B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN

Chi-Yao Hong Subhasree Mandal Mohammad Al-Fares Min Zhu Richard Alimi
Kondapa Naidu B. Chandan Bhagat Sourabh Jain Jay Kaimal Shiyu Liang
Kirill Mendelev Steve Padgett Faro Rabe Saikat Ray Malveeka Tewari
Matt Tierney Monika Zahn Jonathan Zolla Joon Ong Amin Vahdat
Google
b4-sigcomm@google.com

ABSTRACT

Private WANs are increasingly important to the operation of enterprises, telecoms, and cloud providers. For example, B4, Google's private software-defined WAN, is larger and growing faster than our connectivity to the public Internet. In this paper, we present the five-year evolution of B4. We describe the techniques we employed to incrementally move from offering best-effort content-copy services to carrier-grade availability, while concurrently scaling B4 to accommodate 100x more traffic. Our key challenge is balancing the tension introduced by hierarchy required for scalability, the partitioning required for availability, and the capacity asymmetry inherent to the construction and operation of any large-scale network. We discuss our approach to managing this tension: *i)* we design a custom hierarchical network topology for both horizontal and vertical software scaling, *ii)* we manage inherent capacity asymmetry in hierarchical topologies using a novel traffic engineering algorithm without packet encapsulation, and *iii)* we re-architect switch forwarding rules via two-stage matching/hashing to deal with asymmetric network failures at scale.

CCS CONCEPTS

• **Networks** → **Network architectures**; **Routing protocols**; • **Computer systems organization** → **Availability**;

KEYWORDS

Software-Defined WAN, Traffic Engineering

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230545

1 INTRODUCTION

B4 [18] is Google's private backbone network, connecting data centers across the globe (Figure 1). Its software-defined network control stacks enable flexible and centralized control, offering substantial cost and innovation benefits. In particular, by using centralized traffic engineering (TE) to dynamically optimize site to site pathing based on utilization and failures, B4 supports much higher levels of utilization and provides more predictable behavior.

B4's initial scope was limited to loss-tolerant, lower availability applications that did not require better than 99% availability, such as replicating indexes across data centers. However, over time our requirements have become more stringent, targeting applications with service level objectives (SLOs) of 99.99% availability. Specifically, an SLO of X% availability means that both network connectivity (i.e., packet loss is below a certain threshold) *and* promised bandwidth [25] between any pair of datacenter sites are available *X%* of the minutes in the trailing 30-day window. Table 1 shows the classification of applications into service classes with the required availability SLOs.

Matching the reported availability of carrier-grade networks initially appeared daunting. Given the inherent unreliability of long-haul links [17] as well as unavoidable downtime associated with necessary management operations [13], conventional wisdom dictates that this level of availability requires substantial over-provisioning and fast local failure recovery, e.g., within 50 milliseconds [7].

Complicating our push for better availability was the exponential growth of WAN traffic; our bandwidth requirements have grown by 100x over a five year period. In fact, this doubling of bandwidth demand every nine months is faster than all of the rest of our infrastructure components, suggesting that applications derive significant benefits from plentiful cluster to cluster bandwidth. This scalability requirement spans multiple dimensions, including aggregate network capacity, the number of data center sites, the number of TE paths, and network aggregate prefixes. Moreover, we must achieve scale and availability without downtime for existing

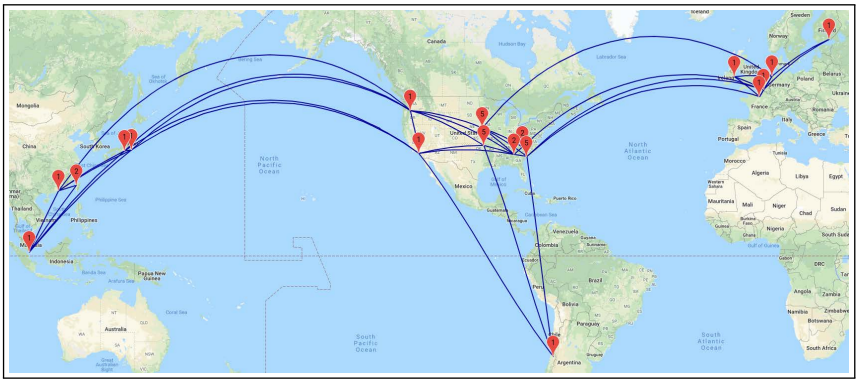


Figure 1: B4 global topology. Each marker indicates a site or multiple sites located in close geographical proximity. B4 consists of 33 sites as of January, 2018.

Service Class	Application Examples	Avail. SLO
SC4	Search ads, DNS, WWW	99.99%
SC3	Photo service backend, Email	99.95%
SC2	Ads database replication	99.9%
SC1	Search index copies, logs	99%
SC0	Bulk transfer	N/A

Table 1: Service classes, their application examples, and the assigned availability SLO.

traffic. As a result, we have had to innovate aggressively and develop novel point solutions to a number of problems.

In this paper, we present the lessons we learned from our five-year experience in managing the tension introduced by the network evolution required for achieving our availability and traffic demand requirements (§2). We gradually evolve B4 into a hierarchical topology (§3) while developing a decentralized TE architecture and scalable switch rule management (§4). Taken together, our measurement results show that our design changes have improved availability by two orders of magnitude, from 99% to 99.99%, while simultaneously supporting an increase in traffic scale of 100x over the past five years (§6).

2 BACKGROUND AND MOTIVATION

In this section, we present the key lessons we learned from our operational experience in evolving B4, the motivating examples which demonstrate the problems of alternative design choices, as well as an outline of our developed solutions.

2.1 Flat topology scales poorly and hurts availability

We learned that existing B4 site hardware topologies imposed rigid structure that made network expansion difficult. As a result, our conventional expansion practice was to add capacity by adding B4 sites next to the existing sites in a close geographical proximity. However, this practice led to three problems that only manifested at scale. First, the increasing site count significantly slowed the central TE optimization algorithm, which was operated at site-level topology. The algorithm run time increased super-linearly with the site count, and this increasing runtime caused extended periods of traffic blackholing during data plane failures, ultimately violating our availability targets. Second, increasing the site count caused scaling pressure on limited space in switch forwarding tables. Third, and most important, this practice complicated capacity planning and confused application developers. Capacity planning had to account for inter-site

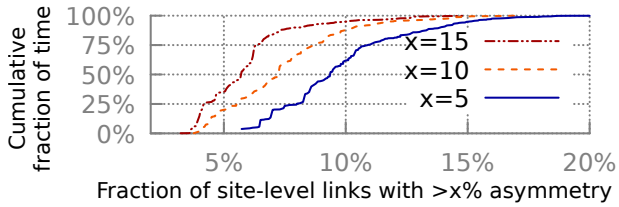


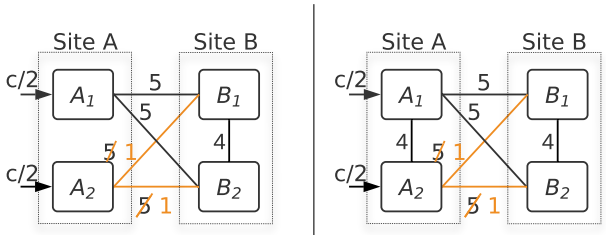
Figure 2: Fraction of site-level links with different magnitude of capacity asymmetry in B4 between October 2017 and January 2018.

WAN bandwidth constraints when compute and storage capacity were available in the close proximity but behind a different site. Developers went from thinking about regional replication across clusters to having to understand the mapping of cluster to one of multiple B4 sites.

To solve this tension introduced by exponentially increasing bandwidth demands, we redesign our hardware topology to a hierarchical topology abstraction (details in Figure 4 and §3). In particular, each site is built with a two-layer topology abstraction: At the bottom layer, we introduce a "supernode" fabric, a standard two-layer Clos network built from merchant silicon switch chips. At the top layer, we loosely connect multiple supernodes into a full mesh topology to manage incremental expansion and inherent asymmetries resulting from network maintenance, upgrades and natural hardware/software failures. Based on our operational experience, this two-layer topology provides scalability, by horizontally adding more supernodes to the top layer as needed without increasing the site count, and availability, by vertically upgrading a supernode in place to a new generation without disrupting existing traffic.

2.2 Solving capacity asymmetry in hierarchical topology

The two-layer hierarchical topology, however, also causes challenges in TE. We find that capacity asymmetry is inevitable at scale due to inherent network maintenance, operation, and data plane device instability. We design our topology to have symmetric WAN capacity at the supernode



(a) $c = 4$ **(b)** $c = 12$

Figure 3: Example of site-level topology abstraction with two sites and two supernodes each. Assume each link is annotated with its active capacity and the ingress traffic is uniformly split between A_1 and A_2. The subfigure labels show the maximum admissible traffic of the site-level link (A, B) subject to link capacity constraints. (a) without sidelinks; (b) with sidelinks.

level, i.e., all supernodes have the same outgoing configured capacity toward other sites. However, Figure 2 shows that 6-20% of the site-level links in B4 still have > 5% capacity asymmetry. We define capacity asymmetry of a site-level link as $(\text{avg}_{\forall i}(C_i) - \min_{\forall i}(C_i))/\text{avg}_{\forall i}(C_i)$, where C_i is the total active capacity of each supernode i toward next site.

