THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Decentralized cloud wide-area network traffic engineering with BlastShield

Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj, *Microsoft*

https://www.usenix.org/conference/nsdi22/presentation/krishnaswamy

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Decentralized cloud wide-area network traffic engineering with BLASTSHIELD

Umesh Krishnaswamy Rachee Singh Nikolaj Bjørner Himanshu Raj

Microsoft

Abstract

Cloud networks are increasingly managed by centralized software defined controllers. Centralized traffic engineering controllers achieve higher network throughput than decentralized implementations, but are a single point of failure in the network. Large scale networks require controllers with isolated fault domains to contain the *blast radius* of faults. In this work, we present BLASTSHIELD, Microsoft's software-defined decentralized WAN traffic engineering system. BLASTSHIELD *slices* the WAN into smaller fault domains, each managed by its own slice controller. Slice controllers independently engineer traffic in their slices to maximize global network throughput without relying on hierarchical or central coordination. BLASTSHIELD is fully deployed in Microsoft's WAN and carries a majority of the backbone traffic. BLASTSHIELD achieves similar network throughput as the previous generation centralized controller and reduces traffic loss from controller failures by 60%.

1 Introduction

Cloud wide-area networks (WANs) enable low-latency and high bandwidth cloud applications like live-video, geo-replication, and other business critical workloads. Cloud WANs are billion-dollar assets, and annually cost a hundred million dollars to maintain. To efficiently utilize their infrastructure investment, cloud providers employ centralized, software-defined traffic engineering (TE) systems. Centralized TE leverages global views of the topology and demands to maximize the network throughput.

Maximum throughput, but at what cost? The paradigm shift in WAN TE from fully decentralized switch-native protocols (*e.g.,* RSVP-TE [4]) to centralized TE controllers was driven by the throughput gains made possible by centralization [16]. After a decade of operating the software-defined WAN (SWAN) in Microsoft's backbone network, we claim that it is more important that the centralized TE controller does not become a single point of failure in the system. The impact of a TE controller fault needs to be lowered along with achieving high throughput.

Controller replication does not guarantee availability. Our operational experience with SWAN has taught us that regardless of good engineering practices (*e.g.,* code reviews, safe deployment, testing and verification), software systems will fail in production in unforeseen ways, often due to complex interactions of multiple faults. While it is hard to eliminate faults, it is crucial to contain the damage when faults inevitably occur. Despite fault-tolerant components of the SWAN TE system and replication of the centralized TE controller, an unforeseen cascade of faults led to an outage of global scope in the SWAN TE system.

In this work, we first describe the operational experiences that led us to migrate away from SWAN, the fully centralized TE system in the Microsoft cloud network (§ 2). Second, to reason about the availability of large-scale wide-area TE systems, we define *blast radius* of a TE controller as the fraction of customer or tier-0 traffic at risk due to its failure. We developed BLASTSHIELD, a WAN TE system that reduces the blast radius by slicing the global cloud WAN into smaller fault domains or *slices* (§ 3). BLASTSHIELD dials back from fully centralized to slice-decentralized TE by striking a balance between the centralized vs. distributed design principles.

BLASTSHIELD slices are independent, and do not rely on hierarchical or central coordination. Multiple WAN slices and controllers raise unique implementation challenges for BLASTSHIELD. In SWAN, a centralized controller with global view of the network, programmed TE routes in all WAN routers. In contrast, BLASTSHIELD slice controllers work independently — each with its own version of code, configuration, and view of the global network topology. Inconsistent views of the network topology can cause routing loops for inter-slice traffic in the cloud WAN. The failure of a slice controller on the path could blackhole traffic. BLASTSHIELD solves these challenges by developing a robust inter-slice routing mechanism that falls back on switch-native protocol routes in case of slice controller failures (§ 4 and § 5).

We have been operating Microsoft's backbone with BLASTSHIELD since 2020. We find that BLASTSHIELD allows us to deploy changes to the network safely without the risk of global impact. While any change in network configuration or software is accompanied by risk, the ability to deploy changes without global risk is a significant advantage. Quantitatively, BLASTSHIELD reduces the risk of traffic loss due to failure of a TE controller by 60%, compared to SWAN (§ 6).

2 Background and Motivation

In this section, we describe an outage in the SWAN network that motivated the design of BLASTSHIELD. This outage was caused by a cascade of several independent failures and its ripple effects persisted long after the root cause was resolved. The experience of resolving this incident urged us to survey the components at risk in SWAN and mechanisms to mitigate the risks. We define metrics to quantify the availability of TE controllers and design a TE system robust to global-scale outages like the one SWAN experienced.

2.1 Bad luck comes in threes

Prior to the development of BlastShield, a series of three unfortunate events occurred causing a SWAN outage of global scope. Global SWAN outages lasting more than a few minutes result in loss of several terabytes of network traffic, and are instantly observed by a global audience.

Controller removes all routes. A partially failed web request triggered the first bug that led the SWAN controller to remove all its TE routes from WAN routers. In the absence of controller routes, the traffic gets routed over shortest paths computed by the IGP [18]. This type of fallback is acceptable at a small scale, but not as a network-wide replacement.

Incorrect IGP shortest paths. Second, there were two links with misconfigured IGP link weights. The misconfiguration was inconsequential while the controller routes were present. When the controller removed its routes, these links incorrectly became a part of many shortest paths, consequently attracting more traffic than their capacity.

Delayed controller response time. An automatic recovery process could have restored the controller routes in 3 minutes, but a second controller bug incorrectly assumed that the recovering routers were undergoing maintenance, and held back programming routes on them. The longer recovery caused some internal workloads to dynamically change their traffic class to a higher tier, worsening the load and congestion in the network. The combination of these three cascading faults amplified the amount of traffic affected by the outage.

With the luxury of hindsight, we extract three key lessons from the SWAN incident:

1. **All changes have risk.** Global changes are antithetical to the availability of large-scale systems. We need an ability to gradually deploy changes, starting with staging which are production-like but without real customers, to low impact, and finally high impact regions. Global centralized TE precludes piece-wise rollout of changes.
2. **Configuration and software bugs are inevitable.** The outage occurred due to configuration and software bugs that escaped sandbox validation. While validation can be effective, it remains inherently best effort. In a nutshell, critical infrastructure like SWAN should not presume perfect pre-deployment validation.
3. **Global optimization does not preclude multiple controllers.** In the scenario, non-leader replicas of the controller had an accurate view of the network, and could have optimized traffic correctly. By partitioning the scope of TE controllers, a faulty leader in one region of the WAN would not impact controllers in other regions.

2.2 Blast Radius, Ripple and Shielding

While faults and small-scale outages occur and get rectified rapidly in our network, what stood out about the SWAN outage incident was its global scope. We define the following terms to quantify the scope of wide-area traffic engineering outages. In later sections, we use these terms to evaluate the reduction in the scope of potential outages when we deploy the new TE system, BlastShield.

Definition 1 (Blast Radius) *is the fraction of customer or tier-0 traffic at risk by a TE controller failure.*

The service level objective (SLO) is the daily average of the hourly percentage of successfully transmitted bytes. Customer or tier-0 traffic has the highest SLO of 99.999%. Discretionary traffic tiers, tier-1 and tier-2, have a lower SLO of 99.9%. Half the traffic in our network is tier-0. The TE controller routes traffic on engineered paths to optimize for congestion, latency, and diversity. When a TE controller fails by withdrawing its routes or programming incorrect routes or stops programming the network, the ensuing tier-0 loss is the blast radius of the controller.

Definition 2 (Blast Ripple) *of a controller failure is the service level degradation experienced by components that are not governed by the failing TE controller.*

The blast or failure of a TE controller can cause *ripples* and impact traffic not managed by the failing controller. The impact of the ripple is proportional to the amount of tier-0 traffic affected that is not managed by the failing controller.

Definition 3 (Blast Shielding) *is the engineering practice that minimizes the blast radius of failing components while meeting operational constraints like cost and complexity.*

We note that blast shielding does not ensure that the overall system is fault tolerant in achieving the service level objective. Fault tolerance allows the system to operate even if its components fail [3]. Table 1 covers mitigation in Microsoft's TE deployment to achieve fault tolerance and blast shielding. We highlight faults that were not addressed in SWAN's original design and are a focus of this work with 🛡.

