Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network

Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat
Google, Inc.
jupiter-sigcomm@google.com

ABSTRACT

We present our approach for overcoming the cost, operational complexity, and limited scale endemic to datacenter networks a decade ago. Three themes unify the five generations of datacenter networks detailed in this paper. First, multi-stage Clos topologies built from commodity switch silicon can support cost-effective deployment of building-scale networks. Second, much of the general, but complex, decentralized network routing and management protocols supporting arbitrary deployment scenarios were overkill for single-operator, pre-planned datacenter networks. We built a centralized control mechanism based on a global configuration pushed to all datacenter switches. Third, modular hardware design coupled with simple, robust software allowed our design to also support inter-cluster and wide-area networks. Our datacenter networks run at dozens of sites across the planet, scaling in capacity by 100x over ten years to more than 1Pbps of bisection bandwidth.

CCS Concepts

•**Networks** → **Data center networks;**

Keywords

Datacenter Networks; Clos topology; Merchant Silicon; Centralized control and management

1. INTRODUCTION

Datacenter networks are critical to delivering web services, modern storage infrastructure, and are a key enabler for cloud computing. Bandwidth demands in the datacenter are doubling every 12-15 months (Figure 1), even faster than the wide area Internet. A number of recent trends drive this growth. Dataset sizes are continuing to explode with more photo/video content, logs, and the proliferation of Internet-connected sensors. As a result, network-intensive data processing pipelines must operate over ever-larger datasets. Next, Web services can deliver higher quality results by accessing more data on the critical path of individual requests. Finally, constellations of co-resident applications often share substantial data with one another in the same cluster; consider index generation, web search, and serving ads.

Ten years ago, we found the cost and operational complexity associated with traditional datacenter network architectures to be prohibitive. Maximum network scale was limited by the cost and capacity of the highest end switches available at any point in time [24]. These switches were engineering marvels, typically recycled from products targeting wide area deployments. WAN switches were differentiated with hardware support/offload for a range of protocols (e.g., IP multicast) or by pushing the envelope of chip memory (e.g., Internet-scale routing tables, off chip DRAM for deep buffers, etc.). Network control and management protocols targeted autonomous individual switches rather than pre-configured and largely static datacenter fabrics. Most of these features were not useful for datacenters, increased cost, complexity, delayed time to market, and made network management more difficult.

Datacenter switches were also built as complex chassis targeting the highest levels of availability. In a WAN Internet deployment, losing a single switch/router can have substantial impact on applications. Because WAN links are so expensive, it makes sense to invest in high availability. However, more plentiful and cheaper datacenter bandwidth makes it prudent to trade cost for somewhat reduced intermittent capacity. Finally, switches operating in a multi-vendor WAN environment with arbitrary end hosts require support for many protocols to ensure interoperability. In single-operator dat-

SIGCOMM '15 August 17-21, 2015, London, United Kingdom

ACM ISBN 978-1-4503-3542-3/15/08.
DOI: http://dx.doi.org/10.1145/2785956.2787508

acenter deployments, the number of required protocols can be substantially reduced.

Inspired by the community's ability to scale out computing with parallel arrays of commodity servers, we sought a similar approach for networking. This paper describes our experience with building five generations of custom data center network hardware and software leveraging commodity hardware components, while addressing the control and management requirements introduced by our approach. We used the following principles in constructing our networks:

Clos topologies: To support graceful fault tolerance, increase the scale/bisection of our datacenter networks, and accommodate lower radix switches, we adopted Clos topologies [2, 9, 15] for our datacenters. Clos topologies can scale to nearly arbitrary size by adding stages to the topology, principally limited by failure domain considerations and control plane scalability. They also have substantial in-built path diversity and redundancy, so the failure of any individual element can result in relatively small capacity reduction. However, they introduce substantial challenges as well, including managing the fiber fanout and more complex routing across multiple equal-cost paths.

Merchant silicon: Rather than use commercial switches targeting small-volume, large feature sets, and high reliability, we targeted general-purpose merchant switch silicon, commodity priced, off the shelf, switching components. To keep pace with server bandwidth demands which scale with cores per server and Moore's Law, we emphasized bandwidth density and frequent refresh cycles. Regularly upgrading network fabrics with the latest generation of commodity switch silicon allows us to deliver exponential growth in bandwidth capacity in a cost-effective manner.

Centralized control protocols: Control and management becomes substantially more complex with Clos topologies because we dramatically increase the number of discrete switching elements. Existing routing and management protocols were not well-suited to such an environment. To control this complexity, we observed that individual datacenter switches played a predetermined forwarding role based on the cluster plan. We took this observation to one extreme by collecting and distributing dynamically changing link state information from a central, dynamically-elected, point in the network. Individual switches could then calculate forwarding tables based on current link state relative to a statically configured topology.

Overall, our software architecture more closely resembles control in large-scale storage and compute platforms than traditional networking protocols. Network protocols typically use soft state based on pair-wise message exchange, emphasizing local autonomy. We were able to use the distinguishing characteristics and needs of our datacenter deployments to simplify control and management protocols, anticipating many of the tenets of modern Software Defined Networking deployments [13]. The datacenter networks described in this paper represent some of the largest in the world, are in deployment at dozens of sites across the planet, and support thousands of internal and external services, including external use through Google Cloud Platform. Our cluster network architecture found substantial reuse for inter-cluster networking in the same campus and even WAN deployments [19] at Google.

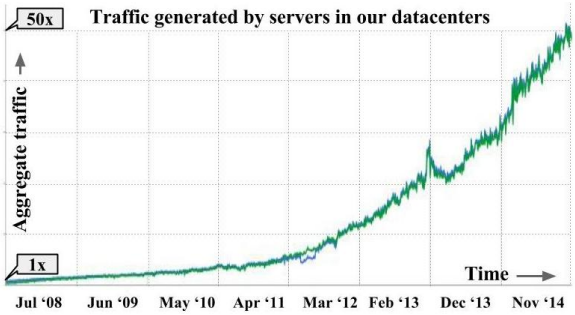

Figure 1: Aggregate server traffic in our datacenter fleet.

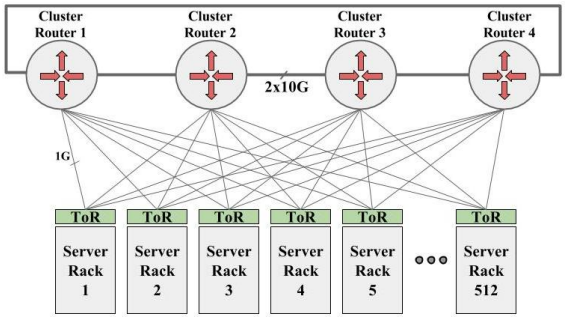

Figure 2: A traditional 2Tbps four-post cluster (2004). Top of Rack (ToR) switches serving 40 1G-connected servers were connected via 1G links to four 512 1G port Cluster Routers (CRs) connected with 10G sidelinks.

2. BACKGROUND AND RELATED WORK

The tremendous growth rate of our infrastructure served as key motivation for our work in datacenter networking. Figure 1 shows aggregate server communication rates since 2008. Traffic has increased 50x in this time period, roughly doubling every year. A combination of remote storage access [7, 14], large-scale data processing [10, 18], and interactive web services [4] drive our bandwidth demands.

In 2004, we deployed traditional cluster networks similar to [5]. Figure 2 depicts this "four-post" cluster architecture. We employed the highest density Ethernet switches available, 512 ports of 1GE, to build the spine of the network (CRs or cluster routers). Each Top of Rack (ToR) switch connected to all four of the cluster routers for both scale and fault tolerance.

With up to 40 servers per ToR, this approach supported 20k servers per cluster. However, high band-

Job Category	B/w (%)
Storage	49.3
Search Serving	26.2
Mail	7.4
Ad Stats	3.8
Rest of traffic	13.3

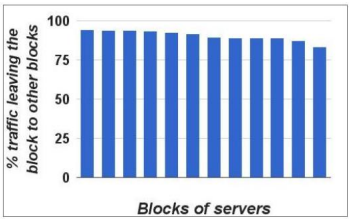

Figure 3: Mix of jobs in an example cluster with 12 blocks of servers (left). Fraction of traffic in each block destined for remote blocks (right).

