Decentralized cloud wide-area network traffic engineering with BLASTSHIELD

Umesh Krishnaswamy,* Rachee Singh, Nikolaj Bjørner, and 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 SDN-based 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. Despite the lack of central coordination, BLASTSHIELD achieves similar network throughput as state-of-the-art centralized deployments. Moreover, BLASTSHIELD reduces throughput loss from the failure of a single controller by over 65%. BLASTSHIELD is deployed in Microsoft's WAN today and carries a majority of the backbone traffic.

1 Introduction

Cloud wide-area networks (WANs) enable low-latency and high bandwidth requirements of client workloads like live-video, business critical applications, and geo-replication 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 [16, 19].

Maximum throughput, but at what cost? The paradigm shift in WAN TE from fully-decentralized switch-native protocols (*e.g.,* RSVP-TE [4, 39]) to centralized TE controllers was driven by the throughput gains made possible by centralization [29]. After a decade of operating SWAN in Microsoft's backbone network, we claim that squeezing the last ounce of network throughput is not necessary. 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. It is possible to build fault-tolerant components, and prevent single point of failures by replicating the centralized controller and its components. However, despite these mechanisms, 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 the service level objective 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 throughput loss due to failure of a TE controller by 65% for 0.07% decrease in network throughput, compared to SWAN (§6).

*Submitted to NSDI'22 operational systems track.

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 Back 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 absence of the controller routes, the traffic gets routed over shortest paths computed by the IGP. 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 weights in the North American region. 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 from 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, Rippling and Shielding

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

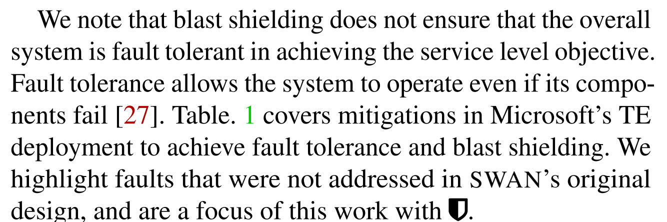
Definition 1 (Blast Radius) *is the fraction of overall TE service level objective (SLO) at risk by a point of failure.*

The blast radius of a TE controller is the customer traffic managed by it. Customer or *tier-0* traffic has the highest SLO. Discretionary traffic tiers, *tier-1* and *tier-2*, have a lower SLO. The TE controller programs routes that steer traffic on traffic engineered paths to optimize for congestion, link diversity or provide granular quality of service. When the TE controller fails, traffic reverts to the shortest path.

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

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 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 [27]. Table. 1 covers mitigations 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

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 SWAN agents perform local repair autonomously from controller. Controller does global repair in the next TE iteration.
Network device disconnects or is unreachable by controller.	SWAN agent retains last programming. Controller reconnects via router management plane. Device treated as down if failure persists. Rollback routes if disconnection is during new route programming.
Invalid, inconsistent, outdated programming by controller.	SWAN 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. ⛉	Restrict access of the controller to defined subgraphs of the WAN. Fast verification of routes before install.
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 ⛉.

replaced SWAN in Microsoft's backbone network. BLASTSHIELD views the WAN as a collection of sites or gateways. Each gateway 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 gateways 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 Figure 1, slice 1 consists of routers in gateways A–D. A slice can have multiple strongly connected

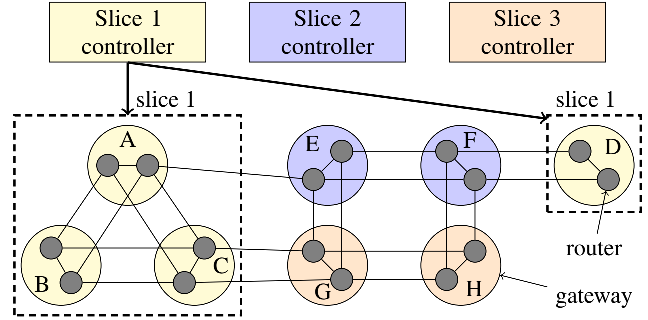


Figure 1: The WAN is divided into slices. Each slice is managed by a dedicated slice controller. Slice 1 consists of routers in gateways A–D, slices 2 and 3 have routers in gateways E–F and G–H.

components of routers. Slice 1 has two strongly connected components, the routers in gateways A–C and D, respectively. Controllers 2 and 3 manage routers in gateways 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 devices 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 other slice controllers. 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 operation 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 five microservices: topology service (TS), bandwidth predictor (BWP), traffic engineering scheduler (TES), route programmer (RP), and admission controller (AC) (Fig. 2). In addition to the controller services that run on off-router compute nodes, a BLASTSHIELD agent (BA) runs on all WAN routers.

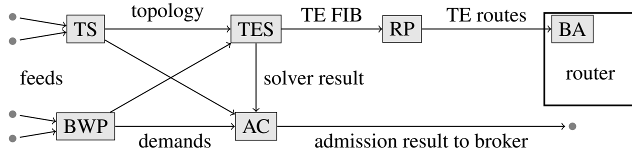


Figure 2: The slice controller is a collection of five microservices: topology service (TS), bandwidth predictor (BWP), traffic engineering scheduler (TES), route programer (RP), and admission controller (AC). BLASTSHIELD agents (BA) run in slice routers.

