Highly Available Bandwidth Guarantees on Highly Utilized Cloud WANs

Muhammed Uluyol[1], Ayush Goel[1], Chi-Yao Hong[2], Harsha Madhyastha[1], Kirill Mendelev[3]*, Dina Papagiannaki[4]*, Sankalp Singh[2], Amin Vahdat[2], Ben Zhang[2], Jonathan Zolla[2]
[1] University of Michigan [2] Google [3] Facebook [4] Microsoft

Abstract

Private wide-area networks (WANs) deployed by large cloud providers enable them to offer predictable bandwidth and latency to their tenants, in contrast to the public Internet. To maximize the WAN's utilization, instead of statically reserving capacity for every tenant, cloud providers dynamically control what traffic is admitted onto the network. But, the availability of promised bandwidth suffers as a result, since a global control plane is fundamentally slow in reacting to changes in traffic demands at scale.

To ensure that efficiency does not come at the expense of predictability, we present HEYP, our new architecture for private WANs. Our high-level approach is that, when any tenant is allowed to send more than its guaranteed bandwidth, its surplus traffic is admitted at a lower quality-of-service (QoS) level. Doing so enables tenants to opportunistically utilize spare capacity without impacting the availability of bandwidth promised to other tenants. However, to fully realize the promise of this approach, we show that one must rethink how traffic is routed across the WAN and account for how congestion control as well as applications will react to QoS changes. Our simulations using traces from a large global, private WAN suggest that HEYP would offer 99% availability for 10× as many bandwidth guarantees as state-of-the-art WAN designs without sacrificing efficiency.

1 Introduction

The public Internet offers no performance guarantees. Therefore, many large cloud providers have deployed their own private wide-area networks (WANs) [16, 38, 40, 43], wherein they provision appropriate network capacity to offer predictable wide-area bandwidth and latency to tenants under a range of failure scenarios and communication patterns. Additionally, via admission control [46] and judicious routing [38, 40], the cloud provider can limit bandwidth interference among tenants.

Since services do not always send traffic at their peak rate, statically configuring a WAN to reserve the bandwidth promised to each tenant and preventing tenants from sending at a higher rate will result in poor network utilization. Cloud providers instead leverage their centralized control of their WANs to dynamically reconfigure routes and admission rate limits in reaction to changes in traffic demands [38, 40, 46]. Based on its global view, a central controller can ensure that any unused capacity that remains after admitting guaranteed demands for bandwidth is shared among tenants' surplus demands as per its business policy. Our simulations using data from BigCloud's large global, private WAN show that such an approach can satisfy 50% more of the traffic demands on average compared to static approaches.

Current WAN architectures for improving network utilization in this manner, however, significantly hamper predictability. For example, in the above-mentioned simulations, dynamically allocating bandwidth offers 99% or higher availability to 10× fewer bandwidth guarantees, as compared to static reservation. A key cause for this dramatically lower predictability is that, to use the bandwidth promised to it, a tenant has to often wait for the central global controller to throttle previously admitted surplus demands of other tenants and reconfigure routes. This is problematic because the speed with which a central controller can react to demand changes is fundamentally limited by two factors: 1) the extremely large scale of global WANs [34, 40, 44, 46, 52], and 2) the need to sequentially apply routing changes in order to prevent inconsistency in routing configurations across switches in the network [38]. These sources of delay will only worsen over time since cloud providers are constantly expanding the number of sites in their WAN [39, 45, 53, 61], and it often takes multiple rounds of reconfigurations for the global controller to correctly estimate and accommodate a tenant's true demand.

To remove this dependence on the global controller for ensuring predictable performance, we argue that any tenant's surplus demands should explicitly be treated differently, and admitted at a lower quality-of-service (QoS) level. When a tenant ramps up its bandwidth usage while staying within its bandwidth guarantee, we can then rely on switches to prioritize its traffic, instead of having to reduce the admission for other tenants utilizing spare capacity. Consequently, global control delays no longer affect the cloud provider's ability to satisfy bandwidth guarantees. With this approach, a tenant bears the risk that its excess traffic is more susceptible to congestion. But, it is, after all, utilizing more bandwidth than was promised to it.

We realize this promise of using QoS downgrade with HEYP (for Highly Efficient, Yet Predictable), our new control plane architecture for private WANs. Our design addresses three challenges that are unique to large cloud provider WANs compared to prior work which has used this approach to offer bandwidth guarantees on the public Internet [22, 23].

First, we show that the use of QoS downgrade calls for a change in how the global controller computes routes, compared to the status quo [38, 40]. A single tenant's demand

* Work done while the author was at Google

is often large enough that capacity from multiple routes must be dedicated to its traffic. Spreading a tenant's high and low priority traffic in the same proportion among all the routes for this tenant constrains which routes can be used to carry low priority traffic, consequently limiting network utilization. Therefore, HEYP installs separate paths for each tenant's promised and surplus bandwidth: stable paths for the former on which capacity is guaranteed irrespective of other tenants' demands, and periodically recomputed paths for the latter to opportunistically utilize unused capacity.

Second, the consequence of using separate paths for high and low priority traffic is that the subset of a tenant's flows which are downgraded cannot be independently determined in each control period. Since latency varies across routes, TCP's congestion control will degrade the performance of any flow which keeps flip-flopping between QoS levels. But, pinning each flow to a specific QoS for a set amount of time limits our ability to respond to demand changes. Instead, we introduce caterpillar hashing, a flow selection mechanism designed to maximize QoS stability. Whenever we need to decrease the fraction of a tenant's traffic that is downgraded, we do so by upgrading the last-downgraded flow, and vice versa to increase the fraction downgraded.

Lastly, in contrast to when every tenant is capped at the bandwidth promised to it, an application may respond to QoS downgrade of its surplus traffic by shifting load towards that subset of its tasks which offer better performance. Since these tasks are more likely to be the ones permitted to send high priority traffic, the net result will be the application sending more high priority traffic than allowed. In response, we can change the QoS assignment, but the application will again react to this change. To converge to a stable QoS assignment for any tenant's traffic, HEYP attempts to identify that subset of the tenant's flows which, if admitted at high priority, cannot ramp up any further due to bottlenecks other than WAN link capacity (e.g., host CPU or NIC).

We evaluate HEYP using testbed experiments and simulations. In our simulations driven by traces obtained from BigCloud, HEYP matches the efficiency obtainable with dynamic bandwidth allocation, and it delivers the availability of bandwidth guarantees afforded by static approaches. We observe similar results when we apply our prototype to an application workload. Tenants that are within their bandwidth guarantees are unaffected by those who have excess traffic, and applications which utilize spare capacity achieve throughput that is within 12% of an optimal approach. Our testbed prototype is open source and will be made available following the publication of this work.

Ethics: This work does not raise any ethical issues.

2 Setting and Motivation

We focus on settings where a WAN administered by a single organization is shared by many tenants. In this setting, we aim to satisfy four goals.

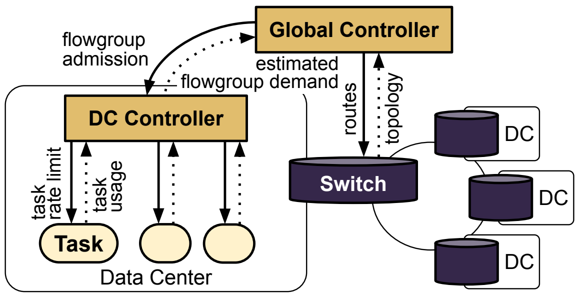


Figure 1: Architecture of a software-defined WAN.

1. Provide predictability by satisfying bandwidth approvals. Based on every tenant's anticipated needs, the provider approves a certain level of bandwidth per tenant between source and destination data centers; we refer to each $(tenant, src, dst)$ tuple as a flowgroup. An *approval* per flowgroup enables more judicious capacity planning compared to guaranteeing every tenant bandwidth in and out of each data center irrespective of its communication pattern [26].

Every approval comes with an associated SLO for the availability of the approved bandwidth, and optionally with guarantees on the length of paths used to route it. A higher availability SLO calls for more redundant bandwidth on the appropriate links to cope with failures; but, in this paper, we consider all approvals as having the same SLO, and we discuss support for multiple SLO levels in §6. We assume approvals are not oversubscribed, so all approvals will be satisfiable as long as the capacity lost due to failures is within the bounds that the network provider wishes to tolerate.

While there exists prior work for making bandwidth approvals resilient to failures [20, 50, 63], we focus on the more commonly occurring risk: rapidly-changing traffic demands.

2. Improve network efficiency by accommodating opportunistic transfers. Our secondary objective is to admit as much of each flowgroup's demand as feasible; a flowgroup's *demand* is the bandwidth it will consume given infinite WAN capacity. The network should typically be able to admit some above-approval demands as it must have spare capacity to tolerate failures, and tenants do not always fully utilize their approvals. Admitting above-approval demands also reduces the risk associated with under-estimation of desired bandwidth. To prevent tenants from becoming dependent on work-conserving bandwidth, they can either be charged for its use [54] or every flowgroup can occasionally be capped at its approval even if there is spare capacity.

3. Support flexible traffic engineering and bandwidth sharing policies. Network policies are rich [46], vary across providers [33, 38, 40, 48, 50], and evolve over time [39]. For example, traffic engineering policies face a tension between optimizing for latency or balanced load, and providers have to choose tradeoffs that are appropriate for their workloads and topology. Rather than dictating particular traffic engineering or bandwidth sharing policies, we aim to be flexible outside of the goals set forth in this section.