3 Slicing the cloud WAN

The global scope of the SWAN outage inspired the design of BlastShield, the WAN traffic engineering system that has replaced SWAN in Microsoft's backbone network. BlastShield views the WAN as a collection of sites. Each site

Fault	Mitigation
Controller hardware, cluster, or site failure.	Automatic migration to geo-redundant cluster.
Network fault, *e.g.,* link failure, forwarding fault, router reboot.	Per-router agents perform local repair autonomously without controller intervention. Controller does global repair in the next TE iteration.
Network device disconnects or is unreachable by controller.	Router agents retain last programming. Controller reconnects via router management plane. Router is treated as down if failure persists. Rollback routes if disconnection is during new route programming.
Invalid, inconsistent, outdated programming by controller.	Router agents perform data plane verification. Controller programs agents with latest inputs every 3 minutes.
TE optimization failure *e.g.,* a controller withdraws its routes, or programs incorrect routes. 🛡	Divide the WAN into subgraphs with a controller per subgraph managing a small fault domain.
Malicious router agent *e.g.,* agent stalls the controller from programming other routers. 🛡	Decrease agent-controller interaction to defined subgraphs of the network.
Byzantine controller fault, *e.g.,* a controller sabotages other controllers. 🛡	Controllers acquire network inputs independently.
Zero-day fault in multiple controllers. 🛡	Diverge configurations in TE controllers.

Table 1: Fault types and their mitigation. New fault types handled by this paper are marked with 🛡.

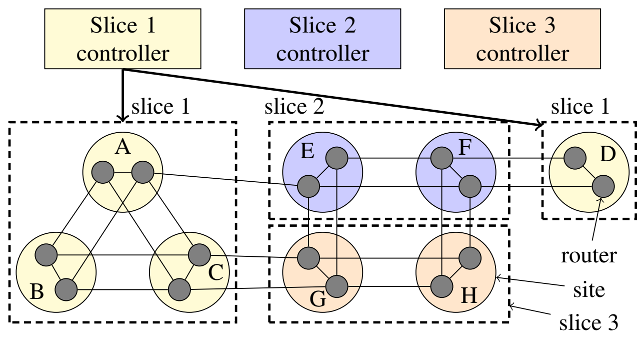


Figure 1: The WAN is divided into slices. Each slice is managed by a dedicated slice controller. Slice 1 consists of routers in sites A–D, slices 2 and 3 have routers in sites E–F and G–H.

consists of multiple WAN routers. WAN routers connect to other routers in the network like the datacenter fabric with a high bandwidth interconnect. WAN routers also transit traffic that is not from a directly connected datacenter. WAN sites at submarine landing terminals and optical transit sites do not have datacenters attached to them.

WAN Slices. BlastShield divides the WAN into *slices* or subgraphs of routers, each controlled by a dedicated slice controller. A slice is a logical partitioning of the WAN into disjoint sets of routers where each router belongs to exactly one slice. A slice can consist of a single router or all routers, or anything in between. Routers do not have any slice-specific configuration. In Fig. 1, slice 1 consists of routers in sites A–D. A slice can have multiple strongly connected components of routers. Slice 1 has two strongly connected components, the routers in sites A–C and D, respectively. Controllers 2 and 3 manage routers in sites E–F and G–H, respectively. The count and composition of slices is not limited by the design but dictated by operational choice.

Enforcing slice isolation. Only the slice's owning controller programs routers in the slice. All traffic from slice routers to any destination is engineered by the slice controller. This includes traffic that originates in datacenters directly connected to slice routers and the traffic originating in upstream slice routers. Each slice is a separate deployment and can be patched independently. Slices can inherit common configuration but BlastShield applies slice-specific configuration independently. Slice controllers do not communicate with another slice controller. This further isolates faults and prevents byzantine controllers bringing the entire system down. Slice controllers operate with a global view of the network by acquiring global topology and demand inputs. Each slice controller makes traffic engineering decisions based on expected conditions in local and remote slices. Controllers anticipate what other controllers do given the same inputs. While deviations between flow allocations computed by different controllers are possible, they are not disruptive to BlastShield's operation.

How many slices? The number of BlastShield WAN slices decide the system's operating point on an important tradeoff between network throughput and blast radius. A single slice enables the TE formulation to achieve maximum network throughput through centralization, but exposes the network to the risk of global blast radius. In contrast, several BlastShield slices reduce the blast radius of slice controllers but may also reduce the achievable network throughput. Additionally, several WAN slices increase the operational overhead of configuring and maintaining slice controllers. There is a sweet spot for the number of slices that limits the risk of changes and keeps operational overhead manageable. We empirically derive the number of BlastShield slices for Microsoft's network and strike a balance between blast radius and network throughput (§ 6).

4 BLASTSHIELD System Design

In this section we present the design of BLASTSHIELD and describe the design choices that motivated our design.

4.1 System overview

Each BLASTSHIELD slice controller is a collection of four services: topology service, demand predictor, traffic engineering scheduler, and route programmer (Fig. 2). In addition to the controller services that run on off-router compute nodes, a router agent runs on all WAN routers.

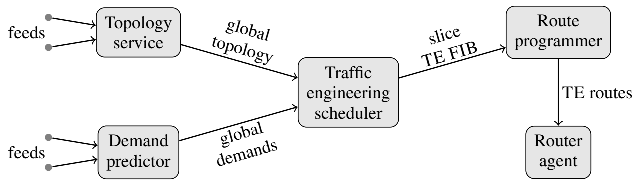


Figure 2: The slice controller consists of topology service, demand predictor, traffic engineering scheduler, and route programmer. Together, they compute traffic engineering routes and program slice routers through router agents.

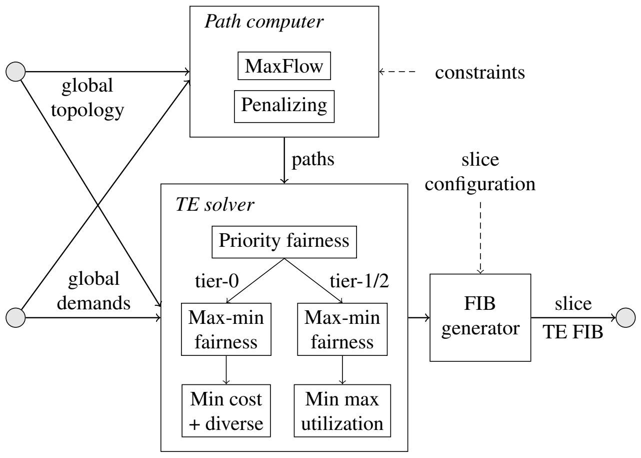


Figure 3: Traffic engineering scheduler computes routes that optimize paths for flows by traffic tier. Each controller performs global optimization based on its view of the entire network, but only programs routers belonging to its slice.

Topology Service synthesizes the global network topology using graph metadata, link state, and router agent input feeds. Graph metadata consists of routers, links, and sites. BGP-LS [15] is the primary source of dynamic link state information *e.g.,* link bandwidths, interface addresses, and segment identifiers [11]. The router agent feed is only used to acquire the health of the router agent; a router must have a functioning agent to be used for traffic engineering.

Demand Predictor predicts upcoming global network demands using a real-time traffic matrices measured by sFlow [26] and host-level packet counters. Each network demand is identified by the tuple: source router, destination site, and traffic class. Traffic class is a differentiated service queue name *e.g.,* voice, interactive, best-effort, or scavenger [5]. Tier-0 traffic uses best-effort or higher traffic classes. Tier-1 and tier-2 use the scavenger traffic class. The data feeds of the demand predictor are independently scaled out and not part of the controller.

Traffic Engineering Scheduler forms the core of the BLASTSHIELD system (Fig. 3). It ingests global network topology and global demands from topology service and demand predictor respectively. The path computer calculates paths using the dynamic topology for the source-destination pairs in the global demands. MaxFlow path computer uses maximum flow algorithms [14], and penalizing path computer computes risk diverse shortest paths using Dijkstra. Path *constraints*, described later in §§ 5.1 and 5.2, limit allowed paths in order to support the routing in BLASTSHIELD.

