Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering

Kok-Kiong Yap Murtaza Motiwala Jeremy Rahe Steve Padgett Matthew Holliman Gary Baldus Marcus Hines Taeeun Kim Ashok Narayanan Ankur Jain Victor Lin Colin Rice Brian Rogan Arjun Singh Bert Tanaka Manish Verma Puneet Sood Mukarram Tariq Matt Tierney Dzevad Trumic Vytautas Valancius Calvin Ying Mahesh Kallahalla Bikash Koley Amin Vahdat
Google
espresso-team@google.com

ABSTRACT

We present the design of Espresso, Google's SDN-based Internet peering edge routing infrastructure. This architecture grew out of a need to exponentially scale the Internet edge cost-effectively and to enable application-aware routing at Internet-peering scale. Espresso utilizes commodity switches and host-based routing/packet processing to implement a novel fine-grained traffic engineering capability. Overall, Espresso provides Google a scalable peering edge that is programmable, reliable, and integrated with global traffic systems. Espresso also greatly accelerated deployment of new networking features at our peering edge. Espresso has been in production for two years and serves over 22% of Google's total traffic to the Internet.

CCS CONCEPTS

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

KEYWORDS

Networking, Peering Routers, Traffic Engineering

ACM Reference format:
Kok-Kiong Yap Murtaza Motiwala Jeremy Rahe Steve Padgett Matthew Holliman Gary Baldus Marcus Hines Taeeun Kim Ashok Narayanan Ankur Jain Victor Lin Colin Rice Brian Rogan Arjun Singh Bert Tanaka Manish Verma Puneet Sood Mukarram Tariq Matt Tierney Dzevad Trumic Vytautas Valancius Calvin Ying Mahesh Kallahalla Bikash Koley Amin Vahdat. 2017. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21–25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098854

SIGCOMM '17, August 21–25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-4653-5/17/08.
https://doi.org/10.1145/3098822.3098854

1 INTRODUCTION

The Internet's peering edge plays a critical and growing role in the architecture of large-scale content providers, driven by High-Definition Video and Cloud Computing. The largest peering edges deliver Terabits/sec of Internet traffic and megawatts of compute/storage. While most of the computing and storage for content providers runs in data centers, the edge supports: i) peering with partner autonomous systems, ii) a server pool of reverse proxies for TCP termination, and iii) caching and content distribution of static content. A properly designed edge architecture supports interactive low latency to a global user population, is the first and most important line of defense against DoS and related attacks, and reduces the buildout of the backbone network back to centrally-located data centers.

The dominant component of an edge architecture is Internet-scale routers. These routers are hardware and software engineering marvels, with at least three critical pieces of functionality. First, the router must scale to hundreds of ports with the highest bandwidth density and packet per second processing rates in each generation. The management and configuration complexity of Internet routers is substantial, so our tendency has been to favor scale up with as few, large routers as possible. Second, the router must support Internet-scale forwarding tables with potentially hundreds of thousands of individual entries down to /24 subnets in the case of IPv4 for global Internet-scale routing. Third, on the incoming path, routers must support complex and large-scale access control lists to support firewall rules and protect against DoS attacks. Otherwise specialized firewalls have to be deployed in peering locations, consuming limited space and power. Finally, the router must support high-end compute for BGP software that can manage hundreds of sessions with remote peers.

In our experience running the network for one of the largest global content providers, the flexibility, availability, and cost efficiency of the peering edge was increasingly limited by these Internet routers. Most critically, we could not introduce new functionality leveraging a global view of traffic or cross-layer optimizations. For example, we measure, in real time, the peer ports most likely to deliver high-bandwidth/low-latency connectivity from our edge by measuring across millions of individual users. Overriding BGP-specified forwarding behavior at fine granularity is limiting. In cases where it is possible, it often requires some change in the vendor software or hardware. The simplest changes can take up to a year to qualify, while more complex changes require either

standardization efforts or new versions of switch silicon. Further, the scale up nature of high-end Internet routers means that any upgrade or failure, however rare, would affect a substantial fraction of our traffic. Finally, the cost per port of Internet routers dominated our edge costs, typically $4-10\times$ relative to the next tier of router with more modest forwarding and ACL table size.

To address these challenges, we used our experience with SDN in a different setting [21] to design and implement Espresso, a new peering edge architecture. The key insight behind Espresso is externalizing most network control from the peering devices, and leaving only a simple data-plane on device—specifically a commodity MPLS switch. In particular, we move packet processing, including routing on Internet-scale forwarding tables and ACLs, to high-performance software packet processors running on the large-scale server infrastructure already present in the edge. Further, we integrate our pre-existing global traffic engineering (TE) system into Espresso to enable fine-grained, BGP-compliant bandwidth management in a manner that would be difficult to implement in a distributed environment and without an end-to-end view of per-flow performance. Finally, we move BGP to a custom stack running on servers, enabling finer-grained partitioning and much more computation power than available in any Internet router.

Taken together, Espresso's design accelerates delivery of innovative networking features to our customers at a previously impossible pace. Coupled with our global TE system, Espresso delivers 13% more user traffic on our infrastructure by integrating with global application-aware TE system as compared to just BGP-based routing, while also improving peer link utilization and end-to-end user experience. For example, the mean time between rebuffers (an important measure for video traffic) improves between 35% to 170%.

2 BACKGROUND AND REQUIREMENTS

Google runs two different WANs. B4 [21], our datacenter-to-datacenter WAN supports global computation. B2 [5] provides connectivity from our datacenters to our peering edge and eventually to end users around the world (Figure 1). Google has one of the largest peering surfaces in the world, exchanging data with Internet Service Providers (ISPs) in over 70 metros.

B2 employs traditional vendor gear and decentralized routing protocols to provide the highest levels of availability. Traditional IP routing operates on low-level information for routing traffic, e.g., BGP announcements and policies. Supporting application-aware fine-grained traffic policies therefore requires complex BGP rules that are hard to manage, reason about [13] or even implement.

In contrast, we built B4 with internally-developed hardware, SDN control and centralized traffic engineering, leveraging the fact that much of our datacenter to datacenter traffic did not require the same availability as B2. As we gained more experience with B4, centralized traffic engineering and SDN, we were able to continuously improve the availability of our SDN WAN while enjoying the benefits of cost efficiency, fine-grained traffic management, and high feature velocity. These capabilities formed the motivation for Espresso: could we bring the benefits of SDN to a portion of B2 while maintaining the requisite availability and interoperability

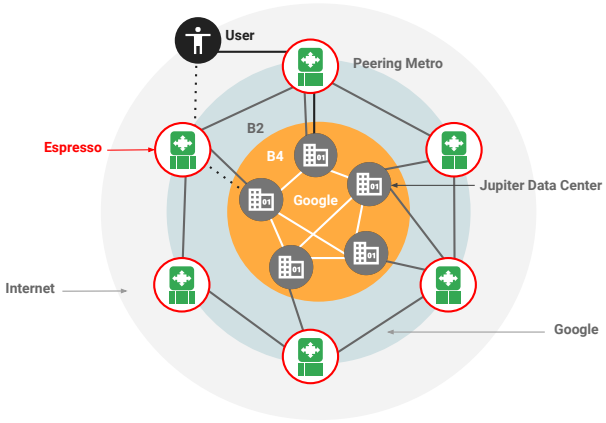


Figure 1: Overview of Google WANs. Espresso runs in Peering Edge Metros that connect to external peers and back to Data Centers via B2. B4 is our SDN WAN that supports high volume traffic between Data Centers.

with arbitrary external hardware and software? Doing so would require us to meet the following requirements:

(1) **Efficiency.** Sustaining Google's rapid Internet traffic growth requires us to reduce the cost of our Internet peering edge network at a faster rate, while simultaneously increasing utilization of the peering ports.
(2) **Interoperability.** Espresso needs to interoperate with the rest of the Internet and the peer networks, supporting all standard Internet protocols, such as BGP and IPv4/IPv6. Previous SDN deployments [11, 21] were internal and only had to interoperate with a controlled number of vendors and/or software.
(3) **Reliability.** Espresso must carry all Google traffic to and from users. Consequently, it must deliver better than 99.999% global availability, or less than five minutes of downtime per year for traffic that is routed through it. Measuring global edge availability presents its own challenges and is beyond the scope of this paper; here, we focus on our techniques for maintaining availability while centralizing portions of our network control.
(4) **Incremental Deployment.** For practical reasons, Espresso must operate alongside existing traditional routing equipment. Given the scale of Google's network, it would be infeasible to forklift current deployments to support Espresso. On the other hand, restricting Espresso to new edge turn ups would limit its utility to Google.
(5) **High Feature Velocity.** Changing product requirements, for example to support peering in heterogeneous Cloud deployments, places a premium on rapid feature development and qualification. Our goal is to deploy a developed feature to production in two weeks, a process that could take up to a year with existing practice. Our experience suggests that end to end feature development and deployment has improved by more than a factor of six with Espresso.

Centrally, a highly available peering edge and high feature velocity are often at odds with each other. Having a highly available system often implies a more slowly-changing system because management operations are often the cause of unavailability [16]. Trying to improve availability beyond the baseline for traditional deployments, while increasing feature velocity, entails substantial architectural care.