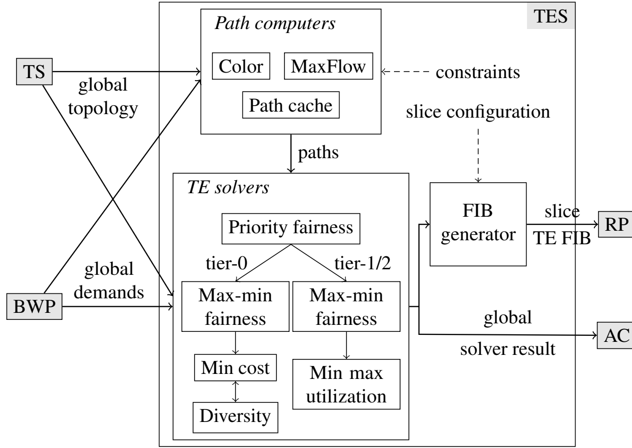


Figure 3: Traffic engineering scheduler computes routes that optimize paths for flows by traffic tier. Each controller performs a global optimization based on its view of the entire network, but only programs routes for devices belonging to its slice.

Topology Service (TS) synthesizes the network topology using graph metadata, link states, and BLASTSHIELD agent input feeds. Graph metadata consists of the list of provisioned routers, their roles in the network, router loopback addresses and links in the network. We use BGP-LS [15] as the primary source of dynamic link state information *e.g.,* link bandwidths, interface addresses, and segment identifiers [8]. The BLASTSHIELD agent feed relays the health of the router agent to the controller. All routers must have a functioning agent to be used for traffic engineering.

Bandwidth Predictor (BWP) predicts upcoming network demands using historical traffic matrices recorded by sFlow [30], IPFIX [32], and bandwidth requests made through the bandwidth broker [16]. IPFIX and sFlow collectors sample packets at WAN entering interfaces to estimate per-minute traffic matrices. BWP uses a linear regression model to predict the upcoming traffic demands in the network using historical traffic matrices from the flow collectors [38]. Bandwidth broker aggregates service-level bandwidth requests made by agents on compute nodes in cloud datacenters. WAN demands are a linear combination of these three predictors.

Each network demand is identified by the (source, destination, application, traffic class) tuple. Source and destination are WAN ingress and egress routers or gateways, respectively. Application is the name of a brokered flow, used only by the admission controller. Traffic class is a differentiated service queue name *e.g.,* best-effort, interactive, scavenger.

Traffic Engineering Scheduler (TES) forms the core of the BLASTSHIELD system (Fig. 3). It ingests global topology and demand inputs from TS and BWP respectively, and some static configuration. A collection of path computers calculates paths using the dynamic topology for the demand source destinations. Too many paths would be inefficient in time and space to compute and optimize with. Too few paths would reduce the choice for placing demands. MaxFlow path computer uses maximum flow algorithms [13], and Color path computer computes risk diverse paths using penalizing shortest path algorithms.

TE solver consists of a chain of linear programming (LP) [6] optimization steps that place demands on weighted cost multiple paths (WCMP) between demand source and destination pairs. It places demands in the tier-0 traffic class on paths with diversity protection that minimize latency subject to approximate max-min fairness [16]. Lower priority demands in tier-1 and tier-2 classes are placed on paths that minimize the maximum link utilization.

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 entire iteration in TES takes around 60 seconds. The network is re-optimized every 3 minutes, or on topology change, whichever occurs first.

Admission Controller (AC) computes the bandwidth that can be granted to brokered flows, called the *admission result*. It defines priority granularly by application and traffic class. Bandwidth broker throttles individual machine flows when the aggregate requested bandwidth exceeds the aggregate granted bandwidth for the demand. Bandwidth broker places brokered tier-0 traffic in a separate network queue since it has already been throttled based on the admission thresholds, and should not experience congestion when mixed with unbrokered tier-0 traffic.

Route Programmer (RP) programs traffic engineering routes in the BLASTSHIELD agent which in turn installs them in the router. RP periodically receives the full set of routes for all slice routers from the TES; this is called the traffic engineering forwarding information base (TE FIB). The FIB is organized into per-device flow and group tables in the OpenFlow [28] format. Every FIB has an associated sequence number which records the version of the FIB. RP performs *make-before-break* updates by first programming the traffic engineered path before placing traffic on it [12]. It programs intermediate FIBs on routers to prevent black-holing traffic during updates to TE paths. FIB programming for the entire slice takes ten seconds. We set scratch capacity aside on links to handle transient flow changes during route programming.

BLASTSHIELD agent (BA) 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. BLASTSHIELD agents have a platform-dependent module to handle this translation. BA verifies tunnels within the slice using probes generated natively or with BFD [21] 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 §6.1. 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 and gather inputs from the network and slice configuration. 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.

Decoupling TE scalability from blast shielding. BLASTSHIELD employs slice controllers to reduce the blast radius of faults in our network but not to improve the scalability of the system. We handle scale along several dimensions, unrelated to blast shielding. For example, BLASTSHIELD aggregates datacenter routes to keep the size of the FIB in check. Thus, despite solving a global TE problem, the run-time complexity scales with the number of gateways, not routes in the network. Another dimension of scale is the number demands that need to be allocated by the TE solver. We find that small demands (<500 Mbps) are 68% by count, but only 1.3% of total bandwidth. The run-time of the LP solver in TES increases with the number of demands, among other factors. We allow small flows to use network shortest paths until their bandwidths exceed a threshold to reduce the input demands to the LP solver. Finally, the sFlow and IPFIX collectors, and bandwidth broker are fully sharded to handle the large scale of the network. They are not part of the slice controller.

Fault tolerant design. All services run on multiple machines in at least two geographically separate clusters. TS instances are fully active, but elect a leader to avoid oscillations if two instances report different topologies due to faults or transients. TES, RP, and AC elect leaders, and switchover in case of failure. RP handles all the faults and inconsistencies that can happen during programming, *e.g.,* BLASTSHIELD 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. BA can react to network faults even when it is disconnected from the RP.

5 Intra-WAN routing with BLASTSHIELD

The routing of *intra-slice* flows in BLASTSHIELD is similar to SWAN's present-day implementation. 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.

