-diameter problem* and has received significant attention in graph theory. The problem is quite difficult and the optimal graphs are only known for very small sizes: the largest degree-diameter graph known to be optimal has $N = 50$ nodes, with degree 7 and diameter 2 [12]. A collection of optimal and best known graphs for other degree-diameter combinations is maintained at [12].

The degree-diameter problem relates to our objective in that short average path lengths imply low resource usage and thus high network capacity. Intuitively, the best known degree-diameter topologies should support a large number of servers with high network bandwidth and low cost (small degree). While we note the distinction between average path length (which relates more closely to the network capacity) and diameter, degree-diameter graphs will have small average path lengths too.

Thus, we propose the best-known degree-diameter graphs as a benchmark for comparison. Note that such graphs do not meet our incremental expansion objectives; we merely use them as a capacity benchmark for Jellyfish topologies. But these graphs (and our measurements of them) may be of independent interest since they could be deployed as highly efficient topologies in a setting where incremental upgrades are unnecessary, such as a pre-fab container-based data center.

For our comparisons with the best-known degree-diameter graphs, the number of servers we attach to the switches was decided such that full-bisection bandwidth

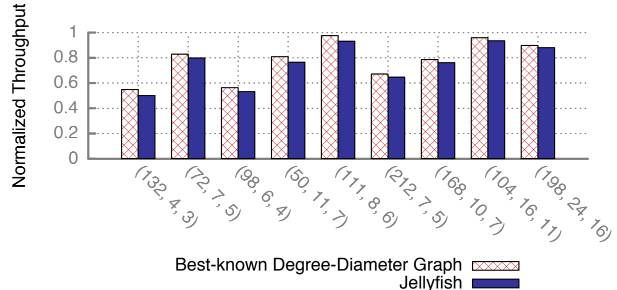


Figure 3: *Jellyfish's network capacity is close to (i.e., ~91% or more in each case) that of the best-known degree-diameter graphs. The x-axis label (A, B, C) represents the number of switches (A), the switch port-count (B), and the network degree (C). Throughput is normalized against the non-blocking throughput. Results are averaged over* 10 *runs.*

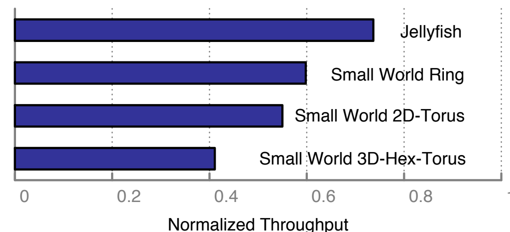


Figure 4: *Jellyfish has higher capacity than the (same-equipment) small world data center topologies [41] built using a ring, a* 2*D-Torus, and a* 3*D-Hex-Torus as the underlying lattice. Results are averaged over* 10 *runs.*

was not hit for the degree-diameter graphs (thus ensuring that we are measuring the full capacity of degree-diameter graphs.) Our results, in Fig. 3, show that the best-known degree-diameter graphs do achieve higher throughput than Jellyfish, and thus improve even more over fat-trees. But in the worst of these comparisons, Jellyfish still achieves ~91% of the degree-diameter graph's throughput. While graphs that are optimal for the degree-diameter problem are not (known to be) provably optimal for our bandwidth optimization problem, these results strongly suggest that Jellyfish's random topology leaves little room for improvement, even with very carefully-optimized topologies. And what improvement is possible may not be worth the loss of Jellyfish's incremental expandability.

Throughput vs. small world data centers (SWDC): SWDC [41] proposes a new topology for data centers inspired by a small-world distribution. We compare Jellyfish with SWDC using the same degree-6 topologies described in the SWDC paper. We emulate their 6-interface server-based design by using switches connected with 1 server and 6 network ports each. We build the three SWDC variants described in [41] at topology sizes as close to each other as possible (constrained by the lattice structure underlying these topologies) across sizes

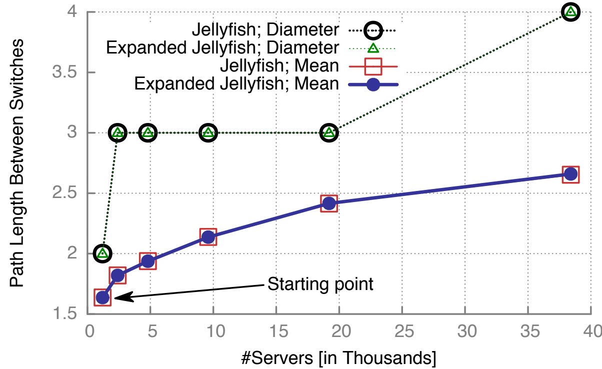


Figure 5: *Jellyfish has short paths: Path length versus number of servers, with $k = 48$ port switches of which $r = 36$ connect to other switches and 12 connect to servers. Each data point is derived from 10 graphs. The diameter is ≤4 at these scales. This figure also shows that constructing Jellyfish from scratch, or using incremental growth yields topologies with very similar path length characteristics.*