Moreover, we find that capacity asymmetry significantly impedes the efficiency of our hierarchical topology—In about 17% of the asymmetric site-level links, we have 100% site-level abstraction capacity loss because at least one supernode completely loses connectivity toward another site. To understand why capacity asymmetry management is critical in hierarchical topologies, we first present a motivating example. Figure 3a shows a scenario where supernodes A_1 and A_2 respectively have 10 and 2 units of active capacity toward site B, resulting from network operation or physical link failure. To avoid network congestion, we can only admit 4 units of traffic to this site-level link, because of the bottlenecked supernode A_2 which has lower outgoing capacity. This indicates that 8 units of supernode-level capacity are wasted, as they cannot be used in the site-level topology due to capacity asymmetry.

To reduce the inefficiency caused by capacity asymmetry, we introduce *sidelinks*, which interconnect supernodes within a site in a full mesh topology. Figure 3b shows how the use of sidelinks increases admissible traffic volume by 3x in this example. The central controller dynamically rebalances traffic within a site to accommodate asymmetric WAN link failures using these sidelinks. Since supernodes of a B4 site are located in close proximity, sidelinks, like other datacenter network links, are much cheaper and significantly more reliable than long-haul WAN links. Such heterogeneous link cost/reliability is a unique WAN characteristic which motivates our design. The flexible use of sidelinks with supernode-level TE not only improves WAN link utilization but also enables incremental, in-place network upgrade, providing substantial up-front cost savings on network deployment. Specifically, with non-shortest-path TE via sidelinks, disabling supernode-level links for upgrade/maintenance does not cause any downtime for existing traffic—It merely results in a slightly degraded site-level capacity.

Evolving the existing site-level TE to a hierarchical TE (site-level, and then supernode-level) turned out to be challenging. TE typically requires tunnel encapsulation (e.g., via MPLS label [34], IP-in-IP encapsulation [30], or VLAN tag [3]), while off-the-shelf commodity switch chips can only hash on either the inner or outer layer packet header. With two layers of tunnel encapsulation, the packet header in both the inner and outer layer has very low entropy, making it impossible to enforce traffic splits. Another option is to overwrite the MAC address on each packet as a tag for supernode-level TE. However, we already reserve MAC addresses for more efficient hashing (§5). To solve this, we design and implement a novel intra-site TE algorithm (§4) which requires no packet tagging/encapsulation. Moreover, we find that our algorithm is scalable, by taking 200-700 milliseconds to run at our target scale, and effective, by reducing capacity loss due to topology abstraction to 0.6% on a typical site-level link even in the face of capacity asymmetry.

A further challenge is that TE updates can introduce routing blackholes and loops. With pre-installed tunnels, steering traffic from one tunnel to another is an *atomic* operation, as only the ingress source node needs to be updated to encapsulate traffic into a different tunnel. However, removing tunnel encapsulation complicates network updates. We find that applying TE updates in an arbitrary order results in more than 38% packet blackholing from forwarding loops in 2% of network updates (§6.3). To address this issue, earlier work enables loop-free update with two-phase commit [31] and dependency tree/forest based approach [28]. More recently, Dionysus [21] models the network update as a resource scheduling problem and uses critical path scheduling to dynamically find a feasible update schedule. However, these efforts assume tunneling/version tagging, leading to the previously described hashing problem for our hierarchical TE. We develop a simple dependency graph based solution to sequence supernode-level TE updates in a provably blackhole/loop free manner without any tunneling/tagging (§4.4).

2.3 Efficient switch rule management

Merchant silicon switches support a limited number of matching and hashing rules. Our scaling targets suggested that we would hit the limits of switch matching rules at 32 sites using our existing packet forwarding pipeline. On the other hand, hierarchical TE requires many more switch hashing entries to perform two-layer, fine-grained traffic splitting. SWAN [16] studied the tradeoffs between throughput and the switch hashing rule limits and found that dynamic tunneling requires much fewer hashing rules than a fixed tunnel set. However, we find that dynamic tunneling in hierarchical

TE still requires 8x more switch hashing rules than available to achieve maximal throughput at our target scale.

We optimize our switch forwarding behavior with two mechanisms (§5). First, we decouple flow matching rules into two switch pipeline tables. We find this hierarchical, two-phase matching mechanism increases the number of supported sites by approximately 60x. Second, we decentralize the path split rules into two-stage hashing across the edge and the backend stage of the Clos fabric. We find this optimization is key to hierarchical TE, which otherwise would offer 6% lower throughput, quite substantial in absolute terms at our scale, due to insufficient traffic split granularity.

3 SITE TOPOLOGY EVOLUTION

Table 2 summarizes how B4 hardware and the site fabric topology abstraction have evolved from a flat topology to a scalable two-stage hierarchy over the years. We next present Saturn (§3.1), our first-generation network fabric, followed by Jumpgate (§3.2), a new generation of site network fabrics with improved hardware and topology abstraction.

3.1 Saturn

Saturn was B4's first-generation network fabric, deployed globally in 2010. As shown in Figure 4, the deployment consisted of two stages: A lower stage of four Saturn chassis, offering 5.12 Tbps of cluster-facing capacity, and an upper stage of two or four Saturn chassis, offering a total of 3.2 Tbps and 6.4 Tbps respectively toward the rest of B4. The difference between cluster and WAN facing capacity allowed the fabric to accommodate additional transit traffic. For availability, we physically partitioned a Saturn site across two buildings in a datacenter. This allowed Saturn sites to continue operating, albeit with degraded capacity, if outages caused some or all of the devices in a single building to become unreachable. Each physical rack contained two Saturn chassis, and we designed the switch to rack mapping to minimize the capacity loss upon any single rack failure.

3.2 Jumpgate

Jumpgate is an umbrella term covering two flavors of B4 fabrics. Rather than inheriting the topology abstraction from Saturn, we recognize that the flat topology was inhibiting scale, and so we design a new custom hierarchical network topology in Jumpgate for both horizontal and vertical scaling of site-level capacity without impacting the scaling requirements of control software. As shown in Figure 4, we introduce the concept of a *supernode* as an intermediate topology abstraction layer. Each supernode is a 2-stage folded-Clos network. Half the ports in the lower stage are external-facing and can be flexibly allocated toward peering B4 sites, cluster fabrics, or other supernodes in the same site. We then build a Jumpgate site using a full mesh topology interconnecting supernodes. These intra-site links are called *sidelinks*. In addition, B4's availability in Saturn was significantly reduced by having a single control domain per site. We had a number of outages triggered by a faulty domain controller that caused widespread damage to all traffic passing through the affected site. Hence, in Jumpgate we partition a site into multiple control domains, one per supernode. This way, we improve availability by reducing the blast radius of any domain controller fault to traffic transiting a single supernode.

We present two generations of Jumpgate where we improve availability by partitioning the site fabric into increasingly more supernodes and more control domains across generations, as shown in Figure 4. This new architecture solves the previously mentioned network expansion problem by incrementally adding new supernodes to a site with flexible sidelink reconfigurations. Moreover, this architecture also facilitates seamless fabric evolution by sequentially upgrading each supernode in place from one generation to the next without disrupting traffic in other supernodes.

Jumpgate POP (JPOP): Strategically deploying transit-only sites improves B4's overall availability by reducing the network cable span between datacenters [1]. Transit sites also improve site-level topology "meshiness," which improves centralized TE's ability to route around a failure. Therefore, we develop JPOP, a low-cost configuration for lightweight deployment in the transit POP sites supporting only transit traffic. Since POP sites are often constrained by power and physical space, we develop JPOP with high bandwidth density 16x40G merchant silicon (Figure 4), requiring a smaller number of switch racks per site.

Stargate: We globally deprecate Saturn with *Stargate*, a large network fabric to support organic WAN traffic demand growth in datacenters. A Stargate site consists of up to four supernodes, each a 2-stage folded-Clos network built with 32x40G merchant silicon (Figure 4). Stargate is deployed in datacenters and can provide up to 81.92 Tbps site-external capacity that can be split among WAN, cluster and sidelinks.

Compared with Saturn, Stargate improves site capacity by more than 8x to keep up with growing traffic demands. A key for this growth is the increasing density of forwarding capacity in merchant silicon switch chips, which enables us to maintain a relatively simple topology. The improved capacity allows Stargate to subsume the campus aggregation network. As a result, we directly connect Stargate to Jupiter cluster fabrics [32], as demonstrated in Figure 5. This architecture change simplifies network modeling, capacity planning and management.

