EBB: Reliable and Evolvable Express Backbone Network in Meta

Marek Denis, Yuanjun Yao, Ashley Hatch, Qin Zhang, Chiun Lin Lim, Shuqiang Zhang, Kyle Sugrue, Henry Kwok, Mikel Jimenez Fernandez, Petr Lapukhov, Sandeep Hebbani, Gaya Nagarajan, Omar Baldonado, Lixin Gao†, Ying Zhang
Meta Platforms, Inc. †University of Massachusetts Amherst

Abstract

We present the design, implementation, evaluation, deployment and production experiences of EBB (Express BackBone), a private WAN (Wide Area Network) connecting Meta's global data centers (DCs). Initiated in 2015, EBB now carries 100% of DC-DC traffic, witnessing remarkable growth over the years. A key design aspect of EBB is its multi-plane architecture, facilitating seamless deployment of a new control plane while ensuring operational simplicity. This architecture allows for efficient failure mitigation, standard maintenance, and capacity expansion by draining one or two planes without impacting service level objectives (SLOs). Another critical design decision is the hybrid model, combining distributed control agents and a central controller. EBB's centralized traffic engineering utilizes an MPLS-TE based solution to allocate paths periodically for different traffic classes based on service requirements, while its distributed control agents enable fast local failure recovery by pre-installing pre-computed backup paths in the data plane. We delve into our eight-year production experience, highlighting the successful deployment of multiple generations of EBB.

CCS Concepts

• **Networks** → **Network architectures**; **Control path algorithms**; **Network management**;

Keywords

Wide-Area Networks, Traffic Engineering, Software-defined Networking

ACM Reference Format:
. 2023. EBB: Reliable and Evolvable Express Backbone Network in Meta. In *Proceedings of ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604860

1 Introduction

Wide area networks (WANs) play a critical role in interconnecting data centers (DCs), facilitating the smooth flow of network traffic across these centers. With the increasing demand for supporting cross-data center replication of rich content like photos and videos, the need for bandwidth between data centers has surged significantly. Additionally, WANs are tasked with handling traffic that varies greatly in terms of bandwidth, availability, and latency requirements. The WAN network must cater to the diverse needs of various service classes, such as user-facing traffic with low latency and high availability demands, as well as bulk network traffic requiring high throughput. Moreover, it must exhibit agility to adapt to changes in configuration, control stack, and the addition of new data centers, ensuring seamless operations in an ever-evolving network landscape.

Meta's WAN network boasts unique characteristics that set it apart from more conventional WAN architectures. One such aspect is the need for the centralized control plane to adapt continuously to frequently changing traffic demands, the addition of new data centers, and the inclusion of new links. To support this evolving control plane, our architecture necessitates the ability to perform live migration and updates of the control plane seamlessly. Another distinguishing feature is the backbone network's swift response to potential link or intermediate node failures. In traditional approaches, the centralized control plane recomputes paths and installs new paths on the data plane to handle failures [14, 15, 30]. However, this approach may result in significant packet loss during the response cycle. Operational simplicity and ease of deployment are paramount in our design decisions. We prioritize the need to maintain a straightforward operational setup while accommodating the demands of multiple traffic classes and continuously evolving the control plane. Recent events, like the large failure incident in Oct 2021 [16], underscore the significance of striking a balance between simplicity and adaptability in our network operations.

We deploy EBB that customizes traffic engineering algorithms and programmable switches to these unique characteristics of the WAN network [25]. As a result, EBB is an MPLS-based software defined network (SDN) with the following unique features.

First, EBB leverages a multi-plane architecture, inspired by our data center network design [5]. This innovative approach divides the physical topology network into multiple parallel topologies, or planes, with each plane receiving a proportion of traffic and possessing a separate centralized control stack. The utilization of parallel planes provides the flexibility to continually evolve our traffic engineering algorithms in the controller. For instance, we can undertake partial deployments of traffic engineering (TE) algorithms in one plane by diverting traffic from that plane and then deploying the new TE algorithm specifically in that plane. Additionally, this allows us to conduct A/B testing on one plane while leaving other planes unaffected, making it easier to compare performance and effectiveness. Moreover, this multi-plane architecture facilitates a phased rollout of new configurations, one plane at a time, to minimize disruption to live traffic.

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604860

Secondly, EBB's control plane employs a hybrid model, combining both distributed control agents and a central controller. Each plane features a centralized controller responsible for executing traffic engineering algorithms, while distributed agents utilize an in-house intradomain routing protocol to discover the network topology. This hybrid approach enables us to efficiently manage traffic flow centrally through traffic engineering path computation. The centralized controller deploys multiple traffic engineering path allocation algorithms, tailored to address the unique characteristics of each traffic class, ensuring optimal routing for diverse service requirements. To enhance resilience, the centralized controller proactively precomputes and pre-installs backup paths for common failure scenarios, allowing for rapid response in the event of link failures. Network failures are detected in a distributed manner, and agents running on network devices are responsible for promptly installing the pre-computed backup paths, ensuring seamless continuity of network services even in the face of disruptions.

Thirdly, our data plane architecture is programmable, going beyond the conventional label switching approach. Notably, the programmable label includes semantic information that indicates the source and destination site, along with traffic classes. This semantic labeling greatly simplifies debugging, monitoring, and measurement activities across the backbone network. The data plane's programmability further allows for the seamless coexistence of primary and backup paths, facilitating rapid switching between these paths in the event of failures. This feature enhances the network's fault tolerance and enables swift response to potential disruptions, ensuring continuous and reliable service delivery.

EBB, with its deployment dating back to 2015, has witnessed remarkable evolution across multiple generations. Currently, our network boasts eight parallel planes, efficiently handling traffic across more than 20 regions. EBB stands as the sole network linking data center regions at Meta, and over the years, it has experienced an impressive 100-fold growth in traffic volume. The network's design prioritizes agility, enabling rapid deployment and iteration of innovative control plane functionalities. EBB's resilience is evident in its capacity to accommodate various failure scenarios and rapidly adapt to fluctuating network traffic demands. In this paper, we offer valuable insights into our experience with the deployment of the evolving centralized controller. Additionally, we share our operational experience accumulated over the years in managing EBB. It is our hope that our experiences will inspire future research and advancements in managing WAN backbone networks, contributing to the continued growth of such critical infrastructures. [1]

2 EBB Topology and Traffic Classes

In this section, we describe the Express Backbone (EBB) topology, traffic characteristics, and deployment scenarios.

2.1 EBB Network Topology

The Express Backbone (EBB) is the wide-area (WAN) backbone network that interconnects data center (DC) sites across the globe. Started from 2015, EBB is built as a separate backbone to carry the inter-region machine-to-machine traffic, in order to better scale with the exponential growth of the inter-DC traffic demand. Indeed, the inter-DC traffic demand has grown by more than 100 times over the last 10 years. The data center-to-user traffic remains at the original backbone called CBB, or Classic Backbone.

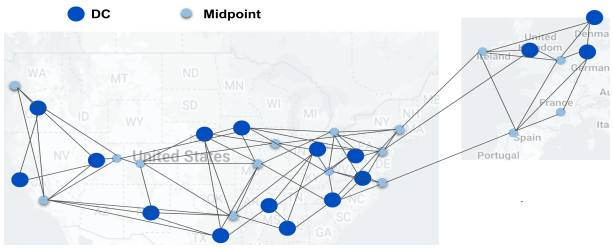

Figure 1: Explanatory EBB topology with selected DCs and circuits (geo-locations are approximate).

EBB contains multiple types of devices in different regions. As illustrated in Figure 1, each node denotes a site, which could be a DC, or mid-point connection node. EBB has over 20 DC nodes and over 20 midpoint nodes. Links between the nodes are Layer 3 IP connections. Each link represents a bundle of physical connectivity. Over the last 8 years, EBB has grown to contain thousands of links.

A primary goal of EBB is to perform traffic engineering for inter-DC traffic. As illustrated in Figure 1, the nodes are either data centers, or midpoint sites that provide connectivity to DC nodes. The TE problem in EBB is to find the paths to connect every pair of source/destination DC sites so as to minimize latency, congestion, and packet loss. Prior to EBB, we used RSVP-TE [3] for fully distributed routing, which caused tens of minutes of convergence time in the worst case. Similar to other SDN efforts [14, 15], we switch to the centralized control for better scalability and performance.

2.2 Traffic Service Classes

Application traffic is classified in a few infrastructure-wide Classes of Service (CoS) - ICP (Infrastructure Control Plane), Gold, Silver and Bronze. ICP class carries the most important network control traffic. Gold class is for user-facing traffic and critical services that are sensitive to latency and availability. Silver class is the default CoS for most applications. The remaining is bronze class for heavy and bulk network consumers. The latter three classes all account for a significant portion of total traffic. Thus, each class deserves dedicated attention from traffic management perspective.

Express backbone implements Strict priority queueing to mitigate network congestion. Higher priority class traffic has higher availability SLOs. Under network congestion, these classes experience different levels of packet loss, with "higher priority" class traffic having a higher preference to be forwarded than "lower priority" ones. Traffic is classified based on IPv6 header's DSCP value, and marked on a distributed host-based stack, based on the marking policies and the entitlements [4]. Such distributed structure enables flexible coordination and innovations between network centralized control and host distributed signaling.

3 EBB Design Overview

In this section, we describe the design principals of EBB, its multi-plane architecture, and an overview of one plane.