TE solver consists of a chain of linear programming optimization steps that place demands on multiple paths with unequal weights between demand source and destination pairs. It places tier-0 demands on paths with diversity protection that minimize latency subject to approximate max-min fairness. Lower priority demands in tier-1 and tier-2 classes are placed on paths that minimize the maximum link utilization. For brevity, we exclude the optimization problem formulations, which are previously described in [6, 16, 21, 25].

The FIB generator mechanically converts the output of the TE solver, called the *solver result*, into TE routes. The *slice configuration* specifies the subset of routers for which routes are generated. The FIB generator transforms the solver result based on the slice configuration, and produces routes only for the routers in the slice. The network is re-optimized every 3 minutes, or on topology change, whichever occurs first.

Route Programmer programs traffic engineering routes in the router agent which in turn installs them in the router. It periodically receives the full set of routes for all slice routers from the traffic engineering scheduler. This is called the traffic engineering forwarding information base (TE FIB). The FIB is organized into per-router flow and group tables (see Fig. 4). The route programmer updates all slice router agents in parallel using an update procedure, called *make-before-break*. The principle is to make all new traffic engineered paths before placing traffic on them. Intermediate FIBs build new paths, transfer traffic to the new paths, and tear down unused paths.

Router Agent runs on all WAN routers. It installs TE routes, monitors the end-to-end liveness of TE paths (*tunnels*), and modifies ingress routes based on liveness information. Route installation on the router requires translating the FIB into router platform-specific API calls. Router agents have a platform-dependent module to handle this translation. The router agent verifies tunnels within the slice using probes generated natively or with BFD [22] from tunnel ingress points.

Flows are unequally hashed to live paths based on the path weight, flow 5-tuple, and traffic class. If a path goes down, the agent proportionally distributes the weight of the down path to remaining up paths. If no path is up, then the ingress route is withdrawn, and packets are forwarded using switch-native protocol routes. This is called *local repair*.

4.2 Design considerations

Global solution at local instances. Each BLASTSHIELD slice controller consumes global network topology and demands. The solver of each controller computes flow allocations for the entire network. Therefore, each slice controller produces the same solver result if its inputs and solver software versions are the same. In practice, inputs and software versions can differ, and we study the impact of these differences in § 6.2. Although a slice controller only programs the WAN routers in its slice, it optimizes flow with a global view. Slice controllers do not communicate with each other but gather inputs from the network. Performing global optimization at each slice controller is beneficial while deploying changes to the network. Some faults involve complex interactions that only occur in unique parts of the WAN. Global inputs increase the coverage of code paths while new software or configuration changes are being deployed in small blast radius slices.

Slices as isolated routing domains. In centralized TE systems, a single controller is responsible for programming all WAN routers with the TE routes. BLASTSHIELD replaces the centralized controller with multiple slice controllers that can only program the routers within their slice. By preventing slice controllers from programming routers outside their slice, we enforce fault isolation between slices. In addition, the routing mechanisms described in § 5 ensure that the failure of one controller does not impede other controllers *e.g.,* the failure of a downstream slice controller on an inter-slice route in the WAN does not lead to blackholing of traffic. Similarly, slice controllers with inconsistent views of the network, route packets to their destination without centralized control.

Fault tolerant design. All services run on multiple machines in at least two geographically separate clusters. Topology service instances are fully active, but elect a leader to avoid oscillations if two instances report different topologies due to faults or transients. The traffic engineering scheduler and route programmer elect leaders, and switchover in case of failure. The route programmer handles all the faults and inconsistencies that can happen during programming, *e.g.,* router agents are unresponsive or have faults before, during, or after route programming. Reliable controller-agent communication is achieved by using network control traffic class, and redundant data and management plane connections. The router agent can react to network faults even when it is disconnected from the router programmer.

Decoupling TE scalability from blast shielding. BLASTSHIELD employs slice controllers to reduce the blast radius of faults in our network. We handle scale along several dimensions, unrelated to blast shielding. But slices also provide the following scaling benefits. The total number of tunnels in the network decreases because an inter-slice path is a sequence of intra-slice tunnels in BLASTSHIELD, whereas in SWAN it required its own tunnel. Second, shorter tunnels decrease tunnel probe round-trip times and speed up local repair.

5 Routing and forwarding in BLASTSHIELD

The routing of *intra-slice* flows in BLASTSHIELD is the same as SWAN. In this section, we describe BLASTSHIELD's extensions to enable routing and forwarding of *inter-slice* flows *i.e.,* flows whose traffic engineered paths span multiple slices. § 5.1 describes inter-slice routing, the approach we deployed, and § 5.2 describes a source routing approach that was evaluated but not deployed.

5.1 Inter-slice routing

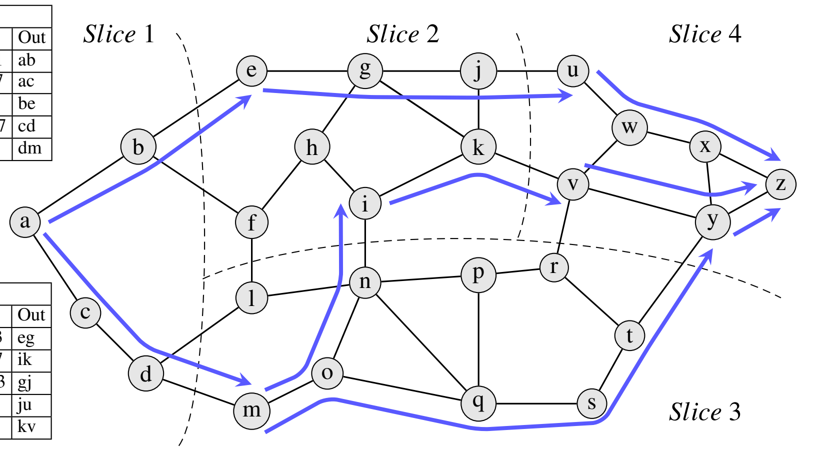
In SWAN, packets are routed using a combination of switch-native protocols and the TE controller. WAN routers connected to the datacenter fabric advertise datacenter routes with themselves as the BGP [27] next hop. BGP receivers recursively lookup the route for this BGP next hop and find multiple available routes: the shortest path route computed by the IGP, or the route programmed by the TE controller which leverages traffic engineered paths. TE routes have higher precedence than the IGP routes. The TE route encapsulates packets using Multiprotocol Label Switching (MPLS) [28] path labels from a label range reserved for the TE controller.

BLASTSHIELD routes inter-slice flows *i.e.,* flows whose traffic engineered paths span multiple slices, using *slice-local encapsulation* till the slice boundary. Slice controllers add encapsulation headers while the packet is within the slice but ensure that the packets arrive at the next slice in their *native encapsulation i.e.,* the encapsulation in which the packets entered the WAN. Each slice controller is only responsible for routing traffic to the ingress router of the next slice. Packets are encapsulated with an MPLS path label at the time of BGP route lookup on the WAN ingress router or the intermediate slice ingress routers. In both scenarios, transit routers forward the packet using the MPLS path label, and the label is popped by the penultimate router — either at a slice boundary or at the destination. Intra-slice traffic is split across TE paths only once at the WAN ingress router. Inter-slice traffic can also be split at the ingress router of an intermediate slice.

