Achieving High Utilization with Software-Driven WAN[*]

Extended Version; MSR-TR-2013-54

Chi-Yao Hong (UIUC) Srikanth Kandula Ratul Mahajan Ming Zhang
Vijay Gill Mohan Nanduri Roger Wattenhofer (ETH)

Microsoft

Abstract— We present SWAN, a system that boosts the utilization of inter-datacenter networks by centrally controlling when and how much traffic each service sends and frequently re-configuring the network's data plane to match current traffic demand. But done simplistically, these reconfigurations can also cause severe, transient congestion because different switches may apply updates at different times. We develop a novel technique that leverages a small amount of scratch capacity on links to apply updates in a provably congestion-free manner, without making any assumptions about the order and timing of updates at individual switches. Further, to scale to large networks in the face of limited forwarding table capacity, SWAN greedily selects a small set of entries that can best satisfy current demand. It updates this set without disrupting traffic by leveraging a small amount of scratch capacity in forwarding tables. Experiments using a testbed prototype and data-driven simulations of two production networks show that SWAN carries 60% more traffic than the current practice.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design
Keywords: Inter-DC WAN; software-defined networking

1. INTRODUCTION

The wide area network (WAN) that connects the datacenters (DC) is critical infrastructure for providers of online services such as Amazon, Google, and Microsoft. Many services rely on low-latency inter-DC communication for good user experience and on high-throughput transfers for reliability (e.g., when replicating updates). Given the need for high capacity—inter-DC traffic is a significant fraction of Internet traffic and rapidly growing [19]—and unique traffic characteristics, the inter-DC WAN is often a dedicated network, distinct from the WAN that connects with ISPs to reach end users [14]. It is an expensive resource, with amortized annual cost of 100s of millions of dollars, as it provides 100s of Gbps to Tbps of capacity over long distances.

However, providers are unable to fully leverage this investment today. Inter-DC WANs have extremely poor efficiency; the average utilization of even the busier links is 40-60%. One culprit is the lack of coordination among the services that use the network. Barring coarse, static limits in some cases, services send traffic whenever they want and however much they want. As a result, the network cycles through periods of peaks and troughs. Since it must be provisioned for peak usage to avoid congestion, the network is under-subscribed on average. Observe that network usage does not have to be this way if we can exploit the characteristics of inter-DC traffic. Some inter-DC services are delay-tolerant. We can tamp the cyclical behavior if such traffic is sent when the demand from other traffic is low. This coordination will boost average utilization and enable the network to either carry more traffic with the same capacity or use less capacity to carry the same traffic.[1]

Another culprit behind poor efficiency is the distributed resource allocation model of today, typically implemented using MPLS TE (Multiprotocol Label Switching Traffic Engineering) [4, 23]. In this model, no entity has a global view and ingress routers greedily select paths for their traffic. As a result, the network can get stuck in locally optimal routing patterns that are globally suboptimal [26].

We present SWAN (Software-driven WAN), a system that enables inter-DC WANs to carry significantly more traffic. By itself, carrying more traffic is straightforward—we can let loose bandwidth-hungry services. SWAN achieves high efficiency while meeting policy goals such as preferential treatment for higher-priority services and fairness among similar services. Per observations above, its two key aspects are *i*) globally coordinating the sending rates of services; and *ii*) centrally allocating network paths. Based on current service demands and network topology, SWAN decides how much traffic each service can send and configures the network's data plane to carry that traffic.

Maintaining high utilization requires frequent updates to the network's data plane, as traffic demand or network topology changes. A key challenge is to implement these updates without causing transient congestion that can hurt latency-sensitive traffic. The underlying problem is that the updates are not atomic as they require changes to multiple switches. Even if the before and after states are not congested, congestion can occur during updates if traffic that a link is supposed to carry after the update arrives before the traffic that is supposed to leave has left. The extent and duration of such congestion is worse when the network is busier and has larger RTTs (which lead to greater temporal disparity in the application of updates). Both these conditions hold for our setting, and we find that uncoordinated updates lead to severe congestion and heavy packet loss.