[1] This work does not raise any ethical issues.

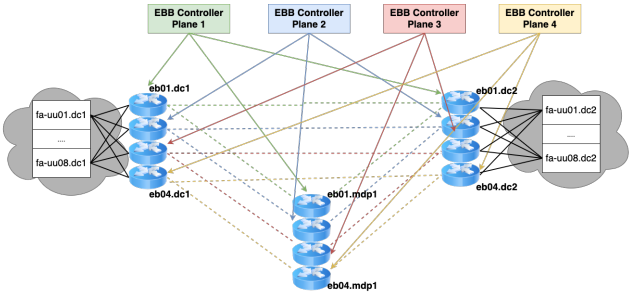

Figure 2: EBB Multiplane Architecture: An example of four planes interconnecting three regions

3.1 Design Principals

EBB uses an MPLS-based SDN architecture with the following design principles.

- *Support constant evolvability of control plane:* The software-defined network (SDN) architecture enables a centralized controller that determines path allocation, which is the key component of the control plane. Yet, the controller algorithm is constantly evolving due to the network upgrade and changing traffic demand. Our multi-plane architecture can readily support the upgrade of the controller algorithm and the necessary egress device drain during operation.
- *Local failure recovery:* The multi-plane architecture provides redundancy among planes so that a plane-level failure such as ingress/egress device drain or a controller failure can be accommodated without bringing live traffic. However, link or router level failure needs to be reacted quickly to with minimum disruption to live traffic. Instead of relying on informing and the reaction of the centralized controller, it is necessary to establish backup paths and pre-installing these backup paths to switches to facilitate local failure recovery. In the absence of the programmed LSP mesh, the separation of centralized TE control and IP routing allows for fallback to IP routing.
- *Support multiclass traffic:* The data center WAN backbone accommodates various network traffic classes. This ranges from critical service traffic with low latency and high availability requirements to bulk best-effort network traffic. Further, traffic demand from different traffic classes evolves over time and might be shaped at ingress points. The path allocation algorithm needs to accommodate these traffic classes based on their requirements.

3.2 Multi-plane Architecture

Similar to our data center network design [5], EBB uses a multi-plane architecture that splits the physical topology network into multiple (in our case, eight) parallel topologies, referred to as *planes*. Figure 2 shows the EBB network architecture with three sites interconnected with four planes. Each plane receives a proportion of traffic and has a separate control stack. The parallel planes allow us to continually evolve our TE algorithms in the controller and perform fast upgrades/rollbacks with minimum disruption to live traffic. Almost identical planes enable A/B testing between the planes and help achieve rapid and safe evolution.

3.2.1 Splitting Traffic to Multiple Planes

EBB inter-connects data centers. We explain how the traffic from source regions get onboarded to the eight planes through multiple routing mechanisms below.

eBGP between DC and EB routers: The datacenter edge routers (e.g., Fabric Aggregation (FA) routers) establish eBGP sessions with EB routers in all planes in the same region. FAs announce all the prefixes within the DC through the eBGP sessions to all the EB routers. As an example in Figure 2, a prefix p in DC1 will be announced from all four FAs. Thus, the traffic to p will be carried via ECMP across all planes.

iBGP mesh between EB routers: Within each plane, EBs form full-mesh iBGP sessions. Each EB propagates all the DC prefixes in its region to remote DCs. In the above example, $eb01.dc2$ learns p's route from $eb01.dc1$ with the nexthop pointed to $eb01.dc1$'s loopback address. Similarly, DC2's EBs in other planes learn p from the corresponding EBs in the same plane at DC1. With this route installed, when $eb01.dc2$ receives traffic destined to p, it knows to send it to $eb01.dc1$.

Controller programs to route traffic into LSPs (Label Switched Path): There are multiple LSPs established between any two EBs, for path diversity and different traffic classes. Here we leverage EBB controller to programs LSPs or the *Nexthop Groups* [2] on the source router. In this example, the controller programs two lookup steps: 1) a map of prefix p and the loopback of eb01.dc1 to a nexthop group; 2) a map of nexthop group to the interface and label stacks of the corresponding LSP.

Open/R for backup EB-EB reachability. Open/R is Meta's in-house IGP solution which computes the shortest paths for each site-pair [12]. It means that Open/R also provides a route for $eb01.dc2$ to reach $eb01.dc1$. This serves as a backup purpose when the LSPs are not programmed due to failures. Thus, this intra-domain path is assigned with a lower preference. The MPLS-based path is used to forward packets as long as it is configured, and Open/R's shortest path serves as a controller failover solution only.

3.2.2 Evolution of Generations

The planar architecture was in the Express backbone's DNA since its inception and is EBB's multiplying factor for reliability. However, when the network's footprint was much smaller, the EBB had only 4 planes, later extended to 8. For a long time, EBB had only one instance of the controller recomputing reprogramming LSP (Label Switched Path) meshes for each separate plane. Even though the computation was running independently, the business logic implemented in the controller's version was shared. That is, a bug unnoticed during the pre-production testing, could surface in the production controller, impacting the whole backbone at once.

To reduce the blast radius, the team improved the architecture of the control stack. Each EBB plane has a dedicated replica of every service, responsible for a single plane. It helps with the isolation of bugs and incidents to a single plane, helps with feature canary, and improves troubleshooting velocity.

In our release engineering pipeline, after rigorous local testing, both in the lab and in pre-prod environment, our systems first deploy a new version of the software on the EBB Plane1. Only after the release is validated, push is continued to the remaining 7 planes.

Figure 3 shows a real-world example of how traffic is shifted to other planes when a plane is drained.

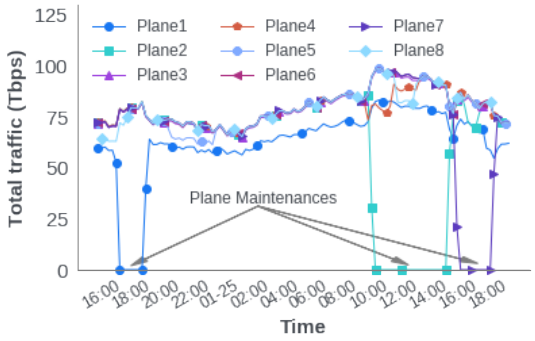

Figure 3: Timeline of plane-level maintenance. When a plane is drained for maintenance, traffic is shifted to other planes.

3.3 Overview of a Plane

Each plane has a dedicated central controller and distributed agents running on network devices, that use an in-house intradomain routing protocol, Open/R, and a full-mesh iBGP. This allows us to control some aspects of traffic flow centrally, e.g., running intelligent path computations. At the same time, we still handle network failures in a distributed fashion, relying on in-band signaling between Open/R key-value store agents deployed on the network nodes. Such a hybrid approach allows the system to be nimble during rapid topology change, for example link failures. As an effect, local LspAgents can immediately reprogram FIBs on the routers to redirect traffic from the LSP transiting through affected links to corresponding precomputed backup LSP, to ensure connectivity and congestion free forwarding. The central controller then has time to evaluate the new topology and compute optimal path allocation, without the urgent need to react rapidly.

Each plane has assigned 6 replicas of the controller, deployed across our data centers to provide geographical redundancy in case of region outage. The replicas are operating in active/passive mode, with only one active at a given time. Since the LSP mesh programming is not atomic, and consists of multiple sequential RPCs, it is very important to ensure mutually exclusive access to the agents running on the network nodes. For that we use distributed locks that ensure safe leader election. The controller is stateless and operates in periodic, independent cycles, each lasting 50-60 seconds. This makes operations easier, as electing new primary replica is as easy as stopping old and starting new process.

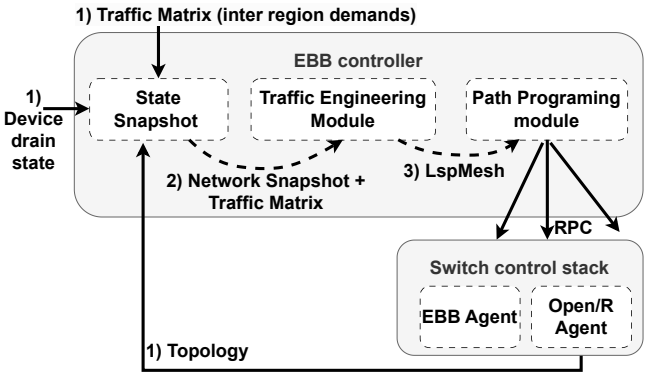

Figure 4: EBB Controller and Switch Modules

The path allocation algorithm can be changed to address the different path requirements. For gold-class traffic, we provide the lowest possible latency, and the provisioned bandwidth is only limited by the physical capacity of the network. For silver-class and bronze-class traffic, EBB provides the best effort service and tries to balance the traffic load over diverse paths to avoid congestion and hot spot as much as possible. We schedule higher-priority traffic ahead of lower-priority traffic.

3.3.1 Controller architecture Figure 4 shows the details of the EBB controller and switch modules. EBB Controller consists of three main modules - (i) State Snapshotter, (ii) Traffic Engineering (TE) module (iii) Path Programming module (often referred as EBB Driver).

State Snapshotter collects requested demands in a form of Traffic Matrix. It also collects real-time topology information from Open/R's key-value store[8]; whose agents run directly on the routers. EBB controller has real-time information about the LAG members that are up, down and what is their current capacity. It also complements the original topology with the drained links, routers or even planes, pulled from the external database. Especially the latter impacts how the paths are computed, de-preferring links, or completely excluding them from the topology graph.