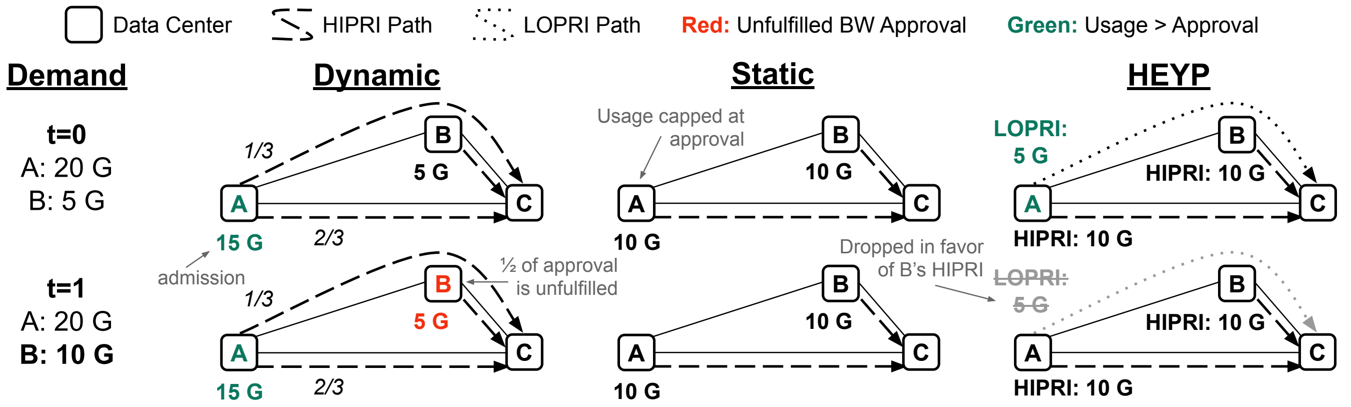


Figure 2: An example where dynamic control prematurely throttles traffic and static allocation fails to fully use network capacity. All links have 10 Gbps capacity. Approvals for both A→C and B→C are 10 Gbps. Initial demands (top) are 20 Gbps for A→C and 5 Gbps for B→C. After limiting it to send at most 15 Gbps, Dynamic sends 33% of A→C's traffic through B and the remaining 67% directly to C. This allows it to fully utilize the network. Later (bottom), when B→C's demand rises to 10 Gbps, Static admits the increased within-approval demand but Dynamic does not. HEYP provides the best of both: it fully utilizes the network with the initial demands and later accommodates the increased demand for B→C.

4. Maintain compatibility with TCP. As it is the most widely-used transport protocol, maintaining compatibility with TCP is necessary to avoid breaking applications. This restriction rules out certain design choices, e.g., because TCP requires most packets to be delivered in order, we cannot spray packets that belong to the same connection over multiple paths which differ in end-to-end latency.

2.1 Dynamic control across and in data centers

To meet these goals, current WANs are architected as shown in Figure 1. Within each data center, a DC controller collects bandwidth *usage* statistics for each flowgroup, aggregating measurements across *tasks*; we consider each instance of an application (i.e., a container or virtual machine) as a set of tasks, where each task sends traffic to a specific DC. From these usage statistics, the controllers *estimate* each flowgroup's demand, e.g., by taking the maximum usage across the past 90 seconds and inflating it by 10% [46]. Using these demand estimates along with the current topology, a global controller periodically adjusts routes to better satisfy demands and determines per-flowgroup *admissions* [46, 55] (i.e., how much aggregate bandwidth to admit for each flowgroup); in practice, separate global controllers may be used for routing and admission control [40, 46]. The DC controllers divide any flowgroup's admission across its tasks and continuously revise this split as demands change. Each task paces its sending rate so as to stay within the programmed rate limit [38, 46, 62].

A key advantage of this approach, compared to distributed approaches such as RSVP-TE [17], comes from the central controller's global visibility. When the network is unable to completely satisfy all demands, the global controller can easily enforce any bandwidth sharing policy desired by the network provider. These include, but are not limited to, allocating bandwidth to tenants in proportion to their payments, max-min fairness, and maximizing throughput.

2.2 Global control delays are a key bottleneck

To understand the sensitivity of dynamic control to delay, consider the example in Figure 2. For simplicity, we ignore failures and route each flowgroup's traffic over the shortest path. If that does not provide enough capacity, we recursively add the next shortest path to the flowgroup's routes.

With Dynamic (left), the global controller first satisfies within-approval demands; it allocates the shortest path for each approval and sets the admissions to 10 Gbps for A→C and 5 Gbps for B→C. It then allocates leftover capacity to surplus demands; it installs a second route for A→C to utilize the spare capacity between B and C, and increases A→C's admission to 15 Gbps.

Although the configuration chosen by Dynamic maximizes demand satisfaction, it risks violating B→C's approval if its demand rises above 5 Gbps. Existing systems which use Dynamic's approach (such as B4 [40, 46] and SWAN [38]), therefore, project demand to be higher than the current usage, e.g., by inflating the usage by 10% [46]. However, this is insufficient to prevent approval violations when demands rise sharply between global reconfigurations (e.g., when B→C's demand increases from 5 to 10 Gbps). Multiple iterations of global reconfiguration are necessary in such cases, since each one only increases the admission by 10%.

One could mitigate the risk of approval violations by improving the responsiveness of Dynamic's global controller, but achieving this is an uphill battle.

- First, there is the issue of scale. The speed with which a global controller can act is fundamentally limited by the scale of a global WAN [46] and the need to sequence routing updates to avoid congestion [38], e.g., routing changes may require tens of seconds to minutes to complete [50]. In addition, the input size to the global controller is rapidly growing. Several large content and cloud providers have added 50–100% more nodes to their WAN over the last 2–6 years [31, 39, 40, 45, 61], and the number of flowgroups grows quadratically in relation to this.

- Second, when the global controller is unavailable, the remaining network components continue to use the last known state [28, 38, 40, 43, 46]. Data from production networks suggests that failures can lead to frequent and long delays. An analysis [30] of over 100 failures in Google's networks attributes 9% of failures to unavailability of the WAN control plane. In 2019, an especially long incident [4] brought down the control plane of Google's backbone network for *over four hours* and caused up to 100% packet loss on certain links.

An alternative approach that eliminates the need for a responsive controller entirely is to use a static configuration that only aims to satisfy approvals. In our example, Static (Figure 2 center) will satisfy the approvals by setting the admissions for both flowgroups to 10 Gbps and configuring each to use only their direct path. Although Static does not take into account either flowgroup's demand at the global level, the DC controller within each data center must dynamically redistribute the admission for each flowgroup across its tasks. The DC controller can react more quickly than the global controller (Appendix C, see also [38, 46]), and is not a significant source of approval violations. As a result, Static ensures that approvals are satisfied regardless of the demand matrix, but it does not admit any above-approval demand.

In §5.1, we quantify the tradeoff between approval and demand satisfaction using the two approaches. The results match the intuition presented here. Static achieves high approval satisfaction and Dynamic provides high demand satisfaction, but each performs poorly in the other metric.

3 Approach and Challenges

To balance the satisfaction of both approvals and demands, the question at hand is: how to retain the benefits of centralized WAN control (i.e., better network utilization and support for flexible bandwidth sharing policies) while addressing its adverse impact on satisfying bandwidth approvals?

Our high level insight is that modifying routes and admissions are not the only measures available in our toolkit for reacting to congestion. In addition, we can leverage support within the network data plane to prioritize the delivery of packets marked with a higher QoS value. This feature can be used to satisfy approvals without involving the global controller, except to handle failures.

A natural approach for using this capability would work as follows. In the common case, the global controller will set the admission for every flowgroup to at least be its approval. Any additional traffic admitted onto the network (to utilize spare capacity) will have its QoS reduced to a lower priority (LOPRI). Any flowgroup's ability to increase its within-approval demand will then not depend on the global controller's ability to react. Instead, network switches will strictly prioritize the delivery of its higher-priority (HIPRI) traffic over any competing above-approval, LOPRI traffic (we discuss other prioritization policies in §6). We would rate limit LOPRI traffic to avoid excessive loss and to ensure that distribution of spare bandwidth is as per business policy.

While this approach shows promise, we need to address three challenges: one on global control and two on control within each data center.

Sharing routes limits efficiency. Each flowgroup's traffic is divided across the routes installed for it in proportion to their weights. When the global controller wants to add an additional path to support, say one-fourth of the above-approval demand, the new path must also admit a quarter of the approval. How should we allocate routes so that this restriction does not limit the efficiency of the network?

QoS churn interacts poorly with congestion control. Each time HEYP migrates a particular TCP flow from HIPRI to LOPRI (or vice versa), it risks changing the RTT for that connection. Such changes, if they occur frequently, will hamper TCP's ability to accurately estimate the bandwidth-delay product, thereby preventing it from fully utilizing available network bandwidth. Therefore, in determining what fraction of a flowgroup's traffic to downgrade, how can the DC controller minimize QoS churn for individual flows?

Uneven bandwidth distribution can lead to harmful app–DC controller interactions. When HEYP downgrades the QoS for part of a flowgroup, the application may, due to congestion, observe worse throughput on its LOPRI flows compared to its HIPRI ones. The application could react by directing more load to its tasks which provide faster responses. As a result, the flowgroup might send more HIPRI traffic than its approval allows and potentially interfere with the approvals of other flowgroups. The DC controller will react by downgrading a different subset of the flowgroup's traffic, but of course, the application can again respond by shifting load. To avoid adverse impact on both the flowgroup in question (unnecessary QoS churn) and other flowgroups (approval violations), how do we ensure that the DC controller converges quickly to a stable QoS assignment that admits only the approval at HIPRI?

4 Design

In this section, we explain how HEYP addresses each of the above-mentioned concerns. HEYP's design is tailored to the needs of large cloud providers. In aiming to satisfy the goals set out in §2, it provides the following key properties.

- Under planned failure scenarios, each flowgroup can ramp up its usage to its approval without any reaction from the global or DC controllers. Within-approval traffic will use paths that meet the specified latency SLO.
- Once a flowgroup exceeds its approval, HEYP will downgrade the flowgroup's excess traffic to LOPRI and rate limit it. The LOPRI routes and admissions are determined using dynamic global control to maximize efficiency.

Inputs: Approvals and demands per flowgroup
Topology annotated with link capacities
Outputs (per flowgroup):
Set of HIPRI routes and set of LOPRI routes
HIPRI admission and LOPRI admission

```
1. Compute HIPRI routes and admissions to satisfy
   approvals
2. Compute unused link capacity by deducting any link
   capacity consumed by within-approval demands
3. Compute LOPRI routes and admissions to satisfy
   above-approval demands
```

Algorithm 1: **Global computation of routes and admissions. Steps 1 and 3 follow provider's allocation policy.**

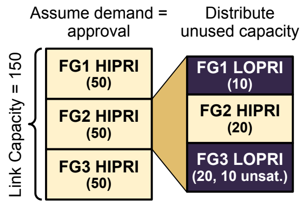
- To avoid degrading the performance of applications that have part of their traffic downgraded to LOPRI, HEYP maximizes the minimum time each task spends at a particular QoS. Applications that want to make the best use of the available LOPRI bandwidth should internally divert work away from bottleneck tasks. Many existing applications – e.g., HTTP proxies [8, 13], bulk data copies [7], and others [1, 9, 11, 12] – have this capability.
- HEYP's DC controller is biased to over-admit HIPRI traffic when usage is concentrated across a small number of tasks. Network operators can account for this by provisioning additional headroom (§4.3). For cloud WANs, we expect that approvals will be large enough for the required headroom to be low.