3 DESIGN PRINCIPLES

To meet our requirements, we employed several design principles crucial to the success of Espresso. As shown in Table 1, Espresso changes the traditional edge routing design by innovating on all three planes: control, data and management.

(1) Espresso employs a **hierarchical control plane** split between local and global controllers. Local controllers apply programming rules and application-specific traffic routing policies that are computed by the global controller. This design is easier to reason about as compared to fully distributed local controllers in metros peering with one another. This approach achieves three main objectives: (i) global traffic optimization to improve efficiency, (ii) improved reliability as the local control plane can operate independently of the global controller, and (iii) fast reaction to local network events, for example on peering port or device failure the local controller performs local repair while awaiting globally optimized allocation from the global controller.
(2) We support **fail static** for high availability [16]. The data plane maintains the last known good state so that the control plane may be unavailable for short periods without impacting packet forwarding. While we are exposed to simultaneous data plane failures while the control plane is unavailable, such failures are typically small in scope, e.g., individual peering link failure. Achieving fail static properties with existing network hardware is challenging because of tight coupling between the control and data plane. For example, it is typically impossible to distinguish between a BGP stack failure and a data plane failure. By externalizing control off peering devices, Espresso is systematically engineered to fail static. Different components in the control plane can be unavailable for varying amounts of time while the data plane and BGP peerings can continue to operate. This design also allows us to upgrade the control plane frequently on a live system without impacting the data plane, which fortuitously provides ample opportunity to test fail static system properties.
(3) Espresso emphasizes **software programmability** via simple hardware primitives (e.g., MPLS pop and forward to next-hop). As such, new features can be introduced without waiting for the vendor to qualify and release a software update. This in turn allows the network to evolve with changing application requirements and also enables innovative networking features to be deployed with high velocity. Separating the control plane from the data plane has the added benefit of allowing the CPU for control protocols to scale independently of hardware packet forwarding capabilities. With commercial Peering Routers, the ratio of control CPU to data plane performance is fixed.
(4) High feature velocity resulting from programmability imposes **testability** as a key design principle for Espresso. Only by rigorously and systematically testing each feature through fully automated end-to-end testing can we achieve feature velocity without sacrificing reliability and availability. Because we implement network functionality in software components, we can perform layers of testing from (i) unit tests to (ii) components to (iii) pairwise interaction to (iv) end-to-end systems and subsystems. This is in sharp contrast to qualification of routers where we can only rely on black box testing coupled to expensive and hard to manage hardware deployments. The layers of testing provide confidence in the safety of new software releases, allowing us to release frequently while staying within our reliability budget.
(5) Supporting exponential growth means that Espresso must be designed for intent-driven **manageability** with controlled configuration updates. This manageability needs to support large scale operation that is safe, automated, and incremental. Such automation is key to sub-linear scaling of the human overhead of running the network and reducing operational errors, the main source of outages [16].

4 DESIGN

In this section, we describe the design of Espresso and how it integrates into the existing Google network. We then detail the design of each component in Espresso and how they adhere to our design principles.

4.1 Background

Peering locations—a.k.a. edge metros—connect Google to end users around the world (Figure 1) through external ISP peers. Figure 3a illustrates this configuration. Prior to Espresso, we used traditional routers, Peering Routers (PR), to connect Google's network with other autonomous systems (AS) in the edge. These PRs support eBGP peerings, Internet-scale FIBs and IP access-control-lists to filter unwanted traffic.

Alongside our routers, we also run Layer 7 reverse proxies at the edge to terminate user connections and to serve cached content. The proxies reduce connection latency to users, reduce the required capacity back to data centers through caching, and improve performance for cacheable content [15, 23]. A typical user request enters Google via one of the PRs and terminates at a local L7 (reverse) proxy. Key to Espresso is using a small subset of this server processing power already running at the edge for programmable packet processing.

4.2 Design Overview

Figure 2 illustrates the basic architecture of an Espresso edge metro, which broadly consists of three subsystems.

(1) A global traffic engineering (TE) system enables application-aware routing at Internet scale. This TE system, consisting of the Global TE controller (GC) and location controllers (LC), programs the packet processors with flow entries that allows dynamic selection of egress port on per-application basis.

Table 1: Espresso: Requirements with the corresponding design principles that help achieve them.

Requirement	Design Principle(s)	Summary
Efficiency	Software Programmability	Centralized TE, application-aware optimization and routing (§4.3)
Interoperability	Software Programmability	eBGP for peering (§4.4.1), support IPv4/IPv6 (§4.2.1)
Reliability	Hierarchical Control Plane, Fail Static, Manageability	Split between local/global control-plane (§4.3), intent-driven configuration system (§4.5)
Security	Software Programmability	Fine-grained DoS & Internet ACL on host packet processors (§4.4.4)
Incremental Deployment	Software Programmability	TE system supports legacy and Espresso peering devices (§4.3)
High Feature Velocity	Software Programmability, Testability	Loosely coupled control-plane, automated testing and release processes (§5)

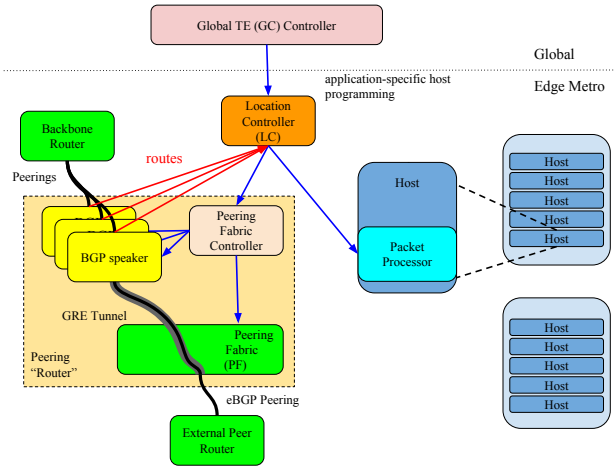


Figure 2: Espresso Control Plane: LC programs Internet-sized FIBs in the packet processors on the edge hosts, and distributes configuration to the Peering Fabric Controllers (PFC). PFC in turns manages PF and BGP speakers. BGP speakers establish eBGP peering with other AS. The local control plane tasks run on the machines hosted in the edge metro.

This programmable packet processing also helps mitigate DoS attacks with finer resolution than possible in hardware routers.

(2) A combination of a commodity MPLS switch (PF) that supports forwarding/tunneling rules and ACLs, and BGP speaker processes that establish BGP peering supports the traditional peering "router" capabilities. Unlike an Internet-scale peering router, the PF has a small TCAM and limited on-box BGP capability. However, it is capable of line rate decapsulation and forwarding of IP GRE and MPLS packets.

(3) Finally, Espresso supports fully automated configuration and upgrades through an intent-driven configuration and management stack. To configure the system, an operator simply changes the intent. Committing the intent triggers the management system to generate, version, and statically verify the configuration before pushing it to all relevant software components and devices. For additional protection, we also canary the configuration with another layer of verification performed by the individual components.

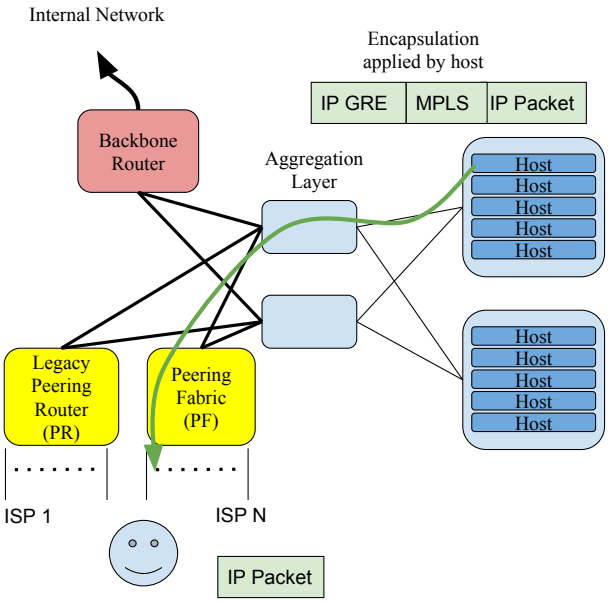
4.2.1 Application-aware routing. A typical user request enters via a peering device and terminates at an edge host (Figure 3b) via standard IP routing. With traditional routing, the response would simply be sent from this edge host to the peering router (PR), which in turn maps the destination IP address to one of its output ports by consulting an Internet-scale FIB constructed by its BGP stack.

Espresso PFs do not run BGP locally and do not have the capacity to store an Internet-scale FIB. We instead store Internet-scale FIBs in the servers, using cheaper DRAM on servers for better scaling with the growth of Internet prefixes. Espresso directs ingress packet to the host using IP GRE where we apply ACLs, see Figure 3b. The hosts also encapsulate *all outbound* packets with a mapping to the PF's output port. That is, we encode the egress port in the packet itself through server packet processors, enabling tremendous hardware and software simplification at the PF.