Traffic Engineering module is a generic purpose module used to compute paths with various Traffic Engineering algorithms. This module, maintained as a library, can also be used as a simulation service where Network Planning teams can estimate risk and test various demands and topologies. Traffic Engineering module outputs a structure internally called LspMesh. It's a representation of the set of all computed paths between all the regions, across all priorities. LspMesh is a graph-based model, representing paths.

Path Programming module is used to orchestrate programming LspMesh object produced by TE module to routers in EBB. It first translates LspMesh into objects specific to the network for the Segment Routing with Binding SID (Segment Identifier), represented by NextHop groups, MPLS routes, mapping from prefixes to the NextHop groups and Class-Based Forwarding rules. These objects are sent to on-router Agents via RPC calls by a state machine in Path Programming module to be programmed in hardware. Algorithms in the state machine guarantee make-before-break that ensures no traffic loss from programming.

3.3.2 Switch Modules As shown in Figure 4, each switch contains two types of modules: Open/R agent and EBB agents.

Open/R Agent We discover the live network topology by running Open/R agents on the network devices. Open/R is the distributed platform that provides both the interior routing and the message bus for the Express Backbone network. The LspAgents learn of topology changes in real-time via the Open/R message bus and react locally to link failures. The central controller also interfaces

with the Open/R agents for the purpose of the full network state discovery. In addition to discovering the network topology, Open/R performs RTT measurements and exports the information to the central controller. Open/R leverages IPv6 link-local multicast for neighbor discovery and RTT measurement.

EBB Agents EBB agents are Meta maintained binaries running on each network device. They expose Thrift-based API, and provide an abstraction layer between the EBB Control and Network Operating System (NOS). Examples of agents are: (i) RouteAgent responsible for programming destination prefix matching configuration and Class Based Forwarding rules, (ii) LspAgent responsible for programming everything related to MPLS traffic forwarding, including NextHop groups, MPLS routes, but also providing composited traffic throughput to the Traffic Matrix Estimator service, (iii) KeyAgent responsible for programming MACSec profiles on circuits (iv) FibAgent responsible for programming FIB based on Open/R's shortest path computation and (v) ConfigAgent responsible for network device state configuration, yet exposing the structured configuration to EBB control stack.

Out of all the EBB agents, LspAgent is one of the most "utilized", as LSP mesh is being reoptimised and reprogrammed periodically to satisfy current demands, given current network topology. Should the topology change occur (for instance due to link flap or fibercut), LspAgents reprogram affected LSPs from primary to backup path. Both paths are precomputed by EBB controller, and transmitted to LspAgents. LspAgents trigger NextHop entry reprogramming after it detects events affecting LSP.

4 Centralized TE Controller

In this section, we describe the centralized TE controller of the Express Backbone (EBB).

4.1 Overview

Each plane runs a centralized traffic engineering controller that collects EBB topology from Open/R, acquires traffic matrix from LspAgents, and runs path assignment algorithms.

In order to discover topology, the TE controller polls the Open/R agents on all routers in each plane for the adjacency lists and link capacities. This results in a directed graph with RTT (round-trip time) and capacity as edge properties.

To measure the traffic matrix among sites in EBB, a separate service, called NHG TM (nexthop group traffic matrix), polls the NHG byte counters from the LspAgent on each router. NHG TM then calculates the demands of all site pairs forming a traffic matrix (TM). Demands for all site pairs in a traffic class are grouped into the demand for that class.

EBB performs traffic class based path allocation and it has three types of LSP meshes: Gold Mesh, Silver Mesh, and Bronze Mesh. We allow multiplexing many traffic classes into a single LSP mesh. For example, both ICP and Gold traffic is mapped to Gold Mesh.

The centralized controller assigns paths for LSP meshes in the order of priority: gold, silver, and bronze. After assigning paths for higher priority classes, the remaining capacity from the previous round forms a "new" topology for the next round.

An LSP mesh is a set of LSPs interconnecting all regions for serving one or two traffic classes and derived from the path allocation algorithm. For each site pair currently, we allocate and program 16 LSPs within an LSP mesh, called an LSP bundle. Each programmed LSP is a direct representation of the calculated path in the Path Allocation module.

The TE controller in each plane can run different TE algorithms. It is also possible to run different TE algorithms for different traffic classes. The path allocation algorithms assign paths for each traffic class separately. In addition to primary paths, the TE controller computes a backup path for each primary path. The backup paths will be installed in LspAgents for failure recovery. In this section, we will describe primary path and backup path allocation algorithms.

4.2 Primary Path Allocation

We will describe the three primary path allocation algorithms and discuss the deployment experience of these algorithms. To provide the lowest possible latency given the bandwidth constraints, we use the constrained shortest path (CSPF) algorithm. CSPF provisions each flow with the required traffic demand considering the physical capacity of the network. Furthermore, CSPF selects the shortest path (in terms of round-trip time) among all paths that can accommodate the traffic demand. In addition to CSPF, we can also run a multi-commodity flow (MCF) based algorithm. It provides the best effort service and tries to balance the traffic load over diverse paths to avoid congestion and hot spot as much as possible. Additionally, we explore a heuristic path allocation algorithm for the best-effort traffic class.

4.2.1 Path Allocation Using CSPF

For the gold LSP mesh, the TE controller runs Constrained Shortest Path First (CSPF) algorithm. The CSPF algorithm runs fast and considers both bandwidth constraints and round-trip time.

For each site pair demand, dividing by LSP bundle size gives per-LSP bandwidth. The CSPF algorithm finds the shortest (in Round trip time, RTT) path between two site pairs subjected to link capacity constraint for each LSP at a time across site pairs in a round-robin fashion. That is, the algorithm goes through each site pair assigning one LSP at a time for fairness. Currently, TE controller allocates 16 LSPs per site pair per traffic class. Note that bundle size determines the granularity of the traffic path allocation.

The link weight in the CSPF algorithm is Open/R derived link metric, RTT. It will find the shortest path and load up the shortest path before moving to the next LSP.

In order to prevent drops in ICP and gold traffic, the path assignment algorithm leaves headroom to absorb bursts. For example, suppose you have a 300G link and gold residual bandwidth is configured to be 50%. Only 150G can be used for the ICP and gold traffic. The remaining 150G serves as "headroom" to absorb bursts. Note that the percentage refers to the remaining bandwidth after the previous rounds of the path allocation, not the overall capacity.

The reserved headroom for each link is to reduce packet loss due to bursts. reservedBwPercentage, configured for each traffic class, limits the percentage of remaining link capacity that can be used by LSPs. reservedBwPercentage is the percentage of free capacity this traffic class can use. For example, the residual capacity of a link for silve traffic is (totalCapacity - bw used by gold raffic) * reservedBwPercentage for silver class. Due to limited space, we show the pseudocode of CSPF in Algorithm 3 and the Round-Robin CSPF in Algorithm 4 in Appendix.

4.2.2 Path Allocation Using Multi-Commodity Flow (MCF) and K-Shorest-Path MCF (KSP-MCF)

In addition to the CSPF algorithm, we evaluated arc-based Multi-Commodity Flow (MCF) and K-Shortest-Path Multi-commodity Flow (KSP-MCF) algorithm to allocate paths for the best effort service traffic classes. The goal of MCF is to balance the traffic load over diverse paths to avoid congestion and hot spot as much as possible. Unlike CSPF, MCF does not guarantee the shortest available paths. That is, MCF may use really long paths. KSP-MCF pre-computes a list of K shortest paths connecting all site pairs, and uses only these K paths as possible paths. It gives MCF-like behavior but also a control of maximum "stretched" latency. We used KSP-MCF for silver and bronze-class path allocation in the first few years primarily for the efficiency gain that allowed us to deliver more low-priority traffic with only a few seconds of extra computation time. Next, we will describe the MCF algorithm and KSP-MCF algorithm.

Arc-based Multi-Commodity Flow (MCF). Our linear programming (LP) formulation of arc-based MCF is similar to problem (2) of [42], with the objective to load balance (minimizing maximum link utilization) while preferring shorter paths (link utilization weighted by the RTT of the link and a small constant, similar to [14]). We group commodities with the same destination but different sources into one commodity with multiple sources and a single destination, which reduces the number of flow variables in the MCF formulation thus reducing computation time greatly. We use CLP [1] to solve the LP problem and the solution is a list of b/w (in floating-point values) for each site pair traffic demand on a list of links. We then convert those link traffic to LSP by quantizing link traffic to LSP bandwidth.

K-Shortest-Path Multi-commodity Flow (KSP-MCF). KSP-MCF precomputes K shortest paths (shortest in terms of RTT) for each router pair in EBB (within each plane) with Yen's algorithm [43] as candidate paths, then solves an LP problem to load balance the traffic over all candidate paths while preferring shorter paths (same objective as MCF and same constraints as SMORE [22]). Then we quantize the optimal LP solution into LSPs that could be programmed on routers by greedily allocating LSPs to the candidate paths with the maximum amount of remaining flows.

4.2.3 Path Allocation Using a Heuristic Algorithm: HPRR

In addition to KSP-MCF, we explore a heuristic path allocation algorithm for the best-effort service traffic classes. Motivated by combinatorial approximation algorithms for the MCF problem in [18, 32] that find a $(1+\epsilon)$ approximation through the IMPROVE-PACKING procedure that iteratively reroutes a tiny fraction of the commodity to a new min-cost solution (shortest path or min-cost flow), where the link cost is an exponential penalty based on link utilization (or width of constraint in their papers), we propose a Heuristic Path ReRouting (HPRR) algorithm as shown in Algorithm 1 that iteratively reroutes every path to a less congested path that is computed with Dijkstra's algorithm where the link cost is exponential to link utilization.