4.1 Separate HIPRI and LOPRI routes for efficiency

To appreciate why the use of QoS downgrade necessitates a change in the global controller's routing strategy, consider the example from Figure 2. To accommodate 15 Gbps of A→C's demand, existing 'Dynamic' controllers would compute and install two routes: one along the direct path and one along the indirect path via B, with the former set to carry one-third of the flowgroup's traffic and the latter two-thirds. If we admit 10 Gbps of A→C's traffic on HIPRI, since that is its approval, then 6.6 Gbps of A→C's HIPRI traffic would go over A-C and 3.3 Gbps over A-B-C. When B→C's within-approval usage increases, we risk violating its approval as it will compete for capacity with A→C's HIPRI traffic.

The problem here is that, if both within- and above-approval traffic are split in the same proportion across paths, we cannot simultaneously satisfy the two properties we want: 1) within-approval demands must be met irrespective of other flowgroups' demands, e.g., A→C should not route within-approval traffic over A-B-C, and 2) above-approval traffic should be able to use any link capacity that is unused by other flowgroups; this constraint is opposite to the previous one: A→C's above-approval traffic must go over A-B-C.

To resolve this issue, in HEYP, we compute *multiple sets* of paths per flowgroup that each meet one of these objectives. We route each flowgroup's within-approval traffic in a manner that statically guarantees no interference with other within-approval traffic. Additionally, we ensure that the routes for above-approval traffic make use of any spare capacity on the network. In our example, we send 10 Gbps of A→C's HIPRI traffic over A-C, and 5 Gbps of A→C's LOPRI traffic over A-B-C. When B→C's demand rises to 10 Gbps, it takes priority over A→C's LOPRI above-approval traffic. Since A→C now sends HIPRI traffic only over the direct link to C, both approvals are satisfied.



(a) Allocation in each phase

	HIPRI	LOPRI
FG1	50 Gbps	10 Gbps
FG2	50 Gbps	0 Gbps
FG3	50 Gbps	20 Gbps

(b) Final admissions

Figure 3: Allocation for three flowgroups from data centers A to B over a direct link of capacity 150 Gbps. Each has a 50 Gbps approval, and demands (in Gbps) are [FG1: 60, FG2: 20, FG3: 80]. In Phase 1, all three approvals fit. Phase 2 distributes 30 Gbps of FG2's unused allocation fairly between FG1 and FG3. The rest of FG3's demand is left unsatisfied.

Global allocation framework. HEYP determines the sets of paths and admissions for each flowgroup as follows. To support many traffic engineering bandwidth sharing policies, HEYP uses existing algorithms as black box functions to provide capacity to within-approval (HIPRI) or above-approval (LOPRI) traffic. Within a particular QoS level, these functions are free to enforce their own policies.

In existing systems, the global WAN controller [38, 40] computes routes for a particular traffic demand matrix in two phases: 1) fit all within-approval demands, and 2) based on the provider's bandwidth sharing policy, accommodate as much above-approval demands as feasible given the capacity that remains. The routes for every flowgroup comprise the union of the routes computed in the two phases and the admission is the sum of capacity allocated on each route.

HEYP's global controller similarly executes in two phases, but both phases differ (Algorithm 1) and the outputs of each phase apply separately to either HIPRI (i.e., within-approval) or LOPRI (i.e., above-approval) traffic.

- **Phase 1: Match Static's approval satisfaction.** First, to ensure that flowgroups can burst up to their approvals, we compute HIPRI routes to fit all *approvals*, not just within-approval demands, while ensuring that path lengths are within guaranteed bounds (§2). In the unlikely scenario that more capacity is lost due to failures and maintenance than what the provider planned for, capacity is shared according to policy (e.g., max-min fairness).
- **Phase 2: Match Dynamic's demand satisfaction.** Next, we determine additional routes based on observed demand. The key is to compute the capacity consumed by Phase 1 based on *within-approval demands*, not approvals. With this, HEYP admits the same volume of above-approval demands as Dynamic. Moreover, when Phase 1 is unable

to fit all approvals into the network, HEYP can admit additional within-approval traffic in this second phase, thereby surpassing Static with respect to approval satisfaction.

In achieving these desirable properties, we are oversubscribing link capacities: HEYP allocates routes based on approvals in Phase 1 but computes the capacity consumed by these routes based on within-approval demands. However, when a link's capacity is oversubscribed, HIPRI traffic will be preferentially delivered, thus ensuring that congestion has no impact on approval satisfaction. HEYP never oversubscribes link capacity in Phase 1 to ensure that an increase in one flowgroup's within-approval demand does not impact the ability to satisfy approvals for other flowgroups. Figure 3 illustrates how the HEYP controller separately computes HIPRI and LOPRI admissions.

Mitigating switch limitations. The degree to which HEYP oversubscribes link capacities is configurable: in Phase 2, the available capacity on each link can be set such that the sum of HIPRI and LOPRI admissions do not exceed a configurable multiple of the link's capacity. For network switches that share buffers between per-QoS queues, this can be used to reduce HIPRI packet drops under a flood of LOPRI traffic.

4.2 Minimizing QoS churn with caterpillar hashing

Once the global controller has determined the HIPRI and LOPRI admissions for a particular flowgroup, the DC controller must assign a QoS level for each of the flowgroup's flows, i.e., each (src IP, src port, dst IP, dst port, protocol) 5-tuple. For this, it first needs to measure the total usage of the flowgroup, and then identify a subset of flows to downgrade such that the sum usage of the remaining flows equals the approval.

Need to minimize QoS churn. A straightforward approach for picking flows to downgrade would be to use a knapsack solver to identify a set of flows whose aggregate usage is closest to the flowgroup's current usage minus the HIPRI admission. However, knapsack solvers make no effort to maintain stable QoS assignments across multiple runs, harming application performance. Every time the QoS assigned to a flow is changed, its bandwidth and latency characteristics change as well. If TCP's congestion control is unable to adapt quickly enough, application performance suffers.

Figure 4 demonstrates the impact of frequently changing QoS between backend servers in one data center and an HTTP proxy in another (see §5.2 for details). Both the latency (90%ile of 600 ms vs 95 ms) and throughput (mean 19K req/s vs 21K req/s) seen by the clients suffer, when compared to scenarios where every flow is pinned to a specific QoS level. The reason for this degradation is that BBR [21], the congestion control used in the experiment, is not able to send data at a rate high enough to avoid large queuing delays. BBR actively probes for new round-trip time (RTT) measurements at most once every 10 seconds (more frequent probes would sacrifice throughput) [21]. So, when the DC controller changes QoS every 5 seconds, BBR incorrectly estimates that 99% of LOPRI flows have the RTT of the HIPRI path, and maintains fewer bytes in flight as a result (average congestion window is over 55% smaller), adding queuing delay. If we change the control period to be one minute, the difference between "Stable QoS" and "Flip Flop" disappears.

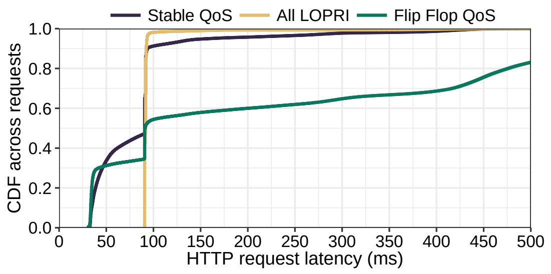


Figure 4: Latency with 8 tasks in three scenarios: all use LOPRI, half use LOPRI (tasks never change QoS), and half use LOPRI (each task flips its QoS once every 5 seconds). HIPRI tasks are rate limited to one-eighth of the demand (see §4.3); if this is removed "Stable QoS" outperforms "All LOPRI".

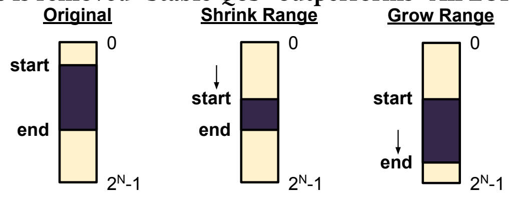


Figure 5: Caterpillar hashing shrinking and growing the subset of flows which are downgraded.

Challenges in minimizing QoS churn. To maintain QoS stability, one could try to 'pin' each flow to its QoS for some minimum threshold of time. However, doing so would impact the accuracy with which the DC controller can downgrade the desired fraction of a flowgroup's traffic, since only a subset of flows would be eligible for QoS changes.

Alternatively, one could hash every flow's identifier and downgrade the traffic of those flows whose hashed identifier falls below a threshold. The DC controller can assign more (less) of a flowgroup's traffic to LOPRI by increasing (decreasing) this threshold. The problem, however, is the *order* in which flows are downgraded and upgraded. When the threshold rises to downgrade additional flows, and later drops to upgrade flows, the most recently downgraded flows would be upgraded; vice-versa when the threshold subsequently is increased again. This behavior maximizes the worst-case QoS churn for individual flows.

Rethinking hashing-based QoS downgrade. In HEYP, we introduce caterpillar hashing as a flow selection mechanism that minimizes QoS churn. Caterpillar hashing chooses which flows to downgrade using a range of the hash space, rather than a threshold. As illustrated by Figure 5, when we need to increase (decrease) the fraction of flows that are downgraded, we grow (shrink) the range by moving the upper (lower) threshold. This behavior upgrades the flows that were downgraded earliest, and therefore, maximizes the minimum time each flow spends at a particular QoS.

Hashing-based approaches randomly select flows for downgrade, and therefore have lower accuracy compared to using a knapsack solver. However, in the following section, we explain how HEYP's DC controller leverages feedback control, and this largely mitigates any concerns about accuracy.