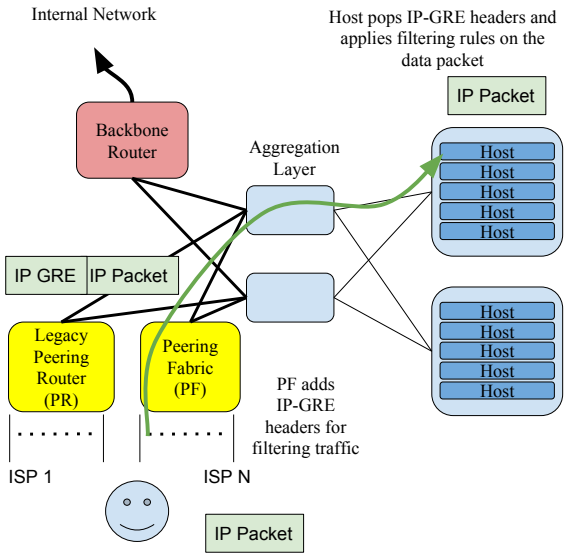
Thus, it is each server's packet processor that maps the ultimate packet destination to a pre-programmed label at the PF using an Internet-scale FIB stored in server memory. Espresso uses IP-GRE and MPLS encapsulation, where IP-GRE targets the correct router and the MPLS label identifies the PF's peering port (Figure 3a). The PF simply decapsulates the IP-GRE header and forwards the packet to the correct next-hop according to its MPLS label after popping the label. An L3 aggregation network using standard BGP forwards the IP-GRE encapsulated packet from the host to the PF. IP-GRE was chosen over destination MAC rewrites, e.g., [4], because it allows us to scale the aggregation network easily.

To program host FIBs, GC gathers routes from all peering devices, both traditional and Espresso peering. It calculates the application-aware FIB, while respecting BGP policy. GC then sends the resulting Internet-sized FIB to the LCs. Because GC has a slower control loop, LC maintains responsibility for quickly reacting to any metro-local network events as shown in Figure 5. Using these rules, the packet processors can implement application-aware routing.

4.2.2 BGP peering and routes propagation. Espresso externalizes eBGP from the peering device (PF) to software processes running on the host servers. To establish a BGP session with an external peer, LC creates an IP-GRE tunnel between the PF and the server running BGP, Figure 2. We partition the responsibility of handling peer BGP sessions among servers at the granularity of each peer, simplifying the process of scaling peering sessions. In addition, this approach allows Espresso to establish a TCP connection directly between the peer router and the eBGP engine without requiring multi-hop peering, which many peers do not support.



(a) Diagram shows path for traffic from the reverse L7 web proxy is directed towards a selected peering port on the PF using IP GRE and MPLS encapsulation.



(b) Diagram shows path for traffic from user to the reverse L7 web proxy is directed with IP GRE encapsulation.

Figure 3: Espresso Metro with support for both legacy peering router (PR) and Espresso peering fabric (PF).

Once the peering is established, Espresso exchanges routes with peers as with any router. LC aggregates these routes across BGP speakers and delivers them to the GC. In turn, GC learns of available peering ports on a per-prefix basis, running a global TE optimization

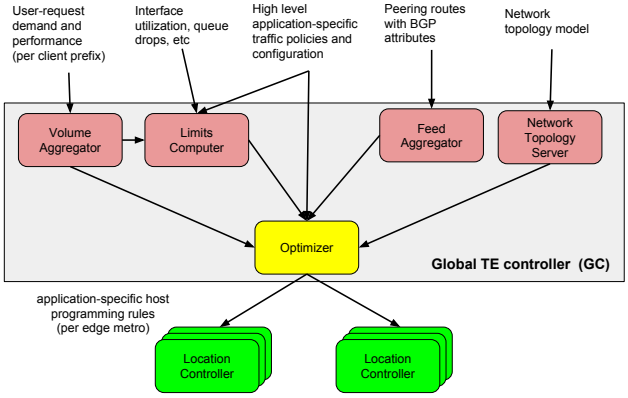


Figure 4: Global Controller system is a distributed system that periodically produces an optimized application-specific programming rules that are distributed via LCs in each edge metro.

to rebalance traffic across available peering ports to serve user traffic. LC also quickly propagate any *failures* (e.g., route withdrawals) to the hosts. Figure 5 shows an example of this operation.

4.3 Application-aware TE System

Espresso's TE system is a hierarhical control plane, divided into a global TE controller (GC) and per-metro location controllers (LC). GC provides application-aware TE decisions through global optimization, while LC provides local fallback and fast reaction to failures to increase Espresso's reliability and responsiveness. Integration with the existing TE system allows for incremental deployment of Espresso, as GC supports both traditional and Espresso peering devices. At a high level, GC's objective is to efficiently allocate user traffic demand given the available resources (peering, backbone, server capacity) while optimizing for application-level metrics such as goodput and latency. GC also strictly observes application-level priority to ensure we allocate bandwidth for higher priority applications before others.

4.3.1 Global Controller. Figure 4 shows an overview of GC's operation. The output of GC is a prioritized list of <PR or PF, egress port> tuples for each <point-of-presence (PoP), client prefix, service class> tuple, where service class encodes the priority of the traffic on the network. We refer to this list as the *egress map*. Packet processors employ this list to control where an application's traffic egresses Google's network. GC only chooses the egress to optimize peering port utilization and does not control pathing between the hosts and the PFs/PRs. Dynamic pathing between hosts and PFs/PRs is beyond the scope of Espresso.

To make application-aware TE, GC's optimizer consumes a number of inputs:

- **Peering routes:** GC determines where user traffic can egress by collecting all peering routes from edge routers. The *Feed Aggregator* collects routes while preserving the BGP attributes to respect BGP peering policies. GC consumes routes from both traditional PRs and the Espresso PFs. GC can then compute egress maps targeting PR or PF, which allows for incremental deployment of new peering capacity on Espresso. Using these routes, GC creates a prioritized list of egresses

for each client prefix, based on attributes such as AS path length, BGP advertisement specificity, operator policies, etc.

- **User bandwidth usage and performance:** To estimate user demand, the layer 7 proxies report connection metrics for each client-prefix and application service class to GC based on the connections they manage. The *Volume Aggregator* aggregates bandwidth, goodput, RTT, retransmits, and queries-per-second reports. A smoothed version of this information serves as an estimate of user demand and path performance in the Optimizer.
  GC determines the appropriate prefix granularity to use for programming client prefixes by joining routing data with observed client latency. The L7 proxies report observed client latency on a per /24 granularity (/48 for IPv6) to GC, and if GC observes different latency characteristics for prefixes within that reported in routing data, it can disaggregate them until the latency characteristics are similar. This enables GC to target client populations with very different network connectivity at the appropriate granularity.
- **Link utilization:** GC collects per-queue link utilization, drops, and link speed from network devices in the peering locations. The *Limits Computer* shown in Figure 4 aggregates this information with user demand to allocate bandwidth targets per link per service class. First, it prioritizes allocation in the order of service classes. Second, it scales the allocation down more aggressively if there are drops in higher priority service classes than in lower priority service classes. This dynamic scaling is critical to maintaining high peering link utilization, while limiting any adverse effect to applications that are latency sensitive, see § 6.2. GC's congestion reaction helps get 17% higher utilization of peering links [2, 12, 21].
  GC also reacts to downstream congestion in the Internet by using host-reported client goodput to compare end-to-end path performance for each client prefix for each egress link. To compare end-to-end path quality, we group host reports based on peering link, AS path and client prefix. The resulting groups have identical BGP and latency characteristics. We use this information to limit the amount of traffic we place on a congested path for a given client prefix, moving traffic to alternate, less congested paths if possible. We show in § 6.2 that enabling this feature significantly improves user-perceived performance, with up to a 170% improvement in user-perceived quality metrics.

GC uses a greedy algorithm to assign traffic to candidate egress device and port using the above inputs. We prefer this greedy algorithm over a more optimized linear program (LP) due to its optimization speed, simplicity and resulting debuggability. We have observed marginal $3-5\%$ improvements in latency to some client prefixes from the LP which is insufficient to justify the additional complexity. We are still investigating in this area, and consider designing an LP-based solution that meets our operational requirements as part of future research.

The greedy algorithm starts with the highest priority traffic and its most preferred candidate mapping. Traffic assignment spills over to subsequent candidates when the remaining traffic exceeds the remaining capacity for the candidate. The algorithm orders egress options based on BGP policy and metrics, user metrics (e.g., RTT and goodput), and cost of serving on the link. For example, if we assign prefix p with expected demand 1 Gbps at location A, to be served via egress peering port E, then GC subtracts 1 Gbps from the available serving limit for peering port E, along with any capacity on the links from A to E. If the complete demand for p cannot be satisfied by serving via port E, then we split the demand between multiple peering ports.

GC also adjusts link capacity to account for traffic not under its control, allowing for incremental deployment. For high priority traffic, GC reserves space for LC and BGP spilling over in case of a link failure. This reserved space is still available for loss-tolerant, lower-QoS traffic.