HPRR is a local search algorithm working as follows. (1) Calculate initial paths that satisfy flow-conservation constraints but may violate flow capacity constraints with any algorithm. (2) Iteratively for each path, compute a new "shortest" path where the link cost is exponential to post-allocation utilization (the link utilization after routing the path through the link) and reroute the path if the new path is less congested with a smaller path utilization (maximum utilization of all links in the path). (3) Terminate the algorithm after a specified number of epochs.

Algorithm 1 Heuristic path rerouting (HPRR) algorithm to minimize path utilization. ($\mathbf{1}_x$ is an indicator function that equals to 1 if x is true or 0 otherwise.)

Input: Network $G(V, E)$, initial paths $P = \{p_i(s_i, t_i, b_i)\}$, metric parameter α, optimization step size σ, number of epochs N
Output: Rerouted paths P

$\forall e \in E, f[e] \leftarrow \sum_{p_i \in P} \mathbf{1}_{e \in p_i} \cdot b_i$ ▷ Initial flow on edge.
for $n \leftarrow 1 \ldots N$ **do** ▷ Reroute all paths in epochs.
  for $p_i \in P$ **do**
    $u_{p_i} \leftarrow \max_{e \in p_i} \frac{f[e]}{capacity[e]}$ ▷ Utilization of p_i.
    if u_{p_i} is low **and** b_i is small **then**
      Continue
    end if
    $u^{\star}_{p_i} \leftarrow u_{p_i} \cdot (1 - \sigma)$ ▷ Target path utilization.
    $\forall e \in E, u'_e = \frac{f[e] + b_i - \mathbf{1}_{e \in p_i} \cdot b_i}{capacity[e]}$ ▷ Utilization if used.
    $\forall e \in E, w[e] = e^{\alpha \cdot (\frac{u'_e}{u^{\star}_{p_i}} - 1)}$ ▷ Exponential cost.
    $p'_i = dijkstra(G, w, s_i, t_i)$
    $u_{p'_i} \leftarrow \max_{e \in p'_i} u'_e$ ▷ Utilization of p'_i.
    if $u_{p'_i} < u_{p_i}$ **then** ▷ Reroute the path.
      $P \leftarrow P \backslash \{p_i\} \cup \{p'_i\}$
      $\forall e \in E, f[e] \leftarrow f[e] + \mathbf{1}_{e \in p'_i} \cdot b_i - \mathbf{1}_{e \in p_i} \cdot b_i$
    end if
  end for
end for
return P

We explain the parameters used in the algorithm. α is a link cost parameter that is set as $\frac{1}{\epsilon} \log H$, where ϵ is the error bound of path utilization at each path rerouting step and H is the maximum number of hops of most paths. This choice of α ensures that the shortest-path subroutine finds a $(1+\epsilon)$ approximation to the local minimum path utilization in each iteration. We set H to be the maximum hops in production. The step size σ should be large enough to ensure optimization progression but not too large to avoid the exponential explosion of edge cost. N is a trade-off between computation time and efficiency based on experimental results. In EBB, we choose $\epsilon = \sigma = 0.05$, $H = 10$, $N = 3$ and $\alpha = 66.4$ accordingly.

Though HPRR provides no guarantee on global optimality, it achieves better efficiency at the cost of more computation time and higher latency stretch in production (see 6.2). Thus HPRR is a preferred algorithm for bronze traffic class which is sensitive to congestion but not latency.

4.2.4 Deployment Experience: Continuous Adaption of TE algorithms

We fully take advantage of the flexibility of pluggable TE algorithms for the centralized controller. Since March 2017 when EBB had only 7 sites, we have been using CSPF for ICP and gold traffic class for its simplicity, scalability, explainability and the low latency of allocated paths, and KSP-MCF for silver and bronze

traffic class for avoiding congestion. We continuously evolve our TE algorithms in the controller based on traffic conditions.

We are running continuous simulation experiments that evaluate the path allocation quality of different algorithms and parameter settings. For example, the computation cost for KSP-MCF solution has been increasing fast with the dramatic growth of network scale over the last few years. In May 2021, experiments showed that it required a "K" of larger than 1000 and more than 20 seconds of extra computation time for KSP-MCF to achieve better efficiency than CSPF. Thus, we decided to switch from KSP-MCF to CSPF for silver and bronze traffic classes for much less computation time with comparable efficiency. Parameters such as the number of LSPs for each flow, reserved bandwidth percentage of CSPF, and the "K" of KSP-MCF are continuously tuned based on the simulation results. More recently, we have deployed HPRR for bronze traffic class in order to improve load balance and avoid congestion.

4.3 Backup Path Allocation

Every primary path has a backup path. When the primary path fails because one or more of its links fail, its backup path is used to deliver its traffic before the next path allocation runs. After all primary paths are allocated, TE controller calculates the backup paths for each primary path with the objective of (1) Each primary and its backup path do not share SRLG (Shared Risk Link Group). (2) Reduce congestion when the primary paths are down.

We propose our backup path allocation algorithm, Reserved Bandwidth Allocation (RBA) algorithm as described in Algorithm 2. It is an improvement of FIR [26]. FIR aims to minimize the *restoration overbuild*, which is the total extra capacity needed for failure recovery. To this end, FIR sets the link weight value based on how much extra reserved bandwidth is needed, and derives the backup path using the shortest path given the link weight. In contrast, RBA minimizes the post-failure link utilization to reduce network congestion upon failure and leave more headroom for traffic with lower priority.

Besides avoiding the same link or same SRLG as the primary path, RBA considers that multiple backup paths share a link and takes into account of the bandwidth required for each link failure. We set the link weight based on how much the reserved bandwidth to cover the primary path failure is related to the residual capacity of the link, and compute the backup path with the shorest-path algorithm. The idea is to set the link weight value large when the reserved bandwidth for the primary path exceeds the residual capacity of the link. We now describe how to compute the required bandwidth and reserved bandwidth and set the link weights accordingly.

Required bandwidth (*reqBw*): $reqBw$ is a matrix where $reqBw[a][b]$ is the total bandwidth required at link b to cover the traffic traversing link a if a fails. Note that we assume that only one link in the primary path fails at a time when assigning backup path. After allocating the backup for a primary path, we add the required bandwidth for each primary and backup link pair with the required bandwidth of the primary path.

Reserved bandwidth (*rsvdBw*): $rsvdBw$ is primary path specific where $rsvdBw_p[b]$ is the amount of bandwidth that needs to be reserved at link b to cover any single-link failure of the primary path p. To compute the reserved bandwidth, for every link in the network, assume it would be used in the backup path for that primary path being considered, and compute how much bandwidth this link needs if any link on the primary path fails. This reserved bandwidth consists of 1) required bandwidth to recover traffic loss from previous primary paths (including higher-priority traffic classes) with determined backup paths that use this link.; 2) the bandwidth of the current primary path that we are computing its backup path for. To compute 1), iterate every link on the primary path, and find out how much bandwidth is required if each primary path's link fails. The required bandwidth in 1) is the maximum among them.

ReservedBwLimit (*rsvdBwLim*): The $rsvdBwLim$ of a link is the its residual capacity after primary path allocation of the corresponding traffic class.

Once we have $rsvdBwLim$ and $rsvdBw$, we can assign a weight for each link by considering two cases as follows.

1. $rsvdBw < rsvdBwLim$, the weight is $\frac{rsvdBw}{rsvdBwLim} \cdot rtt$.
2. $rsvdBw > rsvdBwLim$, the weight is $\frac{rsvdBw - rsvdBwLim}{totalCapacity} \cdot rtt \cdot penalty$.

This penalty is made in a way that the link weight value is large when $rsvdBw$ exceeds $rsvdBwLim$. And if the excess amount $(rsvdBw - rsvdBwLim)$ is the same for two links, then the link with higher total capacity has less penalty than the other link.

Once the weight for each link is derived, we compute the backup path by weighted shortest path using the new weight. Algorithm 2 shows the pseudocode of this algorithm.

Algorithm 2 Reserved Bandwidth Allocation (RBA) algorithm.

Input: Network $G(V, E)$, SRLGs S, primary paths P
Output: Backup paths P'

$P' \leftarrow \{\}$
$\forall a \in E, \forall b \in E, reqBw[a][b] \leftarrow 0$
for $p_i, bw_{p_i} \in P$ **do**
  for $b \in E$ **do**
    if $b \in p_i$ **then**
      $w[b] = INFINITY$
      continue
    else if $srlgs(b) \cap srlgs(p_i) \neq \emptyset$ **then**
      $w[b] = LARGE$
      continue
    end if
    $rsvdBw_{p_i}[b] = bw_{p_i} + \max_{a \in p_i} reqBw[a][b]$
    if $rsvdBw_{p_i}[b] \leq rsvdBwLim[b]$ **then**
      $w[b] = \frac{rsvdBw_{p_i}[b]}{rsvdBwLim[b]} \cdot rtt[b]$
    else
      $w[b] = \frac{rsvdBw_{p_i}[b] - rsvdBwLim[b]}{totalCapacity[b]} \cdot rtt[b] \cdot penalty$
    end if
  end for
  $p'_i \leftarrow dijkstra(G, w, src_{p_i}, dst_{p_i})$
  $P' \leftarrow P' \cup \{p'_i\}$
  $\forall a \in p_i, \forall b \in p'_i, reqBw[a][b] \leftarrow reqBw[a][b] + bw_{p_i}$
end for
return P'

We also extend algorithm 2 to SRLG-Reserved Bandwidth Allocation (SRLG-RBA) by considering required bandwidth at an edge if

any SRLG fails to improve the network resilience to SRLG failures, where $reqBw$ is a matrix and $reqBw[s][b]$ is the total bandwidth required at link b to cover the traffic traversing srlg s if s fails, and $rsvdBw_p[b]$ is the reserved bandwidth at link b to cover any single-SRLG failure that would impact the primary path p.