Routing in SWAN. In SWAN, packets are routed using a combination of switch-native protocols (*i.e.,* BGP [31], IS-IS [18]) and the TE controller. WAN routers connected to the datacenter fabric advertise datacenter routes with themselves as the BGP next-hop. BGP receivers can resolve this BGP next-hop using one of two available routes: the shortest path route computed by IS-IS, or the route programmed by the TE controller which leverages traffic engineered paths. TE routes have higher precedence than IS-IS routes. The TE route encapsulates packets using MPLS labels from a label range reserved for the TE controller. Since IS-IS is configured with segment routing [8], the IS-IS route encapsulates packets in segment identifiers that are also MPLS labels from a different label range. Fig. 4 shows SWAN's packet routing pipeline.

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 (§3). By preventing slice controllers from programming routers outside their

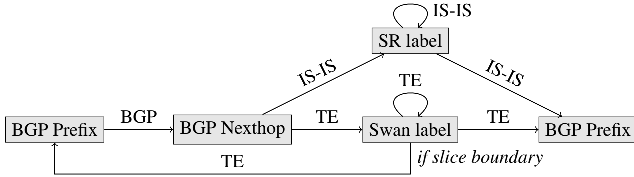


Figure 4: Packet routing pipeline in SWAN. Packets are routed using a combination of protocol and controller routes. Absent the controller, BGP and IS-IS routes forward the packet on shortest paths. The controller programmed TE routes override the IS-IS routes and route to the destination or the next slice.

slice, we enforce fault isolation between slices. 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 black-holing of traffic. Similarly, slice controllers with inconsistent views of the network, route packets to their destination without centralized control.

5.1 Inter-slice routing

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 in MPLS labels either at the time of BGP route resolution on the WAN ingress router or intermediate slice ingress routers. In both scenarios, transit routers forward the packet using the MPLS 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. 5, all four slice TES 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 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. It may appear implausible that controllers could all install the same routes without coordination. In § 6.1, we show results on the alignment of multiple controllers.

Handling intermediate controller failures. 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. For example, if the slice 2 controller withdraws all routes due to a failure, the inter-slice traffic uses shortest paths to the destination. This is the blast ripple of a down controller. In § 6.2, we will discuss how to define slice boundaries to decrease the blast ripple.

5.2 Preventing routing loops

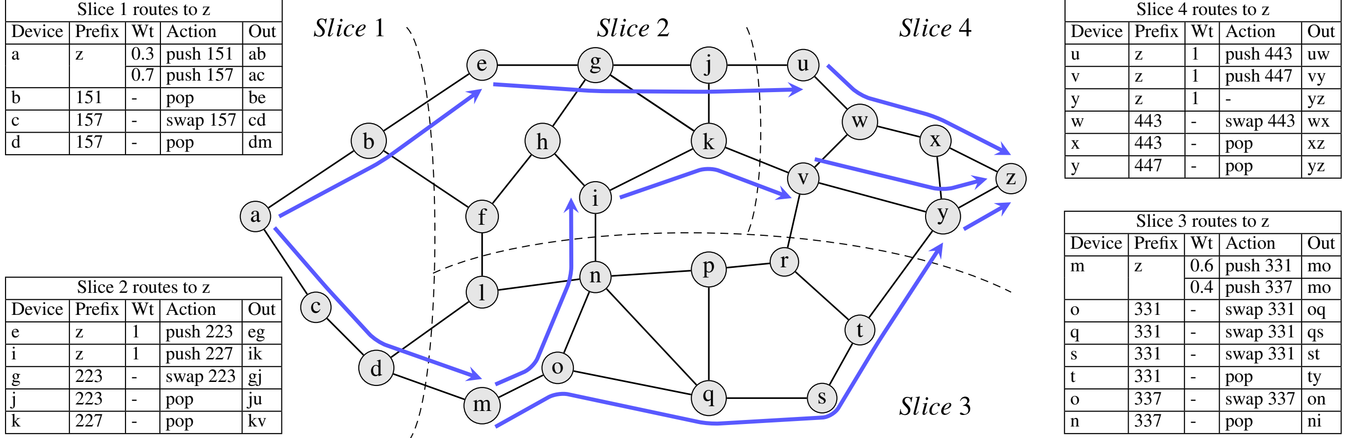
Unlike the TE controller in SWAN, a BLASTSHIELD slice controller is only responsible for routing packets within the slice and not until the packets' final destination. Since each slice is its own routing domain, inconsistent views of the global network graph at different slice controllers can lead to 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 demand source-destination pairs in the network. The constraints ensure loop-free paths, and are calculated offline using a static router-level network graph. Path computers calculate 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, similar to the spanning tree protocol. But, this approach restricts inter-slice paths to go through a few next-hops, creating bottleneck links.

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. Fig. 6 is the slice graph of Fig. 5, formed by removing inter-slice links *be*, *bf*, *dl*, *dm*, *fl*, *in*, *ju*, *kv*, *vr*, and *yt*, 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.

BLASTSHIELD 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. 6, the slice DAG for $s4$ says that paths from any node in $s1$ to any node in $s4$ can only have inter-slice transitions: $s1 \rightarrow s2 \rightarrow s4$, $s1 \rightarrow s3 \rightarrow s4$, and $s1 \rightarrow s3 \rightarrow s2 \rightarrow s4$. No controller, no matter its topology, can use any other inter-slice transition.

Path computers blacklist edges excluded by enter-leave constraints in the dynamic router-level 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



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 5: Inter-slice routing. Router-level cloud WAN topology segregated into slices 1, 2, 3 and 4. The tables represent the TE FIB programmed by slice controllers if they used 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 on slice ingress routers get encapsulated and is split over different paths based on the TE solution. Transit routers guide the packet along the path specified in the packet encapsulations. Packets return to native encapsulation at next slice and WAN exit (*e.g.,* datacenter fabric).