Inter-slice forwarding In Fig. 4, all four slice controllers determine that the demand from *a* to *z* should be placed on paths *abegjuwxz*, *acdmoqstyz*, and *acdmonikvyz* with weights 0.3, 0.42, and 0.28 respectively. Slice 1 programs *abe* with weight

Slice 1 routes to z				
Device	Prefix	Wt	Action	Out
a	z	0.3	push 151	ab
		0.7	push 157	ac
b	151	-	pop	be
c	157	-	swap 157	cd
d	157	-	pop	dm

Slice 2 routes to z				
Device	Prefix	Wt	Action	Out
e	z	1	push 223	eg
i	z	1	push 227	ik
g	223	-	swap 223	gj
j	223	-	pop	ju
k	227	-	pop	kv



Slice 4 routes to z				
Device	Prefix	Wt	Action	Out
u	z	1	push 443	uw
v	z	1	push 447	vy
y	z	1	-	yz
w	443	-	swap 443	wx
x	443	-	pop	xz
y	447	-	pop	yz

Slice 3 routes to z				
Device	Prefix	Wt	Action	Out
m	z	0.6	push 331	mo
		0.4	push 337	mo
o	331	-	swap 331	oq
q	331	-	swap 331	qs
s	331	-	swap 331	st
t	331	-	pop	ty
o	337	-	swap 337	on
n	337	-	pop	ni

Figure 4: Inter-slice routing using an example router-level network graph divided into four slices. The tables represent TE FIBs programmed by slice controllers using inter-slice routing. Each slice controller programs the path segment within its slice. For the path *abegjuwxz*, slice 1 programs *abe*, slice 2 programs *egju*, and slice 3 programs *uwxz*. Traffic arriving at slice ingress routers get encapsulated and split over different paths. Transit routers guide the packet along the path specified by the MPLS label. Packets return to native encapsulation at the next slice and the WAN exit.

0.3, and *acdm* with weight 0.7. Slice 2 programs *egju* and *ikv*. Slice 3 programs *moqsty* with weight 0.6, and *moni* with weight 0.4, and slice 4 programs *uwxz*, *vyz*, and *yz*. Controllers only need to install routes in their slice routers.

If any downstream slice controller fails to program routes to the destination, packets are forwarded using protocol routes along the shortest paths to the destination. Since we enable segment routing [11] with the IGP, the IGP route changes the packet encapsulation and routes the packet to the destination. For example, if the slice 2 controller withdraws all routes due to a failure, the inter-slice traffic uses shortest paths to the destination, *z*. This is the blast ripple of a down controller. In § 6.1, we will discuss how to define slice boundaries to decrease the blast ripple. Downstream slice controllers may have slightly inconsistent views due to network events like link flaps. Inter-slice traffic will be forwarded on shortest paths while the controllers converge. We show results on the alignment of multiple controllers in § 6.2.

Preventing routing loops. Unlike the TE controller in SWAN, a BLASTSHIELD slice controller is only responsible for routing packets within the slice and not until the packets' destination. Since each slice is its own routing domain, inconsistent views of the global network graph in different slice controllers can lead to routing loops.

BLASTSHIELD avoids routing loops by enforcing *enter-leave constraints* on inter-slice next hops. These constraints define the set of inter-slice next hops for all source-destination pairs in the network. The constraints ensure loop-free paths and are calculated offline using a static network graph. The path computer calculates paths on the dynamic network graph, and only allow paths that satisfy the enter-leave constraints. However, enter-leave constraints should not be overly restrictive. For example, a potential approach to preventing routing loops can limit inter-slice next hops to be on the minimum spanning tree from the source router to the destination. But this approach restricts inter-slice paths to go through a few links and causes bottlenecks.

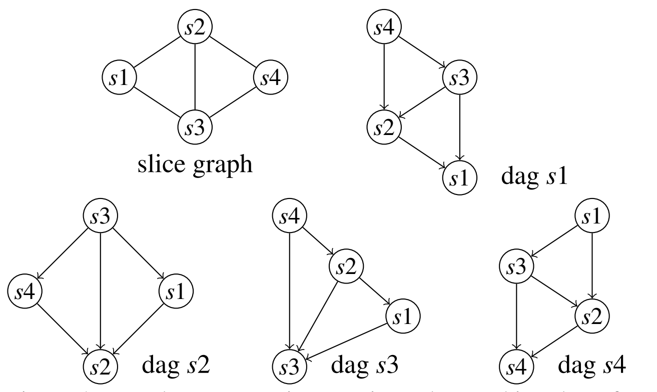


Figure 5: Enter-leave constraints restrict paths to achieve loop-free routing. Slice graph is a component level graph of Fig. 4. Slice DAGs are constructed from shortest path distances in the slice graph. Router-level paths must follow DAG edges when crossing slice boundaries. Path *acdmonikvyz* is allowed for TE because $s1 \to s3 \to s2 \to s4$ is a path in DAG $s4$. Path *abfhinprvyz* is not allowed for TE because $s2 \to s3$ is not present in DAG $s4$.

Computing enter-leave constraints. An offline generator computes enter-leave constraints from the static router-level network graph to prevent inter-slice routing loops. It first constructs a slice graph from the network graph, where each slice node represents a strongly connected component (SCC) after removing all inter-slice links. Figure 5 is the slice graph of Fig. 4, formed by removing inter-slice links *be*, *bf*, *dl*, *dm*, *fl*, *in*, *ju*, *kv*, *rv*, and *ty*, and calculating SCCs. A slice can contribute one or more SCCs as nodes to the slice graph. A link between the slice graph nodes aggregates all links between SCCs in the network graph. Link weights in the slice graph are computed from link weights in the network graph.

The enter-leave constraint generator then constructs per-destination slice DAGs based on the shortest path distances in the slice graph. The enter-leave constraints come out directly from the slice DAGs. In Fig. 5, the slice DAG for $s4$ says that paths from any node in $s1$ to any node in $s4$ can only have inter-slice transitions: $s1 \to s2 \to s4$, $s1 \to s3 \to s4$, and $s1 \to s3 \to s2 \to s4$. No controller, no matter its topology, can use any other inter-slice transition.

The path computer blacklists edges excluded by enter-leave constraints in the dynamic network graph before computing TE paths. Since the slice DAG is loop-free, paths computed by any slice controller are also loop-free. This ensures that even if slice controllers have inconsistent views of the dynamic network graph, they will arrive at loop free routes. Enter-leave constraints place restrictions on TE paths, and reduce the number of paths available to place demands. We evaluate the percentage of allowed paths vs. computed paths without constraints in § 6.1.

Verifying enter-leave constraints. Due to the negative impact of routing loops in production, and because they are global configuration, enter-leave constraints are verified offline before deployment. Enter-leave constraints are updated when there are newly provisioned routers or inter-slice links in the network. They do not need to be updated for newly provisioned intra-slice links.

We use the following formalism to define correct inter-slice routing. Let $\mathcal{R}$ be the set of defined route keys, where route key is a tuple of (router, destination prefix), **end** be the terminating route key, **null** be the undefined route key, and *ttl* be the packet time to live. Let $f : \mathcal{R} \to \mathcal{R}$, where $f(\textbf{null}) = \textbf{null}$, $f(\textbf{end}) = \textbf{end}$. Routing is a repeated application of $f()$, till $f^n(x) = \textbf{end}$ where n ranges over $1 \le n \le ttl$. The collection of TE, BGP, and the IGP routes, and their union are examples of routing functions. The routing function is complete, loops, or blackholes, if:

$$\begin{array}{llll} \forall x, \exists n: & f^n(x) & = \textbf{end} & \text{(complete)} \\ \exists x, n: & f^n(x) & = x & \text{(routing loop)} \\ \exists x, n: & f^n(x) & = \textbf{null} & \text{(blackhole)} \end{array}$$

where x ranges over $\mathcal{R} \setminus \{\textbf{end}, \textbf{null}\}$ and n ranges over $[1..ttl]$. Enter-leave constraints are verified using this formalism to detect routing loops.

5.2 Why not source routing?

In this section, we describe an alternate approach that leverages the capabilities of segment routing (SR) [11], and why we did not adopt this approach.

Loose source routing with SR. SR is a source-based routing technique that allows senders to specify the packets' route through the network by leveraging the MPLS forwarding plane. An SR router subjects arriving packets to a policy and encapsulates the matching packets in an MPLS label stack, each label represents a *segment* in the SR-path. A *node segment* causes the packet to be routed on least-cost paths computed by the IGP to the router identified by the node segment. An *adjacency segment* causes the packet to use a specified link for its next hop.

An IGP path computer models the modified Dijkstra shortest path first algorithm [18]. Coupled with segment identifiers from topology service (§ 4.1), it implements *loose source routing*. In place of explicitly listing adjacency segments of hop-by-hop links of a path, loose source routing uses a node segment when it exactly represents the sequence of the hop-by-hop links of the path. Figure 6 shows an example of loose source routing for the same paths shown in Fig. 4. The path $begjuwxz$ is composed of two shortest path segments $begju$ and $uwxz$. Hence a encapsulates with label stack of $[n(\text{u})\ n(\text{z})]$ to route to z, where $n()$ is the node segment identifier of a router.