5 LSP Meshes

At the network layer, we leverage parts of the MPLS protocol. Paths computed during the Path Allocation are translated directly to a LSP mesh. EBB control stack programs 3 main LSP meshes - gold, silver and bronze mesh. Within a LSP mesh, EBB controller allocates and computes multiple paths for each site-pair. EBB uses an in-house variation of the classic Segment Routing algorithm, described in section 5.2.

5.1 Priority Queues

Express backbone implements Strict Priority Queueing mechanism configured to mitigate network congestion. Each router has a predefined set of rules mapping ranges of DSCP values (denoting priority of the traffic) to various queues. Whenever the network devices buffers are overfilling the router starts dropping lower priority traffic to protect higher priority traffic. In our case Bronze traffic is dropped first to protect Silver, Gold and ICP traffic, however should the congestion persist, such network device drops Silver traffic in order to protect Gold and ICP traffic classes.

5.2 Segment Routing with Binding SID

EBB controller programs 3 LSP meshes independently and concurrently. A component of the EBB controller, the *driver* implements the state machine and follows the algorithm step by step. Driver programs each site-pair independently and opportunistically - programming of siteA-siteB pair success is independent of success of programming of siteM-siteN pair. Since the programming cycle is periodic and executed in reasonably short intervals, we find this simple model robust and easy to reason about.

Segment routing with Binding SID fully utilizes hardware capabilities (i.e maximum MPLS label stack depth supported by the chipset generation), reduces network device forwarding state reprogramming pressure, effectively increasing LSP mesh programming success ratio, and last but not least reduces shared state maintained between routers and EBB control stack to the minimum. The last principle in particular simplifies the overall EBB architecture.

5.2.1 Segment Routing with Static Interface Label Every network device in Express Backbone has a set of MPLS routes programmed during bootstrap. These rules are immutable as long as the device is operational. Each route defines the MPLS label value to match to ingress packets, and the MPLS label action (POP, PUSH, SWAP) and the next hop (expressed as next hop IP, or the identifier of the local NextHop group programmed on a given device). By design, every Port-Channel has a MPLS route associated. These MPLS labels are local to a network device (two or more routers may have label L configured), and are internally called *static interface labels*. MPLS routes on every device have MPLS action set to POP operation, and by default forward the remaining MPLS frame through configured egress interface.

Each path computed during the Path Allocation phase is expressed as an ordered list of egress and ingress interfaces through

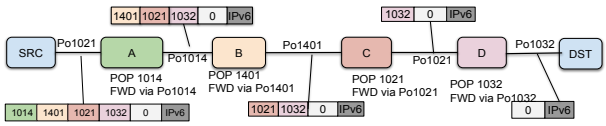

Figure 5: Segment Routing with static interface labels

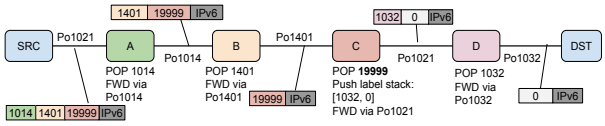

Figure 6: Segment Routing with Binding SID

which the traffic is forwarded. Since the static MPLS labels are statically allocated and known *a priori*, EBB controller is able to map egress interfaces of each segment, and translate into an ordered list of {router, static MPLS label} pairs, and dynamically program source router to encapsulate MPLS label stack on IP packets (see Figure 5). It offers reliable and simple forwarding state programming solution, with only one idempotent RPC call to the source router required. The solution is not feasible for EBB production use case as the number of labels pushed on the label stack is proportional to the length of the LSP. Hardware puts limitations on the maximum labels pushed on the MPLS frame stack. In our case, the limitation is set to maximum of 3 labels on the stack, which guarantees fair hashing entropy based on the 5-tuple values.

5.2.2 Segment Routing with Binding SID In our current path programming scheme, the LSP path is split into a set of segments of N (In our example, $N = 3$) hops. Every N'th hop is programmed by the EBB controller, and configured to route next segment along the LSP.

To avoid MPLS label collision on the routers we define a new type of label - *dynamic label*. The network device responsible for encapsulating the next MPLS segment is called *intermediate node*.

Figure 6 helps visualize the programming configuration. Consider label 19999 as the computed dynamic label for LSP between SRC and DST. For a given LSP and maximum label stack depth of 3, the EBB controller programs two nodes: *intermediate hop* C and SRC. This has MPLS route programmed for label 19999 (dynamic label), and as action pushes another label stack. Compared to original LSP (Figure 5), source node is reprogrammed with label stack of depth 3, however the bottom label is set to label *19999*. Segment Routing with Binding SID allows for programming LSPs of any length, regardless of the hardware imposed limitations. On the other hand, maximizing its capabilities, we reduce *programming pressure* on the nodes - to configure the following LSPs, only two nodes (SRC and C) must be dynamically reprogrammed.

5.2.3 Network-wide Dynamic SID Dynamic SID labels are attributed to the *bundle of LSPs* computed for each site-pair and at the given class. While each LSP is split into segments independently, there is always a chance a node becomes an intermediate node for multiple LSPs from a site-pair bundle. Technically, each intermediate node is reprogrammed with an ingress dynamic MPLS label and the egress NextHop group. This is because the SID-allocated label

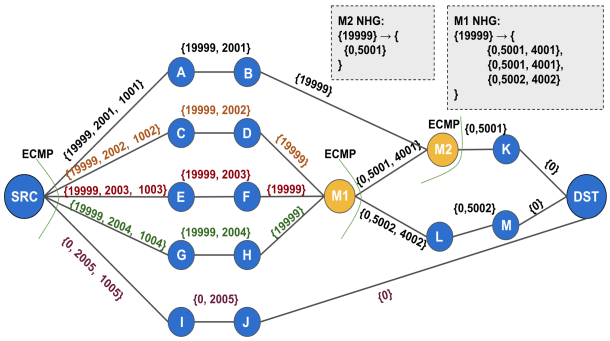

Figure 7: Dynamic Label Switching

encodes the set of LSPs between a given site-pair (and the priority), not a single LSP.

Consider the 5 LSP bundle between SRC and DST in Figure 7 with the maximum label stack depth set to 3. As an effect of the LSP mesh computation in the Path Allocation stage, there are parts with <=3 hops (SRC, G, H, J, DST), but the remaining are longer. For the purpose of the demonstration consider 3 LSPS - (i) (SRC, C, D, M1, M2, J, DST), (ii) (SRC, X, Y, M1, M2 J, DST), (iii) (SRC, E, F, M1, I, K, DST). In the *path split* phase, the driver finds that M1 an intermediate hop for all 3 LSPs. It then programs an MPLS route for ingress label 19999 and the egress NextHop group with 3 entries: (a) (M2, J, DST), (b) (M2, J, DST) (c) (M1, I, K, DST). One can notice entries (a) and (b) are identical, and they match with the subarray of original LSPs (i) and (ii), while entry (c) matches the subarray of the original LSP (iii).

Node M2 is an intermediate node for LSP (SRC, A, B, M2, J, DST), hence the driver programs the MPLS route for ingress label 19999 and NextHop group with single egress entry. Note that traffic admitted to LSPs (i), (ii) or (iii) does not depend on this forwarding state. As per segment splitting traffic that ingress to M2 has a top label with a value 4001.

5.2.4 Dynamic SID label In our model, dynamic labels are dynamically programmed on intermediate hops and have dynamically computed and associated MPLS action and label stack to encapsulate on the packet. The numeric value of the SID label is symmetrically encoded and decoded. We define bit ranges inside the MPLS bit space, and each field encodes the following attributes as shown in Figure 8: (i) label type (static vs dynamic) [1 bit](If the label type is static, the following bits indicate the static label. If the label type is dynamic, the following bits use the format as follows.) (ii) source region [8 bits] (iii) destination region [8 bits] (iv) LSP mesh [2 bits] (v) LSP mesh version [1 bit] (see 5.3).

Symmetric encoding eliminates the need for shared state between the EBB control stack, network device configuration, and EBB agents. This reduces failure domain scope, simplifies the overall failure model and minimizes the number of external dependencies, for example, persistent storage. The solution introduces certain limitations. However at our current scale, we are far away from reaching them. For instance, the maximum number of regions supported in a current scheme is $2^8 = 256$.