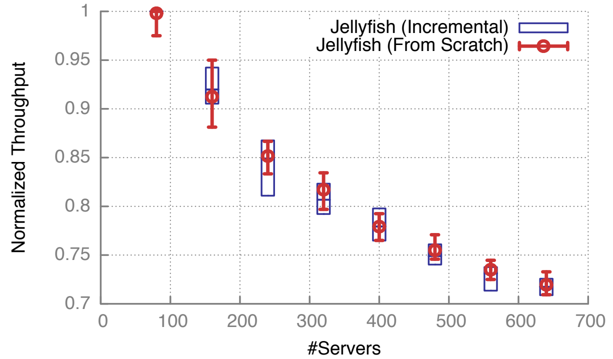


Figure 6: *Incrementally constructed Jellyfish has the same capacity as Jellyfish built from scratch: We built a Jellyfish topology incrementally from* 20 *to* 160 *switches in increments of* 20 *switches, and compared the throughput per server of these incrementally grown topologies to Jellyfish topologies built from scratch using our construction routine. The plot shows the average, minimum and maximum throughput over* 20 *runs.*

we can simulate. Thus, we use 484 switches for Jellyfish, the SWDC-Ring topology, and the SWDC-2D-Torus topology; for the SWDC-3D-Hex-Torus, we use 450 nodes. (Note that this gives the latter topology an advantage, because it uses the same degree, but a smaller number of nodes. However, this is the closest size where that topology is well-formed.) At these sizes, the first three topologies all yielded full throughput, so, to distinguish between their capacities, we oversubscribed each topology by connecting 2 servers at each switch instead of just one. The results are shown in Fig. 4. Jellyfish's throughput is ∼119% of that of the closest competitor, the ring-based small world topology.

Path Length: Short path lengths are important to ensure low latency, and to minimize network utilization. In this context, we note that the theoretical *upper-bound* on the diameter of random regular graphs is fairly small: Bollobás and de la Vega [10] showed that in almost every r-regular graph with N nodes, the diameter is at most $1 + \lceil \log_{r-1}((2+\varepsilon)rN\log N)\rceil$ for any $\varepsilon > 0$. Thus, the server-to-server diameter is at most $3 + \lceil \log_{r-1}((2+\varepsilon)rN\log N)\rceil$. Thus, the path length increases logarithmically (base r) with the number of nodes in the network. Given the availability of commodity servers with large port counts, this rate of increase is very small in practice.

We measured path lengths using an all-pairs shortest-paths algorithm. The average path length (Fig. 5) in Jellyfish is much smaller than in the fat-tree[8]. For example, for RRG(3200, 48, 36) with 38,400 servers, the average path length between switches is <2.7 (Fig. 5), while the fat-tree's average is 3.71 at the smallest size, 3.96 at the size of 27,648 servers. Even though Jellyfish's diameter is 4 at the largest scale, the 99.99$^{\text{th}}$ percentile path-length across 10 runs did not exceed 3 for any size in Fig. 5.

[8]Note that the results in Fig. 5 use 48-port switches throughout, meaning that the only point of direct, fair comparison with a fat-tree is at the largest scale, where Jellyfish still compares favorably against a fat-tree built using 48-port switches and 27,648 servers.

4.2 Flexibility

Arbitrary-sized Networks: Several existing proposals admit only the construction of interconnects with very coarse parameters. For instance, a 3-level fat-tree allows only $k^3/4$ servers with k being restricted to the port-count of available switches, unless some ports are left unused. This is an arbitrary constraint, extraneous to operational requirements. In contrast, Jellyfish permits any number of racks to be networked efficiently.

Incremental Expandability: Jellyfish's construction makes it amenable to incremental expansion by adding either servers and/or network capacity (if not full-bisection bandwidth already), with increments as small as one rack or one switch. Jellyfish can be expanded such that rewiring is limited to the number of ports being added to the network; and the desirable properties are maintained: high bandwidth and short paths at low cost.

As an example, consider an expansion from an RRG(N, k, r) topology to RRG($N+1$, k, r). In other words, we are adding one rack of servers, with its ToR switch u, to the existing network. We pick a random link (v, w) such that this new ToR switch is not already connected with either v or w, remove it, and add the two links (u, v) and (u, w), thus using 2 ports on u. This process is repeated until all ports are filled (or a single odd port remains, which could be matched with another free port on an existing rack, used for a server, or left free). This completes incorporation of the rack, and can be repeated for as many new racks as desired.

A similar procedure can be used to expand network capacity for an under-provisioned Jellyfish network. In this case, instead of adding a rack with servers, we only add the switch, connecting all its ports to the network.

Jellyfish also allows for heterogeneous expansion: nothing in the procedure above requires that the new switches have the same number of ports as the existing switches. Thus, as new switches with higher port-counts become available, they can be readily used, either in racks or to augment the interconnect's bandwidth. There is, of course, the possibility of taking into account heterogeneity explicitly in the random graph construction and to improve upon what the vanilla random graph model yields. This endeavor remains future work for now.

We note that our expansion procedures (like our construction procedure) may not produce uniform-random RRGs. However, we demonstrate that the path length and capacity measurements of topologies we build incrementally match closely with ones constructed from scratch. Fig. 5 shows this comparison for the average path length and diameter where we start with an RRG with 1,200 servers and expand it incrementally. Fig. 6 compares the normalized throughput per server under a random permutation traffic model for topologies built incrementally against those built from scratch. The incremental topologies here are built by adding successive increments of 20 switches, and 80 servers to an initial topology also with 20 switches and 80 servers. (Throughout this experiment, each switch has 12 ports, 4 of which are attached to servers.) In each case, the results are close to identical.