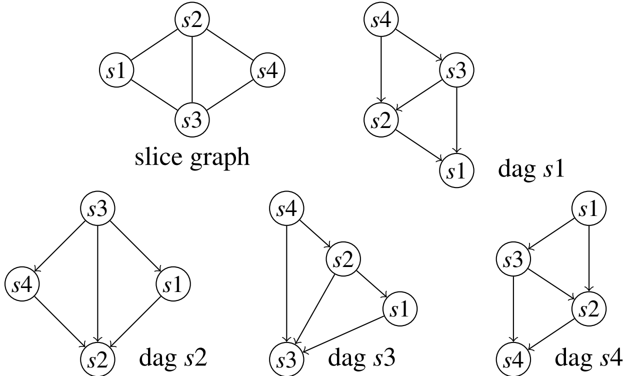


Figure 6: Enter-leave constraints restrict paths to achieve loop-free routing. Slice graph is a component level graph of Fig. 5. Slice DAG is constructed from shortest path distances in the slice graph. Router-level paths must follow DAG edges when crossing slice boundaries. Path *acdmonikvyz* is allowed for TE because $s1 \rightarrow s3 \rightarrow s2 \rightarrow s4$ is a path in DAG $s4$. Path *abfh**in**prvyz* is not allowed for TE because $s2 \rightarrow s3$ is not present in DAG $s4$.

the percentage of allowed paths vs. computed paths without constraints in §6.2.

Verifying inter-slice routing correctness. Due to the negative impact of routing loops in production, loop detection tests are important. This testing has helped us find design flaws and bugs, and is now used to verify constraint configuration before deployment.

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 (device, destination prefix), **end** be the terminating route key, **null** be the undefined route key, and *ttl* be the packet time to live. Let $f : \mathcal{R} \rightarrow \mathcal{R}$, 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 $1 \leq n \leq ttl$. The collection of TE, BGP, and IS-IS routes, and their union are examples of routing functions. The routing function is complete, loops, or blackholes, if:

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

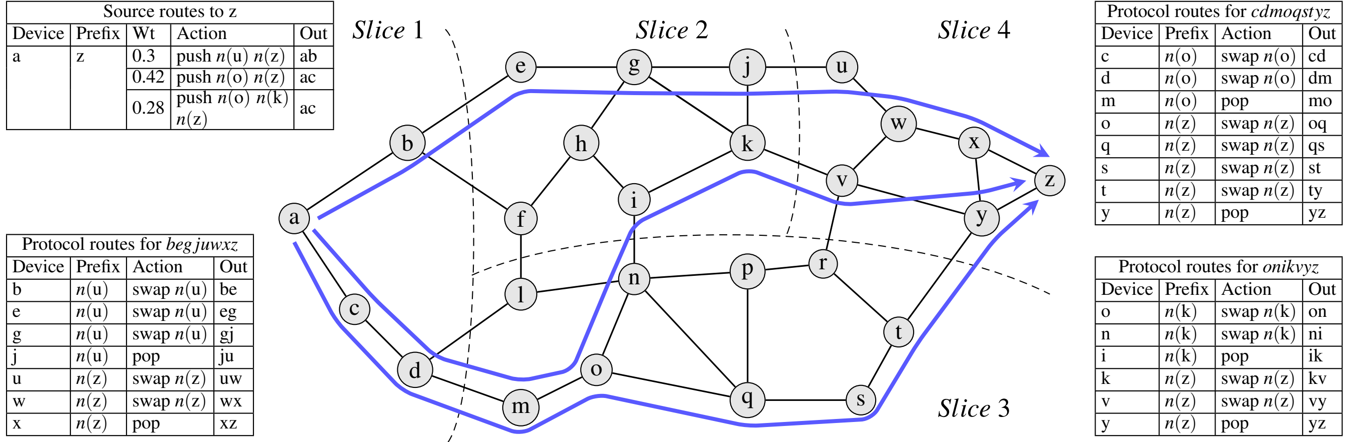
where x ranges over $\mathcal{R} \setminus \{\textbf{end}, \textbf{null}\}$ and n ranges over $[1..ttl]$.

The enter-leave tester uses this formalism to verify constraints. It simulates inconsistent views by feeding slice controllers different network graphs, and checking every route key of the combined FIB for completeness. It also performs graph invariant checks *e.g.,* no inter-slice cycles in the graph after excluding edges in constraints.

5.3 Why not source routing?

An industry-standard mechanism for intra-domain traffic engineering is segment routing (SR) [8]. In this section, we describe an alternate approach that leverages the capabilities of segment routing, and why we did not adopt this approach.

What is segment routing? 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 ingress 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 segment denotes a *forwarding instruction* to traverse one or more hops in the network topology. There are two main types of segments: *adjacency* and *prefix* segments. Prefix segments cause the packet to be routed on the least-cost paths computed by the IGP between a router and a specified prefix. A prefix segment with the router's loopback address is a *node segment*. An adjacency segment causes a packet to traverse a specified link corresponding to an IGP adjacency between two routers. Segments are allocated and signaled by protocols like IS-IS.



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

Protocol routes for *begjuwxz*			
Device	Prefix	Action	Out
b	$n(\mathrm{u})$	swap $n(\mathrm{u})$	be
e	$n(\mathrm{u})$	swap $n(\mathrm{u})$	eg
g	$n(\mathrm{u})$	swap $n(\mathrm{u})$	gj
j	$n(\mathrm{u})$	pop	ju
u	$n(\mathrm{z})$	swap $n(\mathrm{z})$	uw
w	$n(\mathrm{z})$	swap $n(\mathrm{z})$	wx
x	$n(\mathrm{z})$	pop	xz

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

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

Figure 7: Source routing. Slice 1 controller programs ingress routes to z using loose source routing. Switch native protocols like IS-IS take care of transit routes. The path *begjuwxz* is composed of two shortest path segments *begju* and *uwxz*. Hence the label stack for the path is $n(\mathrm{u})$ $n(\mathrm{z})$, where $n()$ is the node segment identifier of a device. Weights of intra-slice links are 1 and inter-slice links are 5.