This challenge recurs in every centralized resource allocation scheme. MPLS TE's distributed resource allocation

[*] The conference version was published in *SIGCOMM'13*.

[1] In some networks, fault tolerance is another reason for low utilization; the network is provisioned such that there is ample capacity even after (common) failures. However, in inter-DC WANs, traffic that needs strong protection is a small subset of the overall traffic, and existing technologies can tag and protect such traffic in the face of failures (§2).

can make only a smaller class of "safe" changes; it cannot make coordinated changes that require one flow to move in order to free a link for use by another flow. Further, recent work on atomic updates, to ensure that no packet experiences a mix of old and new forwarding rules [22, 28], does not address our challenge. It does not consider capacity limits and treats each flow independently; congestion can still occur due to uncoordinated flow movements.

We address this challenge by first observing that it is impossible to update the network's data plane without creating congestion if all links are full. SWAN thus leaves "scratch" capacity s (e.g., 10%) at each link. We prove that this enables a congestion-free plan to update the network in at most $\lceil 1/s \rceil - 1$ steps. Each step involves a set of changes to forwarding rules at switches, with the property that there will be no congestion independent of the order and timing of those changes. We then develop an algorithm to find a congestion-free plan with the minimum number of steps. Further, SWAN does not waste the scratch capacity. Some inter-DC traffic is tolerant to small amounts of congestion (e.g., data replication with long deadlines). We extend our basic approach to use all link capacity while guaranteeing bounded-congestion updates for tolerant traffic and congestion-free updates for the rest.

Another challenge that we face is that fully using network capacity requires many forwarding rules at switches, to exploit many alternative paths through the network, but commodity switches support a limited number of forwarding rules.[2] Analysis of a production inter-DC WAN shows that the number of rules required to fully use its capacity exceeds the limits of even next generation SDN switches. We address this challenge by dynamically changing, based on traffic demand, the set of paths available in the network. On the same WAN, our technique can fully use network capacity with an order of magnitude fewer rules.

We develop a prototype of SWAN, and evaluate our approach through testbed experiments and simulations using traffic and topology data from two production inter-DC WANs. We find that SWAN carries 60% more traffic than MPLS TE and it comes within 2% of the traffic carried by an optimal method that assumes infinite rule capacity and incurs no update overhead. We also show that changes to network updates are quick, requiring only 1-3 steps.

While our work focuses on inter-DC WANs, many of its underlying techniques are useful for other WANs as well (e.g., ISP networks). We show that even without controlling how much traffic enters the network, an ability that is unique to the inter-DC context, our techniques for global resource and change management allow the network to carry 16-25% more traffic than MPLS TE.

2. BACKGROUND AND MOTIVATION

Inter-DC WANs carry traffic from a range of services, where a *service* is an activity across multiple hosts. Externally visible functionality is usually enabled by multiple internal services (e.g., search may use Web-crawler, index-builder, and query-responder services). Prior work [6] and our conversations with operators reveal that services fall into three broad types, based on their performance requirements.

Interactive services are in the critical path of end user experience. An example is when one DC contacts another in the process of responding to a user request because not all information is available in the first DC. Interactive traffic is highly sensitive to loss and delay; even small increases in response time (100 ms) degrade user experience [34].

Elastic services are not in the critical path of user experience but still require timely delivery. An example is replicating a data update to another DC. Elastic traffic requires delivery within a few seconds or minutes. The consequences of delay vary with the service. In the replication example, the risk is loss of data if a failure occurs or that a user will observe data inconsistency.

Background services conduct maintenance and provisioning activities. An example is copying all the data of a service to another DC for long-term storage or as a precursor to running the service there. Such traffic tends to be bandwidth hungry. While it has no explicit deadline or a long deadline, it is still desirable to complete transfers as soon as possible—delays lower business agility and tie up expensive server resources.

In terms of overall volumes, interactive traffic is the smallest subset and background traffic is the largest.