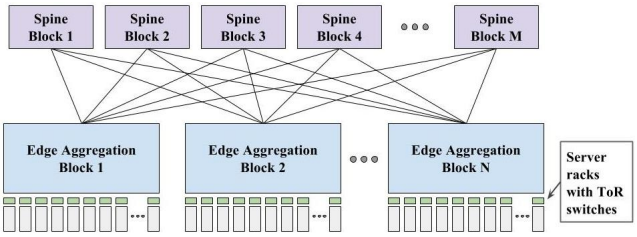

Figure 4: A generic 3 tier Clos architecture with edge switches (ToRs), aggregation blocks and spine blocks. All generations of Clos fabrics deployed in our datacenters follow variants of this architecture.

width applications had to fit under a single ToR to avoid the heavily oversubscribed ToR uplinks. Deploying large clusters was important to our services because there were many affiliated applications that benefited from high bandwidth communication. Consider large-scale data processing to produce and continuously refresh a search index, web search, and serving ads as affiliated applications. Larger clusters also substantially improve bin-packing efficiency for job scheduling by reducing stranding from cases where a job cannot be scheduled in any one cluster despite the aggregate availability of sufficient resources across multiple small clusters.

Maximum cluster scale is important for a more subtle reason. Power is distributed hierarchically at the granularity of the building, multi-Megawatt power generators, and physical datacenter rows. Each level of hierarchy represents a unit of failure and maintenance. For availability, cluster scheduling purposely spreads jobs across multiple rows. Similarly, the required redundancy in storage systems is in part determined by the fraction of a cluster that may simultaneously fail as a result of a power event. Hence, larger clusters lead to lower storage overhead and more efficient job scheduling while meeting diversity requirements.

Running storage across a cluster requires both rack and power diversity to avoid correlated failures. Hence, cluster data should be spread across the cluster's failure domains for resilience. However, such spreading naturally eliminates locality and drives the need for uniform bandwidth across the cluster. Consequently, storage placement and job scheduling have little locality in our cluster traffic, as shown in Figure 3. For a representative cluster with 12 blocks (groups of racks) of servers, we show the fraction of traffic destined for remote blocks. If traffic were spread uniformly across the cluster, we would expect 11/12 of the traffic (92%) to be destined for other blocks. Figure 3 shows approximately this distribution for the median block, with only moderate deviation.

While our traditional cluster network architecture largely met our scale needs, it fell short in terms of overall performance and cost. Bandwidth per host was severely limited to an average of 100Mbps. Packet drops associated with incast [8] and outcast [21] were severe pain points. Increasing bandwidth per server would have substantially increased cost per server and reduced cluster scale.

We realized that existing commercial solutions could not meet our scale, management, and cost requirements. Hence, we decided to build our own custom data center network hardware and software. We started with the key insight that we could scale cluster fabrics to near arbitrary size by leveraging Clos topologies (Figure 4) and the then-emerging (ca. 2003) merchant switching silicon industry [12]. Table 1 summarizes a number of the top-level challenges we faced in constructing and managing building-scale network fabrics. The following sections explain these challenges and the rationale for our approach in detail.

For brevity, we omit detailed discussion of related work in this paper. However, our topological approach, reliance on merchant silicon, and load balancing across multipath are substantially similar to contemporaneous research [2,15]. In addition to outlining the evolution of our network, we further describe inter cluster networking, network management issues, and detail our control protocols. Our centralized control protocols running on switch embedded processors are also related to subsequent substantial efforts in Software Defined Networking (SDN) [13]. Based on our experience in the datacenter, we later applied SDN to our Wide Area Network [19]. For the WAN, more CPU intensive traffic engineering and BGP routing protocols led us to move control protocols onto external servers with more plentiful CPU from the embedded CPU controllers we were able to utilize for our initial datacenter deployments.

Recent work on alternate network topologies such as HyperX [1], Dcell [17], BCube [16] and Jellyfish [22] deliver more efficient bandwidth for uniform random communication patterns. However, to date, we have found that the benefits of these topologies do not make up for the cabling, management, and routing challenges and complexity.

3. NETWORK EVOLUTION

3.1 Firehose 1.0

Table 2 summarizes the multiple generations of our

Challenge	Our Approach (Section Discussed in)
Introducing the network to production	Initially deploy as bag-on-the-side with a fail-safe big-red button (3.2)
High availability from cheaper components	Redundancy in fabric, diversity in deployment, robust software, necessary protocols only, reliable out of band control plane (3.2, 3.3, 5.1)
High fiber count for deployment	Cable bundling to optimize and expedite deployment (3.3)
Individual racks can leverage full uplink capacity to external clusters	Introduce Cluster Border Routers to aggregate external bandwidth shared by all server racks (4.1)
Incremental deployment	Depopulate switches and optics (3.3)
Routing scalability	Scalable in-house IGP, centralized topology view and route control (5.2)
Interoperate with external vendor gear	Use standard BGP between Cluster Border Routers and vendor gear (5.2.5)
Small on-chip buffers	Congestion window bounding on servers, ECN, dynamic buffer sharing of chip buffers, QoS (6.1)
Routing with massive multipath	Granular control over ECMP tables with proprietary IGP (5.1)
Operating at scale	Leverage existing server installation, monitoring software; tools build and operate fabric as a whole; move beyond individual chassis-centric network view; single cluster-wide configuration (5.3)
Inter cluster networking	Portable software, modular hardware in other applications in the network hierarchy (4.2)

Table 1: High-level summary of challenges we faced and our approach to address them.

Datacenter Generation	First Deployed	Merchant Silicon	ToR Config	Aggregation Block Config	Spine Block Config	Fabric Speed	Host Speed	Bisection BW
Four-Post CRs	2004	vendor	48x1G	-	-	10G	1G	2T
Firehose 1.0	2005	8x10G 4x10G (ToR)	2x10G up 24x1G down	2x32x10G (B)	32x10G (NB)	10G	1G	10T
Firehose 1.1	2006	8x10G	4x10G up 48x1G down	64x10G (B)	32x10G (NB)	10G	1G	10T
Watchtower	2008	16x10G	4x10G up 48x1G down	4x128x10G (NB)	128x10G (NB)	10G	nx1G	82T
Saturn	2009	24x10G	24x10G	4x288x10G (NB)	288x10G (NB)	10G	nx10G	207T
Jupiter	2012	16x40G	16x40G	8x128x40G (B)	128x40G (NB)	10/40G	nx10G/ nx40G	1.3P

Table 2: Multiple generations of datacenter networks. (B) indicates blocking, (NB) indicates Nonblocking.

cluster network. With our initial approach, Firehose 1.0 (or FH1.0), our nominal goal was to deliver 1Gbps of nonblocking bisection bandwidth to each of 10K servers. Figure 5 details the FH1.0 topology. Our starting point was 8x10G switches for the fabric and 4x10G switches for ToRs. The fabric switch was deployed with 4x10G ports facing up and 4x10G facing down in all stages but the topmost stage, which had all 8x10G ports facing down. The ToR switch delivered 2x10GE ports to the fabric and 24x1GE south-facing ports of which 20x1GE were connected to servers. Each aggregation block hosted 16 ToRs (320 machines) and exposed 32x10G ports towards 32 spine blocks. Each spine block had 32x10G towards 32 aggregation blocks resulting in a fabric that scaled to 10K machines at 1G average bandwidth to any machine in the fabric.

A key drawback of the topology was the low radix of the ToR switch, which caused issues when links failed. If the left uplink of a source ToR and the right uplink of a destination ToR failed within a MTTR window, machines on these ToRs could not communicate with each other even though they could communicate with other machines - an intransitive disconnect not handled well by applications.

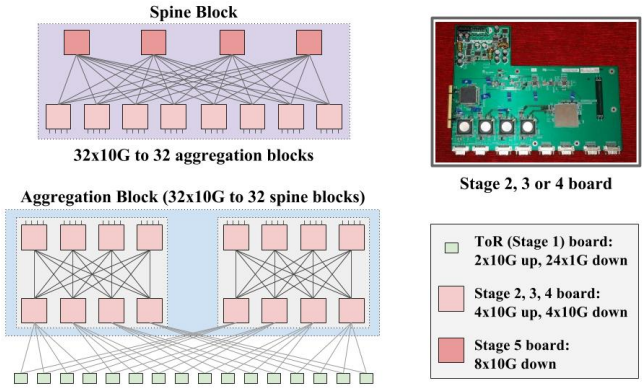

Figure 5: Firehose 1.0 topology. Top right shows a sample 8x10G port fabric board in Firehose 1.0, which formed Stages 2, 3 or 4 of the topology.