Loose source routing. An IGP path computer models the IS-IS shortest path first algorithm [18]. Coupled with segment identifiers from TS (§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. Fig. 7 shows an example of loose source routing for the same paths shown in Fig. 5. The path *begjuwxz* is composed of two shortest path segments *begju* and *uwxz*. Hence *a* encapsulates with label stack of $[n(\mathrm{u})\ n(\mathrm{z})]$ to route to *z*, where $n()$ is the node segment identifier of a device.

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 (LAG) and equal cost multi-path (ECMP) hashing. LAG hashing sprays packets on member links of a port-channel. ECMP 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 ports or links 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, 7, 20]. We note that if the packets ingressing the WAN already are encapsulated in MPLS headers, 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

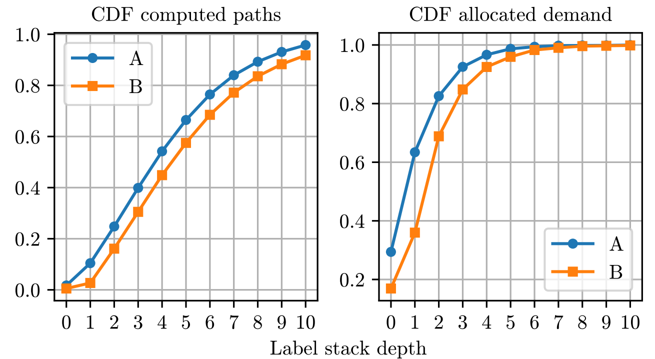


Figure 8: Cumulative density function of label stack depth for (a) computed paths and (b) allocated demands, for inputs A and B of increasing sizes. If depth limit is four, 45% of computed paths and 93% of allocated demands map to allowed paths for input B.

cannot be used for TE. Fig. 8 studies the label stack depth needed to encode paths computed by TES path computers 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 §6.2).

Second, the purpose of BLASTSHIELD is to separate fault domains, but IS-IS is a single fault domain. A fault in IS-IS can ripple to multiple slices, like in the outage described in §2.1. On the other hand, inter-slice routing does not depend on IS-IS. It sets up TE paths without any assistance from dynamic protocols, relies only on BGP, and works across WAN domains.

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.

Time	0	3	6	9	12	15	18	21	24	27	30
Ctlr 1	.5	.5	.5	.5	0	0	.5	.5	.5	0	.4
Ctlr 2	.5	.5	.5	.5	0	0	.2	.5	.4	0	.5

Table 2: The path weights computed by two controllers for a path. Weights have been rounded for readability and time is in minutes. Path weight difference between the two series is 0.17.

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 the Microsoft network. In this section, we evaluate the stochastic effects caused by multiple and independent BLASTSHIELD controllers (§6.1). We show that despite the controllers having different configurations, software versions and network topology snapshots, they arrive at nearly similar flow allocations. Finally, we quantify the cost vs. benefit trade-off of slice-decentralized traffic engineering in terms of the loss of TE throughput and gain in availability as a function of the slice count in BLASTSHIELD (§6.2).

6.1 Stochastic effects of multiple controllers

Prior to the deployment of BLASTSHIELD, the centralized SWAN controller programmed new TE routes for the entire cloud network. In a transient routing state *i.e.,* network state when all new routes have not been programmed and some flows are on old paths while others are on new paths, SWAN did not risk link congestion for two main reasons. First, the time to program routes is very short, only ten seconds, and second, SWAN reserves a small amount of scratch capacity to handle flows in transient routing states. However, BLASTSHIELD replaces the centralized controller with multiple slice controllers that snapshot 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 in an orchestra and not be discordant?

Symphony or cacophany of controllers? Path weights decide the split of traffic across paths and are the final result of the 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 during the measurement period which can cause congestion. Table 2 shows an example of path weight time series of two controllers.

We measure the path weight difference between BLASTSHIELD controllers in scenarios where they are most likely to differ – days when the controllers were operating with different configurations, different software versions, or frequent network topology changes. Fig. 9 shows the histogram of path weight differences in the production network between pairs of controllers. The data shows that 67%, 88%, and 96% of paths have path weight difference of 0, ≤ 0.09, and ≤ 0.18, respectively. The low path weight difference shows that slice controllers can operate without central coordination. We note that additional scratch capacity can be set aside to absorb errors from high path weight differences, but was not required when deploying BLASTSHIELD.

Solver stability. Optimization solvers (*e.g.,* GLOP [14]) can often solve the same or slightly different problem formulation and produce different results that satisfy all problem constraints. Different path weights for *slightly perturbed* inputs creates 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 LP execution to pick first result.

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. We calculate ACF for the hour-long path weight time series of all paths. ACF values range $[-1,1]$, and 1 implies perfect correlation. For example, ρ(3 minutes) for the two path weight series in Table 2 are 0.13 and 0.19 respectively. Network topology changes and model instabilities can both affect the path weight ACF. However, network topology changes are far less frequent compared to model instabilities that can occur in every round of TE computation. Thus, when averaged over a large number of paths and time series, the effect of network changes diminishes and the mean ACF measures model stability. Fig. 10 shows that solver models are quite stable, with mean ACF of 0.75–0.63 for lags of 3–30 minutes.