4 HIERARCHICAL TRAFFIC ENGINEERING

In this section, we start with two straw-man proposals for the capacity asymmetry problem (§4.1). We solve this problem by evolving from flat TE into a hierarchical TE architecture (§4.2) where a scalable, intra-site TE algorithm is

[1] long-span intercontinental cables are more vulnerable, and Layer 1 protection is typically cost prohibitive

Fabric Name	Deployed Year	Location Type	Switch Chip	#Chassis per Site	#Switches per Supernode	Site Capacity	#Switch Racks per Site	#Control Domains
Saturn	2010	Datacenter	24x10G	6 or 8	N/A	5.12/6.4T (WAN); 5.12T (cluster) 2.56/5.12T (inter-chassis)	4	1
JPOP	2013	POP	16x40G	20	24	10.24T (WAN/sidelinks)	4	2
Stargate	2014	Datacenter	32x40G	192	48	81.92T (WAN/cluster/sidelinks)	8	4

Table 2: B4 fabric generations. All the fabrics are built with merchant silicon switch chips.

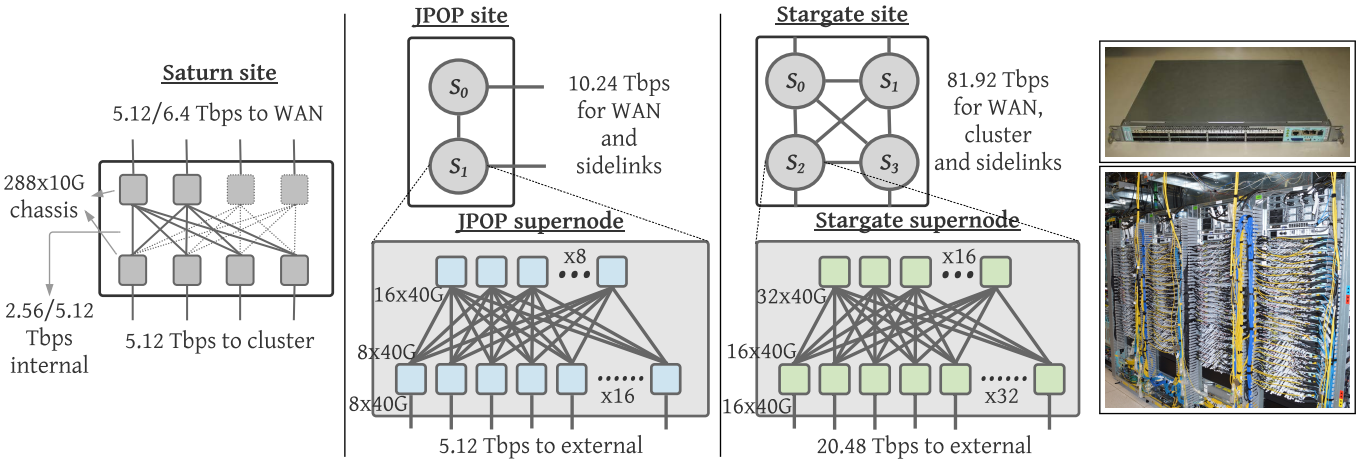


Figure 4: B4 fabric topology evolution from flat Saturn topology to hierarchical Jumpgate topology with a new intermediate topology abstraction layer called a "supernode."

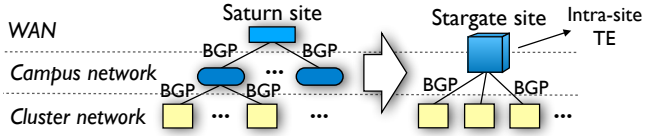


Figure 5: Stargate subsumes campus aggregation network.

developed to maximize site-level link capacity (§4.3). Finally, we present a dependency-graph based algorithm to sequence the supernode-level rule updates (§4.4). Both algorithms are highly scalable, blackhole and loop free, and do not require packet encapsulation/tagging.

4.1 Motivating Examples

Managing capacity asymmetry in hierarchical topologies requires a supernode-level load balancing mechanism to maximize capacity at the site-level topology abstraction. Additionally, we need the solution to be fast, improving availability by reducing the window of blackholing after data plane failures, and efficient, achieving high throughput within the hardware switch table limits. Finally, the solution must not require more than one layer of packet encapsulation. We discuss two straw-man proposals:

Flat TE on supernodes does not scale. With a hierarchical topology, one could apply TE directly to the full supernode-level topology. In this model, the central controller uses IP-in-IP encapsulation to load balance traffic across supernode-level tunnels. Our evaluation in indicates that supporting high throughput with this approach leads to prohibitively high runtime, 188x higher than hierarchical TE, and it also requires a much larger switch table space (details in §6). This approach is untenable because the complexity of the TE problem grows super-linearly with the size of the topology. For example, suppose that each site has four supernodes, then a single site-to-site path with three hops can be represented by $4^3 = 64$ supernode-level paths.

Supernode-level shortest-path routing is inefficient against capacity asymmetry. An alternative approach combines site-level TE with supernode-level shortest path routing. Such two-stage, hierarchical routing achieves scalability and requires only one layer of encapsulation. Moreover, shortest path routing can route around complete WAN failure via sidelinks. However, it does not properly handle capacity asymmetry. For example, in Figure 3b, shortest path routing cannot exploit longer paths via sidelinks, resulting in sub-optimal site-level capacity. One can tweak the cost metric of sidelinks to improve the abstract capacity between site A and B. However, changing metrics also affects the routing for other site-level links, as sidelink costs are shared by tunnels towards all nexthop sites.

4.2 Hierarchical TE Architecture

Figure 6 demonstrates the architecture of hierarchical TE. In particular, we employ the following pathing hierarchy:

- *Flow Group* (FG) specifies flow aggregation as a ⟨Source Site, Destination Site, Service Class⟩ tuple, where we currently map the service class to DSCP marking in the packet

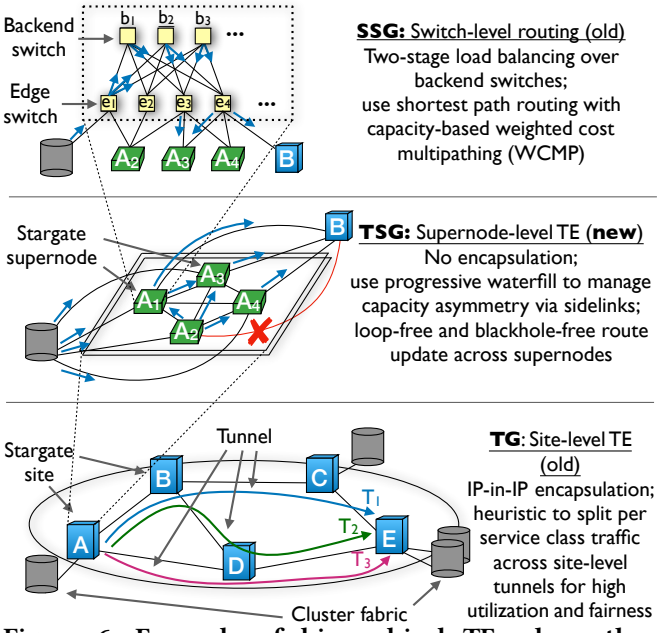


Figure 6: Example of hierarchical TE where the supernode-level TE is newly developed to handle capacity asymmetry via sidelinks.

header. For scalability, the central controller allocates paths for each FG.

- *Tunnel Group* (TG) maps an FG to a collection of tunnels (i.e., site-level paths) via IP-in-IP encapsulation. We set the traffic split with a weight for each tunnel using an approximately max-min fair optimization algorithm (§4.3 in [18]).
- *Tunnel Split Group* (TSG), a new pathing abstraction, specifies traffic distribution *within a tunnel*. Specifically, a TSG is a supernode-level rule which controls how to split traffic across supernodes in the *self-site* (other supernodes in the current site) and the *next-site* (supernodes in the tunnel's next site).
- *Switch Split Group* (SSG) specifies traffic split on each switch across physical links. The domain controller calculates SSGs.