Since we did not have any experience building switches but we did have experience building servers, we attempted to integrate the switching fabric into the servers via a PCI board. See top right inset in Figure 5. However, the uptime of servers was less than ideal. Servers crashed and were upgraded more fre-

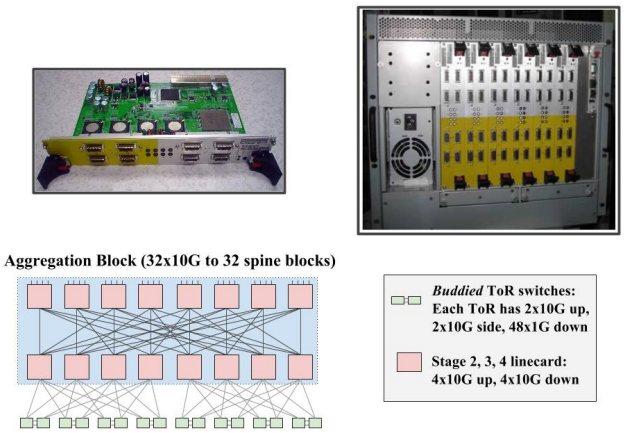

Figure 6: Firehose 1.1 packaging and topology. The top left picture shows a linecard version of the board from Figure 5. The top right picture shows a Firehose 1.1 chassis housing 6 such linecards. The bottom figure shows the aggregation block in Firehose 1.1, which was different from Firehose 1.0.

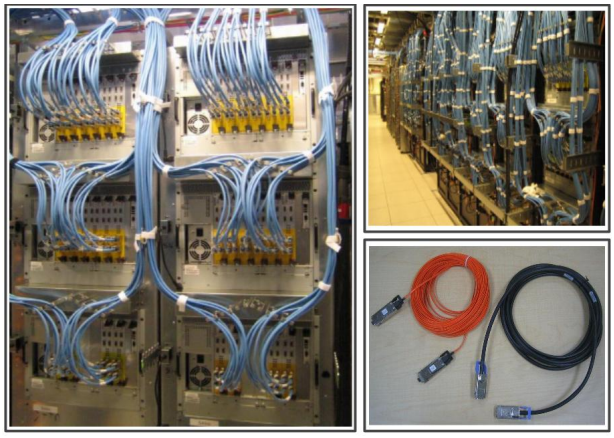

Figure 7: Two Firehose racks (left), each with 3 chassis with bulky CX4 cables from remote racks. The top right figure shows an aisle of cabled racks.

quently than desired with long reboot times. Network disruptions from server failure were especially problematic for servers housing ToRs connecting multiple other servers to the first stage of the topology.

The resulting wiring complexity for server to server connectivity, electrical reliability issues, availability and general issues associated with our first foray into switching doomed the effort to never seeing production traffic. At the same time, we consider FH1.0 to be a landmark effort internally. Without it and the associated learning, the efforts that followed would not have been possible.

3.2 Firehose 1.1: First Production Clos

Our first production deployment of a custom datacenter cluster fabric was called Firehose 1.1 (or FH1.1), a variation of the FH1.0 architecture. We had learned from FH1.0 to not use regular servers to house switch chips. Thus, we built custom enclosures that standardized around the Compact PCI chassis each with 6 independent linecards and a dedicated Single-Board Computer (SBC) to control the linecards using PCI. See insets in Figure 6. The fabric chassis did not contain any backplane to interconnect the switch chips. All ports connected to external copper cables that were wired on the datacenter floor. The linecards were a different form factor of the same boards used in FH1.0 for stages 2-5. We built a separate out-of-band Control Plane Network (CPN) to configure and manage the SBCs of the fabric.

The FH1.1 topology was a variant of the one used in FH1.0. While the spine block was identical to FH1.0, the edge aggregation block illustrated in Figure 6 had a few differences. We used two 4x10G+24x1G switch chips side connected on the board with 2x10G links for ToRs. The resulting configuration was a ToR switch with 4x10G uplinks and 48x1G links to servers. ToRs were developed as separate 1RU switches each with its own CPU controller. To scale to 20k machines with at most 2:1 oversubscription, we decided to *buddy* two ToR switches together. Of the 4x10G uplinks in each ToR, two were connected to the fabric while two were connected sideways to the paired ToR. Traffic from machines under a ToR could use all four uplinks to burst to the fabric, though bandwidth under contention would be lower. The stage 2 and 3 switches within an aggregation block were cabled in a single block (vs. 2 disjoint blocks in FH1.0) in a configuration similar to a Flat Neighborhood Network [11]. With up to 40 machines under each ToR, the FH1.1 aggregation block could scale to 640 machines at 2:1 oversubscription. The changes in the aggregation block allowed Firehose 1.1 to scale to 2x the number of machines while being much more robust to link failures compared to FH1.0.

The copper interconnect for FH1.1 was a significant challenge. Figure 7 shows the chassis in production deployment. Building, testing, and deploying the network was labor intensive and error prone. The 14m length restrictions of our CX4 cables required careful placement of each component of the multistage topology. The longest distances were typically between our ToRs and the next stage of the Firehose switching infrastructure. To improve deployability, we worked on a solution to run fiber only for this stage of the network topology. We collaborated with vendors to develop custom Electrical/Optical/Electrical (EOE) cables for this interconnect. The orange cable in the bottom right of Figure 7 is an EOE cable capable of spanning 100m compared to the bulkier 14m CX4 cable to its right.

A major concern with FH1.1 in production was deploying an unproven new network technology for our mission critical applications. To mitigate risk, we deployed Firehose 1.1 in conjunction with our legacy four-post cluster fabrics as shown in Figure 8. We maintained a simple configuration; the ToR would forward default traffic to the four-post cluster (e.g., for connectivity to external clusters/data centers) while more spe-

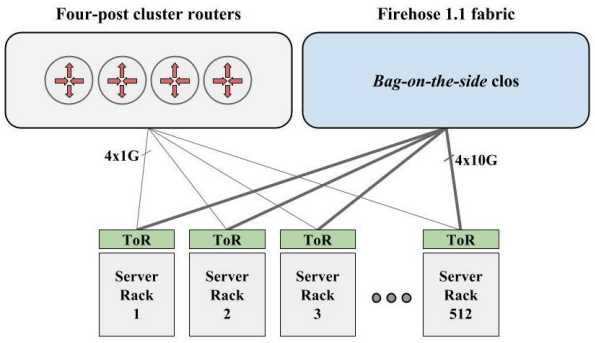

Figure 8: Firehose 1.1 deployed as a *bag-on-the-side Clos* fabric.

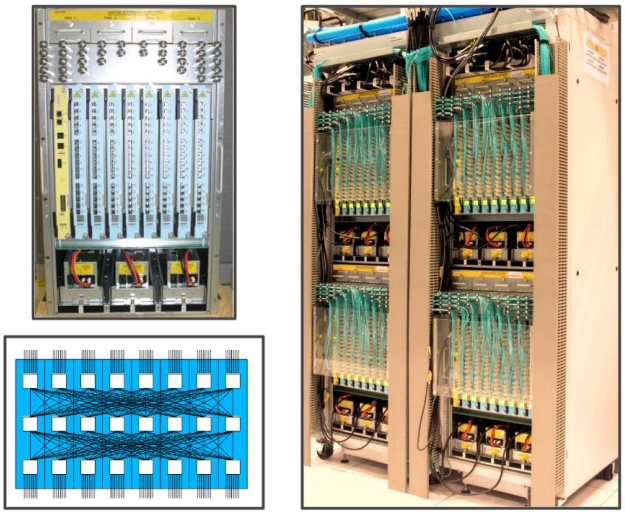

Figure 9: A 128x10G port Watchtower chassis (top left). The internal non-blocking topology over eight linecards (bottom left). Four chassis housed in two racks cabled with fiber (right).

cific intra-cluster traffic would use the uplinks to Firehose 1.1. Since our four-post cluster employed 1G links, we only needed to reserve four 1GE ToR ports. We built a *Big Red Button* fail-safe to configure the ToRs to avoid Firehose uplinks in case of catastrophic failure.

3.3 Watchtower: Global Deployment

Our deployment experience with Firehose 1.1 was largely positive. We showed that services could enjoy substantially more bandwidth than with traditional architectures, all with lower cost per unit bandwidth. Firehose 1.1 went into production with a handful of clusters and remained operational until recently. The main drawback to Firehose 1.1 was the deployment challenges with the external copper cabling.

We used these experiences to design Watchtower, our third-generation cluster fabric. The key idea was to leverage the next-generation merchant silicon switch chips, 16x10G, to build a traditional switch chassis with a backplane. Figure 9 shows the half rack Watchtower

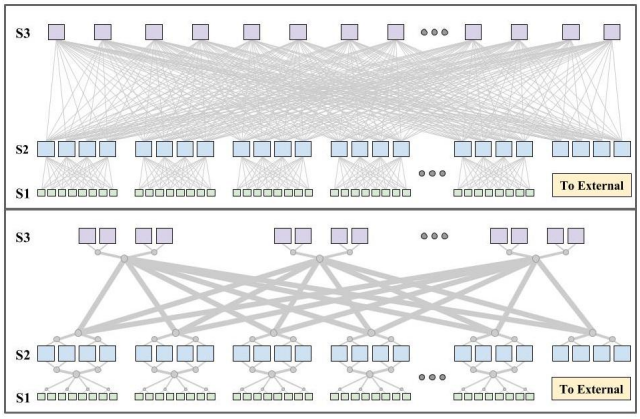

Figure 10: Reducing deployment complexity by bundling cables. Stages 1, 2 and 3 in the fabric are labeled S1, S2 and S3, respectively.

