Jellyfish: Networking Data Centers Randomly

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

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: A Jellyfish interconnect built using the same equipment as a fat-tree, supports as many as 25% more servers at full capacity 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 and evaluate approaches that resolve these challenges effectively, indicating that Jellyfish could be deployed in today's data centers.

1 Introduction

Data centers today form the backbone of cloud operations. A well provisioned data center network is important to ensure that servers do not face bandwidth bottlenecks to utilization; to 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 [20]. As a result, a significant body of work has tackled the problem of building high network capacity interconnects [5, 15–18, 33, 37, 39].

One crucial problem that these designs encounter is incremental expansion of the network, *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-intensive applications. Expansion within a data center is possible through either planned overprovisioning of space and power, or by upgrading old servers to a larger number of more powerful and, at the same time, less power-consuming new servers.

[*]Part of this work was published at HotCloud'11 [36].

Industry experience indicates that incremental expansion is an important problem. Consider the growth of Facebook's data center server population from roughly 30,000 in November 2009 to more than 60,000 by June 2010 [31]. While Facebook has added entirely new data center facilities too, much of this growth involves incrementally expanding existing facilities by "adding capacity on a daily basis" [30]. For instance, Facebook announced that it will double the size of its facility at Prineville, Oregon by early 2012 [14]. A 2011 survey [38] of 300 enterprises that run data centers of a variety of sizes found that "The need to expand data center capacity will continue into 2012 with 84% of firms definitely/probably expanding their operations in 2012." Industry experts have also identified incremental build-out as a useful strategy to reduce up-front capital expenditure [26]. Several industry products advertise incremental expandability of the server pool, including SGI's IceCube (marketed as "The Expandable Modular Data Center" [4]; expands 4 racks at a time) and HP's EcoPod [22] (a "pay-as-you-grow" enabling technology [21]). However, in both cases, no mention is made of how the network supports such expansion of the server pool.

Do current high-bandwidth data center network proposals allow incremental growth? Consider the fat-tree interconnect, as proposed in [5]), as an illustrative example. The entire structure is completely determined by the port-count k 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 be 3,602 servers. Moreover, this "incremental" growth would require replacing all the 48-port switches by 50-port switches. There is, of course, the possibility of making localized changes like replacing a switch with one with larger port count; however, this necessarily makes capacity distribution unfair across the server pool. The only prior work [13] that di-

[1]Other topologies have similar problems: a hypercube [7] allows only power-of-2 sizes, a de Bruijn-like construction [34] allows only power-of-3 sizes, etc.

rectly addresses the problem of incremental expansion, attempts to make the most out of this bad situation – it searches for optimum additions of network equipment to Clos networks. In contrast, we design for expansion, resulting (as we show in §4.2) in significant gains in network capacity for the same (expanding) data center under the same budgetary constraints.

An alternative approach suggested in the literature [18], is based on leaving free ports for future network connections. But the cost of these free ports, is an unnecessary sunk investment for the period in which the network does not expand. 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 interconnect. The proposed interconnect, which we call **Jellyfish**, is a *degree-bounded 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 [5, 16–18, 33, 37]. Jellyfish also allows construction of arbitrary-size networks, unlike past proposals discussed above which limit the network to very coarse design points dictated by their structure.

Somewhat surprisingly, Jellyfish supports *more* servers at full bisection bandwidth with lower mean path length than a fat-tree [5] built using the same network equipment. In addition, as we discuss later, Jellyfish is resilient to failures and miswirings during construction.

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 indicate that Jellyfish can be effectively deployed in today's data centers.

Our key contributions are as follows:

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

- We conduct a comparative study of the bandwidth of several proposed data center network topologies. We find that Jellyfish can support 25% more servers than a fat-tree while using the same switch equipment and providing at least as high bisection bandwidth, and this advantage increases with network size. Moreover, we propose *degree-diameter optimal graphs* as candidate benchmark topologies and show that Jellyfish remains within 14% of these carefully-optimized topologies.

- We demonstrate in packet-level simulations that Jellyfish's bandwidth can be effectively utilized via a practical (indeed, already implemented!) technique, multipath TCP [40] — despite the lack of regular structure that is sometimes used to ease routing in other topologies.

- We demonstrate that Jellyfish provides quantitatively easier incremental expansion than prior work on incremental expansion in Clos networks [13], growing incrementally to a slightly higher capacity network at only 40% of the expense of [13].

- We discuss effective techniques to realize physical layout and cabling of Jellyfish in various deployment scenarios. Jellyfish may require higher cabling cost if cables are on average longer than those of a fat-tree; but when we restrict Jellyfish to use cables of similar length as the fat-tree, it still improves on the fat-tree's bisection bandwidth.

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 under simple routing and congestion control mechansims (§5). We discuss effective cabling schemes and physical construction of Jellyfish in various deployment scenarios (§6), and conclude (§7).

2 Related Work

Several recent data center network proposals for high-capacity networks appropriate special structure for topology and routing. These include folded-Clos (or fat-tree) based designs [5, 16, 33], several designs based on using servers for forwarding [17, 18, 41], and designs using optical networking technology [15, 39]. High performance computing literature has also studied carefully-structured expander graphs [25].

However, none of these architectures address the issue of *incremental expansion* of the network. For some of these (like the fat-tree, for instance), adding servers while preserving the structural properties would require replacing a large number of network elements and extensive rewiring. MDCube [41] allows expansion at a very coarse rate (several thousand servers). DCell and BCube [17, 18] allow expansion to an *a priori* known target size, but require servers with free ports reserved for planned future expansion.

While two recent proposals, Scafida [19] (based on scale-free graphs) and Small-World Datacenters (SWDC) [35] [2], also, employ randomness like Jellyfish, ours is a substantially different random topology which lacks correlation (i.e., structure) among edges. Such structure can cause problems with incremental expansion because it makes it unclear whether the topology retains its characteristics on 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 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 topologies have higher network capacities than SWDC topologies built using the same equipment.