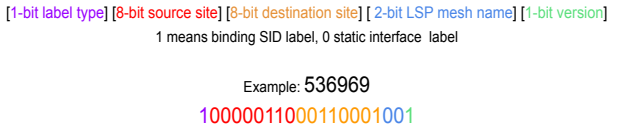

Figure 8: Dynamic Label Format

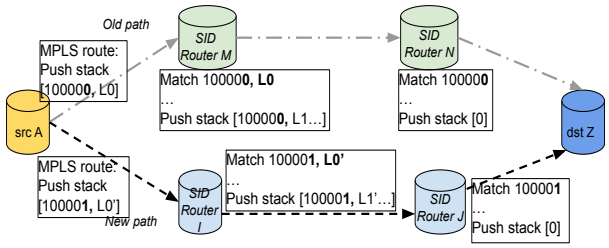

Figure 9: Programming of the site-pair LSPs with make-before-break

5.3 LSP Mesh Update after Path Assignment

Programming of the site-pair is a multi-step process and in order for the operation deemed successful, all N (where N=16) LSPs must be successfully programmed. We use an internally designed variation of the Segment Routing with Binding SID to install routing states. Segment Routing with Binding SID requires multiple nodes to be successfully reprogrammed in a strict order, therefore we developed a label allocation scheme allowing us to guarantee *make-before-break* principles.

To ensure an end-to-end forwarding state EBB controller must program not only the source router but also all intermediate hops. Since MPLS routes and their corresponding NextHop groups are programmed dynamically by the EBB controller, the lack of their presence on the intermediate node would result in traffic blackholing. In other words, for each site pair, all intermediate nodes must be reprogrammed before the source router is reprogrammed. Reprogramming of the site-pair mesh requires multiple RPCs and is not an atomic operation. On the other hand, the traffic constantly being forwarded through the EBB and the *make-before-break* guarantee must be satisfied. To do so, dynamic SID labels have 1 bit allocated to denote the *version* of the LSP mesh.

For each LSP bundle between two sites, in a given time, there is a single SID MPLS label allocated, including its version. When the forwarding state for the new LSP bundle must be reprogrammed, the controller allocates the SID label with unused version and programs MPLS routes and their NextHop groups on the intermediate nodes, and only after previous phase is concluded, reprograms source device. With single bit allocated to a *version* of the site-pair LPS bundle, allocated SID labels have different numeric values, which avoids collision in the traffic forwarding.

Consider a 1-LSP site-pair bundle in figure 9. The programmed LSP between A and Z is (SRC, M, N, DST), and the new computed path is (SRC,I,J,DST). The currently used SID label for such site pair is 100000 (version bit is set to 0). EBB controller computes intermediate nodes and assigns SID value 100001 (version bit is flipped to 1). It then programs intermediate nodes I and J, and once

this step is successful, reprograms source A to forward traffic to I, and encapsulates with label 100001. Once the source node is reprogrammed, the LSP between A an Z is (A,I,J,Z).

5.4 LSP Mesh Update during Failover

EBB controller computes two types of path - for each primary, there is a corresponding backup. The backup path is activated during topology changes and used until the next programming cycle, where controller recomputes LSP mesh with the new topology state. Upon topology change, for example link flap, the event is propagated via Open/R's key-value store. Upon receiving such event, an instance of LspAgent inspects all currently dynamically programmed NextHop groups, iterating over the Nexthop entries. LspAgent maintains an in-memory cache with the whole path (as ordered array of router, interface pairs) and inspects whether Nexthop entry currently forwards traffic through affected path. In such case, such entry is removed from the FIB, symmetrically.

LspAgent maintains the NextHop entry along with both primary and backup paths end to end in memory. Upon topology change, LspAgent inspects if the reachability of the primary path is impacted, and if so programs NextHop entry for the backup path. Primary and backup paths are meant to be completely disjoint, which means that intermediate nodes for primary and corresponding backup paths are mutually exclusive. Hence, the operations of deprogramming of primary and programming of backup paths are happening on separate routers, often in parallel.

A source node is a special case: the same router always removes NextHop entry from the FIB and installs its backup counterpart. Since the SID label decimal value represents an LSP bundle between two sites and for single LSP mesh, we do not distinguish between primary and backup meshes. That's also why intermediate hops participate in the reprogramming of primary and backup paths failover.

6 Evaluation

In this section, we present our evaluation results for traffic engineering with historic and synthetic topology and traffic.

6.1 Traffic Engineering Algorithm Computation Time

Experimental Setting. We evaluate different TE algorithms including CSPF, MCF, HPRR, and KSP-MCF (K equal to 512 and 4096) for primary path computation and Reserved Bandwidth Allocation (RBA) for backup path computation on steady-state (no failure or maintenance) topologies with traffic matrices over last 2 years. We use the same algorithm for all traffic classes in each experiment. We run the experiments on a 32-core KVM with Intel Xeon Processor (Skylake) @ 1.60GHz with 50GB of RAM.

Evaluating TE algorithms in production. Figure 10 shows the number of nodes, the number of edges, and LSPs over time. Figure 11 shows the computation time of different algorithms over time. At the current scale, CSPF is about 15x faster than KSP-MCF and 5 times faster than MCF. The computation time of HPRR (including path initialization with CSPF) is about 1.5 times of CSPF, as many paths are skipped in later iterations when the network is less congested. The computation time for backup path allocation is 2 times of the primary path allocation with CSPF.

During the deployment over the last two years, we *dynamically* switch TE algorithms for each traffic class in the real network to respond to different network conditions. Initially, we use CSPF for the Gold traffic class and KSP-MCF for the Silver and Bronze traffic class. Over time, we adapt the TE algorithm in the controller as follows.

- We discovered a capacity risk related to the silver traffic class in one region. Thus, we increase the value k in the KSP-MCF algorithm for the silver traffic.
- We monitored the runtime performance of the TE algorithm and found it exceeded 30s with a large K (1,000 to 4,000), we decided to switch silver to CSPF for faster TE computation.
- We subsequently switched the TE algorithm for Bronze traffic to CSPF for efficiency and then recently to HPRR.

6.2 Effectiveness of TE

We evaluate the effectiveness of TE with link utilization and latency stretch by running simulations with hourly production-state snapshots of EBB topology and traffic matrices over 2 weeks. We use the same TE algorithm to allocate 16 equally sized paths for all flows in each experiment. We also run the experiment (MCF-OPT) to compute the optimal state with MCF and a large bundle size (512) to reduce the quantization error in the LP solution to LSPs conversion.

Link Utilization. Figure 12 shows the CDF of link utilization percentage of all links at all times for different TE algorithms. We compute link utilization based on the allocated paths and estimated traffic matrix assuming that all traffic is routed. The utilization of more than 100% on a link indicates congestion and excessive traffic will be dropped by priority. At the current EBB network scale, KSP-MCF (even with a large "K" of 4096) is less capacity efficient than MCF and CSPF with more highly utilized links (with utilization over 80%). This is primarily because the "K" is not large enough to provide the needed path diversity for KSP-MCF to find a near-optimal solution. The link utilization distribution is similar for MCF and CSPF when utilization percentage is over 80%. Due to the rounding error when converting the fractional solutions of MCF and KSP-MCF to 16 equally sized paths per flow, the utilization of a few links can be extremely high with MCF and KSP-MCF based approach. A large percentage of links has utilization of 80% in CSPF solution, as we reserved 80% of total link capacity for CSPF to leave headroom for traffic bursts and CSPF would use up all the reserved capacity on the shortest path before moving to a different path. The maximum link utilization of HPRR is much less than CSPF, MCF and KSP-MCF, with the percentage of highly-utilized links close to the optimal state of MCF-OPT. Overall, our backbone link utilization is high due to active control of traffic admission [4].

Latency Stretch. Latency stretch is the ratio of the RTT of the allocated path (RTT_{p_i}) over the shortest-path RTT ($RTT^{\star}_{p_i}$) between the source and destination. For each flow (identified by ingress site, egress site and traffic class) at each time, we compute the average latency stretch and maximum latency stretch of all paths in the LSP bundle.

As the RTT of the shortest paths between some site pairs is so small (only a few milliseconds) that any detour from the shortest path would result in a large latency stretch even though the absolute RTT value is still small enough for services, we normalize latency stretch as $\max\{1, \frac{RTT_{p_i}}{\max\{c, RTT^{\star}_{p_i}\}}\}$, where c is a constant RTT

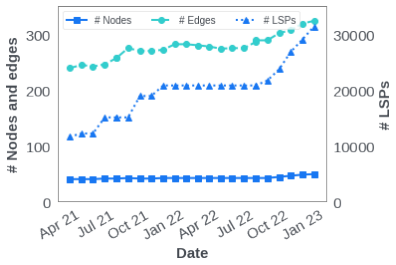

Figure 10: EBB topology size in past 2 years.

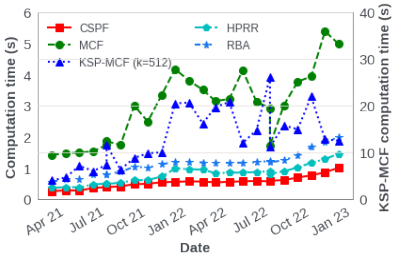

Figure 11: TE computation time (right y-axis for KSP-MCF).

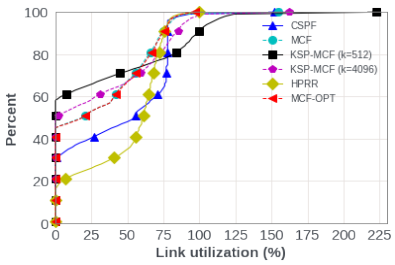

Figure 12: CDF of link utilization

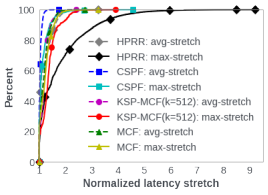

Figure 13: CDF of avg/max latency stretch of gold-class flows.