Packet encapsulations reduce hashing entropy. To achieve balanced utilizations across links in the WAN, the cloud network employs two load balancing mechanisms. Link aggregation group hashing sprays packets on member links of a port-channel. Equal cost multi-path hashing sprays packets on the next hops of a group of traffic engineering routes. The packet processor uses fields from the packet headers to hash the packet to different output ports with the goal of maximizing entropy in the hash calculation. To achieve high entropy, the outermost IPv4/IPv6 source and destination addresses under stack of MPLS header encapsulations should be used to calculate the hash. A deep MPLS label stack can impair the ability of the packet processor to extract the relevant fields in the IP header.

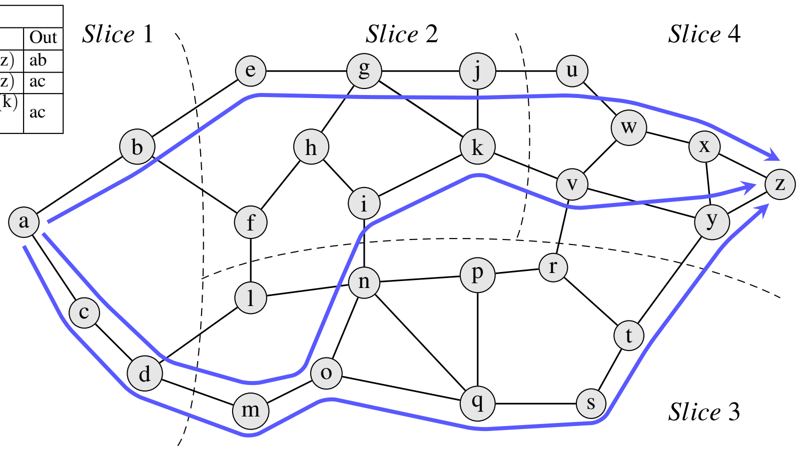
The *depth limit* is the maximum number of MPLS encapsulations a packet can have while still allowing the packet processor to extract the header fields of the original (*i.e.,* prior to MPLS encapsulations) packet. The depth limit is switch platform-dependent [2, 8, 20]. We note that if the packets entering the WAN are already encapsulated in MPLS, the depth limit available to source routing is further reduced.

Why select inter-slice routing? Based on the current generation of platforms across different regions of our cloud WAN, the depth limit is four labels. Paths that require more labels cannot be used for TE. Figure 7 studies the label stack depth needed to encode paths computed by the path computer for current and future evolutions of the WAN. In source routing, 45% of computed paths can be used for TE. For comparison, 69% of computed paths can be used for TE in inter-slice routing (see § 6.1).

Second, in source routing, a downstream slice can only transit upstream flows. In inter-slice routing, the downstream slice is free to rebalance the traffic to correct errors made upstream or mitigate for local slice conditions. This kind of control is not available with source routing.

Source routes to z				
Device	Prefix	Wt	Action	Out
a	z	0.3	push n(u) n(z)	ab
		0.42	push n(o) n(z)	ac
		0.28	push n(o) n(k) n(z)	ac

Protocol routes for *beg juwxz*			
Device	Prefix	Action	Out
b	n(u)	swap n(u)	be
e	n(u)	swap n(u)	eg
g	n(u)	swap n(u)	gj
j	n(u)	pop	ju
u	n(z)	swap n(z)	uw
w	n(z)	swap n(z)	wx
x	n(z)	pop	xz



Protocol routes for *cdmoqstyz*			
Device	Prefix	Action	Out
c	n(o)	swap n(o)	cd
d	n(o)	swap n(o)	dm
m	n(o)	pop	mo
o	n(z)	swap n(z)	oq
q	n(z)	swap n(z)	qs
s	n(z)	swap n(z)	st
t	n(z)	swap n(z)	ty
y	n(z)	pop	yz

Protocol routes for *onikvyz*			
Device	Prefix	Action	Out
o	n(k)	swap n(k)	on
n	n(k)	swap n(k)	ni
i	n(k)	pop	ik
k	n(z)	swap n(z)	kv
v	n(z)	swap n(z)	vy
y	n(z)	pop	yz

Figure 6: Source routing. Slice 1 controller programs ingress routes to z using loose source routing. The IGP with segment routing takes care of transit routes. The path *beg juwxz* is composed of two shortest path segments *beg ju* and *uwxz*. Hence the label stack for the path is n(u) n(z), where n() is the node segment identifier of a router. Weights of intra-slice links are 1 and inter-slice links are 5.

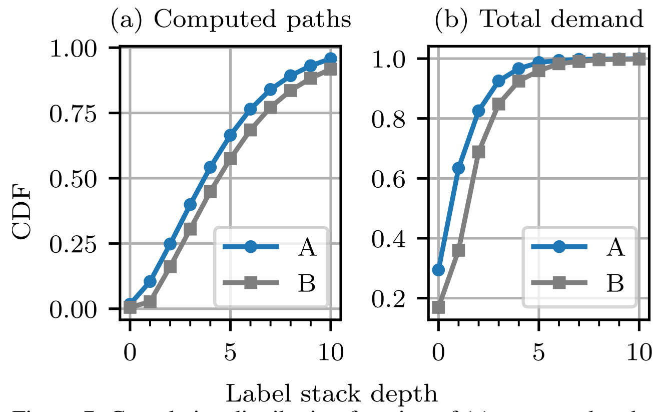


Figure 7: Cumulative distribution function of (a) computed paths and (b) total demand, by label stack depth for inputs A and B of increasing sizes. If depth limit is four, 45% of computed paths and 93% of the total demand map to allowed paths for input B.

6 Evaluating BLASTSHIELD in production

The incremental deployment of BLASTSHIELD began in 2020 and today BLASTSHIELD has replaced the legacy SWAN traffic engineering system in Microsoft's cloud network. In § 6.1, we evaluate the benefits and costs of WAN slicing using demands and topology inputs from the Microsoft backbone network for the month of July 2021. The benefit of slice-decentralized traffic engineering is the reduction in traffic loss from a slice failure. Its cost is the reduction in TE throughput due to enter-leave constraints. We quantify cost and benefit as we incrementally divide the global network into ten slices. In § 6.2, we evaluate the stochastic effects caused by multiple and independent BLASTSHIELD controllers. We show that despite the controllers having different configurations, software versions and network topology snapshots, they arrive at nearly similar flow allocations.

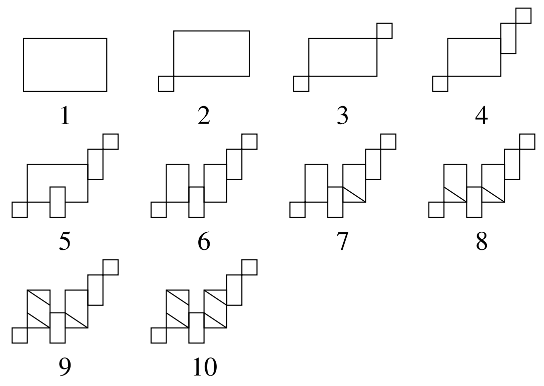


Table 2: Ten slice configurations of the global cloud network. In (1) the entire network is one slice. Slices 2–6 are formed by grouping routers in geographies. Slices 7–10 are created by further subdividing the two largest geographies, Europe and North America.

6.1 Availability vs. throughput trade-offs

We incrementally carve out slices from the global cloud network as shown in Table 2. We consider ten different slicing configurations with increasing number of slices from 1 to 10. Slice configuration 1 represents centralized traffic engineering as in SWAN. Slice configurations 2–6 are formed by drawing slice boundaries around large geographical regions like APAC, EMEA, India, North America, Oceania, and South America. In Table 2, slice configuration 2 represents the network divided into two slices: India and the rest of the world, configuration 3 represents India, Oceania, and the rest of the world, and so on. Slices 7–10 are formed by additionally dividing the two largest geographies, Europe and North America, into smaller slices. In our network, configurations 1–6 tend to have higher intra-slice traffic in comparison to inter-slice traffic. Slices have up to three strongly connected components, arising from disconnected sites and router planes.