LEGUP [13] directly attacks the problem of expansion by attempting to find the optimal upgrades for a Clos network. 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.

In a very recent (August 2011) technical report, Curtis et al. propose REWIRE [12], a heuristic optimization-based method to find high capacity topologies with a given cost budget, taking into account length-varying cable cost. While [12] compares with random graphs, their experiments are very restricted (in both the assumptions made, and the scenarios evaluated), and their results comparing REWIRE with random graphs are inconclusive[4]. Unfortunately, due to the recency of this work, we have to leave a direct quantitative comparison to future work. We note, however, that in §4.2 we do compare against the authors' previous optimization tool, LEGUP [13]; REWIRE has not yet been quantitatively compared against LEGUP.

Random graphs have been examined in the context of communication networks [28] previously. The contribution of our work lies in applying random graphs to allow incremental expansion in data center networks, and in quantifying the efficiency gains such graphs bring over traditional data center topologies.

3 Jellyfish Topology

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: for all i, $k = k_i$ and $r = r_i$. We let N be the number of racks, so 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.

Why should this work? Intuitively, random regular graphs (sampled uniform-randomly from the space of all r-regular graphs) fulfill two key goals. First, they are very efficient: theoretical results show that almost every RRG has low diameter and high bisection bandwidth [8, 10]. Second, they are highly flexible: they can be built with any number of nodes, are naturally extensible to heterogeneous degree distributions, and as we describe, are easy to modify incrementally.

Construction: Formally, RRGs are sampled uniformly from the space of all r-regular graphs. This is a complex problem in graph theory [27]; however, a simple procedure can produce a "sufficiently uniform" random graph which empirically gives us the desired bisection bandwidth and path length distribution. One can simply pick a random pair of nodes with free ports (preferring node-pairs that are not already neighbors), join them with an edge, and repeat until no further edges can be added. If a rack remains with $\geq$ 2 free ports, or if a new rack is

[2] As evidenced by our HotCloud 2011 workshop paper, this work, to appear in SOCC 2011, in October, has been done in parallel to ours.

[3] For instance, using a $2D$-Torus as the lattice implies that maintaining the network structure when expanding an n node network, requires addition of $2\sqrt{n} - 1$ new nodes. The higher the dimensionality of the lattice, the more complicated expansion becomes.

[4] Results in [12] 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, [12] shows random graphs that are disconnected. These significant discrepancies could arise from: (a) [12] 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; (b) evaluating very low bisection bandwidths (**0.04** to 0.37) – at the highest bisection bandwidth evaluated, [12] indicates the random graph, in fact, has *higher* throughput than REWIRE; and (c) separating network port costs from cable costs, resulting in the random graph ending up with too many ports and too few cables to connect them.

added to an existing network, these can be incorporated by removing a random existing link, and linking its endpoints to two free ports. Thus only a single unmatched port might remain across the whole datacenter.

Using the above idea, we generate a topology blueprint for the physical interconnection. We do not suggest allowing human operators to "wire at will", as this may result in poor topologies due to the inherent bias involved (for instance, favoring shorter cables over longer ones). We discuss cabling later in §6.

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 $>86\%$ of the best-known degree-diameter graphs, which we consider benchmark topologies for high capacity at low cost.
- 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 topologies produces topologies identical in throughput and path length characteristics to Jellyfish topologies generated from scratch.
- Jellyfish provides a significant cost-efficiency advantage over prior work (LEGUP [13]) 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 $< 16\%$.

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 other 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[5]. Nevertheless, evaluating other traffic patterns is an important question that we leave for future work. We also note that in a study of several traffic patterns in fat-trees [5], random permutation was of intermediate difficulty among the patterns evaluated, with several patterns producing lower or higher throughput.

Given a traffic matrix, we calculate optimal routing by treating flows as splittable and fluid. This allows us to characterize a topology's raw capacity. The calculation corresponds to a standard multi-commodity network flow problem, which we solve using the CPLEX linear program solver.

For all throughput comparisons, we use the *same switching equipment* (in terms of both number of switches, and ports on each switch) for each pair of topologies compared. Throughput results are always normalized to values between 0 and 1, and averaged over all flows.

For comparisons with the full bisection fat-tree topology, we attempt to find, using a binary search procedure, the 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

Capacity: Bisection bandwidth, denoted by B, is a common measure of network capacity. It measures the *worst-case* bandwidth between two equal-size partitions of the network. This can be normalized to a value between 0 and 1 by dividing it by the total line-rate bandwidth of the servers in one partition.

Jellyfish is a more bandwidth-efficient topology than a fat-tree. Fig. 1(a) shows that to support a given number of servers (x axis) with full bisection bandwidth ($B = 1$), Jellyfish uses significantly fewer switches. For instance,

[5]The flexibility provided by a network which permits such network-oblivious VM placement without a performance penalty is a highly desirable characteristic [20].

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 (as in Fig. 1(a)) or cutting costs by using fewer switches.

Fig. 1(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 [5]) 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.)