2.1 Current traffic engineering practice

Many WANs are operated using MPLS TE today. To effectively use network capacity, MPLS TE spreads traffic across a number of tunnels between ingress-egress router pairs. Ingress routers split traffic, typically equally using equal cost multipath routing (ECMP), across the tunnels to the same egress. They also estimate the traffic demand for each tunnel and find network paths for it using the constrained shortest path first (CSPF) algorithm, which identifies the shortest path that can accommodate the tunnel's traffic (subject to priorities; see below).

With MPLS TE, service differentiation can be provided using two mechanisms. First, tunnels are assigned priorities and different types of services are mapped to different tunnels. Higher priority tunnels can displace lower priority tunnels and thus obtain shorter paths; the ingress routers of displaced tunnels must then find new paths. Second, packets carry differentiated services code point (DSCP) bits in the IP header. Switches map different bits to different priority queues, which ensures that packets are not delayed or dropped due to lower-priority traffic; they may still be delayed or dropped due to equal or higher priority traffic. Switches typically have only a few priority queues (4–8).

2.2 Problems of MPLS TE

Inter-DC WANs suffer from two key problems today.

Poor efficiency: The amount of traffic the WAN carries tends to be low compared to capacity. For a production inter-DC WAN, which we call IDN (§6.1), we find that the average utilization of half the links is under 30% and of three in four links is under 50%.

Two factors lead to poor efficiency. First, services send whenever and however much traffic they want, without regard to the current state of the network or other services. This lack of coordination leads to network swinging between over- and under-subscription. Figure 1a shows the load over a day on a busy link in IDN. Assuming capacity matches peak usage (a common provisioning model to avoid congestion), the average utilization on this link is under 50%. Thus,

[2] The limit stems from the amount of fast, expensive memory in switches. It is not unique to OpenFlow switches; number of tunnels that MPLS routers support is also limited [2].

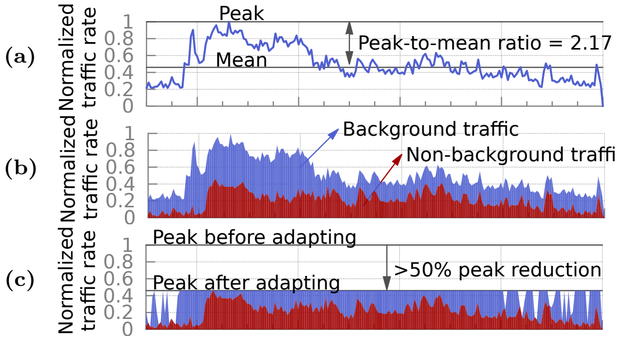
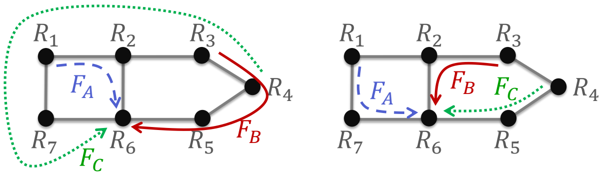


Figure 1: Illustration of poor utilization. (a) Daily traffic pattern on a busy link in a production inter-DC WAN. (b) Breakdown based on traffic type. (c) Reduction in peak usage if background traffic is dynamically adapted.



(a) Local path selection **(b) Globally optimal paths**

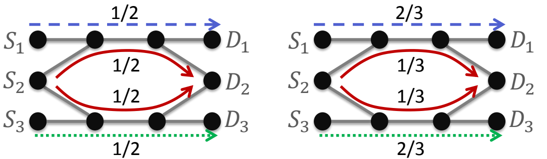
Figure 2: Inefficient routing due to local allocation.

half the provisioned capacity is wasted. This inefficiency is not fundamental but can be remedied by exploiting traffic characteristics. As a simple illustration, Figure 1b separates background traffic. Figure 1c shows that the same total traffic can fit in half the capacity if background traffic is adapted to use capacity left unused by other traffic.