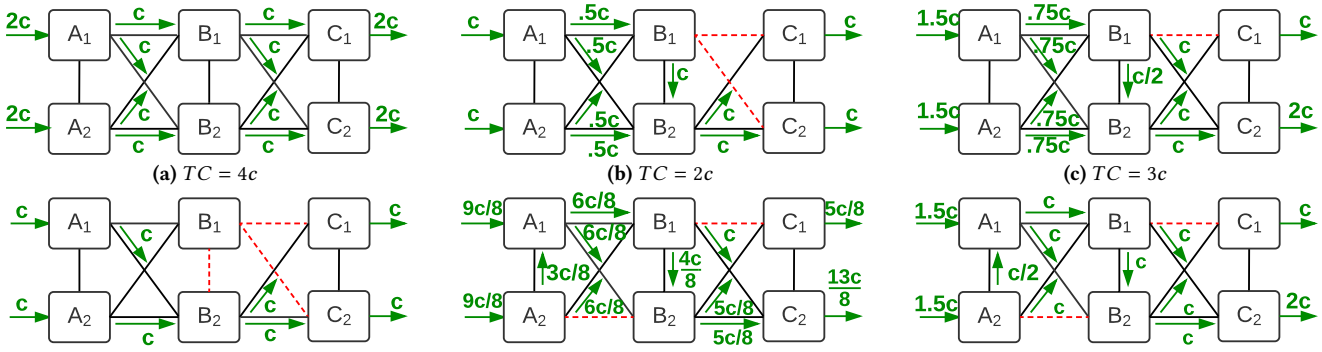
We decouple TG and TSG calculations for scalability. In particular, the TG algorithm is performed on top of a site-level abstract topology which is derived from the results of TSG calculations. TSG calculation is performed using only topology data, which is unaffected by TG algorithm results. We outline the hierarchical TE algorithm as following steps. First, the domain controller calculates supertrunk (supernode-level link) capacity by aggregating the capacity of active physical links and then adjusting capacity based on fabric impairments. For example, the supernode Clos fabric may not have full bisection bandwidth because of failure or maintenance. Second, using supertrunk capacities, the central controller calculates TSGs for each outgoing site-level link. When the outgoing site-level link capacity is symmetric across supernodes, sidelinks are simply unused. Otherwise, the central controller generates TSGs to rebalance traffic arriving at each site supernode via sidelinks to match the outgoing supertrunk capacity. This is done via a fair-share allocation algorithm on a supernode-level topology including only the sidelinks of the site and the supertrunks in the site-level link (§4.3). Third, we use these TSGs to compute the effective capacity for each link in the site-level topology, which is in turn consumed by TG generation (§4.3 in [18]). Fourth, we generate a dependency-graph to sequence TE ops in a provably blackhole-free and loop-free manner (§4.4). Finally, we program FGs, TGs, and TSGs via the domain controllers, which in turn calculate SSGs based on the intra-domain topology and implement hierarchical splitting rules across two levels in the Clos fabric for scalability (§5).

4.3 TSG Generation

Problem statement: Supposing that the incoming traffic of a tunnel is equally split across all supernodes in the source site, calculate TSGs for each supernode within each site along the tunnel such that the amount of traffic admitted into this tunnel is maximized subject to supertrunk capacity constraints. Moreover, an integer is used to represent the relative weight of each outgoing supertrunk in the TSG split. The sum of the weights on each TSG cannot exceed a threshold, T, because of switch hashing table entry limits.

Examples: We first use examples of TSG calculations for fine-grained traffic engineering within a tunnel. In Figure 7a, traffic to supernodes B_i is equally split between two supernodes to the next tunnel site, C_1 and C_2. This captures a common scenario as the topology is designed with symmetric supernode-level capacity. However, capacity asymmetry may still occur due to data-plane failures or network operations. Figure 7b demonstrates a failure scenario where B_1 completely loses connectivity to C_1 and C_2. To route around the failure, the TSG on B_1 is programmed to only forward to B_2. Figure 7c shows a scenario where B_2 has twice the capacity toward C relative to B_1. As a result, the calculated TSG for B_1 has a 2:1 ratio for the split between the *next-site* (C_2) and *self-site* (B_2). For simplicity, the TSG at the next site C does not rebalance the incoming traffic.

We calculate TSGs independently for each site-level link. For our deployments, we assume balanced incoming traffic across supernodes as we observed this assumption was rarely violated in practice. This assumption allows TSGs to be reused across all tunnels that traverse the site-level link, enables parallel TSG computations, and allows us to implement a simpler solution which can meet our switch hardware limits. We discuss two rare scenarios where our approach can lead to suboptimal TSG splits. First, Figure 7d shows a scenario where B_1 loses connectivity to both next site and self site. This scenario is uncommon and has happened only once, as the sidelinks are located inside a datacenter facility and hence much more reliable than WAN links. In this case,



(a) $TC = 4c$ (b) $TC = 2c$ (c) $TC = 3c$

(d) $TC = 0$ or $TC = 2c$ (if B_1 is disabled) (e) $TC = 9c/4$ (f) $TC = 3c$

Figure 7: Example of TSG functionality under different failure scenarios. The traffic presented here is encapsulated into a tunnel $A \rightarrow B \rightarrow C$, and TSGs control how traffic is split at the supernode level (A_i, B_i, and C_i). Each supertrunk is assumed to have capacity c, and the figure label (TC) indicates the maximum tunnel capacity with the given TSG configuration.

the site-level link (B, C) is unusable. One could manually recover the site-level link capacity by disabling B_1 (making all its incident links unusable), but it comes at the expense of capacity from B_1 to elsewhere. Second, Figure 7e shows a scenario with two consecutive asymmetric site-level links resulting from concurrent failure. Because TSG calculation is agnostic to incoming traffic balance at site B, the tunnel capacity is reduced to $9c/4$, 25% lower than the optimal splitting where B_1 forwards half of its traffic toward self-site (B_2) to accommodate the incoming traffic imbalance between B_1 and B_2, as shown in Figure 7f. Our measurements show that this failure pattern happens rarely: to only 0.47% of the adjacent site-level link pairs on average. We leave more sophisticated per-tunnel TSG generation to future work.

TSG generation algorithm: We model the TSG calculation as an independent network flow problem for each directed site-level link. We first generate a graph G_{TSG} where vertices include the supernodes in the source site of the site-level link (S_i, $1 \le i \le N$) and the destination site of the site-level link (D). We then add two types of links to the graph. First, we form a full mesh among the supernodes in the source site: $(\forall i, j \neq i : S_i \leftrightarrow S_j)$. Second, links are added between each supernode in the source site and the destination site: $(\forall i : S_i \leftrightarrow D)$. We associate the aggregate capacity of the corresponding supertrunks to each link. Subsequently, we generate flows with infinite demand from each supernode S_i toward the target site D and try to satisfy this demand by using two kinds of pathing groups (PG) with one hop and two hop paths respectively. We use a greedy exhaustive waterfill algorithm to iteratively allocate bottleneck capacity in a max-min fair manner. We present the TSG generation pseudo code in Algorithm 1.

Theorem 4.1. *The generated TSGs do not form any loop.*

Proof. Assume the generated graph has $K + 1$ vertices: S_i $(0 \le i \le K - 1)$, each represents a supernode in the

Definitions:
$G_{TSG} = (V, E)$: directed graph where vertices include:
- supernodes S_i on source site $S = \{S_i \mid 1 \le i \le N\}$.
- destination site D

$C(u, v)$: capacity of link (u, v) : $\forall (u, v) \in E$
Result:
$TSG(u, v)$: fraction of incoming traffic at u that should be forwarded to v

$C_{\text{Remaining}}(u, v) := C(u, v)$: $\forall (u, v) \in E$
Loop
  ▷ weight: fraction of flows allocated on link
  $weight(u, v) := 0$: $\forall (u, v) \in E$
  $frozen\_flow :=$ NULL
  foreach $S_i \in S$ **do**
    $PG_{2-hop}(S_i) := \{[(S_i, v), (v, D)] \mid v \in S, C_{\text{Remaining}}(S_i, v) > 0, C_{\text{Remaining}}(v, D) > 0\}$
    if $C_{Remaining}(S_i, D) > 0$ **then**
      $weight(S_i, D)$ += 1
    else if $PG_{2-hop}(S_i) \neq \{\}$ **then**
      foreach *path* $P \in PG_{2-hop}(S_i)$ **do**
        foreach $(u, v) \in P$ **do**
          $weight(u, v)$ += $\frac{1}{|PG_{2-hop}(S_i)|}$
    else
      ▷ Stop allocation when any flow failed to find path with remaining capacity
      $frozen\_flow := S_i$
      break
  if $frozen\_flow \neq$ NULL **then break**;
  $E' = \{(u, v) \in E \mid weight(u, v) > 0\}$
  $fair\_share(u, v) := \frac{C_{\text{Remaining}}(u, v)}{weight(u, v)}$: $\forall (u, v) \in E'$
  ▷ BFS: Bottleneck fair share is given by the link which offers minimum fair share
  $BFS := \min_{(u,v) \in E'} fair\_share(u, v)$
  foreach $(u, v) \in E'$ **do**
    $C_{\text{Remaining}}(u, v)$ -= $BFS \times weight(u, v)$
foreach $(u, v) \in E$ **do**
  $TSG(u, v) = \frac{C(u,v) - C_{\text{Remaining}}(u,v)}{\sum_{(u,v') \in E}(C(u,v') - C_{\text{Remaining}}(u,v'))}$

Algorithm 1: TSG generation

source site, and D represents the destination site. Given the pathing constraint imposed in step (1), each flow can only use either the one-hop direct path ($S_i \rightarrow D$) or two-hop paths ($S_i \rightarrow S_{j \neq i} \rightarrow D$), while the one-hop direct path is strictly preferred over two-hop paths. Assume a loop with $\ell > 1$ hops are formed with forwarding sequence, without loss of generality: $<S_0, S_1, ..., S_{\ell-1}, S_0>$. Note that the loop cannot contain destination vertex D. Given the pathing preference and the loop sequence, the link (S_i, D) must be bottlenecked prior to (S_{i+1}, D), and $(S_{\ell-1}, D)$ must be bottlenecked prior to (S_0, D). We observe that bottleneck sequences form a loop, which is impossible given that the algorithm only bottlenecks each link *once*. □

4.4 Sequencing TSG Updates

We find that applying TSG updates in an arbitrary order could cause transient yet severe traffic looping/blackholing (§6.3), reducing availability. Hence, we develop a scalable algorithm to sequence TSG updates in a provably blackhole-/loop-free manner as follows.