Network capacity under expansion: Note that after normalizing by the number of servers $N(k-r)$, the lower bound on Jellyfish's normalized bisection bandwidth (§4.1) is independent of network size N. Of course, as N increases with fixed network degree r, average path length increases, and therefore, the demand for additional per-server capacity increases[9]. But since path length increases very slowly (as discussed above), bandwidth per server remains high even for relatively large factors of growth. Thus, operators can keep the servers-per-switch ratio constant even under large expansion, with minor bandwidth loss. Adding only switches (without servers) is another avenue for expansion which can preserve or even increase network capacity. Our below comparison with LEGUP uses both forms of expansion.

Comparison with LEGUP [14]: While a LEGUP implementation is not publicly available, the authors were kind enough to supply a series of topologies produced by LEGUP. In this expansion arc, there is a budget constraint for the initial network, and for each successive expansion step; within the constraint, LEGUP attempts to maximize network bandwidth, and also may keep some ports free in order to ease expansion in future steps. The initial network is built with 480 servers and 34 switches; the first expansion adds 240 more servers and some switches; and each remaining expansion adds only switches. To build a comparable Jellyfish network, at each expansion step, under the same budget constraints, (using the same cost model for switches, *cabling, and rewiring*) we buy and randomly cable in as many new switches as we can. The number of servers supported is the same as LEGUP at each stage.

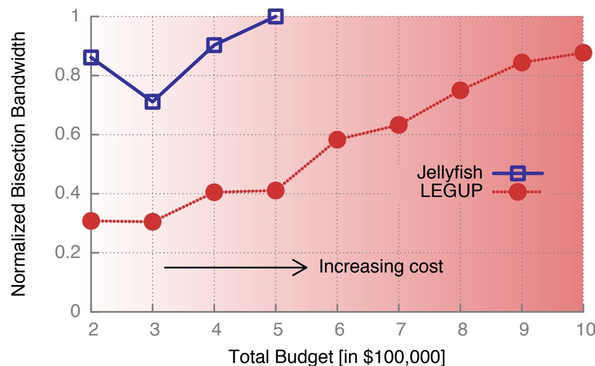


Figure 7: *Jellyfish's incremental expansion is substantially more cost-effective than LEGUP's Clos network expansion. With the same budget for equipment and rewiring at each expansion stage (x-axis), Jellyfish obtains significantly higher bisection bandwidth (y-axis). Results are averaged over* 10 *runs. (The drop in Jellyfish's bisection bandwidth from stage* 0 *to* 1 *occurs because the number of servers increases in that step.)*

LEGUP optimizes for bisection bandwidth, so we compare both LEGUP and Jellyfish on that metric (using code provided by the LEGUP authors [14]) rather than on our previous random permutation throughput metric. The results are shown in Fig. 7. Jellyfish obtains substantially higher bisection bandwidth than LEGUP at each stage. In fact, by stage 2, Jellyfish has achieved higher bisection bandwidth than LEGUP in stage 8, meaning (based on each stage's cost) that Jellyfish builds an equivalent network at cost 60% lower than LEGUP.

A minority of these savings is explained by the fact that Jellyfish is more bandwidth-efficient than Clos networks, as exhibited by our earlier comparison with fat-trees. But in addition, LEGUP appears to pay a significant cost to enable it to incrementally-expand a Clos topology; for example, it leaves some ports unused in order to ease expansion in later stages. We conjecture that to some extent, this greater incremental expansion cost is fundamental to Clos topologies.

[9]This discussion also serves as a reminder that bisection-bandwidth, while a good metric of network capacity, is not the same as, say, capacity under worst-case traffic patterns.

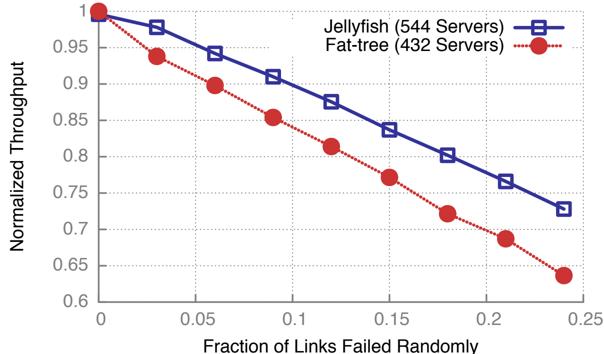


Figure 8: *Jellyfish is highly resilient to failures: Normalized throughput per server decreases more gracefully for Jellyfish than for a same-equipment fat-tree as the percentage of failed links increases. Note that the y-axis starts at* 60% *throughput; both topologies are highly resilient to failures.*

4.3 Failure Resilience

Jellyfish provides good path redundancy; in particular, an r-regular random graph is almost surely r-connected [9]. Also, the random topology maintains its (lack of) structure in the face of link or node failures – a random graph topology with a few failures is just another random graph topology of slightly smaller size, with a few unmatched ports on some switches.

Fig. 8 shows that the Jellyfish topology is even more resilient than the same-equipment fat-tree (which itself is no weakling). Note that the comparison features a fat-tree with fewer servers, but the same cost. This is to justify Jellyfish's claim of supporting a larger number of servers using the same equipment as the fat-tree, in terms of capacity, path length, and *resilience* simultaneously.

5 Routing & Congestion Control

While the above experiments establish that Jellyfish topologies have high capacity, it remains unclear whether this potential can be realized in real networks. There are two layers which can affect performance in real deployments: routing and congestion control. In our experiments with various combinations of routing and congestion control for Jellyfish (§5.1), we find that standard ECMP does not provide enough path diversity for Jellyfish, and to utilize the entire capacity we need to also use longer paths. We then provide in-depth results for Jellyfish's throughput and fairness using the best setting found earlier—k-shortest paths and multipath TCP (§5.2). Finally, we discuss practical strategies for deploying k-shortest-path routing (§5.3).