GC employs *safety* as its fundamental operating principle. This approach has been critical to its success in managing Google's Internet facing traffic while maintaining global scope. The often hidden cost for global optimization is a global failure domain. Every component in GC performs extensive validation at each stage for defense in depth. GC relies on inputs from a number of different systems, like monitoring systems that collect interface counters on peering devices. Given the sheer amount of data consumed by GC and the number of different input sources, GC must verify the basic sanity of all incoming data sources. For example, Limits Computer fails validation if the limits for different links computed change significantly. When validation fails, the Optimizer continues producing maps based on changes to the other input sources, while using the last valid input from Limits Computer for some time. Before publishing the egress map, the optimizer compares it to prior maps, validating traffic-shift rules. We assume maps violating these rules result from some fault in GC, hence we would rather fail-static in these cases. This means we continue to use the last valid map to serve traffic. In our experience, being conservative in validation that could result in serving traffic using stale maps when there is a false positive than risk programming a bad map has resulted in a more reliable operational behavior.

We replicate GC in several datacenters for higher availability. A master election process using a distributed lock system [6], elects one of the GC stacks as master. The Optimizer in the master GC stack sends it output to the LCs in every Espresso Metro. There is a canary process where the LCs canary the new map to a small fraction of packet processors and report their status to the GC. A canary failure would revert the programming to previous valid map and alert an oncall engineer.

We archive both GC inputs and outputs so operators can easily revert to the historical known-good state. Validation reduces the scope and frequency of production outages, while reverting to a known good state helps reduce the duration of outages.

4.3.2 Location Controller. Since we intentionally dampen GC output to minimize churn in the global egress map, Espresso features a local control plane to handle changes such as interface failure and sudden decrease in serving capacity. For this, we run a Location Controller (LC) on host servers in each peering location, which acts as a fallback in case of GC failure.

The main inputs to LC are application-specific forwarding rules from GC, and real-time peer routing feeds via eBGP speakers in the metro. The main output is the application-specific forwarding rules

to the packet processors. Figure 5 shows an example, described below. A single LC scales to the control of approximately 1000 servers, keeping with our general goal of limiting control overhead to approximately 0.1% of our infrastructure costs. We also push the ACL filtering rules via LC to packet processors. Finally as a configuration cache, LC acts as the nexus point for configuration to other control components in the metro, e.g., BGP configuration for the eBGP speakers.

- **Scaling host programming.** To quickly program all packet processors, especially in the event of a network failure like peering port down, we run a number of LC instances in the metro. These instances are designed to be eventually consistent. LC programs all packet processors in an edge metro identically. Hence, to support more packet processors, we can simply add more LC instances.
- **Canary.** To limit the blast radius of an erroneous GC output, LC implements *canarying* to a subset of packet processors in a location. We first deliver a new egress map to a subset of packet processors, proceeding to wider distribution after verifying correct operation in the canary set. We also canary other control operations, e.g., ACL programming.
- **Local Routing Fallback.** Typically 3% of the traffic in a metro does not match any entry in the egress map, e.g., the client prefix has not issued requests recently. To provide a safety net for such traffic, LC calculates BGP best-path using routes from all metro routers. LC programs these routes as the default in packet processors. This approach also allows for a metro to fail static in the event of a failure in GC programming. We maintain a constant trickle of traffic on these paths to provide confidence that this fallback mechanism will work when needed.
- **Fast recovery from failures.** To compensate for the slower responding GC, LC reacts quickly to shift user traffic away from known failed links. LC uses internal priority queues to prioritize *bad* news, e.g., route withdrawals, over GC programming. Coupled with the default routes, we avoid blackholing traffic. A quick response to failure via LC and a slower response to new peers via GC also provides the correct response to a peering flap.

We now discuss the programming example shown in Figure 5. GC generates egress maps for two application service classes to egress from peering ports $A:30$ and $B:10$, where A and B are two peering devices, and the numbers refer to the particular port on the peering device. GC can program traffic for a particular prefix to be split in a weighted manner across multiple peering ports, e.g., for application class 1 and prefix 2.0.0.0/24, traffic is split 30% out of $A:30$ and 70% out of $B:10$. LC sends the appropriate encapsulation rules used by packet processors to send traffic to the target peering port. In the example, to egress from $A:30$, packets must be encapsulated with an outer IP-GRE header with an IP address from the range 42.42.42.0/27, and an MPLS header with label 401. We use a range of addresses for the encap IP header for sufficient hashing entropy as switches in the aggregation layer cannot look beyond the outer IP header for hashing.

LC also receives a real-time feed of the peering routes for each of the peers via BGP speakers. It uses this information to verify

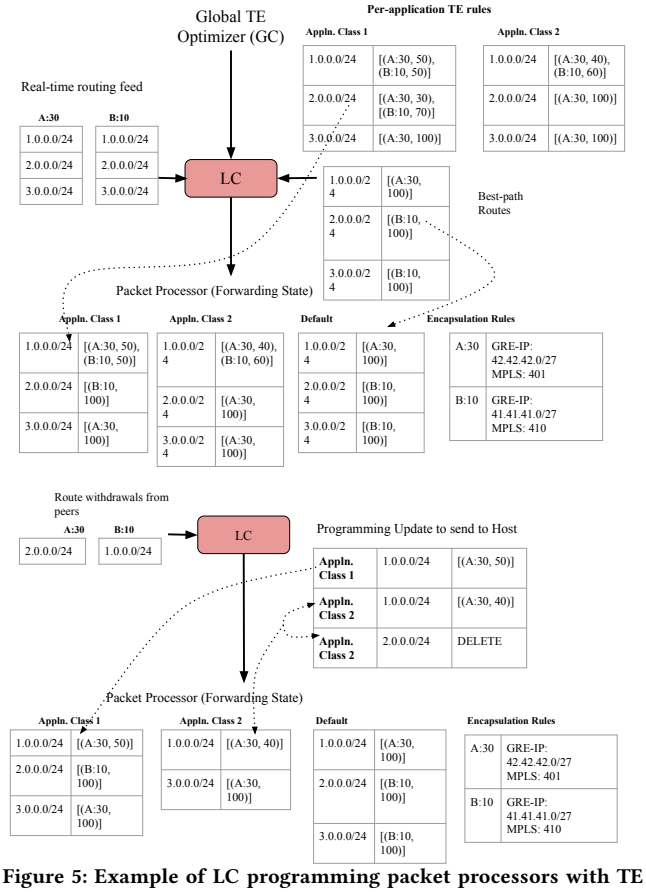


Figure 5: Example of LC programming packet processors with TE assigments from GC, best-path routes and updating the programming based on real-time feed of routing updates (§ 4.3.2). [(A:30, 50), (B:10,50)] refers to a 50:50 split of traffic between peering device A port 30 and peering device B port 10 respectively.

the correct output of GC programming. As shown in Figure 5, when LC receives a route withdrawal for 1.0.0.0/24 for peer $A:30$, it updates the corresponding forwarding rules in each of the application service classes by removing the unavailable egress from the map. In the event that no egress is available in the egress map, LC deletes the programming and the packet processors would instead perform a lookup in the *default* service class programmed from the best-path routes to encapsulate traffic. We keep the LC's response simple so that it only reacts to potential failures rather than make any active traffic steering decisions.

4.4 Peering Fabric

We now discuss how the Peering Fabric (PF) supports the Espresso edge metro. The Peering Fabric (PF) consists of a commodity MPLS switch, our custom BGP stack, Raven, and the Peering Fabric Controller (PFC). The PFC is responsible for programming the switch and managing the eBGP sessions.

4.4.1 Raven BGP Speaker. We needed a server-based BGP speaker for Espresso. We have previously employed Quagga [21] but faced limitations. Specifically, Quagga is single threaded, C-based and

has limited testing support. Hence, we developed *Raven*, a high-performance BGP speaker that runs on the edge servers. Raven allows us to use the large number of CPU cores and ample memory on commodity servers to provide performance. Since Raven is deployed in-house, we can implement only the features we need, which improves performance and results in a leaner, more stable codebase. We also benefit from better unit testing available in our internal codebase. §6.3 presents our comparison between the BGP speaker implementations.

For Raven to peer with external peers, the BGP packets from the peer are encapsulated at the PF and delivered to Raven process running on host machines. Conversely, the packets from Raven are encapsulated and forwarded in the same manner as regular user traffic from the hosts, i.e., IP-GRE and MPLS encapsulation. This ties the control plane (i.e., health of peering session), and data plane (i.e., traffic targeted towards the peer) together. If the data plane from the peer is broken for any reason, the peering session also breaks—resulting in traffic being shifted away from the peering link. We also exploit the packet processors on the host to perform both encapsulation and decapsulation of the BGP packets.

We run multiple Raven instances for each PF, distributing the peers evenly across each instance. Hence, we scale our peering capabilities horizontally by simply deploying more Raven tasks, independent of the hardware and scale of the PF. This partitioning also inherently limits the failure-domain of a single Raven task. Unlike in a traditional router where all peering units can fail together, failure of a Raven task only affects sessions associated with that task.

Since we deploy Raven intances as software services on hosts, they are frequently restarted for upgrades, either of Raven itself or of the host machine. Hence, restarting Raven needs to have minimal impact on traffic. We worked with our peers to enable BGP Graceful Restart [26] such that data traffic continues to flow during Raven restart.