TSG sequencing algorithm: Given the graph G_{TSG} and the result of the TSGs described in §4.3, we create a *dependency graph* as follows. First, vertices in the dependency graph are the same as that in G_{TSG}. We add a directed link from S_i to S_j if S_j appears in the set of next hops for S_i in the target TSG configuration. An additional directed link from S_i to D is added if S_i forwards any traffic directly to the next-site in the target TSG configuration. According to Theorem 4.1, this dependency graph will not contain a loop and is therefore a directed acyclic graph (DAG) with an existing topological ordering. We apply TSG updates to each supernode in the reverse topological ordering, and we show that the algorithm does not cause any transient loops or blackholes during transition as follows. Note that this dependency based TSG sequencing is similar to how certain events are handled in Interior Gateway Protocol (IGP), such as link down, metric change [11] and migrations [33].

Theorem 4.2. *Assuming that neither the original nor the target TSG configuration contains a loop, none of the intermediate TSG configurations contains a loop.*

Proof. At any intermediate step, each vertex can be considered either *resolved*, in which the represented supernode forwards traffic using the target TSG, or *unresolved*, in which the traffic is forwarded using the original TSG. The vertex D is always considered resolved. We observe that the loop cannot be formed among resolved vertices, because the target TSG configuration does not contain a loop. Similarly, a loop cannot be formed among unresolved vertices, since the original TSG configuration is loop-free. Therefore, a loop can only be formed if it consists of at least one resolved vertex and at least one unresolved vertex. Thus, the loop must contain at least one link from a resolved vertex to an unresolved vertex. However, since the vertices are updated in the reverse topological ordering of the dependency graph, it is impossible for a resolved vertex to forward traffic to an unresolved vertex, and therefore a loop cannot be formed. □

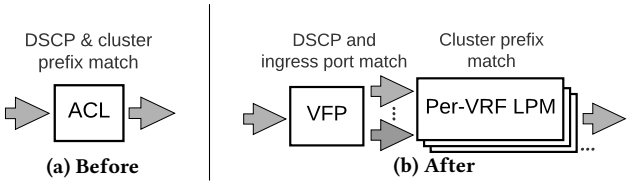


Figure 8: Decoupling FG match into two stages.

Theorem 4.3. *Consider a flow to be* blackholing *if it crosses a down link using a given TSG configuration. Assuming that the original TSG configuration may contain blackholing flows, the target TSG configuration is blackhole-free, and the set of down links is unchanged, none of the intermediate TSG configurations causes a flow to blackhole if it was not blackholing in the original TSG configuration.*

Proof. See the definition of *resolved*/*unresolved* vertex in the proof of Theorem 4.2. Assume a flow $S_i \rightarrow D$ is blackholing at an intermediate step during the transition from the original to the target TSG. Assume that this flow was not blackholing in the original TSG. Therefore, at least one transit vertex for this flow must have been resolved, and the blackhole must happen on or after the first resolved vertex. However, since the resolved vertices do not forward traffic back to unresolved vertices, the backhole can only happen in resolved vertices, which contradicts our assumption. □

5 EFFICIENT SWITCH RULE MANAGEMENT

Off-the-shelf switch chips impose hard limits on the size of each table. In this section, we show that FG matching and traffic hashing are two key drivers pushing us against the limits of switch forwarding rules to meet our availability and scalability goals. To overcome these limits, we partition our FG matching rules into two switch pipeline tables to support 60x more sites (§5.1). We further decouple the hierarchical TE splits into two-stage hashing across the switches in our two-stage Clos supernode (§5.2). Without this optimization, we find that our hierarchical TE would lose 6% throughput as a result of coarser traffic splits.

5.1 Hierarchical FG Matching

We initially implemented FG matching using Access Control List (ACL) tables to leverage their generic wildcard matching capability. The number of FG match entries was bounded by the ACL table size:

$$size_{ACL} \geq num_{Sites} \times num_{Prefixes/Site} \times num_{ServiceClasses}$$

given the ACL table size limit ($size_{ACL}$ = 3K), the number of supported service classes ($num_{ServiceClasses}$ = 6, see Table 1) and the average number of aggregate IPv4 and IPv6 cluster

Stage	Traffic Split	
	Before	After
Ingress edge switches (to backend)	TG, TSG SSG	TG TSG (part 1)
Backend switches (to egress edge switches)		TSG (part 2) SSG

Table 3: Decoupling traffic splitting rules across edge and backend switches.

prefixes per site ($num_{Prefixes/Site} \geq 16$), we anticipated hitting the ACL table limit with ~ 32 sites.

Hence, we partition FG matching into two hierarchical stages, as shown in Figure 8b. We first move the cluster prefix matches to Longest Prefix Match (LPM) table, which is much larger than the ACL table, storing up to several hundreds of thousands of entries. Even though the LPM entries cannot directly match DSCP bits for service class support, LPM entries can match against Virtual Routing Forwarding (VRF) label. Therefore, we match DSCP bits via the Virtual Forwarding Plane (VFP) table, which allows the matched packets to be associated with a VRF label to represent its service class before entering the LPM table in switch pipeline. We find this two-stage, scalable approach can support up to 1,920 sites.

This optimization also enables other useful features. We run TE as the primary routing stack and BGP/ISIS routing as a backup. In face of critical TE issues, we can disable TE at the source sites by temporarily falling back to BGP/ISIS routing. This means that we delete the TE forwarding rules at the switches in the ingress sites, so that the packets can fallback to match lower-priority BGP/ISIS forwarding rules without encapsulation. However, disabling TE end-to-end only for traffic between a single source-destination pair is more challenging, as we must also match cluster-facing ingress ports. Otherwise, even though the source site will not encapsulate packets, unencapsulated packets can follow the BGP/ISIS routes and later be incorrectly encapsulated at transit site where TE is still enabled towards the given destination. Adding ingress port matching was only feasible with the scalable, two-stage FG match.

5.2 Efficient Traffic Hashing By Partitioning

With hierarchical TE, the source site is responsible for implementing TG, TSG and SSG splits. In the original design, we collapsed the hierarchical splits and implemented them on only ingress edge switches. However, we anticipated approaching the hard limits on switch ECMP table size:

$$size_{ECMP} \geq num_{Sites} \times num_{PathingClasses} \times num_{TGs} \times num_{TSGs} \times num_{SSGs}$$

where $num_{PathingClasses} = 3$ is the number of aggregated service classes which share common pathing constraints, $num_{TGs} = 4$ is the tunnel split granularity, $num_{TSGs} = 32$ is the per-supernode split granularity, and $num_{SSGs} = 16$ splits across 16 switches in Stargate backend stage. To support up to 33 sites, we would need 198K ECMP entries while our switches support up to only $size_{ECMP} = 14K$ entries, after excluding 2K BGP/ISIS entries. We could down-quantize the traffic splits to avoid hitting the table limit. However, we find the benefit of TE would decline sharply due to the poor granularity of traffic split.

We overcome per-switch table limitations by decoupling traffic splitting rules across two levels in the Clos fabric, as shown in Table 3. First, the edge switches decide which tunnel to use (TG split) and which site the ingress traffic should be forwarded to (TSG split part 1). To propagate the decision to the backend switches, we encapsulate packets into an IP address used to specify the selected tunnel (TG split) and mark with a special source MAC address used to represent the self-/next-site target (TSG split part 1). Based on the tunnel IP address and source MAC address, backend switches decide the peer supernode the packet should be forwarded to (TSG split part 2) and the egress edge switch which has connectivity toward the target supernode (SSG split). To further reduce the required splitting rules on ingress switches, we configured a link aggregation group (LAG) for each edge switch toward *viable* backend switches. For simplicity, we consider a backend switch is viable if the switch itself is active and has active connections to every edge switch in the supernode.