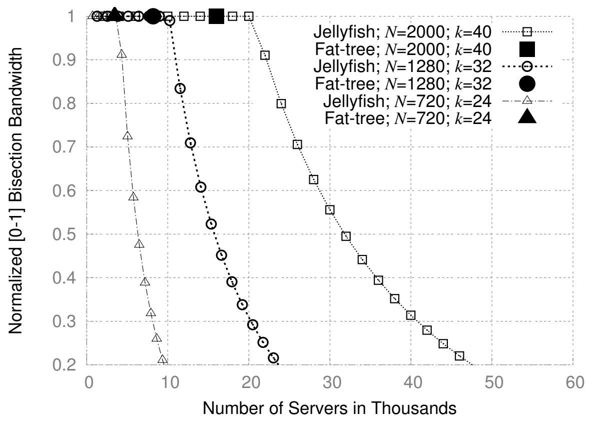
The numbers in Fig. 1(b) and 1(c) are computed by explicitly setting parameters for the fat-tree, and for Jellyfish, by using a lower bound of Bollobás [8]: in almost every r-regular graph with N nodes, every set of $u \leq N/2$ nodes is joined by at least $N(\frac{r}{4} - \frac{\sqrt{r \ln 2}}{2})$ edges to the rest of the graph. Thus, the bisection bandwidth B for RRG(N, k, r) is *at least*

$$\min\left(\frac{N(\frac{r}{4} - \frac{\sqrt{r\ln 2}}{2})}{N(k-r)/2}, 1\right) = \min\left(\frac{r/2 - \sqrt{r\ln 2}}{k-r}, 1\right).$$

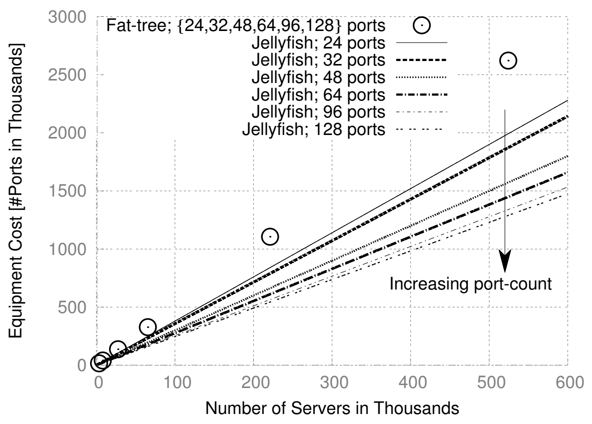
Fig. 1(c) uses the random-permutation traffic model to find the number of servers Jellyfish can support at full capacity as the fat-tree using identical switching equipment. The improvement is as much as 27% more servers than the fat-tree at the largest size (874 servers) we can use CPLEX to evaluate. Also, as with bisection bandwidth, the trend indicates that this improvement increases with scale.

Comparison with Degree-Diameter Graphs: Another capacity comparison we make for Jellyfish, is that with the best known degree-diameter graphs. In the following, we briefly explain what these graphs are, and why this comparison makes sense.

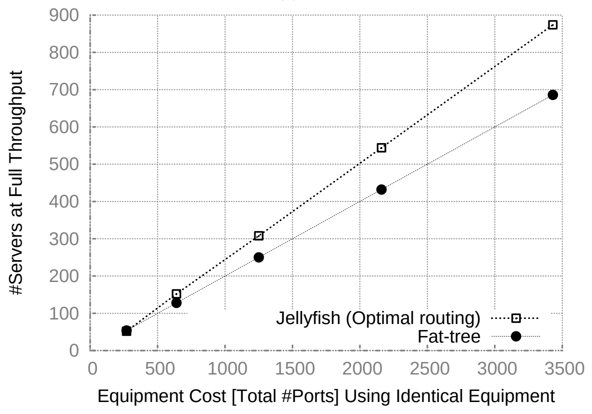
There is a fundamental trade-off between the degree and diameter of a graph of a fixed vertex-set (say of size N). At one extreme is a clique – maximum possible degree ($N-1$), and minimum possible diameter (1). At the other extreme is a disconnected graph with degree 0 and diameter ∞. The problem of 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 [11]. A collection of these optimal graphs and the best known



(a)



(b)



(c)

Figure 1: *Jellyfish offers a virtually continuous design space, and packs 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, 64, 96, 128*) at full bisection bandwidth. Under optimal routing, with random-permutation traffic: (c) Number of servers supported at full capacity using the same switching equipment, for* 6, 8, 10, 12 *and* 14*-port switches. Results are averaged over* 8 *runs.*

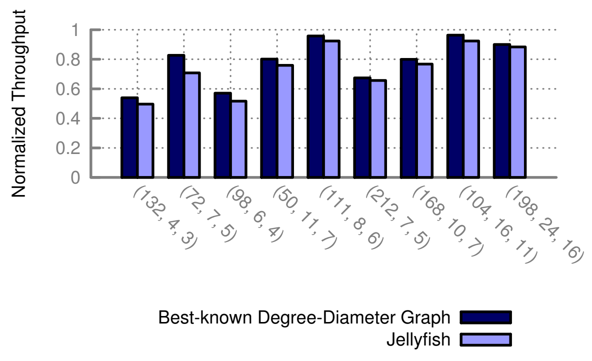


Figure 2: *Jellyfish's network capacity is close to (i.e., $\sim$86% or more in each case) that of the best-known degree-diameter graphs. The x-axis label (A, B, C) represents the number of switches (A), the switch port-count (B), and the network degree (C). Throughput is normalized against the non-blocking throughput. Results are averaged over 3 runs.*

graphs for other degree-diameter combinations is maintained at [11].

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 was not hit for the degree-diameter graphs. (That would be unfair to the degree-diameter graphs because they could still have additional capacity to support some additional servers.)

Our results, in Fig. 2, show that the best-known degree-diameter graphs do indeed achieve higher throughput than Jellyfish, and thus an even bigger improvement over fat-trees. But in the most extreme of these comparisons, Jellyfish still achieves $\sim$86% of the degree-diameter graph's aggregate throughput. In all other cases, Jellyfish achieves over 90% of the degree-diameter graph's throughput. While optimal degree-diameter graphs 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.