5.1 ECMP is not enough

Evaluation methodology: We use the simulator developed by the MPTCP authors for both Jellyfish and fat-tree. For routing, we test: (a) ECMP (equal cost multipath routing; We used 8-way ECMP, but 64-way ECMP does not perform much better, see Fig. 9), a standard strategy to distribute flows over shortest paths; and (b) k-shortest paths routing, which could be useful for Jellyfish because it can utilize longer-than-shortest paths. For k-shortest paths, we use Yen's Loopless-Path Ranking algorithm [2, 46] with $k = 8$ paths. For congestion control, we test standard TCP (1 or 8 flows per server pair) and the recently proposed multipath TCP (MPTCP) [44], using the recommended value of 8 MPTCP subflows. The traffic model continues to be a random permutation at the server-level, and as before, for the fat-tree comparisons, we build Jellyfish using the same switching equipment as the fat-tree.

Summary of results: Table 1 shows the average per server throughput as a percentage of the servers' NIC rate for two sample Jellyfish and fat-tree topologies under different routing and load balancing schemes. We make two observations: (1) ECMP performs poorly for Jellyfish, not providing enough path diversity. For random permutation traffic, Fig. 9 shows that about 55% of links are used by no more than 2 paths under ECMP; while for 8-shortest path routing, the number is 6%. Thus we need to make use of k-shortest paths. (2) Once we use k-shortest paths, each congestion control protocol works as least as well for Jellyfish as for the fat-tree.

The results of Table 1 depend on the oversubscription level of the network. In this context, we attempt to match fat-tree's performance given the routing and congestion control inefficiencies. We found that Jellyfish's advantage slightly reduces in this context compared to using idealized routing as before: In comparison to the same-equipment fat-tree (686 servers), now we can support, at same or higher performance, 780 servers (*i.e.*, 13.7% more that the fat-tree) with TCP, and 805 servers (17.3% more) with MPTCP. With ideal routing and congestion control, Jellyfish could support 874 servers (27.4% more). However, as we show quantitatively in §5.2, Jellyfish's advantage improves significantly with scale. At the largest scale we could simulate, Jellyfish supports 3,330 servers to the fat-tree's 2,662 — a $>$ 25% improvement (after accounting for routing and congestion control inefficiencies).

5.2 k-Shortest-Paths With MPTCP

The above results demonstrate using one representative set of topologies that using k-shortest paths with MPTCP yields higher performance than ECMP/TCP. In this sec-

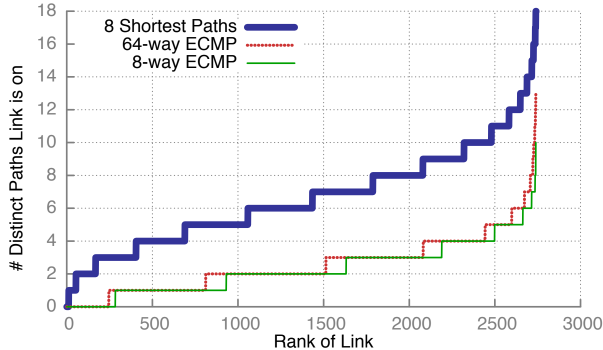


Figure 9: *ECMP does not provide path diversity for Jellyfish: Inter-switch link's path count in ECMP and k-shortest-path routing for random permutation traffic at the server-level on a typical Jellyfish of* 686 *servers (built using the same equipment as a fat-tree supports* 686 *servers). For each link, we count the number of distinct paths it is on. Each network cable is considered as two links, one for each direction.*

Congestion control	Fat-tree (686 svrs) ECMP	Jellyfish (780 svrs) ECMP	Jellyfish (780 svrs) 8-shortest paths
TCP 1 flow	48.0%	57.9%	48.3%
TCP 8 flows	92.2%	73.9%	92.3%
MPTCP 8 subflows	93.6%	76.4%	95.1%

Table 1: *Packet simulation results for different routing and congestion control protocols for Jellyfish (*780 *servers) and a same-equipment fat-tree (*686 *servers). Results show the normalized per server average throughput as a percentage of servers' NIC rate over* 5 *runs. We did not simulate the fat-tree with* 8*-shortest paths because ECMP is strictly better, and easier to implement in practice for the fat-tree.*

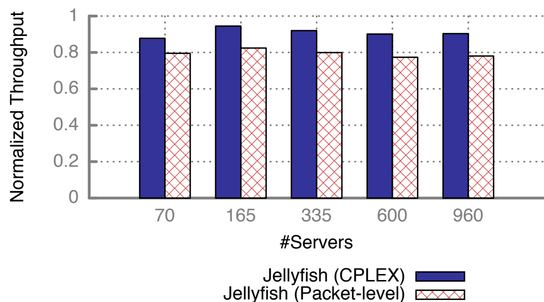


Figure 10: *Simple k-shortest path forwarding with MPTCP exploits Jellyfish's high capacity well: We compare the throughput using the same Jellyfish topology with both optimal routing, and our simple routing mechanism using MPTCP, which results in throughput between* 86% − 90% *of the optimal routing in each case. Results are averaged over* 10 *runs.*

tion we measure the efficiency of k-shortest path routing with MPTCP congestion control against the optimal performance (presented in §4), and later make comparisons against fat-trees at various sizes.