4.3 Mitigating harmful app–controller interactions

Since existing DC controllers configure tasks (§2.1), they could be extended to measure what fraction of usage is above approval, apply caterpillar hashing to select tasks for downgrade, and then compute rate limits for LOPRI tasks. The controller could use caterpillar hashing to ensure that the fraction of a flowgroup's traffic which is downgraded equals 1 - (total usage)/(HIPRI admission), where HIPRI admission is equal to approval, except under extreme failure scenarios. However, this approach can lead to harmful interactions between the DC controller and applications.

Consider the HTTP workload used to generate Figure 4. When LOPRI flows experience congestion, the HTTP proxy would observe longer queues for LOPRI tasks compared to HIPRI tasks, and shift more of its load to the HIPRI tasks. This would cause the flowgroup to have HIPRI usage greater than its approval, since the set of HIPRI tasks is now transmitting bandwidth that used to be spread across a larger set of tasks. However, the DC controller would not react because the fraction of usage above approval is unchanged; after all, the load has simply shifted between tasks. Had the DC controller instead used a knapsack solver, it would have seen that the flowgroup's HIPRI usage is higher than intended and selected a different subset of tasks to downgrade. But, the application will again react by shifting its usage around. To prevent this cat-and-mouse game, which will result in high QoS churn and put approval satisfaction for other flowgroups at risk, let us first consider two strawman approaches.

Strawman 1: Downgrade jobs as a unit. Most cluster management systems have some notion of a job that is used to deploy applications [10, 37, 64]. To downgrade a portion of any flowgroup, if we were to downgrade at the granularity of jobs, the application would be unable to respond in the above manner. However, some applications are composed of multiple jobs, and since the DC controller has no knowledge of which jobs are critical for the application, downgrading an entire job may degrade the user experience.

Strawman 2: Rate limit HIPRI traffic. Alternatively, one could use rate limiting to prevent an application from sending more HIPRI traffic than its approval, as we did in Figure 4. However, despite its use in production WANs, distributed rate limiting suffers from inaccuracy and risks throttling tasks unnecessarily [59] (see also §5.2.3). For LOPRI traffic, we believe the costs of rate limiting are worth the benefit: it enables policy-based sharing and avoids high loss on fully-loaded links. In contrast, for HIPRI traffic, we seek to prevent problematic interactions between applications and the DC controller without rate limiting.

```
Config: upgradeInc = 0.2 (fixed frac. to upgrade)
        propGain = 0.5 (proportional gain)
        errNoise = 0.05 (noise in usage measurement)
        kCoarse = 2 (task err multipler)
Output: fraction of usage to downgrade (upgrade if < 0)
-------------------------------------------------------
if total usage < HIPRI admission then
    return upgradeInc
end
err ← (HIPRI usage - HIPRI admission) ÷ total usage
coarseness ← kCoarse × max task usage ÷ total usage
if 0 < err < max(errNoise, coarseness) then
    return 0
end
return propGain × err
```

Algorithm 2: **Feedback control determines what fraction of usage to downgrade. The configuration parameters were tuned against a range of simulated workloads (§B).**

Search for application bottleneck. To avoid the downsides of these strawman approaches, we use the following observation: applications can respond to QoS downgrade by shifting around load only because their HIPRI tasks are able to handle additional load. Eventually, each task becomes limited by some resource other than WAN link capacity, e.g., the machine's network card. Therefore, if we ensure that all HIPRI tasks are saturated, the application will not shift additional load to HIPRI tasks.

To search for this operating point – where the HIPRI tasks are saturated enough that the application does not shift additional load over from LOPRI tasks – HEYP employs feedback control. Although the DC controller does not know exactly when tasks become saturated, it can iteratively increase the fraction of tasks that are downgraded. We assume that no individual task can saturate an approval, and hence, HIPRI tasks will eventually become saturated.

In each control period, HEYP's DC controller revises the fraction of downgraded tasks in proportion to the relative error in enforcing a flowgroup's HIPRI admission. Using caterpillar hashing, the controller increases (or decreases) the fraction that is downgraded in proportion to (HIPRI usage - HIPRI admission) / flowgroup's overall usage. This simple form of control [65] mitigates the harmful interaction. As an added benefit, it improves the accuracy of the DC controller's selection of tasks to downgrade. If the downgraded tasks combined have too much or too little usage, the feedback controller will observe this error and try to eliminate it.

There remain two concerns that need to be addressed.

- First, when usage is below the HIPRI admission, we do not know what fraction of the usage should be upgraded to HIPRI. It could be that the flowgroup's demand is below the HIPRI admission; in this case, the correct response would be to upgrade all tasks. On the other hand, it could be that the controller has downgraded too much traffic and should simply upgrade a small portion of it. HEYP tries to balance its behavior for these different cases by always upgrading 20% of traffic. This provides a slower, but hopefully acceptable response to the first case (five control periods are needed to upgrade the entire flowgroup) and reduced QoS churn in the second case.

- Second, HEYP ignores excess HIPRI usage in two cases. The first case is when the HIPRI usage is within measurement noise of the HIPRI admission. This threshold can be determined using an online estimator or offline analysis. For simplicity, our prototype uses a static value. The second case is when the HIPRI usage exceeds the HIPRI admission by a small multiple of the maximum task usage. The intuition is that task usages may be too coarse to achieve the desired split, and the maximum task usage serves as an overestimate of the coarseness of all task usages. To prevent the resulting excess HIPRI usage from causing approval violations, the network provider should provision enough headroom to accommodate both cases. As noted at the start of §4, we expect the required headroom for cloud WANs to be low.

Algorithm 2 presents HEYP's final control logic for revising the fraction of tasks to downgrade. In Appendix B, we empirically show that HEYP's DC controller provides low QoS churn and quickly converges to an accurate split under a variety of workloads.

5 Evaluation

We evaluate HEYP's performance in two parts. First, using production traces from BigCloud's WAN and a discrete-event simulator, we evaluate the benefits of HEYP's global controller for satisfying both approvals and demands across data centers. Then, we deploy a prototype of HEYP's DC controller on CloudLab [27] and evaluate its ability to enforce HIPRI admissions (using QoS downgrade) and its utility on an application workload. The primary takeaways from our evaluation are as follows:

- HEYP offers the best combination of approval and demand satisfaction: 99% availability of approved bandwidth for 87–99% of flowgroups (better than even static approval-based allocation) while offering similar demand satisfaction as dynamic allocation, which is able to satisfy only 7–9% of approvals 99% of the time.
- In a sensitivity analysis, we find that dynamic allocation falls short of the approval satisfaction offered by HEYP even if control plane delays are cut by 5× and demands change slowly. In addition, HEYP delivers high approval satisfaction even if demands change twice as fast as in BigCloud's WAN while the approval satisfaction of dynamic allocation is further reduced.
- When applied to an HTTP workload, HEYP offers the best combination of isolation and performance. When competing against a flowgroup with excess traffic, the latency and throughput of a within-approval flowgroup are unchanged compared to static, approval-based rate limiting. In addition, the additional bandwidth HEYP provides to the above-approval flowgroup improves throughput to within 12% of the theoretical max.

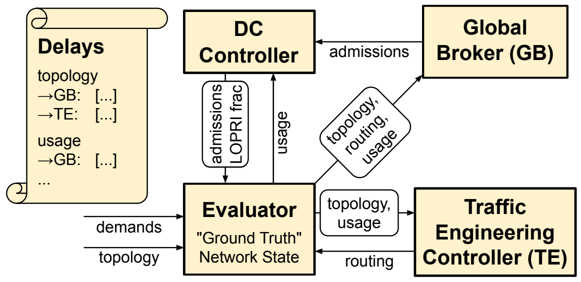


Figure 6: Architecture of our inter-DC simulator.

5.1 Predictability and efficiency across DCs

To evaluate HEYP's impact on sharing bandwidth between flowgroups spread across many data centers, we use a custom, discrete-event simulator. Simulation enables us to evaluate designs that are impossible or difficult to realize (e.g., we consider a hypothetical control plane that acts 5× faster than the state-of-the-art). Our simulator models delays between network components and captures the impact of inaccurate demand estimates (Figure 6). In Appendix A, we describe our simulator in detail, including how we validate its fidelity.

5.1.1 Trace data, allocation algorithms, and metrics

We use traces obtained from BigCloud's WAN containing data for three separate weeks in 2019. Each trace contains snapshots of the topology and demand between data centers – as estimated by the production system – measured once a minute. Bandwidth approvals were derived from production data collected from the BigCloud WAN and adjusted to account for differences between the simulation and production environments. BigCloud ensures that (given fast controller response) approvals can be met under appropriate failure scenarios. For each type of control plane delay (e.g., time taken to install a new set of routes), our simulations mimic the distribution seen in production.

We implement and compare HEYP against the following approaches in our simulator.

- **Static** allocation policy is oblivious to demands. When the Global Broker observes a new topology or routing (resp., when the Traffic Engineering (TE) Controller observes a new topology), it computes new admissions (resp., new routes) given the approvals as demand.
- **Dynamic** policy approximates the behavior of B4 [40, 46] and SWAN [38]. Unlike Static, it reacts not only to topology changes but also when demands change, in order to allocate above-approval traffic after satisfying within-approval demands. For any flowgroup, all traffic traverses one set of paths and uses the same QoS level.
- We also consider two variants of Dynamic which strike intermediate tradeoffs between approval and demand satisfaction. **RA+Dynamic** (for Reserve Approval + Dynamic) assumes that demand = max(approval, demand). Hence, it will allocate at least as much capacity as Static but will

	% of flowgroups with ≥ 99.9% approval satisfaction			% of flowgroups with ≥ 99% approval satisfaction			Mean demand satisfaction (%)		
	Week 1	Week 2	Week 3	Week 1	Week 2	Week 3	Week 1	Week 2	Week 3
Static	37	41	46	81	97	94	55	44	58
RA+Dynamic	34	34	46	80	97	94	70	74	73
Dynamic+JA	3	2	3	9	13	9	81	79	84
Dynamic	3	2	4	7	9	7	86	88	89
HEYP	37	43	51	87	99	97	86	83	90
Legend	0–20 20–40 40–60 60–80 80–100			0–20 20–40 40–60 60–80 80–100			0–60 60–70 70–80 80–90 90–100		

Table 1: Simulation results for BigCloud network traces across three weeks.

attempt to accommodate above-approval traffic when demands change. **Dynamic+JA** (for Dynamic + Jump to Approval) assumes that the demand for any flowgroup being throttled is equal to its approval, thereby preempting the need for multiple iterations of global reconfiguration for within-approval demand to ramp up. The throttling signal is propagated together with the usage information.