6 EVALUATION

In this section, we present our evaluation of the evolved B4 network. We find that our approaches successfully scale B4 to accommodate 100x more traffic in the past five years (§6.1) while concurrently satisfying our stringent availability targets in every service class (§6.2). We also evaluate our design choices, including the use of sidelinks in hierarchical topology, hierarchical TE architecture, and the two-stage hashing mechanism (§6.3) to understand their tradeoffs in achieving our requirements.

6.1 Scale

Figure 9 demonstrates that B4 has significant growth across multiple dimensions. In particular, aggregate B4 traffic was increased by two orders of magnitude in the last five years, as shown in Figure 9a. On average, B4 traffic has doubled every nine months since its inception. B4 has been delivering more traffic and growing faster than our Internet-facing WAN. A key growth driver is that Stargate subsumed the campus aggregation network (§3.2) and started offering huge amounts of campus bandwidth in 2015.

Figure 9b shows the growth of B4 topology size as the number of sites and control domains. These two numbers matched in Saturn-based B4 (single control domain per site) and have started diverging since the deployment of JPOP fabrics in 2013. To address scalability challenges, we considerably reduced the site count by deprecating Saturn fabrics

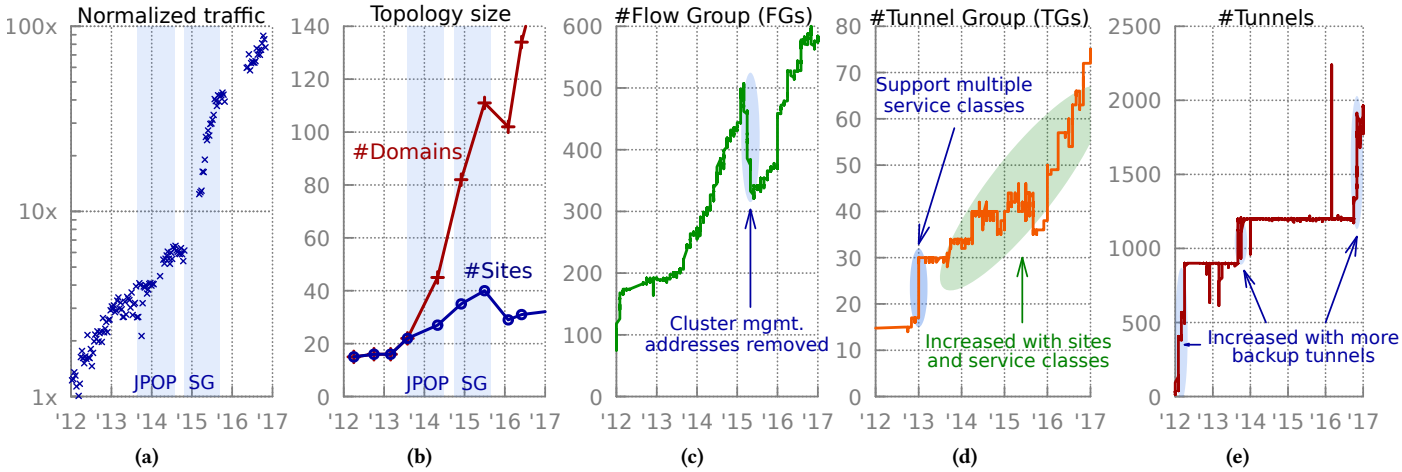


Figure 9: B4 scaling metrics over a five-year span. (a) Total amount of traffic normalized to the minimum value; traffic is aggregated across byte counters on B4's cluster-facing ports. (b) Number of B4 domains and sites. (c) Number of FGs per site. (d) Number of TGs per site. (e) Number of transit tunnels per site. The numbers are averaged and grouped into daily buckets. For per-site numbers (c-e), the largest value is presented across all the sites. JPOP: JPOP deployment period; SG: Stargate deployment period.

with Stargate in 2015. Ironically, this presented scaling challenges during the migration period because both Saturn and Stargate fabrics temporarily co-existed at a site.

Figure 9c shows the number of FGs per source site has increased by 6x in the last five years. In 2015, we stopped distributing management IP addresses for switch and supertrunk interfaces. These IP addresses cannot be aggregated with cluster prefixes, and removing these addresses helped reduce the number of FGs per site by ~ 30%.

B4 supports per service class tunneling. The initial feature was rolled out with two service classes in the beginning of 2013 and resulted in doubling the total number of TGs per site as shown in Figure 9d. After that, the TG count continues to grow with newly deployed sites and more service classes.

The maximum number of transit tunnels per site is controlled by the central controller's configuration. This constraint helps avoid installing more switch rules than available but also limits the number of backup tunnels which are needed for fast blackhole recovery upon data plane failure. In 2016, improvements to switch rule management enabled us to install more backup tunnels and improve availability against unplanned node/link failure.

6.2 Availability

We measure B4 availability at the granularity of FG via two methodologies combined together:

Bandwidth-based availability is defined as the bandwidth fulfillment ratio given by Google Bandwidth Enforcer [25]:

$$\frac{allocation}{\min\{demand, approval\}}$$

where *demand* is estimated based on short-term historical usage, *approval* is the approved minimum bandwidth per SLO contract, and *allocation* is the current bandwidth admitted by bandwidth enforcement, subject to network capacity and fairness constraints. This metric alone is insufficient because of bandwidth enforcement reaction delay and the inaccuracy of bandwidth estimation (e.g., due to TCP backoff during congestion).

Connectivity-based availability complements the limitations of bandwidth fulfillment. A network measurement system is used to proactively send probes between all cluster pairs in each service class. The probing results are grouped into B4 site pairs using 1-minute buckets, and the availability for each bucket is derived as follows:

$$\begin{cases} 1, & loss\_rate \le \alpha \\ 10^{-\beta \times (loss\_rate - \alpha)}, & \text{otherwise} \end{cases}$$

where $\alpha = 5\%$ is a sensitivity threshold which filters out most of the transient losses while capturing the bigger loss events which affect our availability budget. Beyond the threshold, availability decreases exponentially with a decay factor $\beta = 2\%$ as the traffic loss rate increases. The rationale is that most inter-datacenter services run in a parallel worker model where blackholing the transmission of any worker can disproportionately affect service completion time.

Availability is calculated by taking the *minimum* between these two metrics. Figure 10 compares the measured and target availability in each service class. We see that initially B4 achieved lower than three nines of availability at 90th percentile flow across several service classes. At this time, B4 was not qualified to deliver SC3/SC4 traffic other than probing packets used for availability measurement. However, we see a clear improvement trend in the latest state: 4 – 7x in

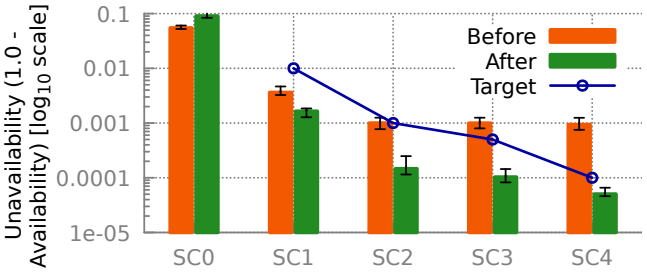


Figure 10: Network unavailability. The y-axis shows the measured and target unavailability using monthly data collected before and after the evolution, where the thick bars show 95th percentile (across site pairs), and the error bars indicate 90th and 99th percentile. x-axis shows service classes ordered by target unavailability.

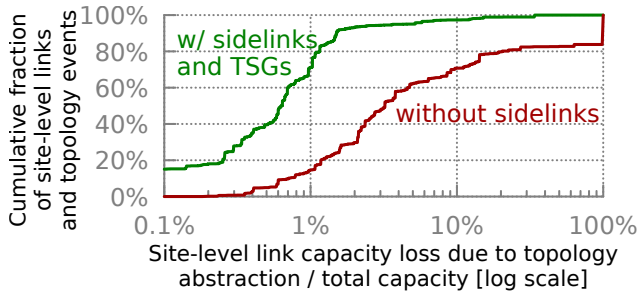


Figure 11: Capacity loss due to capacity asymmetry in hierarchical topology. The loss is normalized to the aggregate capacity at supernode-level. Only asymmetric site-level links are included. The data is collected from 21,921 topology events occurring in May 2017.

SC1/SC2 and 10 – 19x in SC3/SC4. As a result, B4 successfully achieves our availability targets in every service class.

6.3 Design Tradeoffs

Topology abstraction: Figure 11 demonstrates the importance of sidelinks in hierarchical topology. Without sidelinks, the central controller relies on ECMP to uniformly split traffic across supernodes toward the next site in the tunnel. With sidelinks, the central controller uses TSGs to minimize the impact of capacity loss due to hierarchical topology by rebalancing traffic across sidelinks to match the asymmetric connectivity between supernodes. In the median case, abstract capacity loss reduces from 3.2% (without sidelinks) to 0.6% (with sidelinks+TSG). Moreover, we observe that the abstract capacity loss without sidelinks is 100% at 83-rd percentile because at least one supernode has lost connectivity toward the next site. Under such failure scenarios, the vast majority of capacity loss can be effectively alleviated with sidelinks.