Availability gains from decentralized TE. The key benefit

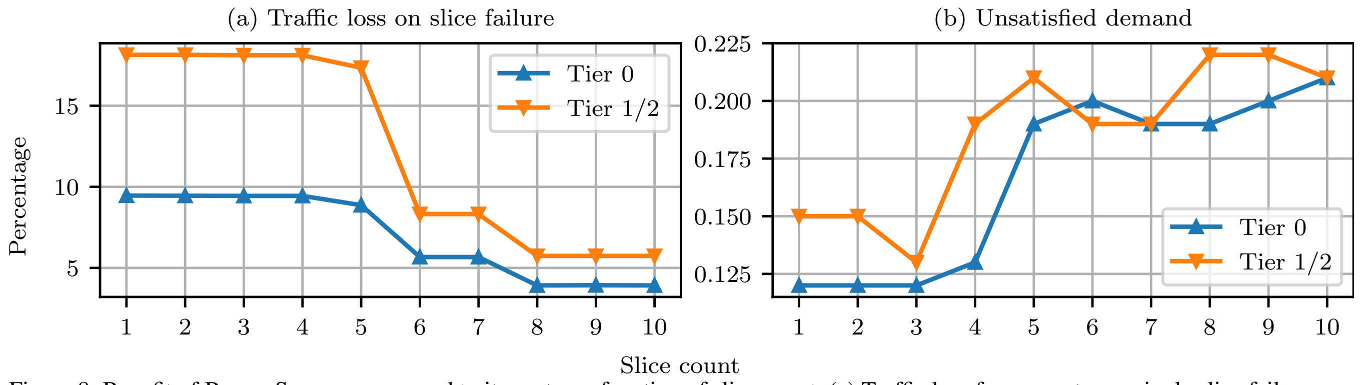


Figure 8: Benefit of BLASTSHIELD compared to its cost as a function of slice count: (a) Traffic loss from worst case single slice failure as a percentage of requested demand, (b) Unsatisfied demand due to enter-leave constraints as a percentage of requested demand.

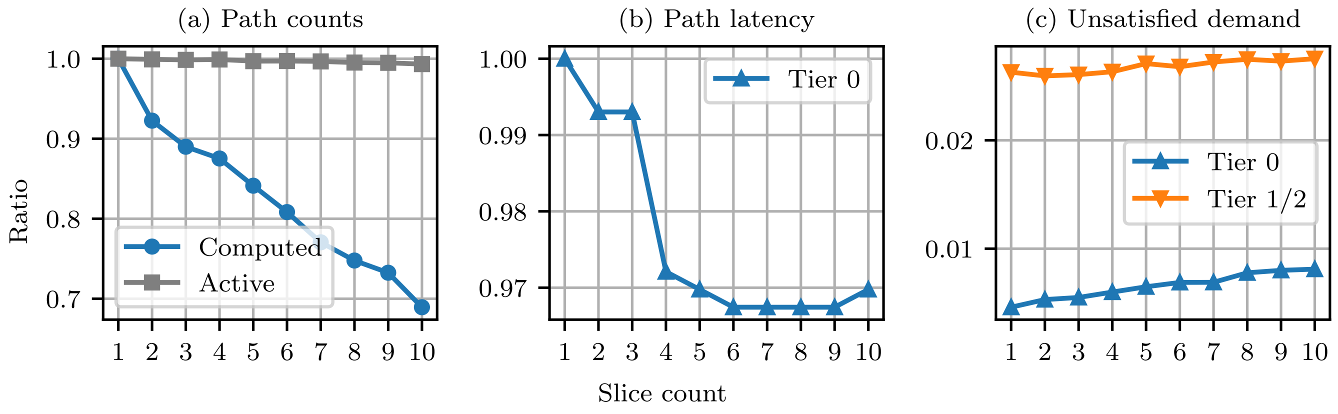


Figure 9: Stress-testing BLASTSHIELD with worst-case failures and 2× demands. (a) Computed paths are the count of paths computed with enter-leave constraints. Active paths are the count of paths used for traffic engineering. (b) Path latency is the traffic weighted average latency of active paths for tier-0 demands. (c) Unsatisfied demand is the unallocated demand per traffic tier. All values, except unsatisfied demand, are normalized to corresponding values for one slice; the latter is a ratio of unsatisfied to requested demand.

of BLASTSHIELD's slicing is the reduction in blast radius when a slice controller fails. We consider the failure where the slice controller removes all programmed TE routes. This causes the traffic to fall back on protocol routes and the ensuing traffic loss is the impact of the slice failure. We measure the traffic loss using a network simulator because the scenarios we are testing cannot be replicated in production. The inputs to the simulation are the production network demands, topology, TE and the IGP routes, and the network simulator models routing, forwarding, and queuing behavior. The simulator is used internally for capacity planning and operational safety checks, and hence is a well-tested proxy for the production network.

Figure 8 (a) shows the impact of the worst-case single slice failure when BLASTSHIELD is operating with 1–10 slices. We keep the demands and topology fixed in this experiment. For each slice configuration, we fail the largest slice by demand. The network uses the IGP routes of the failed slice and TE routes of the remaining slices (if any) to allocate the remaining demands. The traffic losses are caused by congestion due to shortest path routing over IGP routes. There are no losses due to traffic blackholes or routing loops. Figure 8 (a) shows that with ten slices, tier-0 traffic loss due to slice failure, which is the metric for blast radius, decreases by 60%, from 9.5% to 3.9%. Tier-1 and tier-2 traffic loss reduction is greater at 70% (18.1% to 5.7%) because they map to scavenger traffic class and experience more congestion losses when the failed slice uses IGP routes. Slices 2–4 show little improvement because the largest slice can still cause an overly large failure. The improvements come at six and eight slices with the breakup of Europe and North America into separate and smaller slices.

Throughput cost of decentralized TE. The key reason why inter-slice routing in BLASTSHIELD can have lower throughput than SWAN is due to the enter-leave constraints (§ 5.1). These constraints decrease the choice of paths available for placing demands, which in turn decreases the demands that can be allocated. Figure 8 (b) shows unsatisfied demand from enter-leave constraints as a percentage of requested demand. We calculate worst case unallocated demand from 20 variations of the network topology, each variation has multiple shared risk failures that reduce the available capacity. Without constraints, the worst-case unsatisfied demand is 0.27% of the requested demand, and with ten slices it increases to 0.42%. The increase in unsatisfied demand of 0.15% is much smaller than the 18% traffic loss reduction from slice failure. Additional capacity can be provisioned to decrease the unsatisfied demand.

Stress testing BLASTSHIELD. We oversubscribe the network by doubling the bandwidth values in requested demands, and test with variations of the production network with multiple shared risk group failures in hot spots of the topology. The purpose of the stress test is to evaluate the performance of enter-leave constraints in the presence of significant oversubscription. Figure 9 shows the impact of slicing on paths computed by the BLASTSHIELD path computer. Since the constraints enforce a shortest path order when crossing slice boundaries, they exclude paths that would otherwise be allowed. At ten slices, computed paths decrease by 31% when compared with one slice. The number of paths actively used for carrying traffic decreases slightly — by $< 1\%$ due to some demands remaining completely unsatisfied, or diverse paths not getting found. Figure 9 shows that the traffic weighted path latency of tier-0 demands decreases by 3% because the computed paths are skewed towards shortest paths. Finally, unsatisfied demands as a percentage of requested demands increases 16% from 3.1% to 3.6%. Unsatisfied demand increases at half the rate of computed path decrease which is well controlled. In practice, the percentage of computed paths allowed by enter-leave constraints are used to determine whether a slice strategy is appropriate.

6.2 Stochastic effects of multiple controllers

Prior to the deployment of BLASTSHIELD, the centralized SWAN controller programmed new TE routes for the entire cloud network. BLASTSHIELD replaces the centralized controller with multiple slice controllers that snapshot the network topology and demands at different times. Moreover, the controllers may re-run the TE optimization and program their slice routers at different times. We study the impact of the temporally staggered operation of slice controllers to ask: can multiple slice controllers work harmoniously and not be discordant?