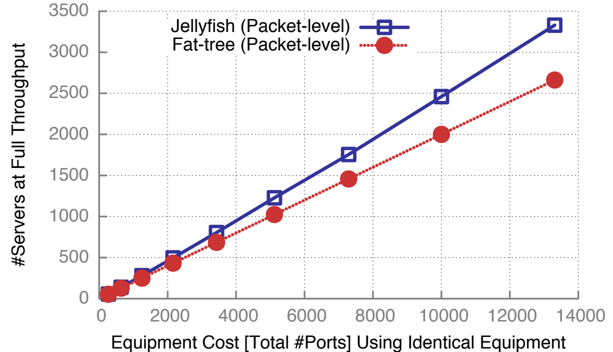


Figure 11: *Jellyfish supports a larger number of servers (>25% at the largest scale shown, with an increasing trend) than the same-equipment fat-tree at the same (or higher) throughput, even with inefficiencies of routing and congestion control accounted for. Results are averages over* 20 *runs for topologies smaller than* 1,400 *servers, and averages over* 10 *runs for larger topologies.*

Routing and Congestion Control Efficiency: The result in Fig. 10 shows the gap between the optimum performance, and the performance realized with routing and congestion control inefficiencies. At each size, we use the same slightly oversubscribed[10] Jellyfish topology for both setups. In the worst of these comparisons, Jellyfish's packet-level throughput is at ∼86% of the CPLEX optimal throughput. (In comparison, the fat-tree's throughput under MPTCP/ECMP is 93-95% of its optimum.) There is a possibility that this gap can be closed using smarter routing schemes, but nevertheless, as we discuss below, Jellyfish maintains most of its advantage over the fat-tree in terms of the number of servers supported at the the same throughput.

Fat-tree Throughput Comparison: To compare Jellyfish's performance against the fat-tree, we first find the average per-server throughput a fat-tree yields in the packet simulation. We then find (using binary search) the number of servers for which the average per-server throughput for the comparable Jellyfish topology is either the same, or higher than the fat-tree; this is the same methodology applied for Table 1. We repeat this exercise for several fat-tree sizes. The results (Fig. 11) are similar to those in Fig. 2(c), although the gains of Jellyfish are reduced marginally due to routing and congestion control inefficiencies. Even so, at the maximum scale of our experiment, Jellyfish supports 25% more servers than the fat-tree (3,330 in Jellyfish, versus 2,662 for the fat-tree). We note however, that even at smaller scale (for instance, 496 servers in Jellyfish, to 432 servers in the fat-tree) the improvement can be as large as ∼15%.

[10] An undersubscribed network may simply show 100% throughput, masking some of the routing and transport inefficiency.

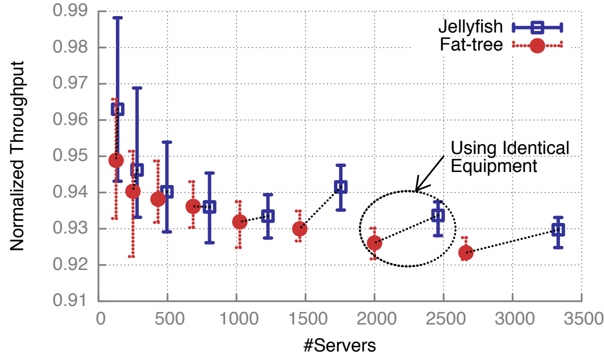


Figure 12: *The packet simulation's throughput results for Jellyfish show similar stability as the fat-tree. (Note that the y-axis starts at* 91% *throughput.) Average, minimum and maximum throughput-per-server values are shown. The data plotted is from the same experiment as Fig. 11. Jellyfish has the same or higher average throughput as the fat-tree while supporting a larger number of servers. Each Jellyfish data-point uses equipment identical to the closest fat-tree data-point to its left (as highlighted in one example).*

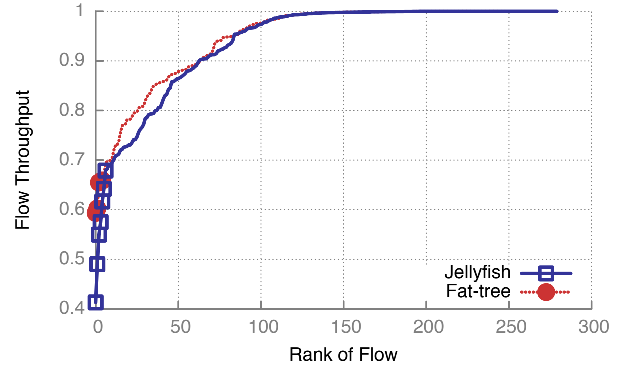


Figure 13: *Both Jellyfish and the fat-tree show good flow-fairness: The distribution of normalized flow throughputs in Jellyfish and fat-tree is shown for one typical run. After the few outliers (shown with points), the plot is virtually continuous (the line). Note that Jellyfish has more flows because it supports a higher number of servers (at same or higher* per-server *throughput). Jain's fairness index for both topologies is* ∼99%.

We show in Fig. 12, the stability of our experiments by plotting the average, minimum and maximum throughput for both Jellyfish and the fat-tree at each size, over 20 runs (varying both topologies and traffic) for small sizes and 10 runs for sizes >1,400 servers.