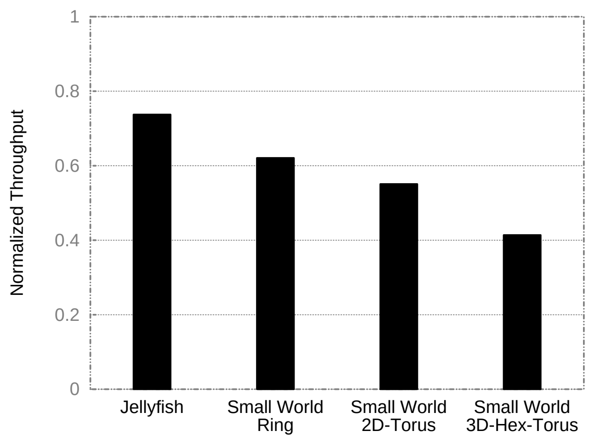


Figure 3: *Jellyfish has higher capacity than the (same-equipment) small world data center topologies [35] built using a ring, a 2D-Torus, and a 3D-Hex-Torus as the underlying lattice. Results are averaged over 5 runs.*

Comparison with small world data centers (SWDC) [35]: We use 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 3 SWDC variants described in [35] at topology sizes as close to each other as possible (constrained by the lattice structure underlying these topologies) across sizes 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 gave full throughput, so, to distinguish between their capacities, we added 2 servers instead of just one, to each switch across all topologies. The results are shown in Fig. 3. Jellyfish throughput is $\sim$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 regular random graphs used by Jellyfish 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 $1 + \lceil \log_{r-1}((2+\epsilon)rN \log N) \rceil$ for any $\epsilon > 0$. Thus, the server-to-server diameter is at most $3 + \lceil \log_{r-1}((2+\epsilon)rN \log N) \rceil$. Thus, the path length increases logarithmically (base r) with the num-

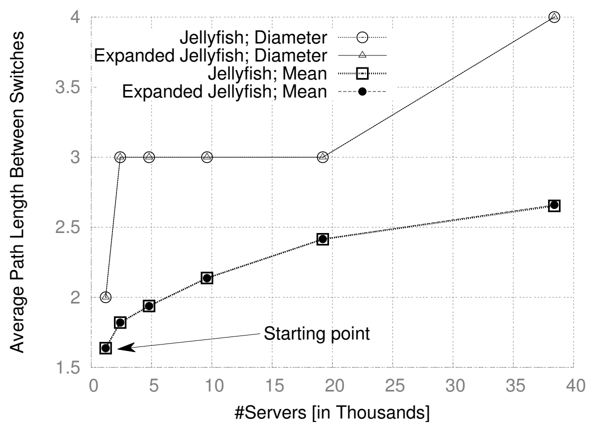


Figure 4: *Jellyfish has short paths: Path length versus number of servers, with $k = 48$ port switches of which $r = 36$ connect to other switches and* 12 *connect to servers. Each data point is derived from* 10 *graphs. For the fat-tree, almost all paths between switches are of length* 4.

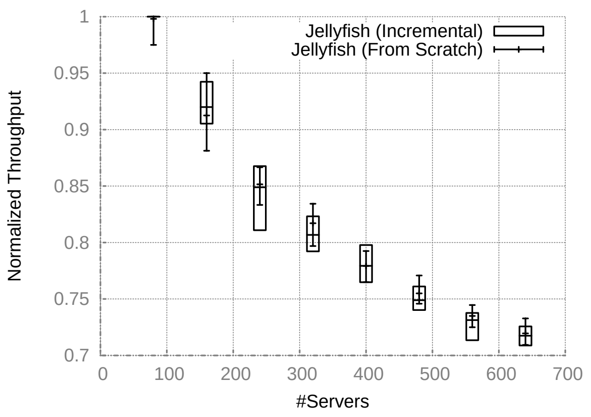


Figure 5: *Incrementally constructed Jellyfish has the same capacity as Jellyfish built from scratch: We built a Jellyfish topology incrementally from* 20 *switches to* 160 *switches in increments of* 20 *switches, and compared the throughput per server of these incrementally grown topologies to Jellyfish topologies built from scratch using our construction routine. The plot shows the average, minimum and maximum throughput over* 20 *runs.*

ber 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 and diameter (Fig. 4) in Jellyfish is much smaller than in the fat-tree. For example, for RRG(3200, 48, 36) with 38,400 servers, the average path length between switches is $<$2.7 (Fig. 4), while the fat-tree's average is $\sim$4. The 99.99^{th} percentile switch-to-switch path-length across 10 runs did not exceed 3 for any topology size in Fig. 4.

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 even 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. 4 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. 5 compares the normalized throughput per server under a random permutation traffic model for topologies built incrementally against those built from scratch. The incremental topolo-

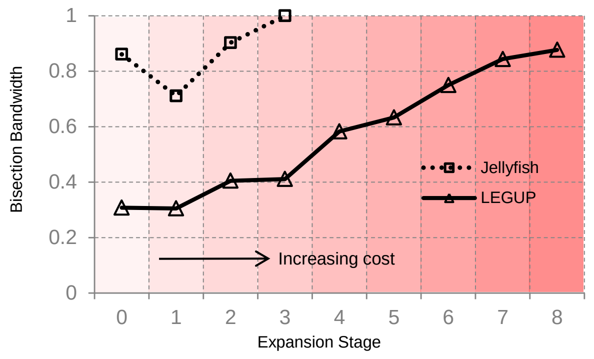


Figure 6: *Incremental expansion with Jellyfish is substantially more cost-effective than in LEGUP's expansion of Clos networks. Using the same equipment and rewiring cost at each stage of expansion (x axis), Jellyfish obtains significantly higher bisection bandwidth (y axis). (The drop in Jellyfish's bisection bandwidth from stage* 0 *to* 1 *occurs because bisection bandwidth is normalized by server capacity, and the number of servers has increased in that step.)*

gies 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 the expression for (the lower bound on) Jellyfish's bisection bandwidth (§4.1) is independent of N, i.e., (the lower bound on) bisection bandwidth stays constant as the network grows. Of course, as N increases with fixed network degree r, average path length increases, and therefore, the demand for additional per-server capacity increases[6]. But since path length increases very slowly (as discussed above), bandwidth per server is likely to remain 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 these forms of expansion.