We reserve 15% scratch capacity in order to support the high SLO of tier-0 traffic. Transient traffic bursts and hashing polarization can cause link utilization to differ significantly from expected values. The scratch capacity is used to avoid congestion losses in these conditions. BLASTSHIELD uses this scratch capacity to deal with differences that arise with multiple controllers.

Symphony or cacophony of controllers? Path weights decide the split of traffic across paths and are the ultimate result of TE optimization. The weight of a path is the fraction of demand placed on it. BLASTSHIELD programs the newly computed path weights every 3 minutes. Since all slice controllers solve the TE problem for the entire network, we measure if the path weights that different controllers compute diverge from each other. We quantify the *path weight difference* as the root mean squared error between path weight time series from two controllers. A path weight difference of zero implies that the controllers are perfectly aligned. Non-zero path weight difference implies that the controllers are setting aside different link bandwidths for a flow which can cause congestion.

We measure the path weight difference between six different BLASTSHIELD controller pairs in the production network over a 30-day period. There were days when the controllers were operating with different configurations, different software versions, in addition to network topology and demand changes that happen throughput the day. Figure 10 shows that only 2% of paths and 3% of total demand have path weight difference of ≥ 0.15. Inter-slice demands make up 48% of paths but 10% of total demand because of the slicing strategy. Since intra-slice traffic dominates, the impact of the path weight difference is limited. The slicing strategy and scratch capacity allow multiple controllers to operate without coordination.

Solver stability. Different path weights for *slightly perturbed* inputs can create an operational challenge for BLASTSHIELD. We constrain the solver models to make their solutions stable — the tier-0 objective function minimizes demand weighted latency after solving for max-min fairness. In practice, this makes the solver results more stable when subjected to input perturbations. We do not allow non-determinism in the TE solver *e.g.,* no parallel primal and dual simplex invocation in the linear programming solver to pick the first result, since they will produce different solution vectors that result in different path weights.

We evaluate the stability of the solver results using the normalized autocorrelation function (ACF) $\rho(\tau)$. ACF is the correlation of a time series to a delayed version of itself, as a function of the delay, τ. In Fig. 11, we calculate ACF for the hour-long path weight time series of all paths in the production network over a 24-hour period. ACF values range $[-1,1]$, and 1 implies perfect correlation.

Demand and network topology changes also affect path weight ACF. So perfect correlation is not possible in an operational network. Figure 11 (a) is an example path weight time series with ACF(30 minutes) of 0.65 showing steady values of the same path weight interspersed by occasional gyrations. Figure 11 (b) shows that mean ACF is 0.75–0.63 for lags of 3–30 minutes. This reaffirms the data in Fig. 10 that path weights from independent BLASTSHIELD controllers are predominantly the same.

7 Discussion

In this section, we discuss our operational experience with BLASTSHIELD and describe safe deployment of software and configuration in BLASTSHIELD slices. We consider the implications of byzantine slice controllers, and the safeguards in place to prevent damage from them.

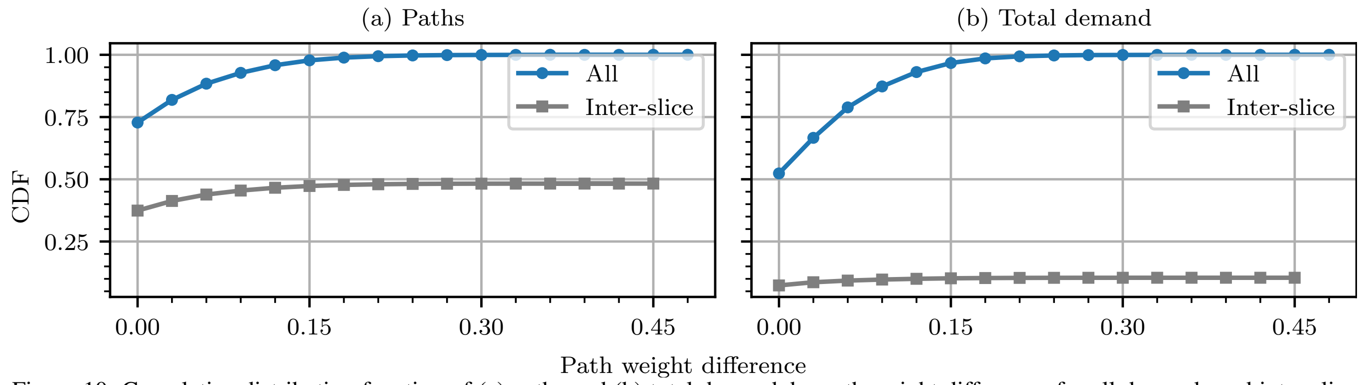


Figure 10: Cumulative distribution function of (a) paths and (b) total demand, by path weight difference, for all demands and inter-slice demands, measured for six controller pairs in the production network over a 30-day period. 98% of paths and 97% of total demand have path weight difference ≤ 0.15. Inter-slice demands make up 48% of paths but 10% of total demand.

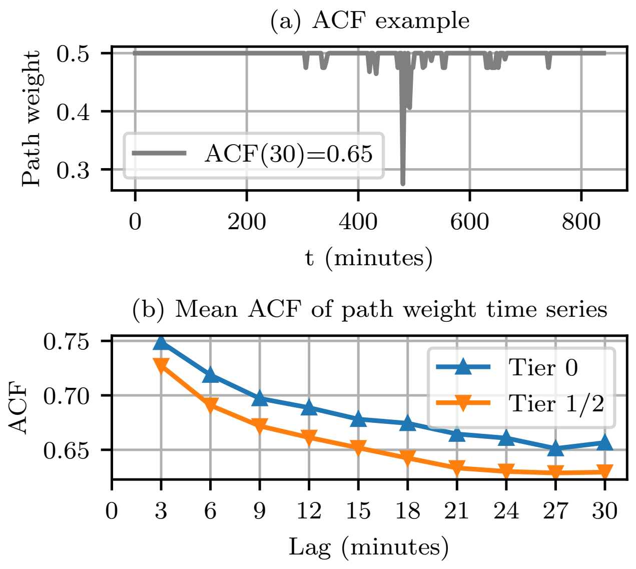


Figure 11: Autocorrelation function (ACF) measures self-similarity with a delayed version, and ranges $[-1,1]$ with 1 being perfect correlation. (a) Example path weight time series with ACF(30 minutes) of 0.65. (b) Mean ACF of path weight time series averaged over all paths in the production network over a 24-hour period by traffic tier.

7.1 Operational experience

BLASTSHIELD has been in operation for two years. Migration from SWAN to BLASTSHIELD was carried out over a number of months. The first step was to deploy inter-slice routing and forwarding functionality in the SWAN controller and router agents. This was the riskiest step and preceded by many months of testing in a virtualized emulation environment of the production network with fault injection. Each slice migration involved preparing a new BLASTSHIELD controller, excluding a set of routers from the slice configuration of the SWAN controller, and adding them to the new controller.

To support deployment of new software and configuration changes, we define slices that range from low to high impact. *Safe deployment* is a partial ordering of slices based on their blast radius. BLASTSHIELD has two staging sites with a staging controller, and new software and configuration is first deployed here. The next slice has the smallest production scope. We assign routers in geo-redundant site pairs to separate slices for additional safety. Deployment progresses to the next slice in the sort order after a sufficient probationary period. The process continues till either all slices receive the new version of software or a failure happens in a slice, which may trigger a rollback of this version from all slices.

Enter-leave constraints have been updated multiple times to pick up newly provisioned routers and links. In one instance, the constraints affected traffic engineering for an inter-datacenter pair by excluding too many links. New constraint configuration to correct the error and reverse an inter-slice traffic flow was deployed without incident.

We have introduced new hardware platforms, router agents, and controller services that would be considered high risk in the SWAN paradigm. BLASTSHIELD allowed us to introduce new implementations in isolated slices with very small blast radius and no inter-slice traffic. Initially the slice only served intra-slice traffic. Inter-slice traffic was introduced after the slice had been in operation for many months. Outages caused by failures in the new slices never had a global impact.