Allocation algorithms. In all approaches, the Global Broker and TE Controller first allocate bandwidth to satisfy within-approval demands, then use residual capacity to satisfy above-approval demands. In either phase, they enforce max-min fair sharing across flowgroups. To compute routes, the TE Controller selects the shortest available path for each flowgroup and computes a max-min fair allocation of bandwidth across these paths. This process loops until either all demands are satisfied or all links are saturated. Traffic for a flowgroup is split across the routes allocated for it in the ratio of the admission computed for each route. When a link fails, flowgroups may experience traffic loss until the controller installs new routes. For more details, see Appendix E.

Metrics. We examine the approval and demand satisfaction of each approach. We consider a flowgroup's approval to be satisfied whenever its usage is ≥ 0.95×min(approval, demand). We compute demand satisfaction as the sum of per-flowgroup usages divided by the sum of their demands. We use this metric – as opposed to link utilization – to measure efficiency because a higher value directly corresponds to a better use of network resources.

5.1.2 Results

Table 1 presents the results for each of the three week-long traces; we consider two commonly studied [20, 39, 69] (99% and 99.9%) availability targets.

Approval satisfaction for HEYP and Static are similar, as expected, since Static's allocation is the same as that used in HEYP's HIPRI allocation. However, Dynamic satisfies up to twice the demand of Static, a result of it's allocation being demand aware. In most cases, HEYP achieves similar demand satisfaction to Dynamic, but HEYP consistently offers significantly higher availability of approved bandwidth. One reason for Dynamic's poor availability is the duration of approval violations: 20% of violations are resolved only after multiple iterations of global control.

Dynamic+JA and RA+Dynamic hit intermediate tradeoffs in between Static and Dynamic. The reason for this is that

Legend: 0–20 20–40 40–60 60–80 80–100

Control Plane Speed	Rate of Demand Change (larger is faster)								
	Dynamic			Static			HEYP		
	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×
5× Faster	59	25	15	88	87	87	96	94	93
Normal	15	9	8	87	87	86	93	91	90
5× Slower	4	4	4	60	60	59	61	59	59

(a) % of flowgroups with ≥ 99% approval satisfaction

Legend: 0–60 60–70 70–80 80–90 90–100

Control Plane Speed	Rate of Demand Change (larger is faster)								
	Dynamic			Static			HEYP		
	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×
5× Faster	95	94	93	55	55	55	88	88	87
Normal	89	86	84	55	55	55	87	86	85
5× Slower	78	75	75	55	55	55	84	83	82

(b) Mean demand satisfaction (%)

Table 2: Performance when varying both the speed at which controllers react and the rate at which demands change.

Dynamic+JA and RA+Dynamic reserve bandwidth based on approvals, even when demands are lower than approvals. Since RA+Dynamic does so always, it offers lower demand satisfaction like Static; whereas, since Dynamic+JA allocates for approval only once a flowgroup is throttled, it offers low approval satisfaction like Dynamic.

Impact of tail latency. In our traces, the time from when the DC Controller detects a change in demand until new admissions (routes) are installed is 3× (1.5×) larger at the 99th percentile than at the median. To investigate whether high tail latency is negatively impacting tail approval satisfaction, we simulate Week 2 with all control delays limited to the 45–55th percentile range of the distribution observed in production. We see little increase in Dynamic's approval satisfaction; only 13% (4%) of flowgroups have 99% (99.9%) approval satisfaction. We conclude that even the median global control delays in such a heavily engineered WAN are too high to accommodate the churn in demand.

Sensitivity to changes in workload and setting. To evaluate each approach in a broader range of settings, we vary the inputs from BigCloud along two dimensions: the rate at which demands change and the control plane's speed (both in the delays incurred and the frequency with which controllers

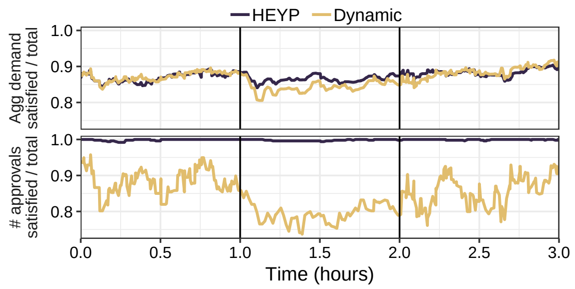


Figure 7: Performance under a global controller outage (starts after 1 hour and lasts for 60 min).

run). Table 2 compares Dynamic, Static, and HEYP on a 48-hour trace during Week 1. We make several observations regarding approval satisfaction:

- Dynamic is highly dependent on timely responses to demand changes. Between the easiest scenario (fast control plane and slow-changing demands) and the hardest (slow control plane and fast-changing demands), fraction of approvals satisfied with Dynamic drop by over 10×.
- Regardless of the control plane's speed, approval satisfaction with HEYP and Static is independent of the rate of demand change. Whereas, Dynamic significantly suffers when demands ramp up faster; even with a fast control plane, the fraction of approvals receiving 99% availability with Dynamic drops by 4× when going from a slow to a fast rate of demand change.

With respect to demand satisfaction, Static is poor across the board since it does not react to changes in demand or admit above-approval traffic. In contrast, a slower control plane significantly decreases demand satisfaction with Dynamic but has no impact on HEYP; by allocating HIPRI routes and admissions based on approvals, not within-approval demands, HEYP allows within-approval usage to ramp up without any action by the global controller. With a faster control plane, both Dynamic and HEYP more quickly adapt to accommodate changing above-approval demands.

Performance under a global controller outage. An extreme case of a slow control plane is when the global controller is down. To evaluate performance under such a scenario, we select a 3-hour window from Week 1 and simulate a failure of both the Global Broker and the TE Controller. No data plane failures take place during the outage.

Figure 7 shows that HEYP consistently satisfies nearly all approvals during the outage, whereas approval satisfaction with Dynamic drops shortly after the control plane outage begins. While both approaches have degraded demand satisfaction during the outage, HEYP satisfies more demand than Dynamic because it pre-allocates capacity to satisfy any increase in within-approval demands. At other times, HEYP and Dynamic provide similar demand satisfaction.

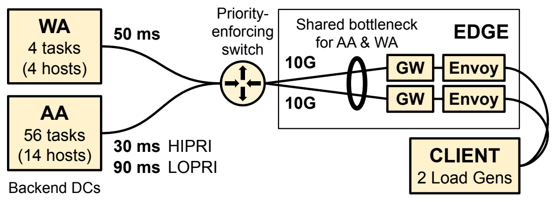


Figure 8: Testbed setup for HTTP workload. We run a separate DC controller for each data center (not shown).

5.2 Testbed evaluation

Next, we deploy a prototype of HEYP's DC controller and study its ability to accurately enforce HIPRI admissions (using QoS downgrade) with low QoS churn. In addition, we examine the impact of QoS downgrade on an application workload, both from the perspective of isolating any within-approval flowgroup and maximizing an above-approval flowgroup's ability to use spare capacity.

5.2.1 Application workload and setup Since web services are highly sensitive to latency inflation and bandwidth shortages, we evaluate HEYP against HTTP workloads and emulate the architecture of production web services. As shown in Figure 8, clients (which use Fortio [14], a load generator) issue requests in an open loop to EDGE, where one of two Envoy [8] proxies examines which backend the request is for and directs it to an appropriate backend server. Upon receiving the proxied request, the backend generates a response that is then forwarded by the proxy back to the client. Each backend task is registered with the local DC controller, and enforces QoS downgrade and rate limiting policies via standard Linux facilities.

We deploy backends onto CloudLab's xl170 machines (10 cores) connected via 10 Gbps links to a Dell S4048-ON switch. The switch is configured to enforce strict priority queuing between HIPRI and LOPRI traffic. The Envoy proxies and Fortio clients run on dedicated c6525-25g machines (16 cores) and are connected to each other via a 25 Gbps network. Each Envoy proxy reaches the backend servers via its own gateway server (xl170) that is connected to both networks. Following existing systems [46], we set the DC controller to compute new QoS assignments and rate limits once every second (we show that this rate is feasible with millions of tasks in Appendix C).

We run two backend services, logically separated into two "data centers": AA (for above approval) and WA (for within approval). We simulate latency between them and EDGE using netem [35]. The approvals for AA→EDGE and WA→EDGE are 2 and 12 Gbps, respectively. WA's approval was chosen to exceed half of the bottleneck link's capacity (20 Gbps, 10 Gbps for each EDGE proxy) so that a max-min fair distribution of the capacity would violate the approval.

We compare HEYP to the following approaches:

- **NoCongestion.** This approach estimates the best throughput and latency that AA can achieve irrespective of whether QoS downgrade or rate limiting is employed, i.e., when

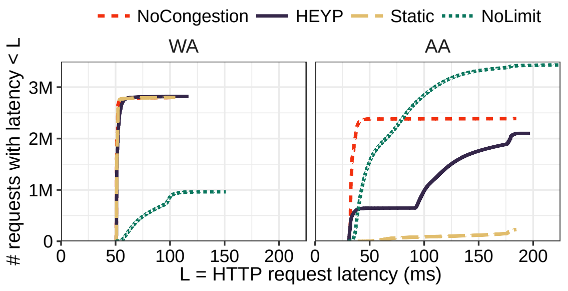


Figure 9: Latency vs number of requests served with that latency for each flowgroup.