# Individual cables	15872
# S2-S3 bundles (16-way)	512
Normalized cost of fiber/m in 16-way bundle	55%
# S2-ToR bundles (8-way)	960
Normalized cost of fiber/m in 8-way bundle	60%
# Total cable bundles	1472
Normalized cost of fiber/m with bundling (capex + opex)	57%

Table 3: Benefits of cable bundling in Watchtower.

chassis along with its internal topology and cabling. Watchtower consists of eight line cards, each with three switch chips. Two chips on each linecard have half their ports externally facing, for a total of 16x10GE SFP+ ports. All three chips also connect to a backplane for port to port connectivity. Watchtower deployment, as seen in Figure 9 was substantially easier than the earlier Firehose deployments. The larger bandwidth density of the switching silicon also allowed us to build larger fabrics with more bandwidth to individual servers, a necessity as servers were employing an ever-increasing number of cores.

Fiber bundling further reduced the cabling complexity of Watchtower clusters. Figure 10 shows a Watchtower fabric deployment without any cable bundling. Individual fibers of varying length need to be pulled from each chassis location, leading to significant deployment overhead. The bottom figure shows how bundling can substantially reduce complexity. We deploy two chassis in each rack and co-locate two racks. We can then pull cable bundles to the midpoint of the co-located racks, where each bundle is split to each rack and then further to each chassis.

Finally, manufacturing fiber in bundles is more cost effective than individual strands. Cable bundling helped reduce fiber cost (capex + opex) by nearly 40% and expedited bringup of Watchtower fabric by multiple weeks. Table 3 summarizes the bundling and cost savings.

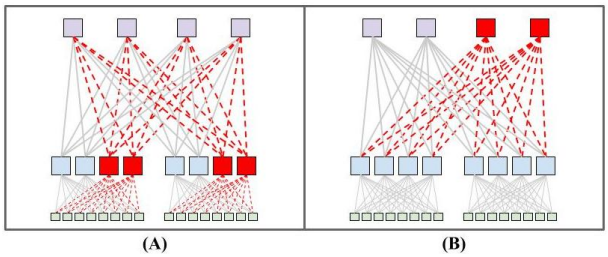

Figure 11: Two ways to depopulate the fabric for 50% capacity.

Figure 10 also depicts how we started connecting our cluster fabric to the external inter cluster networking. We defer detailed discussion to Section 4.

While Watchtower cluster fabrics were substantially cheaper and of greater scale than anything available for purchase, the absolute cost remained substantial. We used two observations to drive additional cost optimizations. First, there is natural variation in the bandwidth demands of individual clusters. Second, the dominant cost of our fabrics was in the optics and the associated fiber.

Hence, we enabled Watchtower fabrics to support depopulated deployment, where we initially deployed only 50% of the maximum bisection bandwidth. Importantly, as the bandwidth demands of a depop cluster grew, we could fully populate it to 100% bisection in place. Figure 11 shows two high-level options, (A) and (B), to depopulate switches, optics, and fiber, shown in red. (A) achieves 50% capacity by depopulating half of the S2 switches and all fiber and optics touching any depopulated S2 switch. (B) instead depopulates half S3 switches and associated fiber and optics. (A) shows 2x more depopulated elements vs. (B) for the same fabric capacity.

(A) requires all spine S3 chassis to be deployed up front even though edge aggregation blocks may be deployed slowly leading to higher initial cost. (B) has a more gradual upfront cost as all spine chassis are not deployed initially. Another advantage of (B) over (A) is that each ToR has twice the burst bandwidth.

In Watchtower and Saturn (Section 3.4) fabrics, we chose option (A) because it maximized cost savings. For Jupiter fabrics (Section 3.5), we moved to option (B) because the upfront cost of deploying the entire spine increased as we moved toward building-size fabrics and the benefits of higher ToR bandwidth became more evident.

3.4 Saturn: Fabric Scaling and 10G Servers

Saturn was the next iteration of our cluster architecture. The principal goals were to respond to continued increases in server bandwidth demands and to further increase maximum cluster scale. Saturn was built from 24x10G merchant silicon building blocks. A

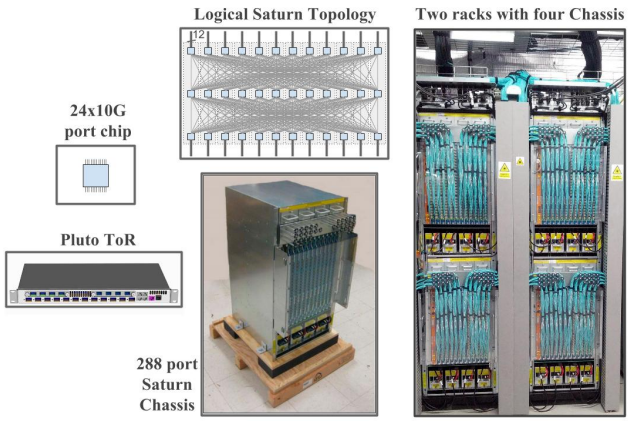

Figure 12: Components of a Saturn fabric. A 24x10G Pluto ToR Switch and a 12-linecard 288x10G Saturn chassis (including logical topology) built from the same switch chip. Four Saturn chassis housed in two racks cabled with fiber (right).

Saturn chassis supports 12-linecards to provide a 288 port non-blocking switch. These chassis are coupled with new Pluto single-chip ToR switches; see Figure 12. In the default configuration, Pluto supports 20 servers with 4x10G provisioned to the cluster fabric for an average bandwidth of 2 Gbps for each server. For more bandwidth-hungry servers, we could configure the Pluto ToR with 8x10G uplinks and 16x10G to servers providing 5 Gbps to each server. Importantly, servers could burst at 10Gbps across the fabric for the first time.

3.5 Jupiter: A 40G Datacenter-scale Fabric

As bandwidth requirements per server continued to grow, so did the need for uniform bandwidth across all clusters in the datacenter. With the advent of dense 40G capable merchant silicon, we could consider expanding our Clos fabric across the entire datacenter subsuming the inter-cluster networking layer. This would potentially enable an unprecedented pool of compute and storage for application scheduling. Critically, the unit of maintenance could be kept small enough relative to the size of the fabric that most applications could now be agnostic to network maintenance windows unlike previous generations of the network.

Jupiter, our next generation datacenter fabric, needed to scale more than 6x the size of our largest existing fabric. Unlike previous iterations, we set a requirement for incremental deployment of new network technology because the cost in resource stranding and downtime was too high. Upgrading networks by simply forklifting existing clusters stranded hosts already in production. With Jupiter, new technology would need to be introduced into the network *in situ*. Hence, the fabric must support heterogeneous hardware and speeds. Because of the sheer scale, events in the network (both planned and unplanned) were expected to

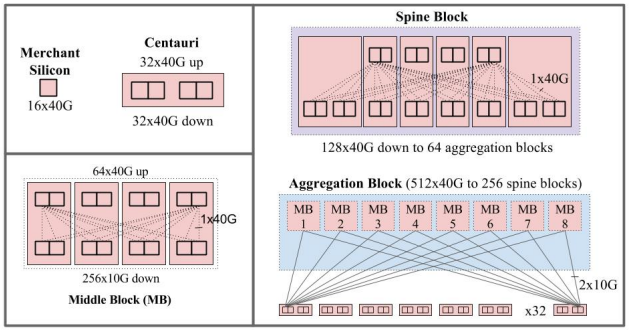

Figure 13: Building blocks used in the Jupiter topology.

Figure 14: Jupiter Middle blocks housed in racks.

be more frequent, requiring Jupiter to react robustly and gracefully to such events.

At Jupiter scale, we had to design the fabric through individual building blocks. However, the size of the building block was a key point of discussion. At one extreme was the Firehose approach, where each switch chip was cabled to others on the datacenter floor. On the other extreme, we could go the way of Watchtower and Saturn fabrics - i.e., build the largest non-blocking, two-stage chassis possible with the current merchant silicon, employing the chassis in various roles within the fabric.

For the first generation of Jupiter (Figure 13), we chose a middle path regarding building block size. Our unit of deployment was a Centauri chassis, a 4RU chassis housing two linecards, each with two switch chips with 16x40G ports controlled by a separate CPU linecard. Each port could be configured in 4x10G or 40G mode. There were no backplane data connections between these chips; all ports were accessible on the front panel of the chassis.