4.4.2 Commodity MPLS Switch. Espresso's commodity MPLS switches support priority-based line-rate packet forwardingbut cannot hold a full Internet-sized routing or ACL table. In Espresso, we program this switch with MPLS forwarding rules. The number of required MPLS forwarding rules corresponds to the number of peerings per router, orders of magnitude smaller than than Internet-size FIB. To program the switch, we extended its OpenFlow agent to program MPLS forwarding and IP-GRE encapsulation/decapsulation rules into the device. Using such rules, we can direct egress packets to the selected peer and encapsulate ingress BGP packets towards the Raven BGP speakers.

4.4.3 Peering Fabric Controller. The PFC acts as the brain of the PF, managing the BGP speakers and switches. PFC manages forwarding on the PF by installing flow rules to decapsulate egress packets and to forward them to the next-hop based on the MPLS label. PFC also installs flow rules to encapsulate BGP packets received from a peer to the server running the correct BGP speaker for that peering session.

PFC maps peers to the available Raven BGP speakers. This map is sticky to minimize BGP session restarts. To gauge individual speaker availability, PFC performs health checks on the speakers and reassigns peering sessions when they go down. Since we loosely couple PFC and the Raven speakers, the peerings remain available even when the PFC is unavailable.

The PFC employs a master-standby architecture to increases Espresso PF reliability in the face of failures. An interesting consequence of externalizing network functionality from the peering device is that the software components are naturally distributed across multiple racks in different power domains, allowing Espresso to be robust against power failure without additional provisioning.

4.4.4 Internet ACL. Google's Internet-facing ACL is large, beyond the capacity of commodity PF switches. To address this challenge, Espresso installs a subset of ACL entries on the PF and uses encapsulation to redirect the remaining traffic to nearby hosts to perform fine-grained filtering. We use traffic analysis to determine what ACL entries are the highest volume in packets per second, and install those on the the PF itself while installing the remainder on the hosts. What we have found is that installing just 5% of the required ACL entries on the PF covers over 99% of the traffic volume.

Even more important than leveraging commodity hardware, Espresso's programmable packet processors can perform more advanced filtering than what is available in any router hardware, commodity or otherwise. This greater flexibility can, for example, support Google's DoS protection system. The Espresso ACL design also allows for rules to be shifted around and new rules to be installed on demand providing increased operational flexibility.

4.5 Configuration and Management

Espresso configuration is based on an automated intent-based system. Currently, we leverage existing configuration languages to express intent in Espresso to minimize churn for operators. Designing a network configuration language for SDN systems is an important area of future work for us.

Upon checking in a human-readable intent for Espresso, the management system compiles it into lower-level configuration data that is more easily consumed by the systems. This configuration data is then delivered verbatim by many of the components in the system. In this way, we verify overall system configuration consistency, and also programmatically ensure the configuration is internally consistent and valid. Changing a configuration schema only requires modifying the consumer(s) of the configuration and not in the configuration management system.

The Espresso configuration management differs from traditional network configuration systems in a number of ways. First, as most of the functionality is in software we can use a declarative configuration language, greatly simplifying higher level configuration and workflow automation. Second, since the data plane (packet processors) is composed of a number of smaller units, we can roll out configuration changes gradually to monitor any impact from bad configuration updates. Finally, the fact that the control plane is composed of a number of discrete units, each performing a specific function allows for strong and repeated validation of configuration flow to provide defense in depth.

In a peering location, the LC provides a cache of the current intent/configuration. Hence, as systems come online or fail over, they always have a local source to load their current configuration, avoiding a high availability SLO for the configuration system. If

LC's connection to the configuration system is lost, the LC will fail-static with the current configuration, alerting operators in parallel to fix the connection. The LC also canaries configuration to a subset of devices within a location and verifies correct behavior before proceeding to wide-scale deployment of a new configuration.

4.5.1 Big Red Buttons. For safety and operational ease, we made the explicit decision to provide a "big red button" (BRB) that operators could quickly, reliably and safely use to turn off some or all parts of the system. These BRBs provide defense in depth at all levels of the system, e.g., (i) we can push a configuration to LC to send all programming via the best-path routes computed locally, (ii) we can move traffic by de-configuring one or more BGP peers, or (iii) we can move all of a peer's traffic by clicking a button on the BGP speaker's monitoring user interface. There are also more fine-grained knobs to disable sending traffic to a particular peering port, or overriding part of GC programming. These BRBs are extensively tested nightly to ensure that they would work when needed. Using the nightly test result, we analyze the performance of these BRB in §6.4.

4.5.2 Network Telemetry. Espresso provides streaming telemetry of data-plane changes and reaction time statistics to events such as peering link failure or route withdrawal. For example, peering link failures are immediately streamed from PFC to the BGP speakers allowing the speaker to withdraw the associated routes quickly, rather than waiting for the BGP session to timeout. LC also uses this signal to update host programming.

Control plane telemetry in Espresso leverages various standard monitoring practices in Google. Every binary in the control plane exports information on a standard HTTP/RPC endpoint, which is collected and aggregated using a system like Prometheus [3].

4.5.3 Dataplane Monitoring via Probing. We continually run end-to-end probes to detect problems. These probes traverse the same path as regular traffic but exercise specific functionality. For example, to verify the proper installation of ACLs at hosts, we send probe packets to the hosts that are encapsulated identically to Internet traffic, allowing us to validate that the systems implementing the ACL forward/drop as expected. We can send probe packets that loop back through the PF and through various links to ensure they reach their destination and are processed correctly.

5 FEATURE AND ROLLOUT VELOCITY

Espresso is designed for high feature velocity, with an explicit goal to move away from occasional releases of software components where many features are bundled together. Espresso software components are loosely coupled to support independent, asynchronous and accelerated releases. This imposes a requirement for full interoperability testing across versions before each release. We achieve this by fully automating the integration testing, canarying and rollout of software components in Espresso.

Before releasing a new version of the Espresso software, we subject it to extensive unit tests, pairwise interoperability tests and end-to-end full system-level tests. We run many of the tests in a production-like QA environment with a full suite of failure, performance, and regression tests that validates the system operation with both the current and new software versions of all control plane components. Once this test suite passes, the system begins an automated incremental global rollout. We also leverage the testing and development infrastructure that is used for supporting all of Google codebase [24].

Table 2: Release velocity of each components (in days) for the past year. Velocity is how frequent a new binary is being rolled out; Qualification is the amount of time it takes for a binary to be qualified for rollout; And Rollout is the time it takes for a new binary to be deployed in all sites.

Component	Velocity	Qualification	Rollout
LC	11.2	0.858 (± 0.850)	5.07 (± 3.08)
BGP speaker	12.6	1.14 (± 1.82)	5.01 (± 2.88)
PFC	15.8	1.75 (± 1.64)	4.16 (± 2.68)

The above features allow Espresso software to be released on a regular weekly or biweekly schedule as shown in Table 2. To fix critical issues, a manual release, testing and rollout can be performed in hours. Since releases are such a common task, we ca quickly and incrementally add features to the system. It is also easy to deprecate an unused feature, which enables us to maintain a cleaner codebase. Rapid evolution of the Espresso codebase is one of the leading contributors to the significantly higher reliability in Espresso compared to our traditional deploymentsc̆iteevolveordie.

Using three years of historical data, we have updated Espresso's entire control plane > 50× more frequently than with traditional peering routers, which would have been impossible without our test infrastructure and a fail-static design that allows upgrades without traffic interruption.

As one example of the benefits, we developed and deployed a new L2 private connectivity solution for our cloud customers in the span of a few months. This included enabling a new VPN overlay, developing appropriate APIs and integrating with the VPN management system, end-to-end testing, and global roll-out of the software. We did so without introducing any new hardware or waiting for vendors to deliver new features. The same work on the traditional routing platforms is still ongoing and has already taken 6× longer.

6 EVALUATION

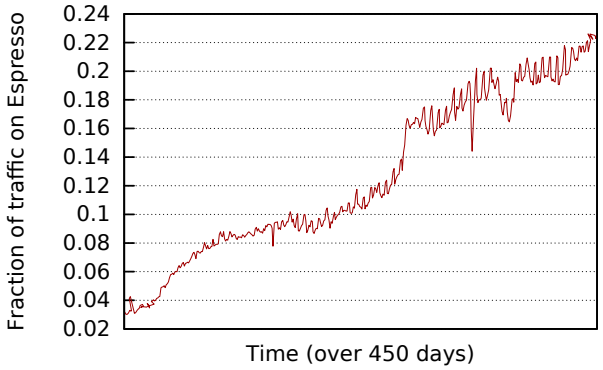
6.1 Growth of Traffic over Time

Espresso is designed to carry all of Google's traffic. We started with lower priority traffic to gain experience with the system, but with time it is now carrying higher priority traffic.