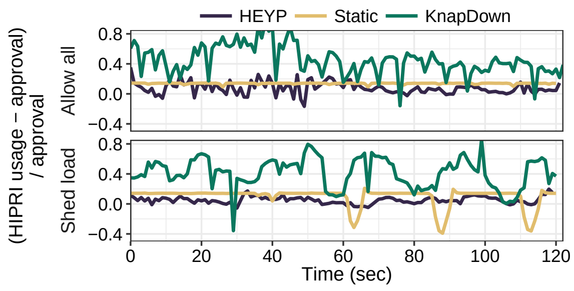


Figure 10: Unlike HEYP and Static, KnapDown admits excess HIPRI traffic when applications redistribute load.

bandwidth is the only constraint. We obtain this estimate by reducing WA's sending rate so that the sum of AA and WA demands are satisfied without overloading any links; empirically, we have determined that the bottleneck link can sustain up to 90% utilization.

- **Static.** By rate limiting each flowgroup to its approval, this approach prioritizes providing isolation for WA at the cost of AA's demand satisfaction. This serves as a baseline for comparing QoS downgrade and rate limiting as admission control mechanisms. Our implementation follows BwE's Job Enforcer [46]; in particular, we have implemented both dynamic oversubscription (based on workload burstiness) and static oversubscription (scale up capacity by 1.25×).
- **KnapDown.** To study the utility of feedback control and caterpillar hashing, we downgrade QoS using a knapsack solver, the initial approach described in §4.2.
- **NoLimit.** To demonstrate that some form of control is needed to satisfy approvals, we consider the effects of using neither QoS downgrade nor rate limiting.

When studying AA's performance, we focus on scenarios where the tenant has configured the application to degrade gracefully under overload, and therefore enable load shedding. To ensure that any requests that are served maintain reasonable latency [6, 19], Envoy routes requests to the least-loaded backend server and eagerly rejects requests for WA when the corresponding backend servers become overloaded. When studying an approach's ability to isolate WA's traffic from a noisy neighbor, we disable load shedding for AA's traffic to ensure that we are not measuring the effects of AA's load shedding, but the network isolation mechanism.

5.2.2 Performance under gradual workload change

We start by examining the performance of a workload that changes gradually, e.g., user traffic increasing over a day. We set AA's demand to 12 Gbps and ramp up WA's demand at a constant rate from 6 to 12 Gbps over 2 minutes. Figure 9 presents the latency and throughput for both backends.

Performance isolation for WA. Static and HEYP both provide strong isolation for WA; latency matches both NoCongestion and the case where AA has no above-approval traffic. With NoLimit, bandwidth is shared based on the behavior of congestion control, not on approvals. As a result, WA's performance degrades when AA, which contains 14× as many tasks as WA, captures more bandwidth than it.

Benefit of above-approval bandwidth for AA. Of the approaches that satisfy WA's approval, we see that HEYP offers the best combination of latency and throughput (throughput is 88% of NoCongestion and 1.8M requests complete within 150 ms) compared to Static (throughput is 9% of NoCongestion and only 120K requests complete in 150 ms). The 12% gap in throughput between HEYP and NoCongestion is due to load shedding; once disabled, AA's throughput with HEYP matches NoCongestion, albeit at an even higher latency (above 400 ms).

Note that the low latency that NoLimit and NoCongestion offer to AA is an artifact of our experimental setup. We inject 60 ms of additional propagation delay for LOPRI traffic to emulate the case where it traverses a longer path than HIPRI traffic. In practice, this should only occur when the global controller observes high utilization on a bottleneck link for HIPRI traffic. In this case, NoLimit and NoCongestion would also need to use the longer path for a portion of their traffic, but our testbed is unable to capture this.

Utility of feedback control on limiting harmful app–controller interactions. For the same workload as Figure 9, Figure 10 shows that the gap between the approval and HIPRI usage for AA is consistently higher when using KnapDown than HEYP. In each instance where KnapDown is able to eliminate all excess HIPRI, we see that AA quickly returns to using more HIPRI than its approval. When the DC controller runs again, KnapDown makes no attempt to maintain stable QoS assignments unlike HEYP, which leverages caterpillar hashing and feedback control. As a result, KnapDown performs 27× the number of QoS changes as HEYP. Of the three approaches shown, HEYP provides the lowest mean absolute error: for the top case, it is within 9% of the approval vs 14% using Static and 45% using KnapDown.

5.2.3 Performance under sudden workload change

Next, we stress test HEYP's ability to keep HIPRI usage near the approval under sudden workload changes. We keep WA's demand static at 12 Gbps and configure AA's demand to rise sharply after 20 seconds from 3 to 9 Gbps, then drop after a minute to 3 Gbps, and rise one last time after another 60 seconds to 9 Gbps.

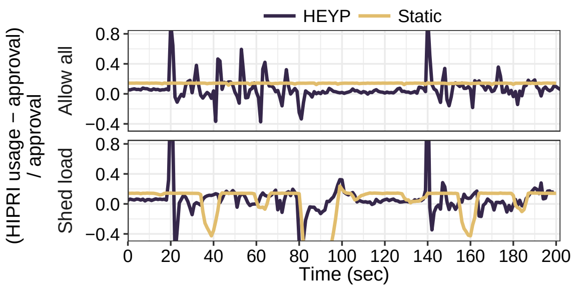


Figure 11: The accuracy of HEYP and Static when trying to keep AA→EDGE's HIPRI usage near its approval under sudden demand changes (at 20s, 80s, and 140s).

Figure 11 presents the HIPRI error for AA when Envoy sheds load, and when it admits all requests. Focusing on Figure 11(top), we see that Static consistently admits 14% more HIPRI usage than approval allows. The excess HIPRI usage is due to Static's oversubscription of bandwidth. With better tuning, Static's accuracy may improve, but if we enable load shedding for AA (see Figure 11(bottom)), we see that Static frequently throttles AA→EDGE below its approval. This illustrates the difficulty in tuning approaches that leverage rate limiting: if we configure Static to oversubscribe the network less, than it may perform better in the former case, but it would throttle even more aggressively in the latter case.

In contrast, HEYP's controller adapts without tuning to the two workloads. The inaccuracy of its rate limiting only impacts how much LOPRI capacity HEYP delivers, not its approval satisfaction. HEYP will satisfy the approval even when it downgrades too much traffic, except when LOPRI is sufficiently congested or throttled. In Figure 11, HEYP satisfies AA→EDGE's approval 96% of the time in the bottom case, and Static satisfies the approval only 80% of the time (both have 100% satisfaction under no load shedding).

6 Discussion

Weighted fair queuing. HEYP's global allocation can be adapted for networks that share bandwidth across QoS levels using weighted fair queuing, rather than strict prioritization. The key is to account for the reservation of bandwidth to LOPRI traffic. For example, if the ratio of weights for HIPRI:LOPRI QoS is 8:2, then LOPRI traffic can use 20% of the link's bandwidth regardless of the HIPRI usage. In this case, we would scale down the link capacities in Phase 1 (§4.1) to 80% of the original values, so that HIPRI traffic always receives its full admission.

Multiple approval SLOs. In this paper, we aim to maximize the satisfaction of a single, high-priority class of approvals. However, HEYP can support multiple levels of prioritized approvals by iteratively allocating routes and admissions for each class, with lower classes using the residual capacity left over from higher classes. The relative importance of a high-priority flowgroup's above-approval traffic compared to a lower-priority flowgroup's within-approval traffic depends on the cloud provider's business policy. For example, if cloud provider wanted to offer two bandwidth approval SLOs on a network with three QoS levels – HIPRI, MEDPRI, and LOPRI – the provider could choose to treat above-approval traffic for higher SLO approvals as equivalent to within-approval traffic for lower SLO approvals, marking both as MEDPRI.

7 Related Work

Software-defined WANs. The rising demand for network bandwidth across data centers has led to the development of many global private WANs, e.g., by Microsoft [38], Google [40], and Facebook [43]. These networks use centralized demand monitoring and traffic engineering to cost-efficiently transfer large volumes of data, though scaling them presents challenges [15, 29, 39]. While HEYP builds on these systems and shares a similar software-defined architecture, it aims to satisfy bandwidth approvals as a primary objective without sacrificing network utilization.

Bandwidth isolation between cloud tenants. Many prior systems aim to guarantee bandwidth between virtual machines in the data center, ranging from approaches that simply isolate tenants from one other [18, 32, 49], to others which provide work conservation [41, 58], to ones that enforce rich notions of fairness across tenants [57]. Adapting these approaches to the WAN setting is not straightforward. Providers have less flexibility with regards to application placement, control plane delays are significantly larger, and bandwidth guarantees are at the granularity of flowgroups, each of which spans a large number of hosts.

BwE [46] and SWAN [38] provide WAN bandwidth isolation by dynamically controlling the sending rates of tenants. HEYP differs from these approaches by combining static and dynamic allocation through the use of QoS downgrade.

Fault-tolerant routing. Many approaches have been proposed to quickly restore network connectivity following a failure [51, 56, 66, 68], and fault-tolerant traffic engineering approaches [20, 42, 50, 63] further aim to ensure that the remaining paths after a failure can support the admitted traffic. These approaches can be used together with HEYP to ensure high approval satisfaction without relying on the global controller reacting to either demand or topology changes.

QoS downgrade. Prior work has used QoS downgrade to provide statistical assurances of capacity to end users of the Internet [22, 23]. Unlike HEYP, these approaches do not scale to the large flowgroups present in data centers. HEYP accounts for the fact that no individual gateway can process all of the traffic belonging to a tenant, and it enables individual flowgroups to consume large quantities of capacity by allowing for any single flowgroup's traffic to be sent along multiple routes. Our use of separate routes for HIPRI and LOPRI traffic, however, introduces the need to maintain QoS stability, which HEYP explicitly aims to provide.

8 Conclusion

Existing control plane architectures for global-scale private WANs are unable to offer highly available bandwidth guarantees at high utilization. A key cause is their dependence on a fundamentally slow central controller to reconfigure the network in response to changing traffic demands. In this paper, we showed how to remove any reliance on the global controller for satisfying bandwidth guarantees by leveraging the data plane's ability to prioritize traffic based on QoS levels. Our HEYP WAN architecture uses the central controller only to maximize efficiency and handle topology changes, and we account for interactions with other layers of the network stack that result from admitting surplus traffic at a lower QoS along a separate set of routes. We showed that HEYP is able to simultaneously offer predictability and efficiency across a range of workloads and settings.

References

[1] 2017. gRPC Load Balancing. (2017). https://grpc.io/blog/grpc-load-balancing/.