6.2 Slicing strategies and TE efficiency

In this section, we describe incremental deployment strategies for BLASTSHIELD— from globally centralized to slice-decentralized traffic engineering. We incrementally carve out slices from the global cloud network as shown in Table 3. 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 3, slice configuration 2 represents the network divided into two slices: India and the rest of the world, configuration 3 represents In-

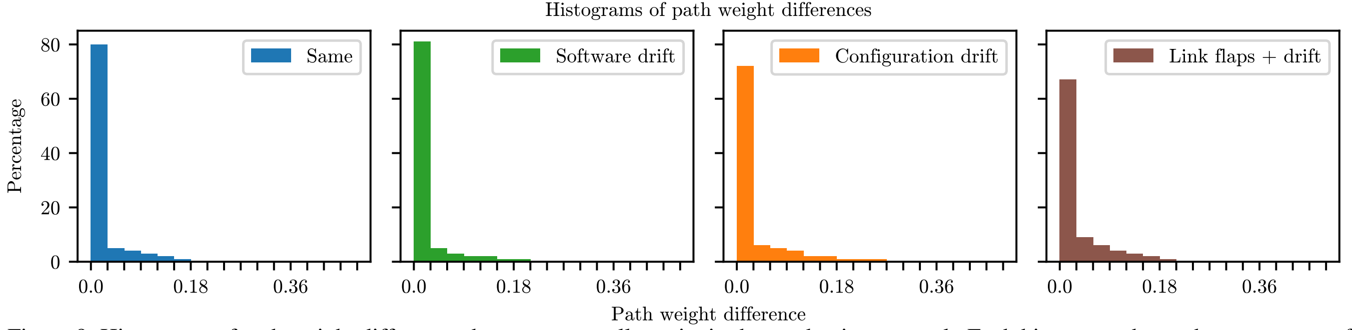


Figure 9: Histograms of path weight differences between controller pairs in the production network. Each histogram shows the percentage of paths by their corresponding path weight difference. **Same** is the histogram for a controller pair with identical software and configuration. **Software drift** and **Configuration drift** are for controller pairs with different software versions and configurations respectively. **Link flaps + drift** is for day when there were a number of fiber events causing rapid topology changes for a controller pair with configuration drift.

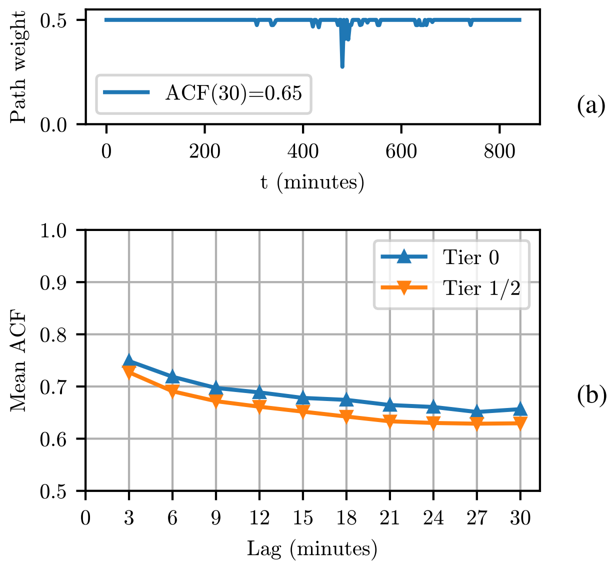


Figure 10: Autocorrelation function (ACF) is calculated on path weight time series of 1 hour, averaged over a day. ACF range is $[-1, 1]$, 1 is perfect correlation. (a) example path weight time series and its lag 30 minutes ACF, (b) Mean ACF of all paths by traffic tier.

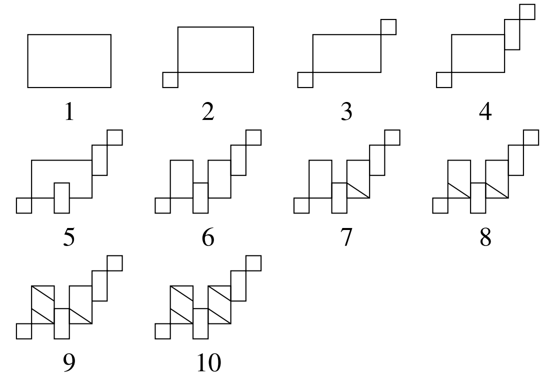


Table 3: Ten slice configurations. In (1) the entire network is one slice. Slices 2–6 are formed by grouping routers in continents or geographies. Slices 7–10 are created by further subdividing the two largest geographies, Europe and North America.

dia, 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 gateways and router planes.

BLASTSHIELD aims to contain the impact of inevitable faults in the cloud WAN by decentralizing traffic engineering into slices. However, decentralization comes at a cost – reduction in the network throughput compared to fully-centralized TE. We compute the benefits and costs of WAN slicing using demands and topology inputs from the Microsoft backbone network for the month of July 2021.

Availability gains from decentralized TE. The key benefit 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 loss of throughput is the impact of the slice failure. We measure the throughput loss under various over-subscribed scenarios using a network simulator that models routing, forwarding, and queuing behavior. Fig. 11 (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. In every simulated failure case, BLASTSHIELD uses the combined FIB consisting of IGP routes of the failed slice, and TE routes of the remaining slices (if any) to allocate the remaining demands. We note that the loss in throughput is due to congestion caused by shortest-path routing over IGP routes. There are no losses due to traffic blackholes or routing loops. Fig. 11 (a) shows that with ten slices, the throughput loss due to slice failure decreases by 65% (-27.6% to -9.6%) across all traffic classes.