Fairness: We evaluate how flow-fair the routing and congestion control is in Jellyfish. We use the packet simulator to measure each flow's throughput in both topologies and show in Fig. 13, the normalized throughput per flow in increasing order. Note that Jellyfish has a larger number of flows because we make all comparisons using the same network equipment and the larger number of servers supported by Jellyfish. Both the topologies have similarly good fairness; Jain's fairness index [25] over these flow throughput values for both topologies: 0.991 for the fat-tree and 0.988 for Jellyfish.

5.3 Implementing *k*-Shortest-Path Routing

In this section, we discuss practical possibilities for implementing k-shortest-paths routing. For this, each switch needs to maintain a routing table containing for each other switch, k shortest paths.

OpenFlow [30]: OpenFlow switches can match end-to-end connections to routing rules, and can be used for routing flows along pre-computed k-shortest paths. Recently, Devoflow [35] showed that OpenFlow rules can be augmented with a small set of local routing actions for randomly distributing load over allowed paths, without invoking the OpenFlow controller.

SPAIN [36]: SPAIN allows multipath routing by using VLAN support in commodity off-the-shelf switches. Given a set of pre-computed paths, SPAIN merges these paths into multiple trees, each of which is mapped to a separate VLAN. SPAIN supports arbitrary topologies, and can enable use of k-shortest path routing in Jellyfish.

MPLS [40]: One could set up MPLS tunnels between switches such that all the pre-computed k-shortest paths between a switch pair are configured to have the same cost. This would allow switches to perform standard equal-cost load balancing across paths.

6 Physical Construction and Cabling

Key considerations in data center wiring include:

- **Number of cables:** Each cable represents both a material and a labor cost.
- **Length of cables:** The cable price/meter is $5-6 for both electrical and optical cables, but the cost of an optical transceiver can be close to $200 [37]. We limit our interest in cable length to whether or not a cable is short enough, i.e., <10 meters in length [17, 26], for use of an electrical cable.
- **Cabling complexity:** Will Jellyfish awaken the dread spaghetti monster? Complex and irregular cabling layouts may be hard to wire and thus susceptible to more wiring errors. We will consider whether this is a significant factor. In addition, we attempt to design layouts that result in aggregation of cables in bundles, in order to reduce manual effort (and hence, expense) for wiring.

In the rest of this section, we first address a common concern across data center deployments: handling wiring

errors (§6.1). We then investigate cabling Jellyfish in two deployment scenarios, using the above metrics (number, length and complexity of cabling) to compare against cabling a fat-tree network. The first deployment scenario is represented by small clusters (∼1,000 servers); in this category we also include the intra-container clusters for 'Container Data Centers' (CDC)[11] (§6.2). The second deployment scenario is represented by massive-scale data centers (§6.3). In this paper we only analyze massive data centers built using containers, leaving more traditional data center layouts to future work.[12]

6.1 Handling Wiring Errors

We envision Jellyfish cabling being performed using a blueprint automatically generated based on the topology and the physical data center layout. The blueprint is then handed to workers to connect cables manually.

While some human errors are inevitable in cabling, these are easy to detect and fix. Given Jellyfish's sloppy topology, a small number of miswirings may not even require fixing in many cases. Nevertheless, we argue that fixing miswirings is relatively inexpensive. For example, the labor cost of cabling is estimated at ∼10% of total cabling cost [37]. With a pessimistic estimate where the total cabling cost is 50% of the network cost, the cost of fixing (for example) 10% miswirings would thus be just 0.5% of the network cost. We note that wiring errors can be detected using a link-layer discovery protocol [32].

6.2 Small Clusters and CDCs

Small clusters and CDCs form a significant section of the market for data centers, and thus merit separate consideration. In a 2011 survey [15] of 300 US enterprises (with revenues ranging from $1B-$40B) which operate data centers, 57% of data centers occupy between 5,000 and 15,000 square feet; and 75% have a power load <2MW, implying that these data centers house a few thousand servers [13]. As our results in §4.1 show, even at a few hundred servers, cost-efficiency gains from Jellyfish can be significant (∼20% at 1,000 servers). Thus, it is useful to deploy Jellyfish in these scenarios.

We propose a cabling optimization (along similar lines as the one proposed in [6]). The key observation is that in a high-capacity Jellyfish topology, there are more than twice as many cables running between switches than from servers to switches. Thus, placing all the switches in close proximity to each other reduces cable length, as well as manual labor. This also simplifies the small amounts of rewiring necessary for incremental expansion, or for fixing wiring errors.

Number of cables: Requiring fewer network switches for the same server pool also implies requiring fewer cables ($15-20\%$ depending on scale) than a fat-tree. This also implies that there is more room (and budget) for packing more servers in the same floor space.

Length of cables: For small clusters, and inside CDC containers using the above optimization, cable lengths are short enough for electrical cables without repeaters.

Complexity: For a few thousand servers, space equivalent to 3-5 standard racks can accommodate the switches needed for a full bisection bandwidth network (using available 64-port switches). These racks can be placed at the physical center of the data center, with aggregate cable bundles running between them. From this 'switch-cluster', aggregate cables can be run to each server-rack. With this plan, manual cabling is fairly simple. Thus, the nightmare cable-mess image a random graph network may bring to mind is, at best, alarmist.