Hierarchical TE with two-stage hashing: Figure 12 quantifies the tradeoff between throughput and runtime as a function of TE algorithm and traffic split granularity. We compare

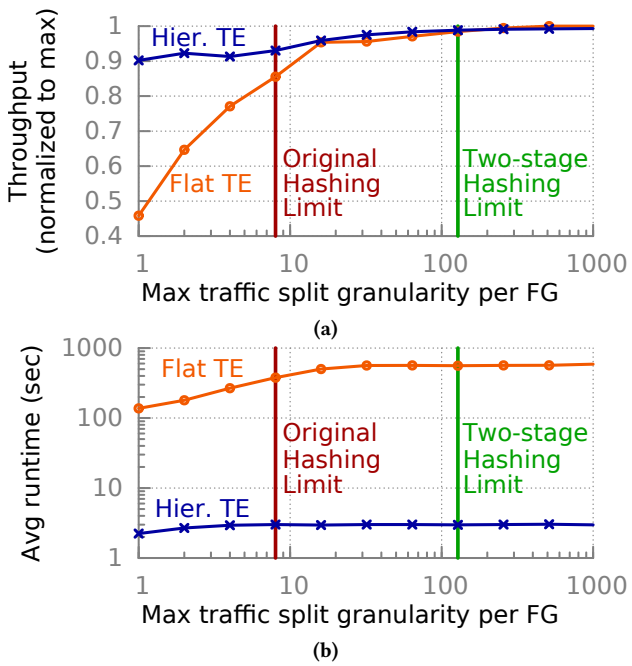


Figure 12: Hierarchical TE with two-stage hashing takes only 2-4 seconds to run and achieves high throughput while satisfying switch hashing limit. The experiment uses topology and traffic usage snapshots sampled from a one-month trace in January 2018.

our new hierarchical TE approach with flat TE, which directly runs a TE optimization algorithm (§4.3 in [18]) on the supernode-level topology. To evaluate TE-delivered capacity subject to the traffic split granularity limit, we linearly scale the traffic usage of each FG measured in Jan 2018, feed the adjusted traffic usage as demand to both flat and hierarchical TE, and then feed the demands, topology and TE pathing to a bandwidth allocation algorithm (see §5 in [25]) for deriving the maximum achievable throughput based on the max-min fair demand allocations.

We make several observations from these results. First, we find that hierarchical TE takes only 2-3 seconds to run, 188 times faster than flat TE. TSG generation runtime is consistently a small fraction of the overall runtime, ranging from 200 to 700 milliseconds. Second, when the maximum traffic split granularity is set to 1, both hierarchical and flat TE perform poorly, achieving less than 91% and 46% of their maximum throughput. The reason is simple: Each FG can only take the shortest available path[2], and the lack of sufficient path diversity leaves many links under-utilized, especially for flat TE as we have exponentially more links in the supernode-level graph. Third, we see that with our original 8-way traffic splitting, hierarchical TE achieves only 94% of

[2]The shortest available path is the shortest path on the *residual topology* where the bandwidth allocated to FGs in higher-priority service classes is excluded.

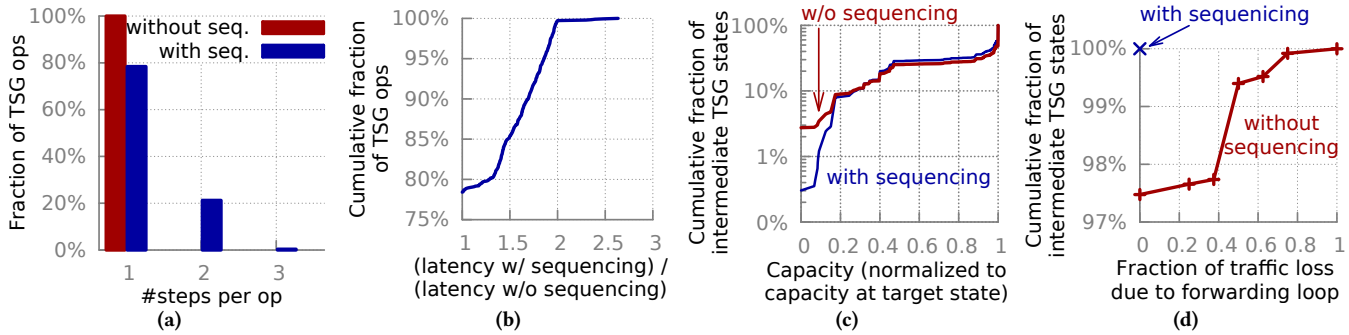


Figure 13: Performance with and without TSG sequencing using 17,167 TSG ops that require updating more than one supernode in August, 2017.

its maximum throughput. By moving to two-stage hashing, we come close to maximal throughput via 128-way TG and TSG split granularity while satisfying switch hashing limits.

TSG sequencing: We evaluate the tradeoffs with and without TSG sequencing using a one-month trace. In this experiment, we exclude 17% of TSG ops which update only one supernode. Figure 13a shows that TSG sequencing takes only one or two "steps" in > 99.7% of the TSG ops. Each step consists of one or multiple TSG updates where their relative order is not enforced. Figure 13b compares the end-to-end latency of TSG ops with and without sequencing. We observe a 2x latency increase at 99.7th percentile, while the worst case increase is 2.63x. Moreover, we find that the runtime of the TSG sequencing algorithm is negligible relative to the programming latency. Figure 13c shows the capacity available at each intermediate state during TSG ops. Without TSG sequencing, available capacity drops to zero due to blackholing/looping in ~ 3% of the cases, while this number is improved by an order of magnitude with sequencing. Figure 13d demonstrates that without sequencing, more than 38% of the ingress traffic would be discarded due to the forwarding loop formed in > 2% of the intermediate states during TSG ops.

7 OPERATIONAL EXPERIENCE AND OPEN PROBLEMS

In this section, we summarize our experiences learned from production B4 network as well as several open problems that remain active areas of work.

Management workflow simplification: Before the evolution, we rely on ECMP to uniformly load-balance traffic across Saturn chassis at each stage (§3.1), and therefore the traffic is typically bottlenecked by the chassis which has the least amount of outgoing capacity. In this design, the admissible capacity of a Saturn site drops significantly in the presence of capacity asymmetry resulting from failure or disruptive network operation [3]. Consequently, we had to manually account for the capacity loss due to capacity asymmetry for disruptive network operations in order to ensure the degraded site capacity still meets the traffic demand requirements. Jumpgate's improved handling of asymmetry using sidelinks and TSGs has reduced the need for manual interaction, as the TE system can automatically use the asymmetric capacity effectively.

By virtue of Jumpgate's multiple independent control domains per site (§3.2), we now restrict operations to modify one domain at a time to limit potential impact. To assess a change's impact on network availability, we perform impact analysis accounting for the projected capacity change, potential network failures, and other maintenance coinciding with the time window of network operation. We tightly couple our software controller with the impact analysis tool to accurately account for potential abstraction capacity loss due to disruptive network operations. Depending on the results, a network operation request can be approved or rejected.

To minimize potential impact on availability, we develop a mechanism called "drain" to shift traffic away from certain network entities before a disruptive change in a safe manner which prevents traffic loss. With the scale of B4, it is impractical for network operators and Site Reliability Engineers to use command-line interface (CLI) to manage each domain controller. Consequently, drain operations are invoked by management tools which orchestrate network operations through management RPCs exposed by the controllers.

Sidelink capacity planning: Sidelinks form a full mesh topology among supernodes to account for WAN capacity asymmetry caused by physical failure, network operation, and striping inefficiency. Up to 16% of B4 site capacity is dedicated to sidelinks to ensure that our TSG algorithm can fully utilize WAN capacity against common failure patterns. However, determining the optimal sidelink capacity that should be deployed to meet bandwidth guarantees is a hard provisioning problem that relies on long-term demand forecasting, cost estimates and business case analysis [5]. We are actively working on a log-driven statistical analysis framework that will allow us to plan sidelink capacity while minimizing costs in order to meet our network availability requirements.

[3] Operations like controller software upgrades are not considered disruptive as they are *hitless* by design: No packet loss and no reduction in capacity.

Imbalance ingress traffic handling: Our TSG algorithm assumes balanced incoming traffic across supernodes in a site (§4.3). This assumption simplifies our design and more importantly, it allows us to meet our switch hardware requirements at scale—Tunnels that shared the same site-level link use a common set of TSG rules programmed in switch forwarding tables. Comparing with per-tunnel TSG allocation, which requires > 1K TSG rules and exceeds our hardware rule limit, our TSG algorithm requires up to N TSG rules, where $N \approx 6$ is the number of peering site. However, it does not handle imbalance of ingress traffic. We plan to study alternative designs such as computing TSGs for each pair of adjacent site-level links. This design will handle ingress traffic imbalance based on capacity asymmetry observed in the upstream site-level link, while requiring a slightly higher number of TSG rules ($N(N-1) = 30$) in switch hardware.