We employed the Centauri switch as a ToR switch with each of the 4 chips serving a subnet of machines. In one ToR configuration, we configured each chip with 48x10G to servers and 16x10G to the fabric. Servers could be configured with 40G burst bandwidth for the first time in production (see Table 2). Four Centauris made up a Middle Block (MB) for use in the aggregation block. The logical topology of an MB was a 2-stage blocking network, with 256x10G links available for ToR connectivity and 64x40G available for connectivity to the rest of the fabric through the spine.

Each ToR chip connects to eight such MBs with dual redundant 10G links. The dual redundancy aids fast reconvergence for the common case of single link failure or maintenance. Each aggregation block exposes 512x40G (full pop) or 256x40G (depop) links towards the spine blocks. Jupiter employs six Centauris in a spine block exposing 128x40G ports towards the aggregation blocks. We limited the size of Jupiter to 64 aggregation blocks for dual redundant links between each spine block and aggregation block pair at the largest scale, once again for local reconvergence on single link failure.

We deploy four MBs in a single networking rack as depicted in Figure 14. Similarly, a spine networking rack houses two pre-wired spine blocks. Cabling on the datacenter floor involves connecting fiber cable bundles between these networking racks and also to ToR switches atop server racks. In its largest configuration, Jupiter supports 1.3 Pbps bisection bandwidth among servers.

4. EXTERNAL CONNECTIVITY

4.1 WCC: Decommissioning Cluster Routers

In this section, we describe how we employed existing cluster networking building blocks to improve the performance and robustness of our inter cluster networking fabrics. Chronologically, this work took place between Watchtower and Saturn.

Through the first few Watchtower deployments, all cluster fabrics were deployed as *bag-on-the-side* networks coexisting with legacy networks (Figure 8). Time and experience ameliorated safety concerns, tipping the balance in favor of reducing the operational complexity, cost, and performance limitations of deploying two parallel networks. Limiting ToR burst bandwidth out of the cluster was particularly restrictive when migrating services or copying large search indexes across clusters.

Hence, our next goal was to decommission the Cluster Routers (CRs) by connecting the fabric directly to the inter-cluster networking layer with Cluster Border Routers (CBRs). This effort was internally called WCC. Figure 15 shows various choices for external connectivity: i) reserve some links from each ToR, ii) reserve ports in each aggregation block, iii) reserve ports in each spine block, iv) build a separate aggregation block for external connectivity. Note that i) was similar to our approach in Firehose 1.1. Further, both options i) and ii) could not improve external burst bandwidth assuming shortest path routing.

However, options iii) and iv) provide the entire pool of external bandwidth to each aggregation block. We

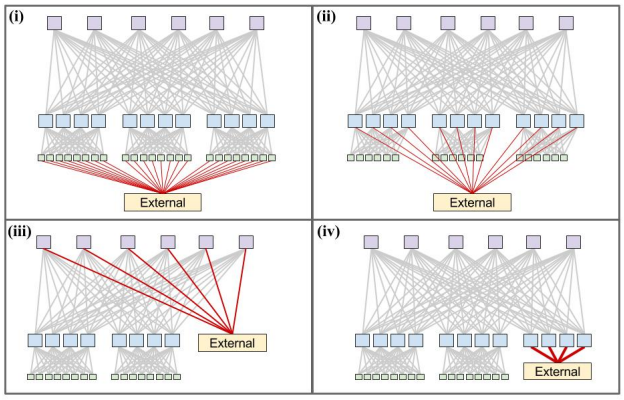

Figure 15: Four options to connect to the external network layer.

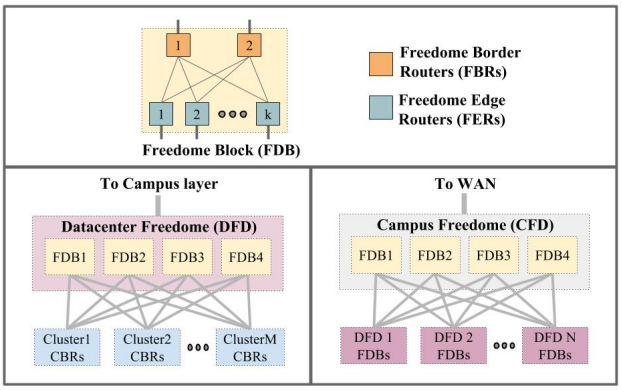

Figure 16: Two-stage fabrics used for inter-cluster and intra-campus connectivity.

chose option iv) because we wanted an isolated layer of switches to peer with external routers rather than spreading peering functionality across the entire set of spine switches. We deemed this approach safer because we wanted to limit the blast radius from an external facing configuration change and because it limited the places where we would have to integrate our in-house IGP (Section 5.2) with external routing protocols.

As a rule of thumb, we allocated 10% of aggregate intra-cluster bandwidth for external connectivity using one to three aggregation blocks. These aggregation blocks were physically and topologically identical to those used for ToR connectivity. However, we reallocated the ports normally employed for ToR connectivity to connect to external fabrics.

We configured parallel links between each CBR switch in these blocks and an external switch as Link Aggregation Groups (LAGs) or trunks. We used standard external BGP (eBGP) routing between the CBRs and the inter-cluster networking switches. CBR switches learned the default route via BGP from the external peers and redistributed the route through Firepath, our intra-cluster IGP protocol (Section 5.2).

WCC enabled the cluster fabric to be truly standalone and unlocked high throughput bulk data transfer between clusters. Moreover, the modular hardware and software of the CBR switch would find application in diverse use cases in our networking hierarchy.

4.2 Inter-Cluster Networking

We deploy multiple clusters within the same building and multiple buildings on the same campus. Given the relationship between physical distance and network cost, our job scheduling and resource allocation infrastructure leverages campus-level and building-level locality to co-locate loosely affiliated services as close to one another as possible. The CBRs developed for WCC enabled clusters to connect to inter cluster networks with massive bandwidth. Each aggregation block supported 2.56Tbps of external connectivity in Watchtower fabrics and 5.76Tbps in Saturn fabrics. However, our external networking layers were still based on expensive and port-constrained vendor gear. The third step in the evolution of our network fabrics involved replacing vendor-based inter cluster switching. Our approach, Freedome, targets massive inter-cluster bandwidth within buildings and the campus at lower cost than existing solutions.

We employed the BGP capability we developed in our cluster routers (Section 5.2.5) to build two-stage fabrics that could speak BGP at both the inter cluster and intra campus connectivity layers. See Figure 16. We configure a collection of routers in blocks called Freedome Blocks as shown in the top figure. Each block exposes 8x more south-facing ports (cluster facing) than north-facing ports (next-level in the hierarchy). Each block has two types of switch roles; the Freedome Edge Routers delivered south-facing ports while the Freedome Border Routers delivered the north-facing ports. The Freedome Block employs eBGP to connect to both north and south facing peers. We use iBGP internal to each block with the Border Routers configured as route reflectors [6].

A Datacenter Freedome typically comprises 4 independent blocks to connect multiple clusters in the same datacenter building. Inter-cluster traffic local to the same building would travel from the source cluster's CBR layer to the Datacenter Freedome, typically staying local to the Edge Router layer, and finally to the CBR layer of the destination cluster. We connect the Freedome Border Router ports to the campus connectivity layer to the north. The bottom left figure in Figure 16 depicts a Datacenter Freedome. We provision 8x more bandwidth for traffic within a building than for traffic between buildings in the same campus.

Recursively, a Campus Freedome also typically comprises 4 independent Freedome Blocks to connect multiple Datacenter Freedomes in a campus on the south and the WAN connectivity layer on the north-facing side. The bottom right figure in Figure 16 depicts a Campus Freedome.

Deploying independent blocks is crucial for maintaining performance on Freedomes since each block can be independently removed from service, or *drained*, and upgraded with an aggregate capacity degradation of 25%. Once we had rolled out the Freedomes for campus networking, the BGP router would also find application in our WAN deployment [19].

5. SOFTWARE CONTROL

5.1 Discussion

As we set out to build the control plane for our network hardware, we faced the following high level tradeoff: deploy traditional decentralized routing protocols such as OSPF/IS-IS/BGP to manage our fabrics or build a custom control plane to leverage some of the unique characteristics and homogeneity of our cluster network. Traditional routing protocols had the advantage of being proven and robust.

We chose to build our own control plane for a number of reasons. First, and most important, existing routing protocols did not at the time have good support for multipath, equal-cost forwarding. Second, there were no high quality open source routing stacks a decade ago. Further, it was a substantial amount of work to modify our hardware switch stack to tunnel control-protocol packets running inline between hardware line cards to protocol processes. Third, we were concerned about the protocol overhead of running broadcast-based routing protocols across fabrics of the scale we were targeting with hundreds or even thousands of switching elements. Scaling techniques like OSPF Areas [20] appeared hard to configure and to reason about [23]. Fourth, network manageability was a key concern and maintaining hundreds of independent switch stacks and, e.g., BGP configurations seemed daunting.