Comparison with LEGUP [13]: 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.

LEGUP attempts to optimize for bisection bandwidth, so we compare both LEGUP and Jellyfish on that metric (using code provided by the authors of [13]) rather than on our previous random permutation throughput metric.

The results are shown in Fig. 6. 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.

4.3 Failure Resilience

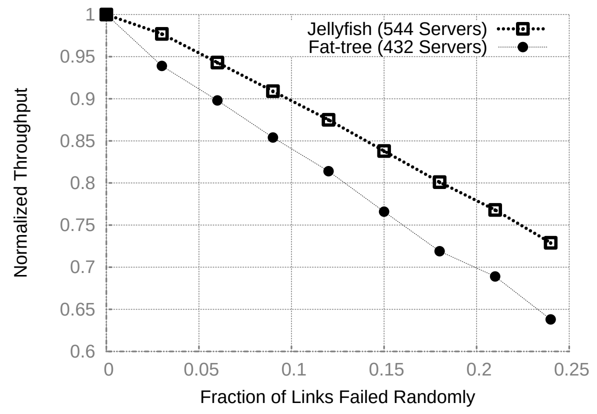
Jellyfish provides good path redundancy; in particular, an r-regular random graph is almost surely r-connected [9]. Also, the randomness of the topology implies that the graph maintains its structure (or rather, the lack of it!) in the face of link or node failures – a random graph topology with a few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

Fig. 7 shows that the Jellyfish topology is even more resilient than the fat-tree (which itself is no weakling). Fig. 7(a) compares a fat-tree and a same-equipment Jellyfish topology as the fraction of links failed (uniformly at random) increases. Fig. 7(b) compares the failure resilience of the fat-tree and Jellyfish as the size of the topology increases (with a fixed failure rate of 9%). 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*.)

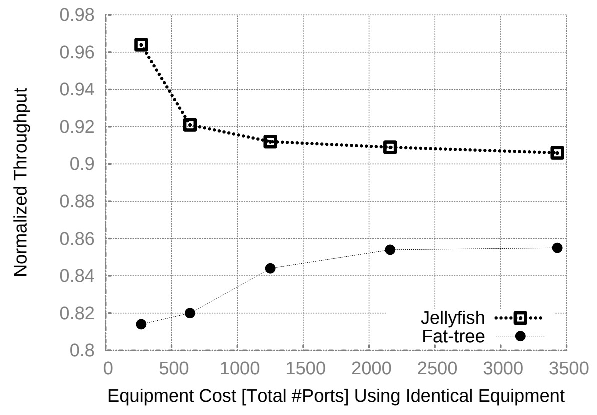
5 Routing & Congestion Control

While we have pointed out that *structure* impedes incremental expansion, we also note that structure lends itself

[6]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.



(a)



(b)

Figure 7: *Jellyfish is highly resilient to failures: a) 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. b) With a fixed link failure rate (*9%*), across increasing topology size, Jellyfish maintains the resilience advantage over the fat-tree. We note that the particular topology used for the experiment in a) is the one with equipment cost* 2,160*, i.e. the last-to-second point in the plot in b). Results are averaged over* 5 *runs.*

to simple and efficient routing schemes. In this section, we test whether the high ideal capacity made available by the Jellyfish topology can be exploited by simple routing and congestion control. Through early experiments, we discovered that Jellyfish (as well as the fat-tree) did not perform well with single-path routing. Hence, we use the recently proposed multipath TCP (MPTCP) [40]. It turns out that a simple routing scheme, when coupled with MPTCP, is able to reach >86%[7] of the *optimal* network throughput as measured using CPLEX. (A 5-7% loss of capacity also occurs for the fat-tree when using MPTCP.)

Routing: We use a simple, standard, k-shortest paths

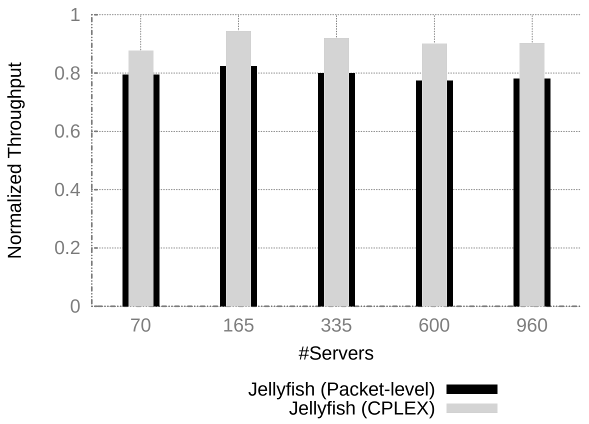


Figure 8: *Simple routing with MPTCP exploits Jellyfish's high capacity well: We compare the throughput using the same Jellyfish topology with both optimal routing, and our simple routing mechanism using MPTCP, which results in throughput between* $86\% - 90\%$ *of the optimal routing in each case. Results are averaged over* 10 *runs.*

algorithm (Yen's Loopless-Path Ranking algorithm [1, 42]) to determine routes. Throughout our experiments, $k = 8$ shortest paths are used. Thus, each switch maintains a routing table containing for each other switch, k shortest paths. Note that a few thousand switches can support several tens of thousands of servers, so routing table sizes are unlikely to be a problem. In any case, our evaluation is primarily a proof-of-concept for routing and congestion control to be able to use the network capacity. There are certainly other routing methods available for use (e.g., source routing, MPLS, or methods based on centralized management by an OpenFlow controller).

Evaluation methodology: We use the packet simulator developed by the MPTCP authors, also using their recommended value of 8 MPTCP subflows throughout our experiments. Our comparisons with the fat-tree use the same number of MPTCP subflows i.e. 8 for both topologies. The traffic model used continues to be random permutation at the server-level, and as before, for the fat-tree comparisons, we use the same switching equipment as the fat-tree.