[2] 2017. NetBench. https://github.com/ndal-eth/netbench. (2017).

[3] 2018. NDP Simulator. https://github.com/nets-cs-pub-ro/NDP/wiki/NDP-Simulator. (2018).

[4] 2019. Google Cloud Networking Incident #19009. (2019). https://status.cloud.google.com/incident/cloud-networking/19009.

[5] 2020. OMNeT++ Discrete Event Simulator. https://omnetpp.org/. (2020).

[6] 2021. Admission control - envoy documentation. (2021). https://www.envoyproxy.io/docs/envoy/v1.20.1/configuration/http/http_filters/admission_control_filter.

[7] 2021. Apache Hadoop Distributed Copy - DistCp Guide. (2021). https://hadoop.apache.org/docs/stable/hadoop-distcp/DistCp.html.

[8] 2021. Envoy Proxy. (2021). https://www.envoyproxy.io/.

[9] 2021. Istio / Traffic Management. (2021). https://istio.io/latest/docs/concepts/traffic-management/.

[10] 2021. Kubernetes. (2021). https://kubernetes.io/.

[11] 2021. Load Balancing | Linkerd. (2021). https://linkerd.io/2.11/features/load-balancing/.

[12] 2021. Loadbalancing reference - v2.5.x | Kong Docs. (2021). https://docs.konghq.com/gateway-oss/2.5.x/loadbalancing/.

[13] 2021. nginx news. (2021). https://nginx.org/.

[14] 2022. Fortio. (2022). https://fortio.org/.

[15] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. 2021. Contracting Wide-area Network Topologies to Solve Flow Problems Quickly. In *NSDI*.

[16] Amazon Web Services, Inc. 2019. Multiple Region Multi-VPC Connectivity. https://aws.amazon.com/answers/networking/aws-multiple-region-multi-vpc-connectivity/. (2019).

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

[18] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *SIGCOMM*.

[19] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.".

[20] 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 *SIGCOMM*. https://doi.org/10.1145/3341302.3342069

[21] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *ACM Queue* (2016).

[22] David Clark and Wenjia Fang. 1998. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on networking* (1998).

[23] David Clark and John Wroclawski. 1997. *An approach to service allocation in the Internet*. Internet Draft, draft-clark-diff-svc-alloc-00. Work in Progress.

[24] Rogério Leão Santos De Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. 2014. Using Mininet for emulation and prototyping software-defined networks. In *IEEE Colombian Conference on Communications and Computing (COLCOM)*.

[25] Nick Duffield, Carsten Lund, and Mikkel Thorup. 2005. Learn more, sample less: control of volume and variance in network measurement. *IEEE Transactions on Information Theory* (2005).

[26] Nick G Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merive. 1999. A flexible model for resource management in virtual private networks. In *SIGCOMM*.

[27] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX ATC*.

[28] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. 2021. Orion: Google's Software-Defined Networking Control Plane. In *NSDI*.

[29] Amitabha Ghosh, Sangtae Ha, Edward Crabbe, and Jennifer Rexford. 2013. Scalable multi-class traffic management in data center backbone networks. *IEEE Journal on Selected Areas in Communications* (2013).

[30] 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*. https://doi.org/10.1145/2934872.2934891

[31] John Graham-Cumming. 2020. Cloudflare outage on July 17, 2020. The Cloudflare Blog. (2020). https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/.

[32] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. 2010. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNext*.

[33] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. 2018. On low-latency-capable topologies, and their impact on the design of intra-domain routing. In *SIGCOMM*.

[34] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. 2015. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *SIGCOMM*.

[35] Stephen Hemminger et al. 2005. Network emulation with NetEm. In *Linux conf au*.

[36] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. 2008. Network simulations with the ns-3 simulator. *SIGCOMM demonstration* (2008).

[37] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*.

[38] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-driven WAN. In *SIGCOMM*. https://doi.org/10.1145/2486001.2486012

[39] 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 *SIGCOMM*.

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

[41] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*.

[42] Chuan Jiang, Sanjay Rao, and Mohit Tawarmalani. 2020. PCF: provably resilient flexible routing. In *SIGCOMM*.

[43] Mikel Jimenez and Henry Kwok. 2017. Building Express Backbone: Facebook's new long-haul network. https://engineering.fb.com/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/. (2017).

[44] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. 2014. Calendaring for wide area networks. In *SIGCOMM*.

[45] Christian Kaufmann. 2018. Making the Internet fast, reliable and secure. LINX Meeting. (2018). https://web.archive.org/web/20210308171115/https://www.linx.net/wp-content/uploads/LINX101-Akamai-ICN-ChristianKaufmann.pdf.

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

[47] Praveen Kumar, Chris Yu, Yang Yuan, Nate Foster, Robert Kleinberg, and Robert Soulé. 2018. YATES: Rapid prototyping for traffic engineering systems. In *SOSR*.

[48] 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 *NSDI*.

[49] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. 2014. Application-driven bandwidth guarantees in datacenters. In *SIGCOMM*.

[50] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic engineering with forward fault correction. In *SIGCOMM*.

[51] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring connectivity via data plane mechanisms. In *NSDI*.

[52] Microsoft. [n. d.]. Microsoft global network. ([n. d.]). https://docs.microsoft.com/en-us/azure/networking/microsoft-global-network.

[53] Microsoft. 2017. Microsoft global network. (2017). https://azure.microsoft.com/en-us/blog/how-microsoft-builds-its-fast-and-reliable-global-network/.

[54] Jeffrey C Mogul and Lucian Popa. 2012. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review* (2012).

[55] Gaya Nagarajan. 2014. Evolution of Facebook Backbone. In *NANOG*.

[56] Ping Pan, George Swallow, and Ping Pan. 2005. *Fast Reroute Extensions to RSVP-TE for LSP Tunnels*. RFC 4090.

[57] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*. https://doi.org/10.1145/2342356.2342396

[58] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. 2013. ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*. https://doi.org/10.1145/2486001.2486027

[59] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. 2007. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*.

[60] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's (datacenter) network. In *SIGCOMM*.

[61] Tanner Ryan. 2021. Cloudflare Backbone: A Fast Lane on the Busy Internet Highway. The Cloudflare Blog. (2021). https://blog.cloudflare.com/cloudflare-backbone-internet-fast-lane/.

[62] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*.

[63] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. 2011. Network architecture for joint failure recovery and traffic engineering. In *SIGMETRICS*.

[64] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*.

[65] Antonio Visioli. 2006. *Practical PID Control*. Springer London.

[66] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. 2010. R3: resilient routing reconfiguration. In *SIGCOMM*.

[67] John Wilkes. 2020. Yet more Google compute cluster trace data. Google research blog. (2020). https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html.

[68] Jiaqi Zheng, Hong Xu, Xiaojun Zhu, Guihai Chen, and Yanhui Geng. 2016. We've got you covered: Failure recovery with backup tunnels in traffic engineering. In *ICNP*.

[69] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. 2021. ARROW: restoration-aware traffic engineering. In *SIGCOMM*.

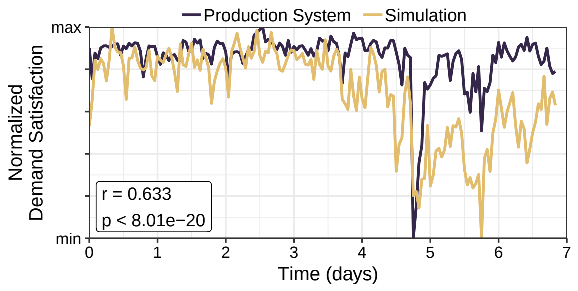


Figure 12: Demand satisfaction (normalized to have the same minimum and maximum values) in production versus simulation. The correlation coefficient (r) and p-value are noted.

A Inter-DC network simulator

We use a custom simulator because existing software has high overhead for evaluating WAN control planes [2, 3, 24, 36], licensing concerns [5], or focuses only on traffic engineering [47]. As in prior work [47], we model the topology at a data center level and apply max-min fair sharing of link bandwidth across flows. In addition, our simulator captures several features that govern the behavior of software-defined WANs.

Network controllers. As in B4 [40, 46], our simulator employs separate global controllers to select routes and admissions: the Traffic Engineering (TE) Controller and the Global Broker, respectively. Since we model traffic at the data center level, the simulated DC Controllers cannot configure individual tasks. Instead, we try to capture the impact that partitioning traffic into HIPRI and LOPRI has on approval and demand satisfaction. For example, if 30% of demand is marked LOPRI and usage drops to the approval, 30% of demand will remain LOPRI until the DC Controller revises the split. Appendix F contains the logic for each controller.

Modeling delays and inconsistency of state. We model the WAN as a set of processes that share no state. Processes send messages to a each other, scheduling their arrival at a future time. This model captures both the delays in controller response and any inconsistency of state across controllers.

Capturing demand uncertainty. An Evaluator process (see Figure 6) tracks the network state and computes metrics (e.g., demand satisfaction). The Evaluator broadcasts changes in any flowgroup's *usage* to all controllers. As a result, controllers may not observe rapid increases in a flowgroup's *demand* until several control periods have passed.

Validation. To confirm that the data output by our simulator is meaningful, we compare the mean, hourly demand satisfaction reported by BigCloud's production system against our simulated adaptation of it (see Dynamic in §5.1). Figure 12 shows that there is a statistically significant, positive correlation between the demand satisfaction observed in our simulation and in production. While the production system contains additional heuristics to improve performance, our simulation is a reasonably good predictor for the demand satisfaction seen in production: time frames in which the production system has higher (lower) demand satisfaction are also times in which the simulator performs well (poorly).

B Large-scale simulation of HEYP's DC controller

We use monte carlo simulation to study the behavior of HEYP's DC controller across a wider range of workloads than those run in §5.2.

Setup. In each run, we generate a static set of per-task demands according to a desired distribution and repeatedly invoke the DC controller against it to either downgrade traffic (if all tasks are HIPRI) or upgrade traffic (if all tasks are LOPRI). We set the approval to one-half of the expected demand, cap each host to send at most 40 Gbps, and set the available LOPRI bandwidth to 25% of the aggregate demand.