Second, the local, greedy resource allocation model of MPLS TE is inefficient. Consider Figure 2 in which each link can carry at most one flow. If the flows arrive in the order F_A, F_B, and F_C, Figure 2a shows the path assignment with MPLS TE: F_A is assigned to the top path which is one of the shortest paths; when F_B arrives, it is assigned to the shortest path with available capacity (CSPF); and the same happens with F_C. Figure 2b shows a more efficient routing pattern with shorter paths and many links freed up to carry more traffic. Such an allocation requires non-local changes, e.g., moving F_A to the lower path when F_B arrives.

Partial solutions for such inefficiency exist. Flows can be split across two tunnels, which would divide F_A across the top and bottom paths, allowing half of F_B and F_C to use direct paths; a preemption strategy that prefers shorter paths can also help. But such strategies do not address the fundamental problem of local allocation decisions [26].

Poor sharing: Inter-DC WANs have limited support for flexible resource allocation. For instance, it is difficult to be fair across services or favor some services over certain paths. When services compete today, they typically obtain throughput proportional to their sending rate, an undesirable outcome (e.g., it creates perverse incentives for service developers). Mapping each service onto its own queue at routers can alleviate problems but the number of services (100s) far exceeds the number of available router queues. Even if we had infinite queues and could ensure fairness on the data plane, network-wide fairness is not possible without controlling which flows have access to which paths. Consider Figure 3 in which each link has unit capacity and each service ($S_i \rightarrow D_i$) has unit demand. With link-level fairness, $S_2 \rightarrow D_2$ gets twice the throughput of other services. As we show, flexible sharing can be implemented with a limited number of queues by carefully allocating paths to traffic and control the sending rate of services.



(a) Link-level **(b) Network-wide**

Figure 3: Link-level fairness $\neq$ network-wide fairness.

3. SWAN OVERVIEW AND CHALLENGES

Our goal is to carry more traffic and support flexible network-wide sharing. Driven by inter-DC traffic characteristics, SWAN supports two types of sharing policies. First, it supports a small number of priority classes (e.g., Interactive > Elastic > Background) and allocates bandwidth in strict precedence across these classes, while preferring shorter paths for higher classes. Second, within a class, SWAN allocates bandwidth in a max-min fair manner.

SWAN has two basic components that address the fundamental shortcomings of the current practice. It coordinates the network activity of services and uses centralized resource allocation. Abstractly, it works as:

1. All services, except interactive ones, inform the SWAN controller of their demand between pairs of DCs. Interactive traffic is sent like today, without permission from the controller, so there is no delay.
2. The controller, which has an up-to-date, global view of the network topology and traffic demands, computes how much each service can send and the network paths that can accommodate the traffic.
3. Per SDN paradigm, the controller directly updates the forwarding state of the switches. We use OpenFlow switches, though any switch that permits direct programming of forwarding state (e.g., MPLS Explicit Route Objects [3]) may be used.

While the architecture is conceptually simple, we must address three challenges to realize this design. First, we need a scalable algorithm for global allocation that maximizes network utilization subject to constraints on service priority and fairness. Best known solutions are computationally intensive as they solve long sequences of linear programs (LP) [9, 25]. Instead, SWAN uses a more practical approach that is approximately fair with provable bounds and close to optimal in practical scenarios (§6).

Second, atomic reconfiguration of a distributed system of switches is hard to engineer. Network forwarding state needs updating in response to changes in the traffic demand or network topology. Lacking WAN-wide atomic changes, the network can drop many packets due to transient congestion even if both the initial and final configurations are uncongested. Consider Figure 4 in which each flow is 1 unit and each link's capacity is 1.5 units. Suppose we want to change the network's forwarding state from Figure 4a to 4b, perhaps to accommodate a new flow from R_2 to R_4. This change requires changes to at least two switches. Depending on the order in which the switch-level changes occur, the network reaches the states in Figures 4c or 4d, which have a heavily