A unique possibility allowed by the assembly-line nature of CDCs, is that of fabricating a random-connect *patch panel* such that workers only plug cables from the switches into the panel in a regular easy-to-wire pattern, and the panel's internal design encodes the random interconnect. This could greatly accelerate manual cabling.

Whether or not a patch panel is used, the problems of layout and wiring need to be solved only *once* at design time for CDCs. With a standard layout and construction, building automated tools for verifying and detecting miswirings is also a one-time exercise. Thus, the cost of any additional complexity introduced by Jellyfish would be amortized over the production of many containers.

Cabling under expansion: Small Jellyfish clusters can be expanded by leaving enough space near the 'switch-cluster' for adding switches as servers are added at the periphery of the network. In case no existing switch-cluster has room for additional switches, a new cluster can be started. Cable aggregates run from this new switch-cluster to all new server-racks and to all other switch-clusters. We note that for this to work with only electrical cabling, the switch-clusters need to be placed within 10 meters of each other as well as the servers. Given the constraints the support infrastructure already places on such facilities, we do not expect this to be a significant issue.

As discussed before, the Jellyfish expansion procedure requires a small amount of rewiring. The addition of every two network ports requires two cables to be moved (or equivalently, one old cable to be disconnected and two new cables to be added), since each new port will

[11] As early as 2006, The Sun Blackbox [3] promoted the idea of using shipping containers for data centers. There are also new products in the market exploiting similar physical design ideas [4, 5, 24].

[12] The use of container-based data centers seems to be an industry trend, with several players, Google and Microsoft included, already having container-based deployments [17].

be connected to an existing port. The cables that need to be disconnected and the new cables that need to be attached can be automatically identified. Note that in the 'switch-cluster' configuration, all this activity happens at one location (or with multiple clusters, only between these clusters). The only cables not at the switch-cluster are the ones between the new switch and the servers attached to it (if any). This is just *one* cable aggregate.

We note that the CDC usage may or may not be geared towards incremental expansion. Here the chief utility of Jellyfish is its efficiency and reliability.

6.3 Jellyfish in Massive-Scale Data Centers

We now consider massive scale data centers built by connecting together multiple containers of the type described above. In this setting, as the number of containers grows, most Jellyfish cables are likely to be between containers. Therefore, inter-container cables in turn require expensive optical connectors, and Jellyfish can result in excessive cabling costs compared to a fat-tree.

However, we argue that Jellyfish can be adapted to wire massive data centers with lower cabling cost than a fat-tree, while still achieving higher capacity and accommodating a larger number of servers. For cabling the fat-tree in this setting, we apply the layout optimization suggested in [6], *i.e.,* make each fat-tree 'pod' a container, and divide the core-switches among these pods equally. With this physical structure, we can calculate the number of intra-container cables (from here on referred to as 'local') and inter-container cables ('global') used by the fat-tree. We then build a Jellyfish network placing the same number of switches as a fat-tree pod in a container, and using the same number of containers. The resulting Jellyfish network can be seen as a 2-layered random graph—a random graph within each container, and a random graph between containers. We vary the number of local and global connections to see how this affects performance in relation to the unrestricted Jellyfish network.

Note that with the same switching equipment as the fat-tree, Jellyfish networks would be overprovisioned if we used the same numbers of servers. To make sure that any loss of throughput due to our cable-optimization is clearly visible, we add a larger number of servers per switch to make Jellyfish oversubscribed.

Fig. 14 plots the capacity (average server throughput) achieved for 4 sizes[13] of 2-layer Jellyfish, as we vary the number of local and global connections, while keeping the total number of connections constant for a topology. Throughput is normalized to the corresponding unrestricted Jellyfish. The throughput drops by <6% when

[13] These are very far from massive scale, but these simulations are directed towards observing general trends. Much larger simulations are beyond our simulator's capabilities.

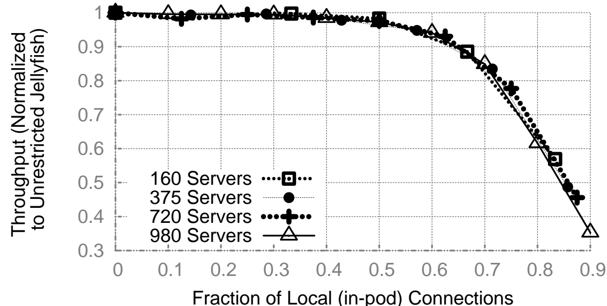


Figure 14: *Localization of Jellyfish random links is a promising approach to tackle cabling for massive scale data centers: As links are restricted to be more and more local, the network capacity decreases (as expected). However, when* 50% *of the random links for each switch are constrained to remain inside the pod, there is* <3% *loss of throughput.*

60% of the network connections per switch are 'localized'. The percentage of local links for the equivalent fat-tree is 53.6%. Thus, Jellyfish can achieve a higher degree of localization, while still having a higher capacity network; recall that Jellyfish is 27% more efficient than the fat-tree at the largest scale (§4.1). The effect of cable localization on throughput was similar across the sizes we tested. For the fat-tree, the fraction of local links (conveniently given by $0.5(1+1/k)$ for a fat-tree built with k-port switches) *decreases* marginally with size.

Complexity: Building Jellyfish over switches distributed uniformly across containers will, with high probability, result in cable assemblies between every pair of containers. A 100,000 server data center can be built with ∼40 containers. Even if *all* ports (except those attached to servers) from each switch in each container were connected to other containers, we could aggregate cables between each container-pair leaving us with ∼800 such cable assemblies, each with fewer than 200 cables. With the external diameter of a 10GBASE-SR cable being only 245um, each such assembly could be packed within a pipe of radius <1cm. Of course, with higher over-subscription at the inter-container layer, these numbers could decrease substantially.