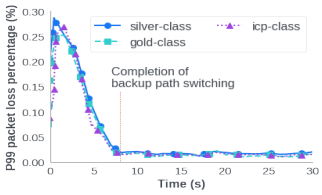

Figure 14: Recovery process from a small SRLG failure.

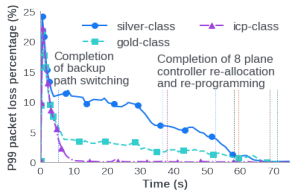

Figure 15: Recovery process from a large SRLG failure.

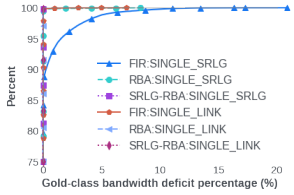

Figure 16: CDF of gold-class bandwidth deficit percentage

that is small enough for any service. Figure 13 shows the CDF of normalized (with c being 40 ms) per-flow average and maximum latency stretch of all gold-class flows between data centers for different TE algorithms. HPRR has the most latency stretch. CSPF has the least average latency stretch. While the maximum latency stretch of CSPF is similar to or larger than MCF and KSP-MCF, as round-robin CSPF would have to allocate longer paths when there isn't enough capacity in the shorter paths. We use CSPF as the primary path allocation for gold-class traffic for its low average latency stretch and simplicity.

6.3 Failure Recovery

6.3.1 Loss during failure recovery. EBB recovers from network topology failures in three phases.

1. At the beginning of the failure, all traffic on the failed links is dropped due to a black hole.
2. LspAgents detect the failure and switch affected primary paths to available backup paths in a few seconds. Depending on the efficiency of the backup paths, traffic is still susceptible to congestion loss.
3. At the next programming cycle, TE controller recomputes and reprograms the paths and the network fully recovers.

Figure 14 shows the recovery process from a small SRLG failure. It took 7.5 seconds for all routers to switch to backup paths after receiving the link-down report. There was no congestion loss for ICP, Gold and Silver classes after switching to backup paths. Figure 15 shows the recovery process from an impactful SRLG failure when FIR was used as the backup path algorithm. All traffic classes suffered adverse drops upon the SRLG failure. In 3 to 6 seconds, LspAgents completed the backup path switching. The backup path switch mitigated all the ICP drops within 5-7 seconds though Gold and Silver showed prolonged congestion until Controller had the chance to compute and program new meshes.

6.3.2 Efficiency of backup paths The efficiency of backup paths determines the magnitude of prolonged congestion loss after the backup path switch within the same programming cycle. Historically we used FIR, and then developed RBA and SRLG-RBA that is more efficient for single-link failures and single-SRLG failures. We compare the efficiency of FIR, RBA and SRLG-RBA under all possible single-link and single-SRLG failures with simulation.

We run simulations with hourly topology snapshots and traffic matrices over 2 weeks. First, we allocate primary paths (using CSPF) and backup paths using different backup path algorithms. Then we simulate for each possible single-link failure and single-SRLG failure, and report the per-traffic class bandwidth deficit ratio (total amount of traffic that cannot be accepted without congestion / total amount of traffic) of each backup path algorithm upon each failure. Figure 16 shows that RBA almost eliminates gold-class congestion under single-link failures, and SRLG-RBA almost eliminates gold-class congestion under both single-link and single-SRLG failures.

7 Operational Experiences

One fundamental philosophy of EBB's design is for operational optimization - the plane architecture which we adapt from our fabrics allowed for decoupled upgrades, experimentation, and testing; the hybrid architecture provides safe controller fall-back, allowing us to more quickly and confidently iterate and update controller software and algorithms. Host-based marking and switch-based enforcement give fewer touch-points where traffic is impacted, which is easier to debug. We highlight two key learning below.

7.1 Circular Dependency of other Core Infra Services

EBB Controller plays a critical role in the reliability of Meta's entire infrastructure. However, to build scalable services, the controller unavoidably has to leverage other Meta's distributed services. In one such example, the controller leverages the pub/sub service Scribe [19] to collect traffic statistics. In one outage, there was

severe network congestion that caused Scribe service to fail. The controller was supposed to recompute the path to alleviate the congestion in the next TE cycle. However, it is blocked by the step of writing additional data through the Scribe API. The circular dependency caused the network and the Scribe service to be blocked by each other. The mitigation solution was updating the controller to temporarily bypass the Scribe call.

Implication After this incident, we changed to use all async calls to read and write to Scribes. In addition, we systematically study the circular dependency on other infrastructure services. We conducted the *dependency failure testing* and integrated it into our release pipeline. In addition, for other services, if possible, we build a separate binary and run a local copy on the same machine as the controller to minimize dependency on the network communication. Besides these engineering practices, this example also calls for new failure modeling and automatic analysis for circular dependency in large-scale network systems such as backbone networks. Instead of discovering circular dependency based on occurred outages, we argue that it is essential to build an automatic analysis of circular dependency in the release pipeline.

7.2 Failure Recovery

Although EBB's planar design is targeted for strong reliability, unexpected mistakes and catastrophic failure can still happen. Thus, we develop a series of mitigation and automated recovery process from the worst-case failures. Such mitigation has helped rescue the network and all Meta's services from severe outages. In one outage, a minor configuration change to enable a security feature was pushed to all eight planes. Note that we only do staging on one plane for major disruptive configuration changes and this specific change has passed the normal canary phase. However, this security feature caused unexpected link flaps on all EBB links, leading to high packet loss and bringing all our services down. The high loss was detected around 5 minutes after the configuration rollout by our monitoring services and a rollback was triggered automatically. The outage was recovered within 10 minutes.

Implication This incident reveals the importance of auto-recovery system design. The auto-recovery system needs to consider the worst-case scenarios and is able to be activated automatically. Further, when designing the auto-recovery system, we need to model the mean time to recovery in order to respond to failures in a timely manner.

However, the recovery is more challenging when the connectivity is completely lost. In a recent well-known outage [16], a misconfiguration caused all eight planes of EBB to be drained, i.e., completely offline. It effectively disconnected all data centers, where the controller resides. Even remote access to the EBB routers was hindered because the authentication services in the data centers were not accessible. Such an outage was extremely rare but once it happened it required manual or even physical access to recover. Even worse, when the backbone is recovered, all data center services initiate the communications simultaneously, which could overwhelm the network again. Thanks to Meta's continuous disaster recovery drills [40], all services gradually recovered smoothly after EBB became available again.

Implication This incident reinforces that such large-scale network configuration changes necessarily bring out the worst-case scenarios. Dependency among different services needs to be modeled before deploying configuration changes. Further, necessary recovery mechanisms during disaster scenarios need to be built considering the unique perspective of software-defined networks.

8 Related Work

Software Defined Networking: In the past decade, the SDN based centralized control for backbone networks has been studied extensively both in academia and industry [11, 14, 15, 35]. B4 [15] introduces the SDN network in Google WAN, and SWAN [14] reveals the SDN solution in Microsoft backbone. The distinction of EBB is its multi-plane architecture that allows continuous evolution of the centralized controller and provides seamless deployment of new centralized control.

Traffic engineering: Traffic engineering in the WAN has been studied in various settings including datacenter WANs and ISP networks[7, 10, 13–15, 21, 23, 24, 27, 27, 31, 36, 37, 41]. Similar traffic control is done at the edge networks [34]. Instead of just focusing on TE algorithms, this paper points out the necessity of evolving TE algorithms as the network scenario changes, and shares our experience of how we select specific TE algorithms in production.

Network Management: A number of studies have focused on addressing the management of SDN networks. In particular, managing route update is the key to avoid congestion and loss in SDN networks [9, 17, 28, 29, 38]. [17] proposes a two-phase commit method to ensure network-wide consistency in SDN networks. Others have focused on verification of routing correctness in SDN networks [6, 20, 33]. The management systems from Meta have been introduced in [4, 39, 44]. These systems are closely integrated with EBB for management and bandwidth control.

9 Conclusion

The wide-area Backbone network plays a crucial role in the reliability and performance of large-scale Internet services. In this paper, we introduce Express Backbone (EBB), an SDN-like network control stack with a hybrid design to ensure fast fault recovery. We share EBB's TE solutions and the philosophy behind its transformation through generations. Further, we present a unique MPLS-based data plane design that tackles existing hardware's limitations. We share eight years of operational experiences with EBB and hope to inspire future research for hybrid network control, failure modeling, and evolvable traffic engineering.

Acknowledgements

Many people in the Backbone Engineering organization at Meta have contributed to the Express Backbone program over the years. We would like to acknowledge Vipul Deokar, Gurinder Grewal, Brendan Cleary, Jaime Miles, Ranjeeth Dasineni, Will Collier-Byrd, Mark McKillop, Alberto Herrero Mediavilla, Andrey Golovanov, Anton Marchenko, Pavel Kiselev, Alexandru Manea, Palak Mehta, Rasmus Jönsson, Jonathan Weis, Sanjay Murthy, Nisha Ananda, Priyanka Sangtani, Iuliana Mitac, James Paussa, Srivani Ward, Madhan Kumar Ramachandran, Rong Rong, Vitaly Neganov and Saikat Ray. In addition, we sincerely thank the anonymous reviewers for their valuable feedback on earlier versions of this paper.