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

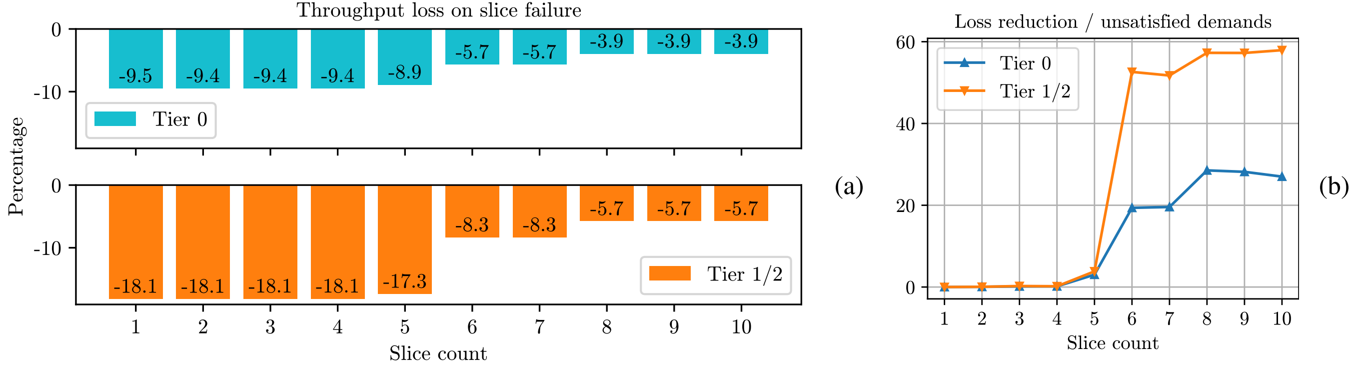


Figure 11: (a) Throughput loss from worst case single slice failure as a function of number of slices. Loss is measured by simulation and reported as a percentage of requested demand per traffic tier. (b) Ratio of slice failure throughput loss reduction to unsatisfied demands due to enter-leave constraints points to a sweet spot of eight slices with this slicing strategy.

placing demands, which in turn decreases the demands that can be allocated. To quantify the cost vs. benefit tradeoff of decentralized traffic engineering, we compute the ratio of the worst-case throughput reduction due to a single slice failure and the unsatisfied demand. Fig. 11 (b) compares the ratio of throughput loss due to slice failure and unsatisfied demand from enter-leave constraints – higher ratio implies a better operating point on the benefits vs. cost tradeoff. 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 break up of Europe and North America into separate and smaller slices. Beyond eight slices, the loss reduction to unsatisfied demand ratio starts to decline. Hence, the ideal number of slices with this strategy is eight.

Stress testing BLASTSHIELD. We oversubscribe the network by doubling the bandwidth values in requested demands, and failing the worst-case shared risk groups that bring down multiple links in hot spots of the topology. The oversubscription amplifies the downsides of enter-leave constraints. Figure 12 shows the impact of slicing on paths computed by BLASTSHIELD path computers. Since the constraints enforce a shortest-path order when crossing slice boundaries, they exclude paths that would otherwise be allowed. The count of computed paths decreases with each additional slice. At ten slices, computed paths decreases by 31% when compared with one slice. Of the computed paths, 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 12 shows that the bandwidth weighted path latency of tier-0 demands decreases by 3% because TE computed paths are skewed towards shortest paths. Finally, unsatisfied demands as a percentage of requested demands increases by 16% (from 3.1% to 3.6%).

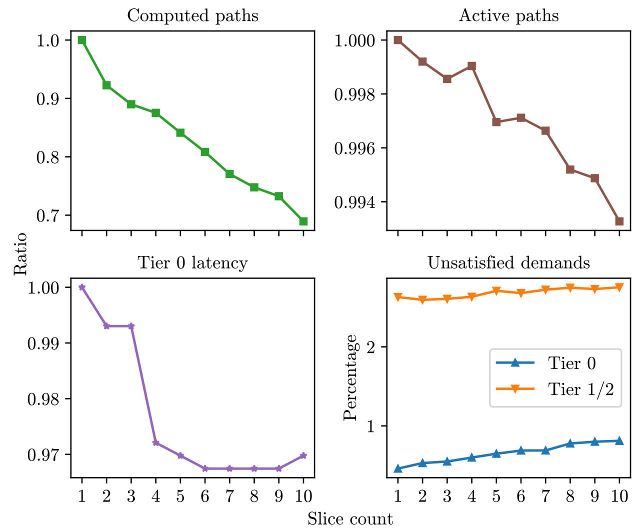


Figure 12: Stress-testing BLASTSHIELD with worst-case failures and 2x demands. Computed paths is the count of paths computed with enter-leave constraints. Active paths is the count of paths selected by TE solver. Latency is average path latency for tier-0 demands. Unsatisfied demand is the unallocated demand per traffic tier. All values, except unsatisfied demand, are normalized to corresponding values for one slice; the latter is a percentage of requested demand.

7 Discussion

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

7.1 Safe deployment with slices

Cloud platforms are partitioned into *rings* of different scopes. Deployment starts in a ring with the smallest scope, *e.g.,* staging, and progresses to larger scopes. We use probationary periods for evaluating changes, watchdogs to alert on failures, and rollbacks to control progress. BLASTSHIELD offers the same approach for the WAN with slices.

We begin testing new software and configuration with sandboxes that replicate the production WAN [26]. We inject faults in the sandbox before releasing the change to production. This

process was used for turning on new slices in BLASTSHIELD.

We define slices that range from low to high impact. *Safe deployment* is a partial ordering of slices based on their blast radius. The initial slices have the smallest production scope. We assign devices of paired regions used for geo-redundancy to separate slices. 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.

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 [23]. 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 BLASTSHIELD agents in other slices, and thus are unaffected by unreliable agents in other slices. Access control lists on slice routers prevent other slice controllers 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. A byzantine slice can violate this assumption. The impact of a byzantine controller's actions is limited to the remote traffic from the byzantine slice. We provision WAN links to incorporate such controller failures. Moreover, BLASTSHIELD reduces the cost of capacity augments since WAN traffic patterns can inform the creation of slices that are self-contained and minimize inter-slice traffic [35].