Our approach was driven by the need to route across a largely static topology with massive multipath. Each switch had a predefined role according to its location in the fabric and could be configured as such. A centralized solution where a route controller collected dynamic link state information and redistributed this link state to all switches over a reliable out-of-band Control Plane Network (CPN) appeared to be substantially simpler and more efficient from a computation and communication perspective. The switches could then calculate forwarding tables based on current link state as deltas relative to the underlying, known static topology that was pushed to all switches.

Overall, we treated the datacenter network as a single fabric with tens of thousands of ports rather than a collection of hundreds of autonomous switches that had to dynamically discover information about the fabric. We were, at this time, inspired by the success of large-scale distributed storage systems with a centralized manager [14]. Our design informed the control architecture for both Jupiter datacenter networks and Google's B4 WAN [19]. Details of Jupiter's control architecture are beyond the scope of this paper.

5.2 Routing

We now present the key components of Firepath, our routing architecture for Firehose, Watchtower, and Saturn fabrics. A number of these components anticipate some of the principles of modern Software Defined Networking, especially in using logically centralized state and control. First, all switches are configured with the baseline or intended topology. The switches learn actual configuration and link state through pair-wise neighbor discovery. Next, routing proceeds with each switch exchanging its local view of connectivity with a centralized Firepath master, which redistributes global link state to all switches. Switches locally calculate forwarding tables based on this current view of network topology. To maintain robustness, we implement a Firepath master election protocol. Finally, we leverage standard BGP only for route exchange at the edge of our fabric, redistributing BGP-learned routes through Firepath.

5.2.1 Neighbor Discovery to Verify Connectivity

Building a fabric with thousands of cables invariably leads to multiple cabling errors. Moreover, correctly cabled links may be re-connected incorrectly after maintenance such as linecard replacement. Allowing traffic to use a miscabled link can lead to forwarding loops. Links that fail unidirectionally or develop high packet error rates should also be avoided and scheduled for replacement. To address these issues, we developed *Neighbor Discovery* (ND), an online liveness and peer correctness checking protocol.

Neighbor Discovery (ND) uses the configured view of cluster topology together with a switch's local ID to determine the expected peer IDs of its local ports. It regularly exchanges its local port ID, expected peer port ID, discovered peer port ID, and link error signal. Doing so allows ND on both ends of a link to verify correct cabling.

The Interface Manager (IFM) module on each switch's embedded stack continuously monitors the ND state of each port, declaring a port up to the routing process only if it is both PHY UP and ND UP. Linecard LEDs display the ND status of each port to assist physical debugging in the field. Our monitoring infrastructure also collects and displays all link status on various dashboards. ND also serves as a keepalive protocol to ensure peers are alive and functional. If the remote software has crashed or shut down, peer ND instances will eventually report the failure to the interface manager, which in turn will declare the interface down to the routing process.

5.2.2 Firepath

We support Layer 3 routing all the way to the ToRs via a custom Interior Gateway Protocol (IGP), Firepath. Each ToR implements a Layer 2 subnet, i.e.,

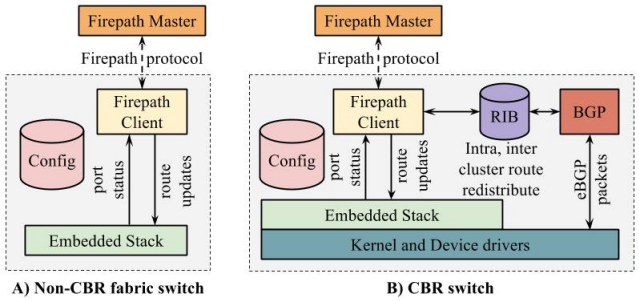

Figure 17: Firepath component interactions.

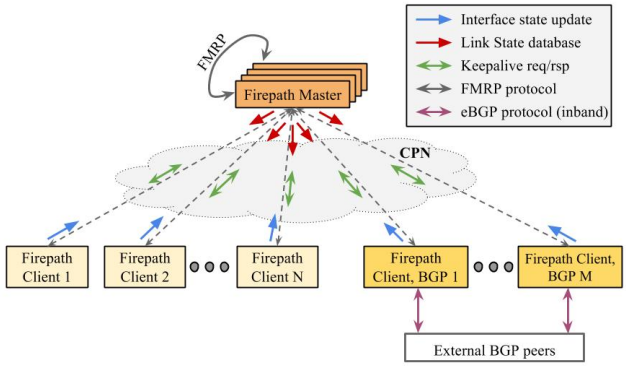

Figure 18: Protocol messages between Firepath client and Firepath master, between Firepath masters and between CBR and external BGP speakers.

all machines under one ToR are part of a broadcast domain. The L3 subnets assigned to ToRs are aligned to aid aggregation in limited forwarding tables in merchant silicon.

Firepath implements centralized topology state distribution, but distributed forwarding table computation with two main components. A Firepath client runs on each fabric switch, and a set of redundant Firepath masters run on a selected subset of spine switches. Clients communicate with the elected master over the Control Plane Network (CPN). Figure 17 shows the interaction between the Firepath client and the rest of the switch stack. Figure 18 illustrates the protocol message exchange between various routing components.

At startup, each client is loaded with the static topology of the entire fabric called the cluster config. Each client collects the state of its local interfaces from the embedded stack's interface manager and transmits this state to the master. The master constructs a Link State Database (LSD) with a monotonically increasing version number and distributes it to all clients via UDP/IP multicast over the CPN. After the initial full update, a subsequent LSD contains only the diffs from the previous state. The entire network's LSD fits within a 64KB payload. On receiving an LSD update, each client computes shortest path forwarding with Equal-Cost Multi-Path (ECMP) and programs the hardware forwarding tables local to its switch. To prevent overwhelming the

Failure Event	Recovery time (ms)	Convergence type
S2-S3 link	125	Local; affected S2, S3 chassis route around failed link
S3 chassis	325	Local; adjacent S2s route around failed S3
ToR-S2 link	4000	Non-local; all S3s avoid one S2 for impacted ToR
S2 chassis	325	Local; adjacent S3s, ToRs route around failed S2

Table 4: Summary of convergence times in a Saturn cluster.

clients, the master throttles the number of LSD changes it sends to clients.

The master also maintains a keepalive protocol with the clients. It sends periodic heartbeat messages with its master ID and the current LSD version number. If a client loses synchronization with the master, e.g., by missing an LSD message, it requests a full LSD update.

5.2.3 *Path Diversity and Convergence on Failures*

For rapid convergence on interface state change, each client computes the new routing solution and updates the forwarding tables independently upon receiving an LSD update. Since clients do not coordinate during convergence, the network can experience small transient loss while the network transitions from the old to the new state. However, assuming churn is transient, all switches eventually act on a globally consistent view of network state.

Table 4 shows the reaction time to route around component failures. Due to high path diversity, most failures require only local convergence, i.e., elements adjacent to the failure typically have multiple other viable next hops to the eventual destination. The switch's embedded stack can quickly prune the failed link/next hop from an ECMP group containing the impacted link. The ToR-S2 link failure requires non-local convergence and hence takes longer. In this case, all S3 chassis must avoid one particular S2 chassis for the IP prefix of the impacted ToR switch. Even this case can be optimized if ToRs have multiple links to an S2 switch.

Firepath LSD updates contain routing changes due to planned and unplanned network events. The frequency of such events in a typical cluster (from Figure 3) is approximately 2,000 times/month, 70 times/day, or 3 times/hour.

5.2.4 *Firepath Master Redundancy Protocol*

The centralized Firepath master is a critical component in the Firepath system. It collects and distributes interface states and synchronizes the Firepath clients via a keepalive protocol. For availability, we run redundant master instances on pre-selected spine switches. Switches know the candidate masters via their static configuration.

The Firepath Master Redundancy Protocol (FMRP) handles master election and bookkeeping between the active and backup masters. The active master maintains a keepalive with the backup masters and ensures that the current LSD is in sync with the backup masters on a CPN FMRP multicast group. On startup, a master enters an *Init* state where it invalidates its LSD and waits to hear from an existing master. If it hears a keepalive from an existing master, it enters a *backup* state. Otherwise, it enters the *electing* state where it broadcasts an election request to other master candidates. Typically, the winner of the election is the master with the latest LSD version. Alternately, a preemption mode elects the master based strictly on a priority such as highest IP address. Newly elected masters enter a *master* state.