Routing and Congestion Control Efficiency: First, we set out to measure how well our simplistic routing works with MPTCP, as compared to the optimal performance discussed in §4. Using both the packet-level simulator, and the optimizer, we compute the throughput obtained with and without routing and congestion control inefficiencies. At each size, we use the *same* slightly oversubscribed (to make the comparison clear) Jellyfish topology for both setups. The results are shown in Fig. 8. Even in the worst of these comparisons, the packet level throughput is at ~86% of the CPLEX optimal throughput for Jellyfish. For the fat-tree, this throughput is 93-95% of

[7]A gap of <14% of the optimal throughput is reasonable; prior work has shown that using currently deployed network protcols (TCP; VLB over ECMP), this gap is 23% for the fat-tree [6].

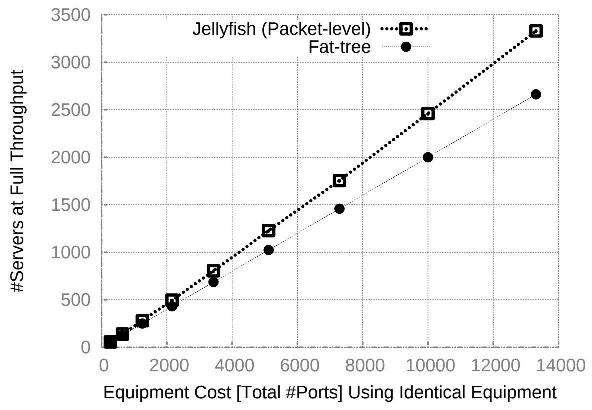


Figure 9: *Jellyfish supports a larger number of servers (>25% at the largest scale shown, with an increasing trend) than the same-equipment fat-tree at the same (or higher) throughput, even with inefficiencies of routing and congestion control accounted for. Results are averages over 20 runs for topologies smaller than 1,400 servers, and averages over 10 runs for larger topologies.*

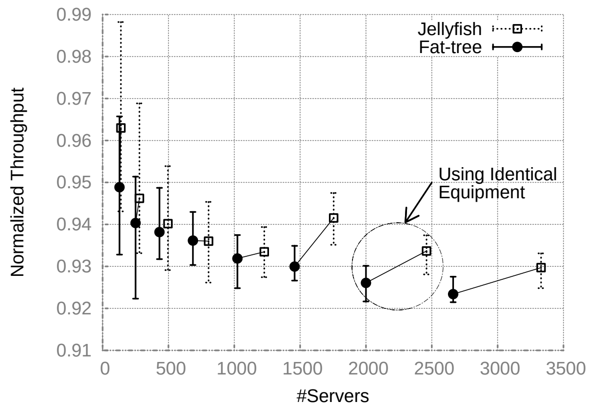


Figure 10: *The packet simulation's throughput results for Jellyfish show similar stability as the fat-tree. (Note that the y-axis starts at 91% throughput.) Average, minimum and maximum throughput-per-server values are shown. The data plotted is from the same experiment as Fig. 9. Jellyfish has the 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).*

the optimal for the fat-tree. 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 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 a binary search method) a 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. We repeat this exercise for several fat-tree sizes. The results (Fig. 9) reveal that Jellyfish can support significantly more servers at the same per-server capacity as the fat-tree. Moreover, this advantage becomes more pronounced with larger scale. 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 $\sim$15%.

We also show in Fig. 10 the stability of our throughput experiments, by plotting the average, minimum and maximum throughput for both Jellyfish and the fat-tree at each size, over 20 runs for small sizes and 10 runs for sizes larger than 1,400 servers. (Although using a larger number of runs increases the spread of the extremes for the smaller topologies, we used a smaller number of runs for the larger topologies in the interest of running time. We note that results were more stable for large topologies, with a standard deviation of less than 0.5% of the mean value. For smaller sizes, this value was smaller than 2% of the mean in each case.)

Fairness: We also evaluate how flow-fair the routing and congestion control is for Jellyfish. We use the packet simulator to measure each flow's throughput in both topologies and show in Fig. 11, 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. We also computed Jain's fairness index [23] over the same flow throughput values

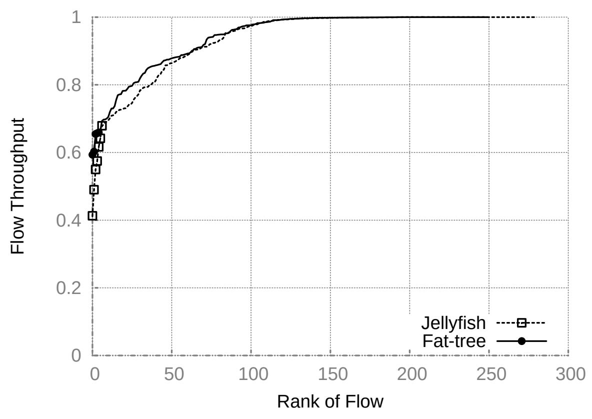


Figure 11: *Both Jellyfish and the fat-tree show good flow-fairness: The distribution of normalized flow throughputs in Jellyfish and fat-tree is shown for one typical run. After the few outliers (shown with points), the plot is virtually continuous (hence the line). Note that Jellyfish has more flows because it supports a higher number of servers (at same or higher* per-server *throughput). Jain's fairness index for both topologies is* $\sim$99%.

as in the plot for both topologies: 0.991 for the fat-tree and 0.988 for Jellyfish.

6 Physical Construction and Cabling

Key considerations in wiring data center networks include:

- **Number of cables:** Each cable is 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 [32]. Thus, we limit our interest in cable length to whether a cable is short enough, i.e., <10 meters in length [15, 24], for use of an electrical cable or not.
- **Cabling complexity:** Will Jellyfish awaken the dread spaghetti monster? Complex 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.