We note that non-byzantine controller faults are also possible. A 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

Software-defined centralized TE: Cloud providers have embraced software-defined, centralized TE controllers to assign flow in their WANs to maximize their utilization, guarantee fairness, and prevent congestion [16, 19, 22, 33, 34, 36, 37].

Fully-decentralized TE: Early work on traffic engineering in WANs relied on switch-native protocols like MPLS RSVP-TE [3, 39]. Predominantly deployed in ISP networks, operators have used global optimization of IGP link weights to achieve desired traffic engineering properties in their network [10, 11]. BLASTSHIELD proposes an SDN-based slice-decentralized approach to strike a balance between fully-decentralized and centralized TE solutions.

Hierarchial TE: In [17], Google presented the updated design of their B4 WAN TE system. The new design leverages hierarchical TE for scalability in their WAN. BLASTSHIELD does not rely on hierarchical WAN topology.

Solver scalability: [1] improves solver runtime with network contractions. BLASTSHIELD focusses on controller failures, and our results show that current LP solvers can solve our constrained problem formulations within seconds. Once the scale becomes a challenge, BLASTSHIELD can leverage NCFlow [1] to replace the slice optimization formulation.

Wide-area failure recovery: [9] considered safe reallocation of routes without incurring congestion or breaking reachability under network and demand changes. Bringing these ideas to TE, K-wise failure resiliency was developed in [5, 24, 25]. These works are complementary to BLASTSHIELD and can be incorporated in the optimization formulation that slice controllers solve.

9 Conclusion

In this work, we motivate the design of a partially 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, 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.

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 NSDI*, pages 175–200. USENIX Association, April 2021.

[2] Port channels and LACP load balancing hashing algorithms. https://www.arista.com/en/um-eos/eos-port-channels-and-lacp (accessed August 2021).

[3] D. O. Awduche. MPLS and traffic engineering in IP networks. *IEEE Communications Magazine*, 37(12):42–47, 1999.

[4] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP tunnels. https://tools.ietf.org/html/rfc3209, December 2001. RFC 3209.

[5] 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.

[6] Vašek Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[7] Implementing Cisco Express Forwarding, December 2020. 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 August 2021).

[8] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Brune Decraene, Stephane Litkowski, and Rob Shakir. Segment routing architecture. https://tools.ietf.org/html/rfc8402, July 2018. RFC 8402.

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

[10] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional IP routing protocols. *IEEE Communications Magazine*, 40(10):118–124, 2002.

[11] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 2, pages 519–528 vol.2, 2000.

[12] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 279–291, New York, NY, USA, 2011. Association for Computing Machinery.

[13] 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.

[14] OR-Tools – Google optimization tools. https://github.com/google/or-tools.

[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. https://tools.ietf.org/html/rfc7752, March 2016. RFC 7752.

[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 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 SIGCOMM*, pages 74–87, August 2018.

[18] ISO/IEC 10589:2002 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). https://www.iso.org/standard/30932.html, November 2002.

[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 SIGCOMM*, pages 3–14, August 2013.

[20] Understanding the algorithm used to load balance traffic on MX series routers, March 2021. https://www.juniper.net/documentation/us/en/software/junos/sampling-forwarding-monitoring/topics/

concept/hash-computation-mpcs-understanding.html (accessed August 2021).

[21] Dave Katz and Dave Ward. Bidirectional Forwarding Detection. https://tools.ietf.org/html/rfc5880, June 2010. RFC 5880.

[22] 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 *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 157–170, Renton, WA, April 2018. USENIX Association.

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

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

[25] Hongqiang Harry Liu and Jian Li. O(n) compression of bounded M-sum constraints in traffic engineering with forward fault correction. http://www.hongqiangliu.com/uploads/5/2/7/4/52747939/ffc-improve.pdf, 2014.

[26] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Fault tolerance, September 2021. https://csrc.nist.gov/glossary/term/Fault_tolerance.

[28] OpenFlow switch specification. https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, March 2015.

[29] Abhinav Pathak, Ming Zhang, Y. Charlie Hu, Ratul Mahajan, and Dave Maltz. Latency inflation with mpls-based traffic engineering. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, page 463–472, New York, NY, USA, 2011. Association for Computing Machinery.

[30] Peter Phaal and Marc Levine. sFlow version 5. https://sflow.org/sflow_version_5.txt, July 2004.

[31] Yakov Rekhter, Tony Li, and Susan Hares. A Border Gateway Protocol 4 (BGP-4). https://tools.ietf.org/html/rfc4271, January 2006. RFC 4271.

[32] Ganesh Sadasivan, Nevil Brownlee, and Benoit Claise. Architecture for IP flow information export. https://tools.ietf.org/html/rfc5470, March 2009. RFC 5470.

[33] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with Edge Fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431. ACM, 2017.

[34] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with Cascara. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[35] Rachee Singh, Nikolaj Bjorner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective capacity provisioning in wide area networks with shoofly. SIGCOMM '21, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, and Phillipa Gill. RADWAN: Rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 547–560, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Shih-Hao Tseng, Saksham Agarwal, Rachit Agarwal, Hitesh Ballani, and Ao Tang. Codedbulk: Inter-datacenter bulk transfers using network coding. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 15–28. USENIX Association, April 2021.

[38] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. Learning to route. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 185–191, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Xipeng Xiao, A. Hannan, B. Bailey, and L. M. Ni. Traffic engineering with MPLS in the internet. *Netwrk. Mag. of Global Internetwkg.*, 14(2):28–33, March 2000.