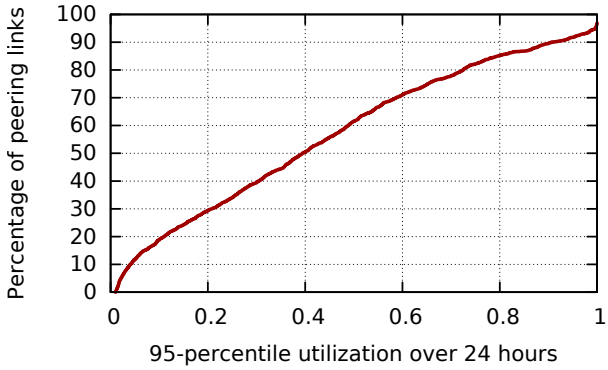
To date, Espresso is carrying more than 22% of Google's outbound Internet traffic, with usage increasing exponentially. Figure 6a shows that an increasing fraction of total traffic is carried on Espresso. E.g., in the last two months, traffic on Espresso grew 2.24× more than the total.

6.2 Application-aware Traffic Engineering

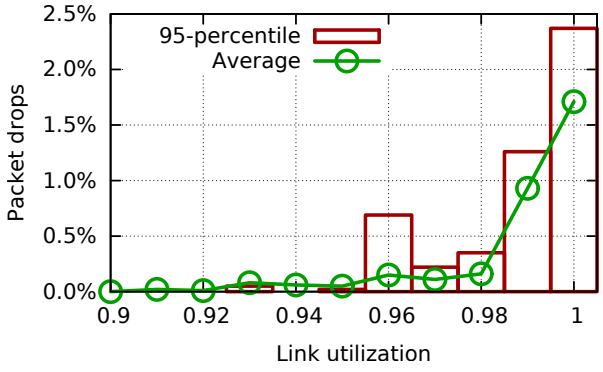
We discuss some benefits of a centralized application-aware TE. The global nature of GC means that it can move traffic from one peering point to another, even in a different metro, when we encounter capacity limits. We classify this as *overflow*, where GC serves the excess user demand from another location. Overflow allows Google to serve, on average, 13% more user traffic during peaks than would



(a) Fraction of total traffic carried on Espresso over time.



(b) CDF of 95-percentile peering link utilization over a day.



(c) Packet drops by link utilization

Figure 6: Total traffic and peering link utilization on Espresso.

otherwise be possible. GC can find capacity for overflow either on a different peering in the same edge metro or could also spill it to a different point-of-presence (PoP), as it has global visibility. We observe that over 70% of this overflow is sent out to a different metro. Figure 7 shows the distribution of overflow by client ISPs, as a fraction of what could be served without TE. The overflow fraction is fairly small for most ISPs, however for few very capacity-constrained ISPs we find the GC overflowing over 50% of the total traffic to those ISPs from non best location.

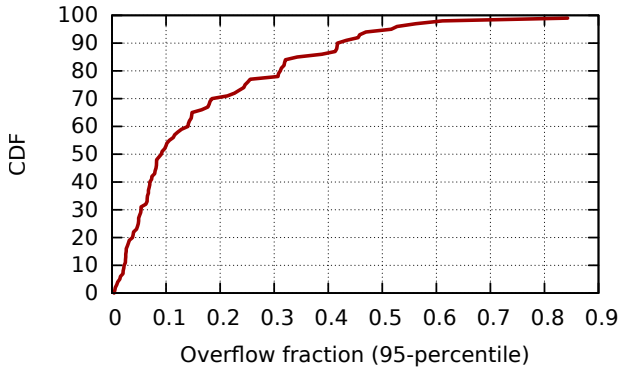


Figure 7: CDF of overflow (over a day) managed by GC for top 100 client ISPs. X axis shows the fraction of total traffic to the ISP that is served from a non best location.

As described in § 4.3.1, GC caps loss-sensitive traffic on peering links to allow for errors in bandwidth estimation. However, GC can safely push links close to 100% utilization by filling any remaining space with lower-QoS loss-tolerant traffic. For this traffic, GC ignores estimated link capacity, instead dynamically discovering the traffic level that produces a target low level of queue drops. Figure 6b shows that over 40% of peering links have a 95^{th} percentile utilization exceeding 50%, with 17% of them exceeding 80% utilization. This is higher than industry norms for link utilization [2, 12, 21]. GC sustains higher peak utilization for a substantial number of peering links without affecting users, easing the problem of sustaining exponential growth of the peering edge.

Figure 6c focuses on observed packet drops for the highly utilized peering links. GC manages packet drops to $< 2\%$ for even peering links with 100% utilization. The drops are in the lower-QoS traffic that is more loss-tolerant, as GC will react aggressively to reduce limit if higher-QoS drops are observed.

GC monitors client connection goodput to detect end-to-end network congestion, including those beyond Google's network. GC can then move traffic to alternate paths with better goodput. This congestion avoidance mechanism dramatically improved our video streaming service's user-experience metrics for several ISPs, which has been shown to be critical to user engagement [10].

Table 3 shows examples of improvements to user experience observed by our video player when we enabled congestion detection and reaction in GC. Mean Time Between Rebuffers (MTBR) is an important user metric taken from the client side of Google's video service. MTBR is heavily influenced by packet loss, typically observed in some ISPs that experience congestion during peak times. For the examples presented here, we did not observe any congestion on the peering links between Google and the ISP. This demonstrates the benefit of a centralized TE system like GC, which considers the least congested *end-to-end* path for a client prefix through a global comparison of all paths to the client across the peering edge.

6.3 Comparison of BGP speakers

Early in the Espresso project, we had to choose between an open-source BGP stack, e.g., Quagga [1], Bird [14], or XORP [19], or extend Raven, an internally-developed BGP speaker. We settled

Table 3: Improvements in Goodput and Mean Time Between Rebuffers (MTBR) observed for video traffic when we enabled GC reaction to end-to-end congestion detection and reaction.

ISP	Change in MTBR	Change in Goodput
A	10 → 20 min	2.25 → 4.5 Mbps
B	4.6 → 12.5 min	2.75 → 4.9 Mbps
C	14 → 19 min	3.2 → 4.2 Mbps

on Quagga as the leading open source candidate and conducted a detailed comparison to Raven. This choice is in-part driven by the fair amount of effort we have spent in optimizing Quagga in B4 [21].

One of the most important metrics was the BGP advertising and withdrawal convergence time, for both IPv4 and IPv6 routes. Raven consistently outperformed Quagga for IPv4, converging 3.5–5.0× faster with 3 million routes (Figure 8a) and performing as well for IPv6 (Figure 8b). Raven also consistently used less memory and CPU than Quagga, e.g., for one million IPv4 routes, Raven used less than half the memory of Quagga. We saw similar savings in IPv6 routes. Raven also has lower latency because it does not write routes into the kernel. Further, Quagga is single-threaded and does not fully exploit the availability of multiple cores on machines.

We also compared Raven with the BGP stack on a widely used commercial router. Based on IPv4 and IPv6 routes drawn from a production router, we created route sets of different sizes. We compared the advertisement and withdrawal performance of Raven (Adj-RIB-in post-policy) and the commercial router over several runs for these route sets. The average convergence latency (Figure 8c) showed that Raven significantly outperformed the router in both dimensions. This performance was partly made possible by the approximately 10x CPU cores and memory available on our servers relative to commercial routers.

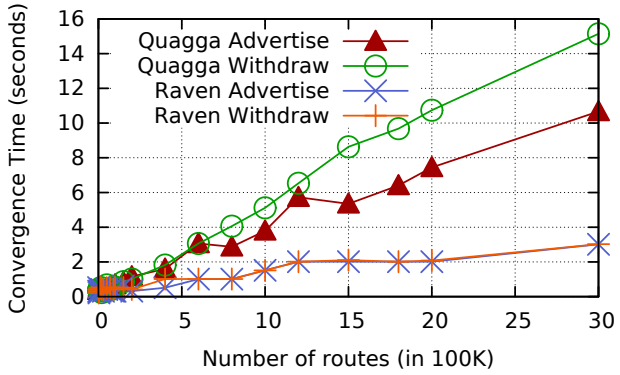
6.4 Big Red Button

In this section, we evaluate the responsiveness of two of our "big red buttons" mechanisms: (i) we can disable a peering by clicking a button on the BGP speaker's user interface for a temporary change, and (ii) permanently drain traffic by de-configuring one or more BGP peers via an intent change and config push.

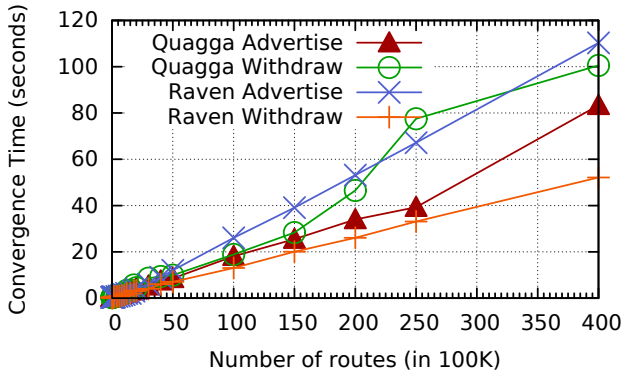
In case (i), we measure the time between clicking the button on the BGP speaker's user interface and the routes for that peer being withdrawn by Raven. This takes an average of 4.12 s, ranging for 1.60 to 20.6 s with std. dev. of 3.65 s. In case (ii), we measure the time from checking in intent to the time routes are withdrawn. This takes an average of 19.9 s, ranging for 15.1 to 108 s with std. dev. of 8.63 s. This time includes checking in an intent change to a global system, performing various validation checks, propagating the drain intent to the particular device and finally withdrawing the appropriate peering sessions.