FMRP has been robust in production over multiple years and many clusters. Since master election is *sticky*, a misbehaving master candidate does not cause changes in mastership and churn in the network. In the rare case of a CPN partition, a multi-master situation may result, which immediately alerts network operators for manual intervention.

5.2.5 Cluster Border Router

Our cluster fabrics peer with external networks via BGP. To this end, we integrated a BGP stack on the CBR with Firepath. This integration has two key aspects: i) enabling the BGP stack on the CBRs to communicate inband with external BGP speakers, and ii) supporting route exchange between the BGP stack and Firepath. Figure 17B shows the interaction between the BGP stack, Firepath, and the switch kernel and embedded stack.

For i) we created a Linux network device (netdev) for each external trunk interface running eBGP. As shown in Figure 18, BGP protocol packets flow across inband links; we use the embedded stack's packet I/O engine to vector these control packets via the netdevs to the BGP stack running on the embedded stack.

For ii) a proxy process on the CBR exchanges routes between BGP and Firepath. This process exports intra-cluster routes from Firepath into the BGP RIB and picks up inter-cluster routes from the BGP RIB, redistributing them into Firepath. We made a simplifying assumption by summarizing routes to the cluster-prefix for external BGP advertisement and the /0 default route to Firepath. In this way, Firepath manages only a single route for all outbound traffic, assuming all CBRs are viable for traffic leaving the cluster. Conversely, we assume all CBRs are viable to reach any part of the cluster from an external network. The rich path diversity inherent to Clos fabrics enables both these simplifying assumptions.

5.3 Configuration and Management

Next, we describe our approach to cluster network configuration and management prior to Jupiter. Our primary goal was to manufacture compute clusters and network fabrics as fast as possible throughout the entire fleet. Thus, we favored simplicity and reproducibility over flexibility. We supported only a limited number of fabric parameters, used to generate all the information needed by various groups to deploy the network, and built simple tools and processes to operate the network. As a result, the system was easily adopted by a wide set of technical and non-technical support personnel responsible for building data centers.

5.3.1 Configuration Generation Approach

Our key strategy was to view the entire cluster network top-down as a single static fabric composed of switches with pre-assigned roles, rather than bottom-up as a collection of switches individually configured and assembled into a fabric. We also limited the number of choices at the cluster-level, essentially providing a simple menu of fabric sizes and options, based on the projected maximum size of a cluster as well as the chassis type available.

The configuration system is a pipeline that accepts a specification of basic cluster-level parameters such as the size of the spine, base IP prefix of the cluster and the list of ToRs and their rack indexes. It then generates a set of output files for various operations groups: i) a simplified bill of materials for supply chain operations; ii) rack layout details, cable bundling and port mapping for datacenter operations; iii) CPN design and switch addressing details (e.g., DNS) for network operations; iv) updates to network and monitoring databases and systems; v) a common fabric configuration file for the switches; and vi) summary data to feed graphical views to audit the logical topology and cluster specifications.

We distribute a single monolithic cluster configuration to all switches (chassis and ToRs) in the cluster. Each switch simply extracts its relevant portion. Doing so simplifies configuration generation but every switch has to be updated with the new config each time the cluster configuration changes. Since cluster configurations do not change frequently, this additional overhead is not significant and often necessary since Firepath requires global topology state.

5.3.2 Switch Management Approach

We designed a simple management system on the switches. We did not require most of the standard network management protocols. Instead, we focused on protocols to integrate with our existing server management infrastructure. We benefited from not drawing arbitrary lines between server and network infrastructure; in fact, we set out to make switches essentially look like regular machines to the rest of fleet. Examples include the image management and installation, large scale monitoring, syslog collection, and global alerting.

The embedded stack exports a single Common Management Access Layer (CMAL) interface for external systems to manage the device. We limit administra-

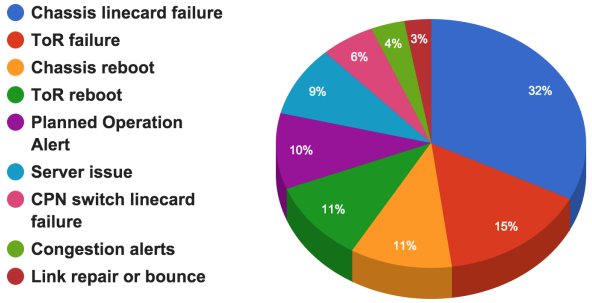

Figure 19: Alerts in Firehose/Watchtower fabrics over 9 months in '08-'09.

tive updates to draining or disabling specific ports. Since there are multiple software components running on each switch, they must all simultaneously accept a new switch configuration. Hence, we employ a standard two-phase verify-commit protocol for components on the switch orchestrated by CMAL to deploy new switch configurations.

Management clients retrieve switch status through a simple API. Important services include a local CLI for an operator to read switch status for debugging, a minimal SNMP agent to support legacy SNMP monitors, and a specific monitoring agent that exports data to the network and machine monitoring system. This last client allows us to reuse all the scalable monitoring, alerting, time-series databases (TSDB) systems built to manage our server machine fleet, saving a huge amount of work. Figure 19 presents a sample breakdown of the type of monitoring/alerts observed in our clusters for a period of 9 months in 2008-2009. The high incidence of chassis linecard failures was due to memory errors on a particular version of merchant silicon and is not reflective of a trend in linecard failure rates.

5.3.3 Fabric Operation and Management

For fabric operation and management, we continued with the theme of leveraging the existing scalable infrastructure built to manage and operate the server fleet. We built additional tools that were aware of the network fabric as a whole, thus hiding complexity in our management software. As a result, we could focus on developing only a few tools that were truly specific to our large scale network deployments, including link/switch qualification, fabric expansion/upgrade, and network troubleshooting at scale. Also important was collaborating closely with the network operations team at Google to provide training before introducing each major network fabric generation, expediting the ramp of each technology across the fleet.

Figure 20 summarizes our approach to fabric software upgrades. Rather than support in-service firmware upgrade on our switches, we exploit fabric redundancy for upgrades. We would like the degradation in fabric capacity not to exceed 25%. The figure shows two ways to upgrade the fabric chassis in multiple steps in the Clos

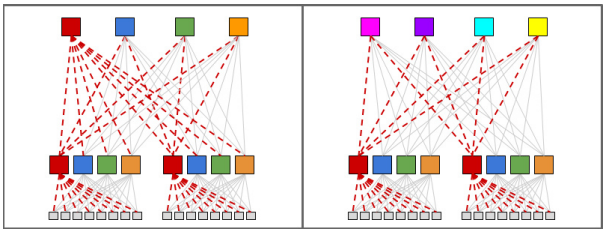

Figure 20: Multi-color fabric chassis upgrade.

topology. The left figure divides all chassis into four sets. When upgrading the red set, links in dashed red are disabled. However, the figure illustrates that the fabric capacity degrades to 56.25% (75%*75%). The right figure shows a more graceful but more time consuming upgrade process involving eight sets. Upgrading one switch at a time would take too long.

Troubleshooting misbehaving traffic flows in a network with such high path diversity is daunting for operators. Therefore, we extended debugging utilities such as traceroute and ICMP to be aware of the fabric topology. This helped with triangulating switches in the network that were potentially blackholing flows. We proactively detect such anomalies by running probes across servers randomly distributed in the cluster. On probe failures, these servers automatically run traceroutes and identify suspect failures in the network.

6. EXPERIENCE

6.1 Fabric Congestion

Despite the capacity in our fabrics, our networks experienced high congestion drops as utilization approached 25%. We found several factors contributed to congestion: i) inherent burstiness of flows led to inadmissible traffic in short time intervals typically seen as incast [8] or outcast [21]; ii) our commodity switches possessed limited buffering, which was sub optimal for our server TCP stack; iii) certain parts of the network were intentionally kept oversubscribed to save cost, e.g., the uplinks of a ToR; and iv) imperfect flow hashing especially during failures and in presence of variation in flow volume.