8 RELATED WORK

Software-defined WAN: Several companies have also applied SDN to inter-datacenter connections, such as Facebook Express Backbone [20], Microsoft SWAN [16], Viptela [1], and VMWare VeloCloud [2]. All of these products employ a logically centralized SDN control platform. Hence, we believe that our experiences in evolving B4's control/data plane abstractions for availability and scalability can be applied to these efforts as well.

Decentralized, hierarchical network abstractions: We decouple site-level topology, TE architecture and switch forwarding rule management with hierarchical abstractions. The concept of hierarchical routing has been widely adopted historically (e.g., [24]), and several decentralized, hierarchical SDN routing architectures have been proposed more recently (e.g., Fabric [8] and Recursive SDN [29]). At a high level, B4 is built upon these concepts, and the novelty of our work lies in managing the practical challenges faced when applying these well-known concepts to a globally deployed software-defined WAN at massive scale. In particular, we propose efficient TE abstractions to achieve high availability at scale despite capacity asymmetry.

TE and WAN resource scheduling: Tempus [22] performs online temporal scheduling to minimize completion time of bulk transfers. Pretium [19] couples WAN TE with dynamic pricing to achieve efficient network usage. SOL [14] casts network problems, including TE, into a novel path-based optimization formulation. FFC [26] proactively minimizes congestion under data/control plane faults by embedding fault tolerance constraints into the TE formulation. These novel ideas are complementary and can be leveraged to improve B4's control and data plane.

Network update and switch rule management: A significant volume of work on consistent network update ensures properties such as packet connectivity, loop freedom, packet coherence, and capacity loss in SDN [6, 10, 12, 16, 18, 21, 27, 28, 31] and IGP routing [11, 33]. Unlike these efforts, our TSG sequencing solves a restricted use case using a simple, scalable mechanism without requiring any packet tagging. Moreover, we decouple switch matching and hashing rules into stages across forwarding tables and switches. The idea of decoupling forwarding rules has been used in several previous works to minimize the number of rules (e.g., One big switch abstraction [23]) and optimize update speed (e.g., BGP Prefix Independent Convergence [4] and SWIFT [15]). Our work is unique in unveiling the operational challenges faced in scaling the switch rule management for enabling hierarchical TE in a large scale SDN WAN.

9 CONCLUSION

This paper presents our experience in evolving B4 from a bulk transfer, content copy network targeting 99% availability to one that supports interactive network services requiring 99.99% availability, all while simultaneously increasing the traffic volume by a hundredfold over five years. We describe our lessons in managing the tension introduced by network expansion required for growing traffic demands, the partitioning of control domains for availability, and the capacity asymmetry inevitable in any large-scale network. In particular, we balance this tension by re-architecting B4 into a hierarchical topology while developing decentralized TE algorithms and switch rule management mechanisms to minimize the impact of capacity asymmetry in the hierarchical topology. We believe that a highly available wide area network with plentiful bandwidth offers unique benefits to many cloud services. For example, we can enable synchronous replication with transactional semantics on the serving path through services like Spanner [9] and analytics over massive data sets without having huge processing power co-located with all of the data in the same cluster (i.e., data can be reliably fetched across clusters on demand). On the other hand, the increasing availability requirements and the continuous push to ever-larger scale drive us to continue improving B4 beyond four nines of availability.

ACKNOWLEDGMENT

We acknowledge Ashby Armistead, Farhana Ashraf, Gilad Avidov, Arda Balkanay, Harsha Vardhan Bonthalala, Emilie Danna, Charles Eckman, Alex Goldhammer, Steve Gribble, Wenjian He, Luis Alberto Herrera, Mahesh Babu Kavuri, Chip Killian, Santa Kunchala, Steven Leung, John Mccullough, Marko Milivojevic, Luly Motomura, Brian Poynor, Suchitra Pratapagiri, Nagarushi Rao, Arjun Singh, Sankalp Singh, Puneet Sood, Rajababru Thatikunta, David Wetherall, Ming Zhang, Junlan Zhou, and many others for their significant contributions to the evolution of B4. We also thank our shepherd Changhoon Kim, anonymous reviewers, Nandita Dukkipati, Dennis Fetterly, Rebecca Isaacs, Jeff Mogul, Brad Morrey, and Dina Papagiannaki for their valuable feedback.

REFERENCES

[1] 2017. Viptela Inc. http://viptela.com/. (2017).

[2] 2018. VeloCloud Networks, Inc. http://www.velocloud.com/. (2018).

[3] IEEE Standard 802.1Q. 2011. IEEE standard for local and metropolitan area networks–media access control (MAC) bridges and virtual bridged local area networks. (2011).

[4] Ed. A. Bashandy, C. Filsfils, and P. Mohapatra. 2018. BGP Prefix Independent Convergence. IETF Internet Draft. (2018).

[5] Ajay Kumar Bangla, Alireza Ghaffarkhah, Ben Preskill, Bikash Koley, Christoph Albrecht, Emilie Danna, Joe Jiang, and Xiaoxue Zhao. 2015. Capacity Planning for the Google Backbone Network. In *International Symposium on Mathematical Programming (ISMP'15)*.

[6] Sebastian Brandt, Klaus-Tycho Foerster, and Roger Wattenhofer. 2016. On Consistent Migration of Flows in SDNs. In *INFOCOM'16*.

[7] Deborah Brungard, Malcolm Betts, Satoshi Ueno, Ben Niven-Jenkins, and Nurit Sprecher. 2009. Requirements of an MPLS transport profile. RFC 5654. (2009).

[8] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. 2012. Fabric: A Retrospective on Evolving SDN. In *HotSDN'12*.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *OSDI'12*.

[10] Klaus-Tycho Foerster, Ratul Mahajan, and Roger Wattenhofer. 2016. Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes. In *IFIP Networking'16*.

[11] Pierre Francois and Olivier Bonaventure. 2005. Avoiding Transient Loops during IGP convergence in IP Networks. In *INFOCOM'05*.

[12] Soudeh Ghorbani and Matthew Caesar. 2012. Walk the Line: Consistent Network Updates with Bandwidth Guarantees. In *HotSDN'12*.

[13] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. In *SIGCOMM'16*.

[14] Victor Heorhiadi, Michael K. Reiter, and Vyas Sekar. 2016. Simplifying Software-defined Network Optimization Using SOL. In *NSDI'16*.

[15] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. 2017. SWIFT: Predictive Fast Reroute. In *SIGCOMM'17*.

[16] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-driven WAN. In *SIGCOMM'13*.

[17] Gianluca Iannaccone, Chen-nee Chuah, Richard Mortier, Supratik Bhattacharyya, and Christophe Diot. 2002. Analysis of Link Failures in an IP Backbone. In *ACM SIGCOMM Workshop on Internet Measurment'02*.

[18] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined WAN. In *SIGCOMM'13*.

[19] Virajith Jalaparti, Ivan Bliznets, Srikanth Kandula, Brendan Lucier, and Ishai Menache. 2016. Dynamic Pricing and Traffic Engineering for Timely Inter-Datacenter Transfers. In *SIGCOMM'16*.

[20] Mikel Jimenez and Henry Kwok. 2017. Building Express Backbone: Facebook's New Long-haul Network. https://code.facebook.com/posts/1782709872057497/. (2017).

[21] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of Network Updates. In *SIGCOMM'14*.

[22] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. 2014. Calendaring for Wide Area Networks. In *SIGCOMM'14*.

[23] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. 2013. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *CoNEXT'13*.

[24] L. Kleinrock and F. Kamoun. 1977. Hierarchical Routing for Large Networks, Performance Evaluation and Optimization. *Computer Networks* 1, 3 (January 1977), 155–174.

[25] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM'15*.

[26] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic Engineering with Forward Fault Correction. In *SIGCOMM'14*.

[27] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM'13*.

[28] Ratul Mahajan and Roger Wattenhofer. 2013. On Consistent Updates in Software Defined Networks. In *HotNets'13*.

[29] James McCauley, Zhi Liu, Aurojit Panda, Teemu Koponen, Barath Raghavan, Jennifer Rexford, and Scott Shenker. 2016. Recursive SDN for Carrier Networks. *SIGCOMM Comput. Commun. Rev.* 46, 4 (Dec. 2016), 1–7.

[30] Charles E. Perkins. 1996. IP Encapsulation within IP. RFC 2003. (1996).

[31] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. In *SIGCOMM'12*.

[32] 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. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM'15*.

[33] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. 2011. Seamless Network-wide IGP Migrations. In *SIGCOMM'11*.

[34] Arun Viswanathan, Eric C. Rosen, and Ross Callon. 2001. Multiprotocol Label Switching Architecture. RFC 3031. (2001).