6.5 Evaluating Host Packet Processing

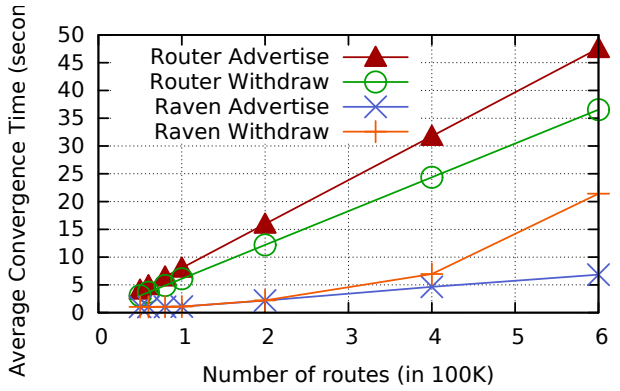
Key to Espresso is host-based packet processing to offload Internet scale routing to the end hosts. This section demonstrates that a well-engineered software stack can be efficient in memory and CPU overhead.



(a) IPv4 Convergence between Quagga and Raven BGP speaker.



(b) IPv6 Convergence between Quagga and Raven BGP speaker.

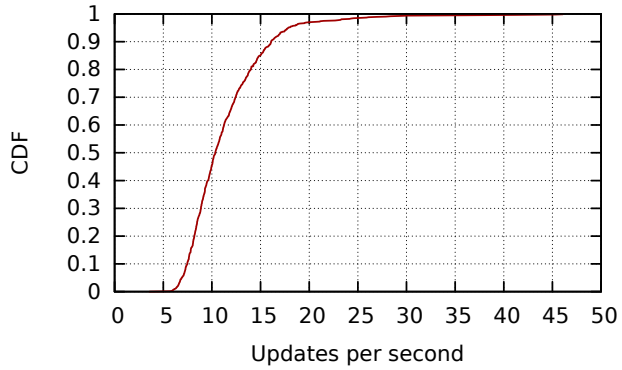


(c) Convergence between Commercial Router and Raven BGP speaker.

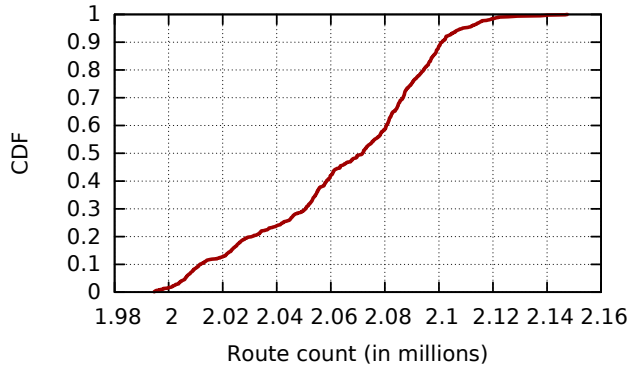
Figure 8: Comparison of convergence times of Raven BGP speaker vs. other BGP implementations.

Figure 9a shows the CDF of the programming update rate from LC to one host; the host received 11.3 updates per second on average (26.6 at 99th percentile). We also measure update processing overhead. Update processing is serialized on a single CPU and takes only 0.001% on average and 0.008% at 99th percentile of its cycles.

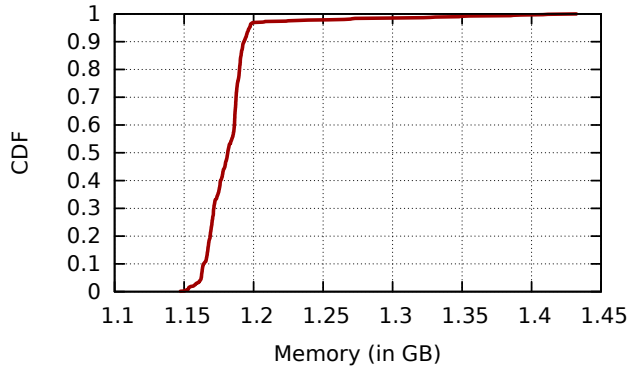
An on-host process translates the programming update into an efficient longest prefix match (LPM) data structure for use in the data path. We share this immutable LPM structure among all of



(a) Routing update arrival rate.



(b) Route count (IPv4 and IPv6).



(c) Memory use of LPM and control plane structures.

Figure 9: Performance of packet processor. Data collected over a 3 month period from a live production host.

the packet processing CPUs, allowing lock-free packet processing. We first install programming updates in a shadow LPM structure, updating all packet processing path pointers to point to the new LPM afterward. We use a compressed multibit trie to implement IPv4 LPM and a binary trie to implement IPv6 LPM. See [31] for a survey of possible LPM implementations. Figure 9b shows the route counts, and Figure 9c the memory usage in an Espresso deployment over a period of 3 months. The LPMs contain approximately 1.9 million IPv4/IPv6 prefixes-service classes tuple. The LPMs and control plane structures used 1.2 GB of RAM on average and 1.3 GB of RAM at 99^{th} percentile. We attribute occasional spikes in memory use to the background threads triggered by our profiling infrastructure.

We also evaluate the CPU overhead of both LPM lookups and packet encapsulation on a production machine at peak load. At peak, the machine transmits 37 Gbps and 3 million packets per second. On average, the LPM lookups consume between 2.1% to 2.3% of machine CPU, an acceptable overhead. For expediency, we used simple binary trie LPM implementation, which can be improved upon.

7 EXPERIENCE

Perhaps the largest meta-lesson we have learned in deploying various incarnations of SDN is that it takes time to realize the benefits of a new architecture and that the real challenges will only be learned through production deployment. One of the main drivers for our emphasis on feature velocity is to support our approach of going to production as quickly as possible with a limited deployment and then iterating on the implementation based on actual experience. In this section, we outline some of our most interesting lessons.

(1) **Bug in a shared reporting library.** All Espresso control-plane components use a shared client library to report their configuration state to a reporting system for management automation. We added this reporting system after the initial Espresso design but neglected to add it to regression testing. A latent bug triggered by a failure in the reporting system caused all the control plane jobs to lock up due to process thread exhaustion. In turn, the control-plane software components across all Pilot peering locations failed. This caused traffic to be automatically drained from the Espresso PF ports, failing the traffic back to other peering locations. Espresso's interoperability with the traditional peering edge allowed routing to work around the Espresso control plane failure and prevented any user-visible impact. This control-plane outage however reinforced the need for *all* control and management plane components to be included in integration testing.

(2) **Localized control plane for fast response.** During the initial deployment of Espresso, we observed an interesting phenomenon: new routing prefixes were not being utilized quickly after peering turnup, leading to under-utilization of available capacity. The root-cause was slow propagation of new prefixes to the GC and subsequent incorporation in the forwarding rules. This is a feature in GC, implemented to reduce churn in the global map computed by GC.
To reduce the time needed to utilize newly learned prefixes without introducing unintended churn in the GC, we augment the local control plane to compute a set of default forwarding rules based strictly on locally collected routing prefixes within an edge metro. Espresso uses this default set for traffic that is not explicitly steered by GC, allowing new prefixes to be utilized quickly. This default also provides an in-metro fallback in the event of a GC failure, increasing

system reliability. This lesson demonstrates how a hierarchical control plane can manage global optimization while maintaining or improving reliability.

(3) **Global drain of PF traffic.** To aid incremental deployment of Espresso and emergency operational procedures, GC can disable/enable use of a set of Espresso PF devices as an egress option for traffic. An operator pushed erroneous GC configuration that excluded all Espresso PF devices from the set of viable egress options. Fortunately, we had sufficient spare capacity available, and GC successfully shifted traffic to other peering devices so that there was minimal user-visible impact. This outage though again demonstrated the perils of a global control system, and the need to have additional safety checks. GC supports a simulation environment to evaluate the effects of a drain, but the simulation results were overlooked, and the configuration was still pushed. To improve availability, we bolstered the simulation environment by making large traffic shifts show up as failures, and introduced additional safety checks in GC to prohibit draining significant traffic from peering devices.

(4) **Ingress traffic blackholing.** Espresso's most user visible outage resulted from our phased development model. Before implementing ACL-based filtering in end hosts, we restricted announcement of prefixes to Espresso peers to the subset that we could support filtering with the limited PF hardware TCAMs. An inadvertent policy change on one of our backbone routers caused external announcement of most of Google prefixes via the PFs. Some of the attracted traffic, e.g., VPN, did not find an "allow" match in the PF (which has limited TCAM) and was blackholed. We subsequently fixed our BGP policies to prevent accidental leaking of routes to Espresso peers, evaluated the limited ACLs on PF to allow additional protocols, and expedited the work to deploy complete Internet ACL filtering in end-hosts. This outage demonstrates the risks associated with introducing new network functionality, and how subtle interactions with existing infrastructure can adversely affect availability.