We used several techniques to alleviate the congestion in our fabrics. First, we configured our switch hardware schedulers to drop packets based on QoS. Thus, on congestion we would discard lower priority traffic. Second, we tuned the hosts to bound their TCP congestion window for intra-cluster traffic to not overrun the small buffers in our switch chips. Third, for our early fabrics, we employed link-level pause at ToRs to keep servers from over-running oversubscribed uplinks. Fourth, we enabled Explicit Congestion Notification (ECN) on our switches and optimized the host stack response to ECN signals [3]. Fifth, we monitored application bandwidth requirements in the face of oversubscription ratios and could provision bandwidth by deploying Pluto ToRs

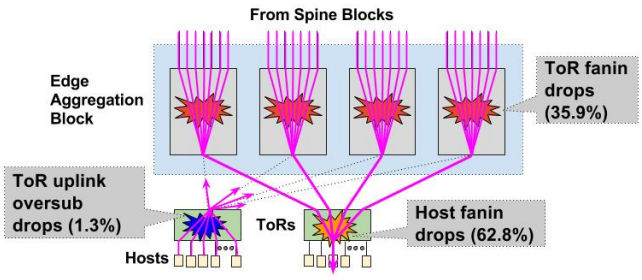

Figure 21: Congestion hotspots in a Saturn fabric.

with four or eight uplinks as required. Similarly, we could repopulate links to the spine if the depop mode of a fabric was causing congestion. Sixth, the merchant silicon had shared memory buffers used by all ports, and we tuned the buffer sharing scheme on these chips so as to dynamically allocate a disproportionate fraction of total chip buffer space to absorb temporary traffic bursts. Finally, we carefully configured switch hashing functionality to support good ECMP load balancing across multiple fabric paths.

Our congestion mitigation techniques delivered substantial improvements. We reduced the packet discard rate in a typical Clos fabric at 25% average utilization from 1% to $< 0.01\%$. Figure 21 shows the breakdown among the three principal sources of congestion in a representative Saturn cluster with 10G hosts. The largest source of loss comes from *host fanin* - traffic fanning in from the ToRs to certain hosts. The next biggest source is *ToR fanin*, which may be caused by imperfect hashing and incast communication to specific ToRs. Finally, a relatively small fraction of discards is due to oversubscription of ToR uplinks towards the fabric. Further improving fabric congestion response remains an ongoing effort.

6.2 Outages

While the overall availability of our datacenter fabrics has been satisfactory, our outages fall into three categories representing the most common failures in production: i) control software problems at scale; ii) aging hardware exposing previously unhandled failure modes; and iii) misconfigurations of certain components.

6.2.1 *Control software problems at large scale*

In the first example, a datacenter power event caused the entire fabric to restart simultaneously. However, the control software did not converge without manual intervention. The instability took place because our liveness protocol (ND) and route computation contended for limited CPU resources on embedded switch CPUs. On entire fabric reboot, routing experienced huge churn, which in turn led ND not to respond to heartbeat messages quickly enough. This in turn led to a snowball effect for routing where link state would spuriously go from up to down and back to up again. We stabilized the network by manually bringing up a few blocks at a time.

Going forward, we included the worst case fabric reboot in our test plans. Since the largest scale datacenter could never be built in a hardware test lab, we launched efforts to stress test our control software at scale in virtualized environments. We also heavily scrutinized any timer values in liveness protocols, tuning them for the worst case while balancing slower reaction time in the common case. Finally, we reduced the priority of non-critical processes that shared the same CPU.

6.2.2 *Aging hardware exposes unhandled failure modes*

Over years of deployment, our inbuilt fabric redundancy degraded as a result of aging hardware. For example, our software was vulnerable to internal/backplane link failures, leading to rare traffic blackholing. Another example centered around failures of the Control Plane Network (CPN). Each fabric chassis had dual redundant links to the CPN in active-standby mode. We initially did not actively monitor the health of both the active and standby links. With age, the vendor gear suffered from unidirectional failures of some CPN links exposing unhandled corner cases in our routing protocols. Both these problems would have been easier to mitigate had the proper monitoring and alerting been in place for fabric backplane and CPN links.

6.2.3 *Component Misconfiguration*

A prominent misconfiguration outage was on a Freedome fabric. Recall that a Freedome chassis runs the same codebase as the CBR with its integrated BGP stack. A CLI interface to the CBR BGP stack supported configuration. We did not implement locking to prevent simultaneous read/write access to the BGP configuration. During a planned BGP reconfiguration of a Freedome block, a separate monitoring system coincidentally used the same interface to read the running config while a change was underway. Unfortunately, the resulting partial configuration led to undesirable behavior between Freedome and its BGP peers.

We mitigated this error by quickly reverting to the previous configuration. However, it taught us to harden our operational tools further. It was not enough for tools to configure the fabric as a whole; they needed to do so in a safe, secure and consistent way.

7. CONCLUSION

This paper presents a retrospective on ten years and five generations of production datacenter networks. We employed complementary techniques to deliver more bandwidth to larger clusters than would otherwise be possible at any cost. We built multi-stage Clos topologies from bandwidth-dense but feature-limited merchant switch silicon. Existing routing protocols were not easily adapted to Clos topologies. We departed

from conventional wisdom to build a centralized route controller that leveraged global configuration of a predefined cluster plan pushed to every datacenter switch. This centralized control extended to our management infrastructure, enabling us to eschew complex protocols in favor of best practices from managing the server fleet. Our approach has enabled us to deliver substantial bisection bandwidth for building-scale fabrics, all with significant application benefit.

8. ACKNOWLEDGEMENTS

Many teams contributed to the success of the datacenter network within Google. In particular, we would like to acknowledge the Platforms Networking (PlaNet) Hardware and Software Development, Platforms Software Quality Assurance (SQA), Mechanical Engineering, Cluster Engineering (CE), Network Architecture and Operations (NetOps), Global Infrastructure Group (GIG), and Site Reliability Engineering (SRE) teams, to name a few. We would also like to thank our shepherd Ming Zhang as well as the anonymous SIGCOMM reviewers for their useful feedback.

9. REFERENCES

[1] Ahn, J. H., Binkert, N., Davis, A., McLaren, M., and Schreiber, R. S. HyperX: topology, routing, and packaging of efficient large-scale networks. In *Proc. High Performance Computing Networking, Storage and Analysis* (2009), ACM, p. 41.

[2] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review* (2008), vol. 38, ACM, pp. 63–74.

[3] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center TCP (DCTCP). *ACM SIGCOMM computer communication review 41*, 4 (2011), 63–74.

[4] Barroso, L. A., Dean, J., and Holzle, U. Web search for a planet: The Google cluster architecture. *Micro, Ieee 23*, 2 (2003), 22–28.

[5] Barroso, L. A., and Hölzle, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 4*, 1 (2009), 1–108.

[6] Bates, T., Chen, E., and Chandra, R. Bgp route reflection: An alternative to full mesh internal bgp (ibgp). RFC 4456, RFC Editor, April 2006. http://www.rfc-editor.org/rfc/rfc4456.txt.

[7] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 143–157.

[8] Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking* (2009), ACM, pp. 73–82.

[9] Clos, C. A Study of Non-Blocking Switching Networks. *Bell System Technical Journal 32*, 2 (1953), 406–424.

[10] Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[11] Dietz, H. G., and Mattox, T. I. KLAT2's flat neighborhood network. *Proceedings of the Extreme Linux track in the 4th Annual Linux Showcase, Atlanta, GA* (2000).

[12] Farrington, N., Rubow, E., and Vahdat, A. Data center switch architecture in the age of merchant silicon. In *Proc. HOT Interconnects, 2009. 17th IEEE Symposium on* (2009), pp. 93–102.

[13] Feamster, N., Rexford, J., and Zegura, E. The Road to SDN: An Intellectual History of Programmable Networks. *ACM Queue 11*, 12 (December 2013).

[14] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 29–43.

[15] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: a scalable and flexible data center network. In *Proc. ACM SIGCOMM Computer Communication Review* (2009), pp. 51–62.

[16] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. BCube: A high performance, server-centric network architecture for modular data centers. In *Proc. ACM SIGCOMM* (2009), pp. 63–74.

[17] Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., and Lu, S. Dcell: a scalable and fault-tolerant network structure for data centers. *ACM SIGCOMM Computer Communication Review 38*, 4 (2008), 75–86.

[18] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. ACM SIGOPS Operating Systems Review* (2007), pp. 59–72.

[19] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., and Vahdat, A. B4: Experience with a globally-deployed software defined WAN. In *Proc. ACM SIGCOMM* (2013), pp. 3–14.

[20] Moy, J. OSPF Version 2. STD 54, RFC Editor, April 1998. http://www.rfc-editor.org/rfc/rfc2328.txt.

[21] Prakash, P., Dixit, A. A., Hu, Y. C., and Kompella, R. R. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In *Proc. NSDI* (2012), pp. 413–426.

[22] Singla, A., Hong, C.-Y., Popa, L., and Godfrey, P. B. Jellyfish: Networking Data Centers Randomly. In *NSDI* (2012), vol. 12, pp. 17–17.

[23] Thorup, M. OSPF Areas Considered Harmful. IETF Internet Draft 00, individual, April 2003. http://tools.ietf.org/html/draft-thorup-ospf-harmful-00.

[24] Vahdat, A., Al-Fares, M., Farrington, N., Mysore, R. N., Porter, G., and Radhakrishnan, S. Scale-Out Networking in the Data Center. *IEEE MICRO*, 4 (August 2010), 29–41.