How much each of the above considerations matters, depends on the deployment scenario in question. Thus, in this section we describe physical packaging and wiring approaches for three deployment scenarios for Jellyfish: (a) as an interconnect for small data centers ($\sim$1,000 servers); (b) as the intra-container interconnect for a 'Container Data Center' (CDC) [2–4, 41]; (c) as a network interconnect for a massive-scale data center.

We note that small data centers and CDCs form a significant section of the market for data centers. In a 2011 survey [38] 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 [12].

For each deployment scenario, we also discuss how cabling works with incremental expansion.

6.1 Jellyfish in Small Data Centers

As our results in §4.1 show, even at a few hundred servers, cost-efficiency gains from Jellyfish can be significant ($\sim$20% at 1,000 servers). Thus, it is useful to deploy Jellyfish in this scenario.

Number of cables: For a $\sim$1,000 server data center, Jellyfish uses $\sim$15% fewer cables than a fat-tree.

Length of cables: At such sizes, the cable lengths will be short enough to use electrical cables without repeaters. Nevertheless, we propose an optimization (along similar lines as the one proposed in [5]) based on the observation 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.

Complexity: For 1,000 servers, space equivalent to 3 standard racks can accommodate all the switches necessary to build a full bisection bandwidth network (using 48-port switches available today). These 3 racks can then 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 rack of servers in the data center. In this example scenario, 20 server-racks will suffice, leaving us with 20 aggregate cable assemblies, each with 50 cables, running from the switch-cluster to the server-racks. Thus, the nightmare cable-mess image a random graph network may spring to mind is, at best, alarmist.

Also, the manual work involved includes connecting the 20 bundles of 50 cables from the switches to the servers in a trivial manner. The *only* constraint is for each switch to contribute 14-15 ports to this total of 1,000 connections. The remaining cabling task is to connect the random graph component. This part of the cabling plan can be computer-generated based on the topology and physical layout of devices, and handed to workers to connect.

Handling Mis-wiring: While some human errors are likely in cabling, these are easy to detect and fix. Given Jellyfish's sloppy topology, a small number of miswirings need not even require fixing in many cases. Nevertheless, an estimate [32] of the labor cost of cabling puts it at $\sim$10% of cable cost at this scale. The cable cost itself would be only a fraction of the network cost; assuming that fraction is a rather high 50%, the cost of correcting (for example) 10% mis-wirings would be just 0.5% of the network cost. We note that running a link-layer topology discovery protocol [29] yields enough information to check the resulting cabling against a computer-generated connection plan and detect errors.

Cabling under expansion: Jellyfish can be expanded in such a setting either by leaving enough space near the 'switch-cluster' for addition of more switches as more 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 switch-cluster to all server-racks and to the 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 sequence of edge swaps. After an automated computation of the network cables that need to be moved and new ones that need attachment, these can be run parallel to existing cable bundles, or in the case of new switch-clusters, new cable aggregates can be started. Addition of each two ports requires two cables to be moved (one end of an existing cable is connected to one of the two new ports, and a new cable is connects the orphaned attachment point of the old cable to the second new port). Note that with the 'switch-cluster' configuration, all this activity happens at one location (or with multiple clusters, the activity happens 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.

6.2 Intra-container Jellyfish

As early as 2006, The Sun Blackbox [2] promoted the idea of using shipping containers for the construction of data centers. There are also new products in the market exploiting similar physical design ideas [3, 4, 22].

Much of the appeal of container data centers is in their 'deploy and forget' nature. These are ideal for enterprises that seek quickly deployable and low-maintainence systems that literally work out of the box. Besides providing high capacity, the network must also be reliable, so as to not require frequent intervention inside the container. The utility of using Jellyfish in such a scenario is its efficiency and reliability. As we have shown in §4.1 and §4.3, Jellyfish provides high capacity, low latency, and high failure resilience, all at low cost. In comparison with the fat-tree, Jellyfish uses less switching equipment to support the same server-pool at higher performance.

Number of cables: Given the roughly linear switch-server curve in Fig. 9, 25% more servers with same switching equipment as a fat-tree also translates to 20% less switching equipment (and hence, cabling) for supporting the same server-pool size. This implies that there is more room (and budget) for packing servers in a single container.

Length of cables: The optimization of placing all the switches in close proximity near the center of the container is useful in this scenario too. Given a shipping container's dimensions, this also results in all connections being short enough for use of electrical cabling throughout.

Complexity: 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 significantly accelerate the cabling process.

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 mis-wirings 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: We note that the CDC usage may or may not be geared towards incremental expansion. There are certainly products in the market which bring the modularity idea containers brought to mega-data centers, to containers themselves – allowing gradual build-up of a container with smaller container-modules [4]. The more common scenario, however, appears to be standard, packed containers, where the chief utility of Jellyfish is its efficiency and reliability. Nevertheless, Jellyfish expansion ideas apply to the modular containers in similar fashion to their application to small data centers. If patch panels are used in CDCs as suggested, then they play a role similar to the switch-cluster in the small data center: most rewiring can be completed at the patch panel.

6.3 Jellyfish in Massive-Scale Data Centers

Massive scale data centers may be built by connecting together multiple containers of the type described above. (This is already an industry trend, with several players, Google and Microsoft included, already having container-based deployments [15].) Inside the containers, the same arguments for number, length, and complexity of cables apply as discussed before. However, extending Jellyfish to such settings naïvely, might result in excessive cabling costs: As the number of containers grows, almost all cables are likely to be between containers, thus necessitating the use of expensive optical connectors. Thus, for these scales, our comparisons with the fat-tree might be considered unfair in absence of accounting for cabling.