Slice controller environments are used by additional services to decrease their blast radius. For example, discretionary flows can be throttled at the sending host to control congestion in the network. Bandwidth is allocated to discretionary flows by global optimization but each controller only serves bandwidth pools for a smaller fault domain.

7.2 Byzantine slice controllers

A byzantine controller is an unreliable controller that is disseminating false information or sabotaging the operation of other slices in the network [24]. A controller that only impacts its own traffic is not considered byzantine in our analysis.

Resistance to byzantine slice controllers is baked into the BLASTSHIELD design. BLASTSHIELD does not allow any inter-controller interaction. Each controller uses its own services to get demand and topology inputs. It calculates TE routes by sensing the state of the network, and does not rely on communication with other controllers. Route programmers of a WAN slice do not communicate with router agents in other slices, and thus are unaffected by unreliable agents in other slices. Access control lists on slice routers prevent another slice controller from attempting to program them.

Despite these protections, a byzantine controller may route traffic in a way that causes congestion in downstream slices. A slice controller estimates the demands at the slice boundary based on the assumption that all slices are well behaved *i.e.,* they use the same algorithm and configuration as itself. Byzantine slice can violate this assumption. The impact of a byzantine controller's actions are limited to the remote traffic from the byzantine slice. WAN traffic patterns inform the creation of slices that minimize inter-slice traffic [30].

We note that non-byzantine controller faults are also possible. Faulty controller may withdraw all its routes and congest links in its own or other slices. A faulty controller may loop or blackhole packets. While we have safety checks and routing constraints that prevent such conditions, if a controller manages to bypass the checks, human intervention is required. We mitigate these failures by pausing the faulty controllers, and restoring the network programming to last known good FIB.

8 Related work

B4 [17, 19] and EBB [10] are two examples of operational networks that use software-defined traffic engineering. [17] states that site-level domain controllers were large blast radius and faults caused widespread impact to traffic passing through the affected site, which led them to divide a site into two or four control domains, each managed by a separate domain controller. Similarly, in BLASTSHIELD, we assign routers in a site to separate slice controllers. [17] uses a central controller to calculate tunnel split groups and the sequencing of traffic engineering operations, and a large fleet of domain controllers to do route calculation and programming. BLASTSHIELD does not use any central controllers and each slice controller performs global traffic engineering calculation and slice-local route programming. It should be noted that the network architectures of BLASTSHIELD and B4 are quite different. [10] uses a centralized controller and segment routing, which we evaluated but did not select because of label stack depth and lack of control in intermediate slices.

Prior work on software-defined traffic engineering [1, 7, 23, 25, 29] focus on the optimization problem of maximizing utilization, guarantee fairness, preventing congestion under faults, or dynamic pricing without considering how they would be deployed. They all assume a centralized controller will perform the optimization for the entire network without considering what happens when the controller fails. BLASTSHIELD can be used in conjunction with these works to make them deployable in operational networks.

Inter-slice routing is similar to pathlet routing [13] but without any controller interaction or dissemination protocol. [9, 12] study consistent updates and loop avoidance with a centralized controller, but not multiple controllers with inconsistent views. BLASTSHIELD adopts a stricter approach of not communicating with another controller to avoid additional failure modes from faults in the communication, and because the information a controller needs can be acquired from the network.

9 Conclusion

In this work, we motivate the design of a decentralized traffic engineering system for large-scale cloud WANs using our operational experience with SWAN. We propose BLASTSHIELD, Microsoft's new global TE system that decentralizes the TE controller with WAN slicing and implements loop-free inter-slice routing. BLASTSHIELD achieves similar throughput as fully centralized TE implementations while significantly reducing the blast radius of faults in TE controllers. We have been operating Microsoft's WAN with BLASTSHIELD, and it has substantially lowered the risk of configuration changes causing large outages.

Acknowledgements. We thank our colleagues for their significant contributions to BLASTSHIELD: Amin Ahmadi Adl, Ashlesha Atrey, Jeff Cox, Shubhangi Gupta, Guruprasad Hiriyannaiah, Luis Irun-Briz, Karthick Jayaraman, Srikanth Kandula, Pranav Khanna, Sonal Kothari, Nishschay Kumar, Erica Lan, Dave Maltz, Paul Mattes, Antra Mishra, Zahira Nasrin, Paul Pal, Francesco De Paolis, Rohit Pujar, Rejimon Radhakrishnan, Prabhakar Reddy, Newton Sanches, Anubha Sewlani, Sailaja Vellanki, Wei Wang, and Li-Fen Wu. We also thank our shepard, Stefan Schmid, and the anonymous reviewers who gave us invaluable feedback.

References

[1] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *Proceedings of USENIX NSDI*, pages 175–200, April 2021.

[2] Port channels and LACP load balancing hashing algorithms. https://www.arista.com/en/um-eos/eos-port-channels-and-lacp, accessed February 2022.

[3] Algirdas Avižienis. Fault-tolerant systems. *IEEE Transactions on Computers*, 25(12):1304–1312, December 1976.

[4] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP tunnels. RFC 3209, December 2001.

[5] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. RFC 2475, December 1998.

[6] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of ACM SIGCOMM*, pages 29–43, August 2019.

[7] Yiyang Chang, Chuan Jiang, Ashish Chandra, Sanjay Rao, and Mohit Tawarmalani. Lancet: Better network resilience by designing for pruned failure sets. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3), December 2019.

[8] Implementing Cisco Express Forwarding. https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5500/ip-addresses/66x/b-ip-addresses-cg-ncs5500-66x/m-implementing-cisco-express-forwarding-ncs5500.html, accessed February 2022.

[9] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't touch this: Consistent network updates for multiple policies. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 133–143, June 2016.

[10] Mikel Jimenez Fernandez and Henry Kwok. Building express backbone: Facebook's new long-haul network, May 2017. https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/.

[11] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Brune Decraene, Stephane Litkowski, and Rob Shakir. Segment routing architecture. RFC 8402, July 2018.

[12] Klaus-Tycho Forster, Ratul Mahajan, and Roger Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *IFIP Networking*, May 2016.

[13] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *Proceedings of ACM SIGCOMM*, pages 111–122, August 2009.

[14] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. Network flow algorithms. In Bernhard Korte, Lásló Lovász, Hans Jürgen Prömel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI Layout (Algorithms and Combinatorics)*, volume 9, pages 101–164. Springer-Verlag, 1990.

[15] Hannes Gredler, Jan Medved, Stefano Previdi, Adrian Farrel, and Saikat Ray. North-bound distribution of link-state and traffic engineering (TE) information using BGP. RFC 7752, March 2016.

[16] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of ACM SIGCOMM*, pages 15–26, August 2013.

[17] 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, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *Proceedings of ACM SIGCOMM*, pages 74–87, August 2018.

[18] Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473). ISO/IEC 10589:2002, November 2002. https://www.iso.org/standard/30932.html.

[19] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of ACM SIGCOMM*, pages 3–14, August 2013.

[20] Understanding the algorithm used to load balance traffic on MX series routers. https://www.juniper.net/documentation/us/en/software/junos/sampling-forwarding-monitoring/topics/concept/hash-computation-mpcs-understanding.html, accessed February 2022.

[21] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *Proceedings of ACM SIGCOMM*, pages 253–264, August 2005.

[22] Dave Katz and Dave Ward. Bidirectional Forwarding Detection. RFC 5880, June 2010.

[23] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *Proceedings of USENIX NSDI*, pages 157–170, April 2018.

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[25] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of ACM SIGCOMM*, pages 527–538, August 2014.

[26] Peter Phaal and Marc Levine. sFlow version 5, July 2004.

[27] Yakov Rekhter, Tony Li, and Susan Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[28] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol label switching architecture. RFC 3031, January 2001.

[29] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with Cascara. In *Proceedings of USENIX NSDI*, pages 201–216, April 2021.

[30] Rachee Singh, Nikolaj Bjørner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective capacity provisioning in wide area networks with Shoofly. In *Proceedings of ACM SIGCOMM*, pages 534–546, August 2021.