Demand distributions. We simulate 200 tasks and distribute demand across tasks according to one of the following distributions (all have a mean usage per task of 2 Gbps):

- UNI: The demand of each task is chosen between 0 and 2 Gbps uniformly at random.
- EM-5%: The top 5% of tasks have demand chosen between 30 and 34 Gbps uniformly at random. The remaining tasks have demand between 0 and 842 Mbps, also chosen uniformly at random.
- EXP: The demand of each task is chosen from an exponential distribution.
- FB15: We generate demands for the four types of WAN-using applications at Facebook [60], and scale them so that the distribution mean is the desired 2 Gbps. We assume the fraction of tasks belonging to each application is proportion to its demand, and evenly spread each application's demand across its tasks with a random value of 5% noise added.

Results. On average, HEYP's DC controller converges – defined as the DC controller taking no actions for 5 consecutive periods – in under 18 control periods 95% of the time (Table 3), and we found that it always converged. For EXP, FB15, and UNI, overage (excess HIPRI usage) was approximately 5% of the approval and no shortage (volume of usage we failed to admit at HIPRI) remained once the controller converged, but intermediate states exhibited higher overage (mean up to 14%) and shortage (mean up to 19%. The higher amounts of overage compared to shortage are a direct consequence of HEYP's bias to prefer it (§4.3). EM-5% exhibits higher amounts of overage – after converging the mean is up to 15% – due to the coarseness of demands. Each "elephant" task carries approximately 8% of the demand, and so the DC controller stops reacting once overage is twice this value.

C Scalability of DC controller

The faster the DC controller can react to changes in demand, the more accurate it can enforce admissions and avoid throttling. However, in today's clouds, individual tenants may run

Demand Dist.	EM-5%		EXP		FB15		UNI	
Init. % Downgraded	0	100	0	100	0	100	0	100
Convergence Time (#periods)	4.72	8.8	12.38	14.89	7.83	10.07	17.13	14.27
No. of QoS changes undone	3.89	5.2	6.56	7.05	3.98	4.77	9.58	5.39
No. of Oscillations	0.25	0.23	1.88	1.61	0.81	0.66	2.59	1.63
Final Overage (%)	15	11	4	4	6	4	3	3
Final Shortage (%)	0	0	0	0	0	0	0	0
Intermediate Overage (%)	23	2	10	2	14	2	7	2
Intermediate Shortage (%)	1	13	0	7	0	10	0	7

Table 3: Performance of HEYP's DC controller when downgrading part of a flowgroup across a range of simulated settings. To compute the value of each cell, we take the mean value across 100 monte carlo runs. Convergence time is measured in control periods. Overage and shortage are measured both once the controller has converged ("Final Overage") and during the period before convergence ("Intermediate Overage"). Compare with the number of QoS changes that are unintended against the number of QoS changes expected for each case (100, since we expect to downgrade or upgrade roughly half of the tasks in each case).

millions of tasks [67]. For this reason, we optimize the reaction time of our prototype's DC controller. The partitioning of traffic into HIPRI and LOPRI, done via caterpillar hashing and feedback control, is constant time. Therefore, the main scalability bottlenecks are in the collection of task-level usage, and the broadcasting of task-level QoS. We optimize the former by using threshold sampling [25], this enables over a 50× reduction in input data without sacrificing much accuracy. The latter is dependent on the rate of demand change, e.g., if the usage doubles from the approval over the course of 5 seconds, then the controller will need to downgrade approximately half of the tasks within that time frame. Tasks whose QoS is unchanged do not need to be contacted.

Evaluation Setup. To evaluate how quickly our prototype DC controller can react to changes in demand for a single flowgroup, we feed usage data for 1 million simulated tasks (the usage data is transmitted via RPCs to a real DC controller, but no such tasks exist). The flowgroup's demand cycles between 50% of its approval to 150% of its approval and back every 10 seconds. Each task carries one one-millionth of the flowgroup's overall demand, and we configure the system to sample usage from approximately 1.6% of tasks. The DC controller runs once every 400 ms.

Results. Figure 13 shows that HEYP's DC controller is able to respond to changes within 500 ms. If the DC controller perfectly eliminated over usage (or under usage) of HIPRI every iteration, this would imply that the DC controller would bound HIPRI usage to within 10% of the approval (for workloads which grow or shrink their demand by 10% of their approval every 500 ms). However, HEYP's feedback control requires more than one iteration to obtain perfect accuracy, as each iteration attempts to eliminate half of the error, and so the error may persist for several seconds.

The gradually increasing slope is a result of smearing the arrival times of usage data. In order to avoid alternating between periods of no work and overload at the DC controller, we configure each task to sleep a random period before transmitting their initial usage data to the DC controller. This spreads the load on the DC controller over time and enables it to react to a greater number of tasks, albeit with unequal response time.

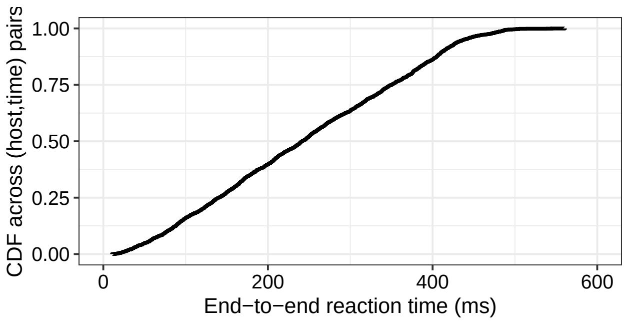


Figure 13: Delay between tasks sending usage to the DC controller and receiving a QoS assignment.

D Additional results for sensitivity analysis

Below are the percent of flowgroups with 99.9% approval satisfaction for the same settings as Table 2.

Legend: 🟥 0–20 🟧 20–40 🟨 40–60 🟩 60–80 🟩 80–100

Control Plane Speed	Rate of Demand Change (larger is faster)								
	Dynamic			Static			HEYP		
	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×
5× Faster	12	7	4	76	72	67	91	87	85
Normal	6	5	3	46	45	43	48	46	45
5× Slower	3	2	2	36	34	35	38	36	36

E Route allocation algorithm used in §5.1

Algorithm 3 provides a detailed description of the routing algorithm used by all approaches in §5.1. It is similar to the greedy algorithm used by B4 [40] and prioritizes satisfying any within-approval demands before above-approval ones.

```
Configuration parameters:
    Maximum number of paths (i.e. path budget) per flowgroup
Inputs:
    Approvals and demands per flowgroup
    Topology annotated with link capacities
Outputs (per flowgroup):
    Routes and admissions
---------------------------------------------------------------
// Initialization
1  $PathAdmissions \leftarrow \{\}$
// Start by satisfying within-approval demands
2  For all flowgroups $f$, $D_f \leftarrow \min(\text{demand}_f, \text{approvals}_f)$
// Main path allocation loop
3  while some flowgroup has positive demand and some link has positive capacity do
4      $CurPaths \leftarrow \{\}$
5      foreach flowgroup $f$ with $D_f > 0$ do
6          $p \leftarrow$ next shortest path that avoids links with no capacity
7          if no such $p$ exists or if adding $p$ to $Routes_f$ exceeds the path budget then
8              $D_f \leftarrow 0$
9          else
10             $CurPaths_f \leftarrow p$
11     end
12     Compute a max-min fair allocation of link capacity to satisfy $D$ using $CurPaths$
13     Add allocations to admissions and subtract from $D_f$
14     $PathAdmissions_{f,p} \leftarrow PathAdmissions_{f,p}$ + admission
15 end
// We have satisfied any within-approval demands (if possible), try to satisfy above-approval demands
16 Set $D_f \leftarrow \text{demand}_f - \sum_p PathAdmissions_{f,p}$
17 Repeat loop on Lines 3–15
18 foreach flowgroup $f$ do
19     if $demand_f = 0$ then
20         Use the shortest route and set admission to 0
21     else
22         Use routes in $PathAdmissions_f$ with each getting a share of traffic proportionate to the path admission
23         Set admission to $\sum_p PathAdmissions_{f,p}$
24 end
```

Algorithm 3: **Route computation algorithm.**

F Discrete-event simulation control logic

Our simulation in §5.1 diverges from the design described in §4 in two ways. First, route and admission computation are performed by two separate controllers (Algorithms 4 and 5) similar to B4 [40, 46]). Second, because the the network traces lack per-task data, the simulated DC controller can only partition traffic based on usage (Algorithm 6).

```
Inputs:
    Approvals and demands per flowgroup
    Topology annotated with link capacities
    Route allocation function (see Algorithm 3)
Outputs (per flowgroup): HIPRI and LOPRI routes
---------------------------------------------------------------
1. HIPRI routes, HIPRI admissions
   ← AllocateRoutes(approvals, approvals, link capacities)
2. Compute unused link capacity by deducting any link
   capacity consumed by the volume of each flowgroup's
   demand that is under the HIPRI admission
3. HIPRI routes, _ ← AllocateRoutes(demands - HIPRI
   admissions, approvals - HIPRI admissions, link capacities
   from Step 2)
```

Algorithm 4: **Allocating routes separately from admissions while accounting for any oversubscription caused by failures.**

```
Inputs:
    Approvals and demands per flowgroup
    HIPRI and LOPRI routes per flowgroup
    Topology annotated with link capacities
Outputs (per flowgroup): HIPRI and LOPRI admissions
---------------------------------------------------------------
1. Set the HIPRI admissions to a max-min fair allocation of
   bandwidth to satisfy approvals using the HIPRI routes
2. Compute unused link capacity by deducting any link
   capacity consumed by the volume of each flowgroup's
   demand that is under the HIPRI admission
3. Set the LOPRI admissions to a max-min fair allocation of
   bandwidth to satisfy any residual demand using the
   LOPRI routes
```

Algorithm 5: **Allocating admissions separately from routes while accounting for any oversubscription caused by failures.**

```
Inputs:
    Usage and admission per (flowgroup, QoS)
    Demand per flowgroup
    Current fraction of demand marked as HIPRI per flowgroup
Outputs (per flowgroup):
    New fraction of demand to mark as HIPRI
---------------------------------------------------------------
foreach flowgroup do
    $t \leftarrow \min(\text{demand}, \text{HIPRI} + \text{LOPRI admissions})$
    Set new HIPRI fraction to $\min(1, \text{HIPRI limit}/t)$
end
```

Algorithm 6: **Splitting traffic into QoS levels.**