Cabling under expansion: In massive-scale data centers, expansion can occur through addition of new containers, or expansion of containers (if permissible). Laying out spare cables together with the aggregates between containers is helpful in scenarios where a container is expanded. When a new container is added, new cable aggregates must be laid out to every other container. Patch panels can again make this process easier by exposing the ports that should be connected to the other containers.

7 Conclusion

We argue that random graphs are a highly flexible architecture for data center networks. They represent a novel approach to the significant problems of incremental and heterogeneous expansion, while enabling high capacity, short paths, and resilience to failures and miswirings.

We thank Chandra Chekuri, Indranil Gupta, Gianluca Iannaccone, Steven Lumetta, Sylvia Ratnasamy, Marc Snir, and the anonymous reviewers for helpful comments; Andy Curtis for providing code for bisection bandwidth calculation, and sharing LEGUP topologies for comparison; and the MPTCP authors for sharing their simulator. This work was supported in part by National Science Foundation grant CNS 10-40396.

References

[1] CPLEX Linear Program Solver. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[2] An implementation of k-shortest path algorithm. http://code.google.com/p/k-shortest-paths/.

[3] Project blackbox. http://www.sun.com/emrkt/blackbox/story.jsp.

[4] Rackable systems. ICE Cube modular data center. http://www.rackable.com/products/icecube.aspx.

[5] SGI ICE Cube Air expandable line of modular data centers. http://sgi.com/products/data_center/ice_cube_air.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[7] L. N. Bhuyan and D. P. Agrawal. Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on Computers*, 1984.

[8] B. Bollobás. The isoperimetric number of random regular graphs. *Eur. J. Comb.*, 1988.

[9] B. Bollobás. Random graphs, 2nd edition. 2001.

[10] B. Bollobás and W. F. de la Vega. The diameter of random regular graphs. In *Combinatorica 2*, 1981.

[11] A. Broder and E. Shamir. On the second eigenvalue of random regular graphs. In *FOCS*, 1987.

[12] F. Comellas and C. Delorme. The (degree, diameter) problem for graphs. http://maite71.upc.es/grup_de_grafs/table_g.html/.

[13] A. R. Curtis, T. Carpenter, M. Elsheikh, A. Lopez-Ortiz, and S. Keshav. REWIRE: an optimization-based framework for unstructured data center network design. In *INFOCOM*, 2012.

[14] A. R. Curtis, S. Keshav, and A. Lopez-Ortiz. LEGUP: using heterogeneity to reduce the cost of data center network upgrades. In *ACM CoNEXT*, 2010.

[15] Digital Reality Trust. What is driving the us market? http://goo.gl/qiaRY, 2001.

[16] Facebook. Facebook to expand Prineville data center. http://goo.gl/fJAoU.

[17] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM*, 2010.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[19] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[20] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[21] L. Gyarmati and T. A. Trinh. Scafida: A scale-free network inspired data center architecture. In *SIGCOMM CCR*, 2010.

[22] J. Hamilton. Datacenter networks are in my way. http://goo.gl/Ho6mA.

[23] HP. HP EcoPOD. http://goo.gl/8A0Ad.

[24] HP. Pod 240a data sheet. http://goo.gl/axHPp.

[25] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, Digital Equipment Corporation, 1984.

[26] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH*, 2008.

[27] F. T. Leighton. Introduction to parallel algorithms and architectures: Arrays, trees, hypercubes. 1991.

[28] A. Licis. Data center planning, design and optimization: A global perspective. http://goo.gl/Sfydq.

[29] B. D. McKay and N. C. Wormald. Uniform generation of random regular graphs of moderate degree. *J. Algorithms*, 1990.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR*, 2008.

[31] A. B. Michael, M. Nolle, and G. Schreiber. A message passing model for communication on random regular graphs. In *International Parallel Processing Symposium (IPPS)*, 1996.

[32] Microsoft. Link layer topology discovery protocol. http://goo.gl/bAcZ5.

[33] R. Miller. Facebook now has 30,000 servers. http://goo.gl/EGD2D.

[34] R. Miller. Facebook server count: 60,000 or more. http://goo.gl/79J4.

[35] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. DevoFlow: cost-effective flow management for high performance enterprise networks. In *Hotnets*, 2010.

[36] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies. In *NSDI*, 2010.

[37] J. Mudigonda, P. Yalagandula, and J. Mogul. Taming the flying cable monster: A topology design and optimization framework for data-center networks. 2011.

[38] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[39] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A cost comparison of datacenter network architectures. In *ACM CoNEXT*, 2010.

[40] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, 2001.

[41] J.-Y. Shin, B. Wong, and E. G. Sirer. Small-world datacenters. *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[42] A. Singla, A. Singh, K. Ramachandran, L. Xu, and Y. Zhang. Proteus: a topology malleable data center network. In *HotNets*, 2010.

[43] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time optics in data centers. In *SIGCOMM*, 2010.

[44] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for Multipath TCP. In *NSDI*, 2011.

[45] H. Wu, G. Lu, D. Li, C. Guo, and Y. Zhang. MDCube: a high performance network structure for modular data center interconnection. In *ACM CoNEXT*, 2009.

[46] J. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 1971.