To bring fairness to this comparison, we restrict Jellyfish to the same physical constraints and available cable lengths. For the fat-tree we apply the same optimization as suggested in [5] for laying out the fat-tree at such scale[8]. With this optimized layout, we know the number of cables inside containers (henceforth, 'local') and

[8]The core idea was to make each fat-tree 'pod' a container, and to divide the core-switches among these pods equally.

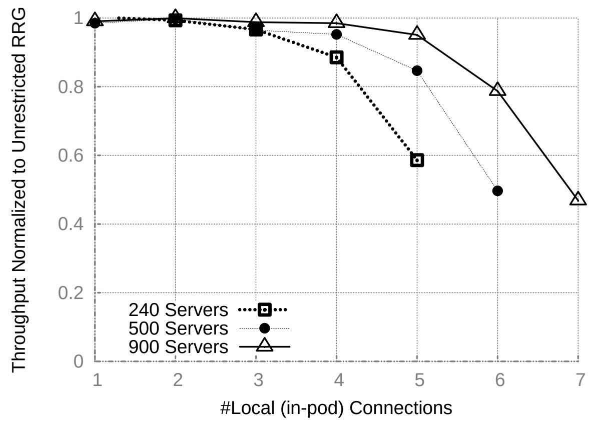


Figure 12: *Localization of Jellyfish random links is a promising approach to tackle cabling for massive scale data centers: As links are restricted to be more and more local, the network capacity decreases (as expected). However, at the largest scale shown, with* 5 *of* 8 *random links for each switch constrained to remain inside the pod, there is only* 5% *loss of throughput.*

outside ('global').

With the same number of switches and the same physical structuring (in terms of number of pods and equal number of switches per pod) as the fat-tree, we build Jellyfish networks varying the number of local and global connections to see how this affects performance in relation to the unrestricted Jellyfish network. By restricting Jellyfish, we mean that we only allow the local connections to be picked to nodes inside the pod (randomly), and the rest only to nodes outside (randomly). This effectively results in a 2-layer random graph.

With the same switching equipment as the fat-tree, Jellyfish networks would be overprovisioned with this design. Thus, we add a larger number of servers per switch to make the topology over-subscribed. Then we vary the number of local and global connections (such that the sum is constant, and the same as the fat-tree) and measure performance in relation to the unrestricted Jellyfish.

The results are shown in Fig. 12. For the largest size in this test, throughput does not drop significantly until 6 out of the total of 8 network links are restricted to be inside the pod. In this scenario, 2-layering (with 5 out of 8 links 'localized') reduces Jellyfish throughput to 95% of its optimal. In the same setting, in the fat-tree, the fraction of local links is $13/24 = 0.542 < 5/8 = 0.62$. Thus, we can achieve a higher degree of localization, while still having a higher capacity network. (Note that our results from §4.1 show that Jellyfish is 26% more efficient than the fat-tree at this scale.)

Separate from the favorable fat-tree comparison, we note that the localization strategy reduces the number of global cables (in expectation) from 11 of every 12 links (each pod has the same number of switches;there are 12 pods) to 3 of every 8 links – a 59% decrease, for the loss of 5% of network capacity. More evaluation of this aspect, in particular, to determine whether the *fraction* of links one can localize without losing significant throughput keeps increasing with network size is on our agenda for future work. For the fat-tree layout, the answer is known: the fraction of local links (which is conveniently given by $0.5(1 + 1/k)$) decreases with size.

Complexity: Building a random graph between switches at the inter-container layer will, with high probability, result in cable assemblies running between every pair of containers. A 100,000 server data center can be built with ~40 containers. Even if *all* the ports (except those attached to servers) from each switch in each container were to be connected to other containers, we could aggregate cables between each pair of containers leaving us with roughly 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 be decreased several times.

Cabling under expansion: In massive-scale data centers, expansion can occur both through addition of new containers and expansion of containers (if permissible). The random connections for each layer are added independently by the standard Jellyfish procedure. Laying out spare cables together with the aggregates between containers is helpful in scenarios where a container is being expanded. When a new container is added, new aggregates must be laid out to every other container. Patch panels can again make this process easier. In each container, all the global-connection ports from switches can be connected from the inside to a patch panel (using short electrical cables) which is easily accessible. In this scenario, the use of patch panels is limited to making these connections accessible for later change, and not for encoding the random interconnect.

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, still enabling high capacity, short paths, and resilience to failures and miswirings.

References

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

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

[3] Rackable systems. ice cube modular data center. http://www.rackable.com/products/icecube.aspx.

[4] Sgi ice cube air expandable line of modular data centers. http://www.sgi.com/products/data_center/ice_cube_air/.

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

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. The case for fine-grained traffic engineering in data-centers. In *INM/WREN*, 2010.

[7] L. N. Bhuyan and A. D. P. 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] F. Comellas and C. Delorme. The (degree, diameter) problem for graphs. http://maite71.upc.es/grup_de_grafs/table_g.html/.

[12] A. R. Curtis, T. Carpenter, M. Elsheikh, A. Lopez-Ortiz, and S. Keshav. Rewire: An optimization-based framework for data center network design. Technical report, August 2011.

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

[14] Facebook. Facebook to expand prineville data center. http://goo.gl/fJAoU.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] HP. Hp ecopod. http://goo.gl/8A0Ad.

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

[23] 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.

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

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

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

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

[28] 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.

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

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

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

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

[33] 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.

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

[35] J.-Y. Shin, B. Wong, and E. G. Sirer. Small-world datacenters. *ACM SOCC*, 2011.

[36] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Network data centers randomly. In *HotCloud*, 2011.

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

[38] D. R. Trust. What is driving the us market? white paper, 2011. http://goo.gl/qiaRY.

[39] 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.

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

[41] H. Wu, G. Lu, D. Li, C. Guo, and Y. Zhang. Mdcube: A high performance network structure for modular data center interconnection. In *CoNext*, 2009.

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