References

[1] [n.d.]. COIN-OR Linear Program Solver. ://www.coin-or.org/Clp/.

[2] [n.d.]. Configuring Next-Hop Groups to Use Multiple Interfaces to Forward Packets Used in Port Mirroring. https://www.juniper.net/documentation/us/en/software/junos/sampling-forwarding-monitoring/topics/concept/policy-configuring-next-hop-groups.html.

[3] [n.d.]. RSVP-TE. https://en.wikipedia.org/wiki/RSVP-TE.

[4] Satyajeet Singh Ahuja, Vinayak Dangui, Kirtesh Patil, Manikandan Somasundaram, Varun Gupta, Mario Sanchez, Guanqing Yan, Max Noormohammadpour, Alaleh Razmjoo, Grace Smith, Hao Zhong, Abhinav Triguna, Soshant Bali, Yuxiang Xiang, Yilun Chen, Prabhakaran Ganesan, Mikel Jimenez Fernandez, Petr Lapukhov, Guyue Liu, and Ying Zhang. 2022. Network Entitlement: Contract-Based Network Sharing with Agility and SLO Guarantees *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 250–263.

[5] Alexey Andreyev. 2014. Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 155–168.

[7] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. 2019. TEAVAR: Striking the Right Utilization-Availability Balance in WAN Traffic Engineering. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 29–43. https://doi.org/10.1145/3341302.3342069

[8] Facebook. 2018. KvStore - Store and Sync. https://openr.readthedocs.io/Protocol_Guide/KvStore.html.

[9] Klaus-Tycho Förster, Ratul Mahajan, and Roger Wattenhofer. 2016. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *2016 IFIP Networking Conference, Networking 2016 and Workshops, Vienna, Austria, May 17-19, 2016*. 1–9. https://doi.org/10.1109/IFIPNetworking.2016.7497232

[10] Bernard Fortz, Jennifer Rexford, and Mikkel Thorup. 2002. Traffic engineering with traditional IP routing protocols. *IEEE communications Magazine* 40, 10 (2002), 118–124.

[11] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *SIGCOMM* (Florianopolis, Brazil). 15 pages. https://doi.org/10.1145/2934872.2934891

[12] Saif Hasan, Petr Lapukhov, Anuj Madan, and Omar Baldonado. 2017. Open/R: Open routing for modern networks. https://engineering.fb.com/2017/11/15/connectivity/open-r-open-routing-for-modern-networks/.

[13] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu Bollineni, 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. 2018. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-defined WAN. In *ACM SIGCOMM (2018)*.

[14] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-Driven WAN. In *Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Hong Kong*.

[15] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined Wan. *SIGCOMM* (2013), 12 pages.

[16] Santosh Janardhan. 2021. More details about the October 4 outage. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/.

[17] X. Jin, H. Harry Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. 2014. Dynamic scheduling of network updates. In *ACM SIGCOMM (2014)*, Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy (Eds.). 539–550.

[18] David Karger and Serge Plotkin. 1995. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. 18–25.

[19] Manolis Karpathiotakis, Dino Wernli, and Milos Stojanovics. 2017. Scribe: Transporting petabytes per hour via a distributed, buffered queueing system. https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/.

[20] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 9.

[21] Praveen Kumar, Chris Yu, Yang Yuan, Nate Foster, Robert Kleinberg, and Robert Soulé. 2018. YATES: Rapid Prototyping for Traffic Engineering Systems. In *Proceedings of the Symposium on SDN Research* (Los Angeles, CA, USA) *(SOSR '18)*. ACM, New York, NY, USA, Article 11, 7 pages. https://doi.org/10.1145/3185467.3185498

[22] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. 2018. Semi-oblivious traffic engineering: The road not taken. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 157–170.

[23] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Ciun L. Lim, and Robert Soulé. 2018. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *USENIX NSDI (2018)*.

[24] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. 2011. Inter-datacenter bulk transfers with netstitcher. In *Proceedings of the ACM SIGCOMM 2011 Conference*. 74–85.

[25] George Leopold. 2017. Building Express Backbone: Facebook's new long-haul network. http://code.facebook.com/posts/1782709872057497/.

[26] Guangzhi Li, Dongmei Wang, C. Kalmanek, and R. Doverspike. 2002. Efficient distributed path selection for shared restoration connections. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 1. 140–149 vol.1. https://doi.org/10.1109/INFCOM.2002.1019255

[27] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic Engineering with Forward Fault Correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 527–538.

[28] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. zUpdate: Updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. 411–422.

[29] Ratul Mahajan and Roger Wattenhofer. 2013. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. 1–7.

[30] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review* 38, 2 (2008), 69–74.

[31] Abhinav Pathak, Ming Zhang, Y Charlie Hu, Ratul Mahajan, and Dave Maltz. 2011. Latency inflation with MPLS-based traffic engineering. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. 463–472.

[32] Serge A Plotkin, David B Shmoys, and Éva Tardos. 1995. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research* 20, 2 (1995), 257–301.

[33] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 953–967.

[34] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. 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* (Los Angeles, CA, USA) *(SIGCOMM '17)*. ACM, New York, NY, USA, 418–431. https://doi.org/10.1145/3098822.3098853

[35] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 183–197.

[36] Kandula Srikanth, Dina Katabi, Bruce Davie, and Anna Charny. 2005. Walking the tightrope: responsive yet stable traffic engineering. (2005).

[37] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. 2011. Network Architecture for Joint Failure Recovery and Traffic Engineering. In *ACM SIGMETRICS (2011)*.

[38] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A Network-State Management Service. *SIGCOMM Comput. Commun. Rev.* 44, 4 (aug 2014), 563–574.

[39] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. 2016. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 426–439.

[40] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, et al. 2018. Maelstrom: mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 373–389.

[41] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. 2006. COPE: Traffic engineering in dynamic networks. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. 99–110.

[42] Dahai Xu, Mung Chiang, and Jennifer Rexford. 2011. Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering. *IEEE/ACM Transactions on networking* 19, 6 (2011), 1717–1730.
[43] Jin Y Yen. 1970. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of applied mathematics* 27, 4 (1970), 526–530.
[44] Yang Zhou, Ying Zhang, Minlan Yu, Guangyu Wang, Dexter Cao, Yu-Wei Eric Sung, and Starsky H. Y. Wong. 2022. Evolvable Network Telemetry at Facebook. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 961–975.

Appendices

Appendices are supporting material that has not been peer-reviewed.

A TE Algorithms in Detail

CSPF. Algorithm 3 shows the pseudocode for CSPF. The complexity of CSPF algorithm is $O(|P|\cdot(|V|+|E|)\cdot\log|V|$, where $|P|$, $|V|$, $|E|$ are the number of LSPs, vertices and edges respectively. This allocation runs once the last round of running is complete. Algorithm 4 shows the round robin path allocation procedure using CSPF.

HPRR. The time complexity of HPRR is $O(N\cdot|P|\cdot(|V|+|E|)\cdot\log|V|$ as it is running Dijkastra's algorithm in N epochs. The algorithm is pseudo-polynomial in nature, allowing for an undetermined number of path rerouting epochs. In the context of EBB, we have determined that rerouting path for three epochs yields satisfactory results.

RBA. The time complexity of RBA is $O(|P|\cdot(|V|+|E|)\cdot\log|V|)$. Assuming average number of hops of the paths is H, the time to compute $rsvdBw$ and $reqBw$ is $|P|\cdot|E|\cdot H$ and $|P|\cdot H^2$ respectively. Considering that H is relatively small in a well-connected graph when compared with $|E|$, $|V|$ and $\log|V|$, the time to compute the metrics and update the tables is at a smaller order of magnitude than the the computation time of shortest paths.

Algorithm 3 CSPF

Input: Network $G(V,E)$, flow $f(s,t,bw)$, constraint C
Output: A path p from s to t

$Q \leftarrow (s,0)$
$\forall v \in V, v \neq s, dist[v] \leftarrow INFINITY$
$dist[s] \leftarrow 0$
$\forall v \in V, prev[v] \leftarrow UNDEFINED$
while Q is not empty **do**
  $u \leftarrow Q.extractMin()$
  for $v \in neighbors(u)$ **do**
    if $C(f, e_{u,v})$ and $bw \leq e_{u,v}.freeCapacity$ **then**
      $dist_{v,u} \leftarrow dist[u] + e_{u,v}.metric$
      if $dist_{v,u} < dist[v]$ **then**
        $dist[v] \leftarrow dist_{v,v}$
        $prev[v] \leftarrow u$
        $Q.update(v, dist_{v,u})$
      end if
    end if
  end for
end while
if $prev[t]$ is UNDEFINED **then**
  return $\emptyset$
end if
$p \leftarrow \{\}, v \leftarrow t$
while $prev[v]$ is not UNDEFINED **do**
  $p \leftarrow p \cup \{e_{prev[v],v}\}$
  $v = prev[v]$
end while
return p

Algorithm 4 Round-robin CSPF

Input: Network $G(V,E)$, Flows F, constraint C, bundle size B
Output: Allocated paths P

$P \leftarrow \{\}$
for $n \in \{0, 1, ..., B\}$ **do**
  for $f_i \in F$ **do**
    $p_i^n \leftarrow CSPF(G, f_i, C)$
    $P \leftarrow P \cup \{p_i^n\}$
    for $e \in p_i^n$ **do**
      $e.freeCapacity \leftarrow e.freeCapacity - bw$
    end for
  end for
end for
return P