8 RELATED WORK

Recently, several large-scale SDN deployments [11, 21, 29] have been discussed in the literature. These deployments only had to interoperate with a controlled number of vendors and/or software. SDN peering [17, 18] provides finer-grained policy but does not allow application-specific traffic engineering. This work is also limited to the PoP scale while Espresso targets a globally available peering surface. None of these deployments target high-level availability required for a global scale public peering. A number of efforts [4, 28] target more cost-effective hardware for peering; we show our approach end-to-end with an evaluation of a large-scale deployment.

Espresso employs centralized traffic engineering to select egress based on application requirements and fine-grained communication patterns. While centralized traffic engineering [7, 8, 20–22] is not novel, doing so considering end-user metrics at Internet scale has not been previously demonstrated. We also achieve substantial flexibility by leveraging host-based encapsulation to implement the centralized traffic engineering policy by programming Internet-sized FIBs in packet processors rather than in the network, taking earlier ideas [9, 25] further.

Edge Fabric [27] has similar traffic engineering goals as Espresso. However, their primary objective is to relieve congested peering links in a metro while Espresso aims at fine-grained global optimization of traffic. Consequently, Edge Fabric has independent local and global optimization which does not always yield globally optimal traffic placement, and relies on BGP to steer traffic. On the other hand, Espresso integrates egress selection with our global TE system allowing us to move traffic to a different serving location and to perform fine-grained traffic engineering at the hosts. Espresso also considers downstream congestion which Edge Fabric is exploring as future work. Another key difference lies in Espresso's use of commodity MPLS switches, in contrast with Edge Fabric that relies on peering routers with Internet-scale forwarding tables. We have found that such peering routers comprise a substantial portion of our overall network costs.

Espresso's declarative network management is similar to Robotron [30]. We further build a fully automated configuration pipeline where a change in intent is automatically and safely staged to all devices.

9 CONCLUSIONS

Two principal criticisms of SDN is that it is best suited to walled gardens that do not require interoperability at Internet scale and that SDN mainly targets cost reductions. Through a large-scale deployment of Espresso—a new Internet peering architecture—we shed light on this discussion in two ways. First, we demonstrate that it is possible to incrementally evolve, in place, a traditional peering architecture based on vendor gear and maintain full interoperability with peers, their myriad policies, and varied hardware/protocol deployments at the scale of one of the Internet's largest content provider networks. Second, we show that the value of SDN comes from the capability and its software feature velocity, with any cost savings being secondary.

In particular, Espresso decouples complex routing and packet-processing functions from the routing hardware. A hierarchical control-plane design and close attention to fault containment for loosely-coupled components underlie a system that is highly responsive, highly reliable, and supports global/centralized traffic optimization. After more than a year of incremental rollout, Espresso supports six times the feature velocity, 75% cost-reduction, many novel features and exponential capacity growth relative to traditional architectures. It carries more than 22% of all of Google's Internet traffic, with this fraction rapidly increasing.

ACKNOWLEDGMENT

Many teams has contributed to the success of Espresso and it would be impossible to list everyone that has helped make the project successful. We would like to acknowledge the contributions of G-Scale Network Engineering, Network Edge (NetEdge), Network Infrastructure (NetInfra), Network Software (NetSoft), Platforms Infrastructure Engineering (PIE), Site Reliability Engineering SRE, including, Yuri Bank, Matt Beaumont-Gay, Bernhard Beck, Matthew Blecker, Luca Bigliardi, Kevin Brintnall, Carlo Contavalli, Kevin

Fan, Mario Fanelli, Jeremy Fincher, Wenjian He, Benjamin Helsley, Pierre Imai, Chip Killian, Vinoj Kumar, Max Kuzmin, Perry Lorier, Piotr Marecki, Waqar Mohsin, Michael Rubin, Erik Rubow, Murali Suriar, Srinath Venkatesan, Lorenzo Vicisano, Carmen Villalobos, Jim Wanderer, Zhehua Wu to name a few. We also thank our reviewers, shepherd Kun Tan, Jeff Mogul, Dina Papagiannaki and Anees Shaikh for their amazing feedback that has made the paper better.

REFERENCES

[1] 2010. GNU Quagga Project. www.nongnu.org/quagga/. (2010).
[2] 2013. Best Practices in Core Network Capacity Planning. White Paper. (2013).
[3] 2017. Prometheus - Monitoring system & time series database. https://prometheus.io/. (2017).
[4] Joo Taveira Arajo. 2016. Building and scaling the Fastly network, part 1: Fighting the FIB. https://www.fastly.com/blog/building-and-scaling-fastly-network-part-1-fighting-fib. (2016). [Online; posted on May 11, 2016].
[5] Ajay Kumar Bangla, Alireza Ghaffarkhah, Ben Preskill, Bikash Koley, Christoph Albrecht, Emilie Danna, Joe Jiang, and Xiaoxue Zhao. 2015. Capacity planning for the Google backbone network. In *ISMP 2015 (International Symposium on Mathematical Programming)*.
[6] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 335–350.
[7] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. 2005. Design and Implementation of a Routing Control Platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 15–28. http://dl.acm.org/citation.cfm?id=1251203.1251205
[8] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: Taking Control of the Enterprise. *SIGCOMM Comput. Commun. Rev.* 37, 4 (Aug. 2007), 1–12. https://doi.org/10.1145/1282427.1282382
[9] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. 2012. Fabric: A Retrospective on Evolving SDN. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*. ACM, New York, NY, USA, 85–90. https://doi.org/10.1145/2342441.2342459
[10] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. 2011. Understanding the Impact of Video Quality on User Engagement. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 362–373. https://doi.org/10.1145/2018436.2018478
[11] Sarah Edwards, Xuan Liu, and Niky Riga. 2015. Creating Repeatable Computer Science and Networking Experiments on Shared, Public Testbeds. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 90–99. https://doi.org/10.1145/2723872.2723884
[12] Nick Feamster. 2016. Revealing Utilization at Internet Interconnection Points. *CoRR* abs/1603.03656 (2016). http://arxiv.org/abs/1603.03656
[13] Nick Feamster, Jay Borkenhagen, and Jennifer Rexford. 2003. Guidelines for interdomain traffic engineering. *ACM SIGCOMM Computer Communication Review* 33, 5 (2003), 19–30.
[14] O. Filip. 2013. BIRD internet routing daemon. http://bird.network.cz/. (May 2013).
[15] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. 2013. Reducing Web Latency: the Virtue of Gentle Aggression. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM '13)*. http://conferences.sigcomm.org/sigcomm/2013/papers/sigcomm/p159.pdf
[16] 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 *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 58–72. https://doi.org/10.1145/2934872.2934891
[17] Arpit Gupta, Robert MacDavid, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. 2016. An Industrial-scale Software Defined Internet Exchange Point. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=2930611.2930612
[18] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean Patrick Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. 2014. SDX: A Software Defined Internet Exchange. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 579–580. https://doi.org/10.1145/2740070.2631473
[19] Mark Handley, Orion Hodson, and Eddie Kohler. 2003. XORP: An Open Platform for Network Research. *SIGCOMM Comput. Commun. Rev.* 33, 1 (Jan. 2003), 53–57. https://doi.org/10.1145/774763.774771
[20] 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 *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 15–26.
[21] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM* 43, 4, 3–14.
[22] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. 2015. Practical, real-time centralized control for cdn-based live video delivery. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 311–324.
[23] Abhinav Pathak, Y Angela Wang, Cheng Huang, Albert Greenberg, Y Charlie Hu, Randy Kern, Jin Li, and Keith W Ross. 2010. Measuring and evaluating TCP splitting for cloud services. In *International Conference on Passive and Active Network Measurement*. Springer Berlin Heidelberg, 41–50.
[24] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59, 7 (June 2016), 78–87. https://doi.org/10.1145/2854146
[25] Barath Raghavan, Martín Casado, Teemu Koponen, Sylvia Ratnasamy, Ali Ghodsi, and Scott Shenker. 2012. Software-defined Internet Architecture: Decoupling Architecture from Infrastructure. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets-XI)*. ACM, New York, NY, USA, 43–48. https://doi.org/10.1145/2390231.2390239
[26] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter. 2007. Graceful Restart Mechanism for BGP. RFC 4724 (Proposed Standard). (Jan. 2007). http://www.ietf.org/rfc/rfc4724.txt
[27] Brandon Schlinker, Hyojeong Kim, Timothy Chiu, Ethan Katz-Bassett, Harsha Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. Engineering Egress with Edge Fabric. In *Proceedings of the ACM SIGCOMM 2017 Conference (SIGCOMM '17)*. ACM, New York, NY, USA.
[28] Tom Scholl. 2013. Building A Cheaper Peering Router. NANOG50. (2013). nLayer Communications, Inc.
[29] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2016. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *Commun. ACM* 59, 9 (Aug. 2016), 88–97. https://doi.org/10.1145/2975159
[30] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 426–439. https://doi.org/10.1145/2934872.2934874
[31] David E Taylor. 2005. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)* 37, 3 (2005), 238–275.