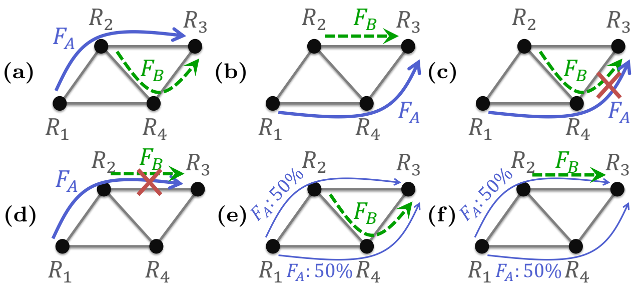


Figure 4: Illustration of congestion-free updates. Each flow's size is 1 unit and each link's capacity is 1.5 units. Changing from state (a) to (b) may lead to congested states (c) or (d). A congestion-free update sequence is (a)⇝(e)⇝(f)⇝(b).

congested link and can significantly hurt TCP flows as many packets may be lost in a burst.

To avoid congestion during network updates, SWAN computes a multi-step *congestion-free* transition plan. Each step involves one or more changes to the state of one or more switches, but irrespective of the order in which the changes are applied, there will be no congestion. For the reconfiguration in Figure 4, a possible congestion-free plan is: *i*) move half of F_A to the lower path (Figure 4e); *ii*) move F_B to the upper path (Figure 4f); and *iii*) move the remaining half of F_A to the lower path (Figure 4b).

A congestion-free plan may not always exist, and even if it does, it may be hard to find or involve a large number of steps. SWAN leaves scratch capacity of $s \in [0, 50\%]$ on each link, which guarantees that a transition plan exists with at most $\lceil 1/s \rceil - 1$ steps (which is 9 if s=10%). We then develop a method to find a plan with the minimal number of steps. In practice, it finds a plan with 1-3 steps when s=10%.

Further, instead of wasting scratch capacity, SWAN allocates it to background traffic. Overall, it guarantees that non-background traffic experiences no congestion during transitions, and the congestion for background traffic is bounded (configurable).

Third, switch hardware supports a limited number of forwarding rules, which makes it hard to fully use network capacity. For instance, if a switch has six distinct paths to a destination but supports only four rules, a third of paths cannot be used. Our analysis of a production inter-DC WAN illustrates the challenge. If we use k-shortest paths between each pair of switches (as in MPLS), fully using this network's capacity requires k=15. Installing these many tunnels needs up to 20K rules at switches (§6.5), which is beyond the capabilities of even next-generation SDN switches; the Broadcom Trident2 chipset will support 16K OpenFlow rules [32]. The current-generation switches in our testbed support 750 rules.

To fully exploit network capacity with a limited number of rules, we are motivated by how the working set of a process is often a lot smaller than the total memory it uses. Similarly, not all tunnels are needed at all times. Instead, as traffic demand changes, different sets of tunnels are most suitable. SWAN dynamically identifies and installs these tunnels. Our dynamic tunnel allocation method, which uses an LP, is effective because the number of non-zero variables in a basic solution for any LP is fewer than the number of constraints [24]. In our case, we will see that variables include the fraction of a DC-pair's traffic that is carried over a tunnel and the number of constraints is roughly the number of priority classes times the number of DC pairs. Because SWAN supports three priority classes, we obtain three tunnels with non-zero traffic per DC pair on average, which is much less than the 15 required for a non-dynamic solution.

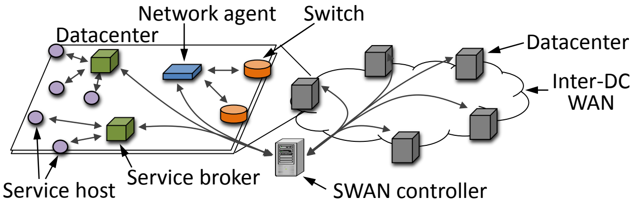


Figure 5: Architecture of SWAN.

Dynamically changing rules introduces another wrinkle for network reconfiguration. To not disrupt traffic, new rules must be added before the old rules are deleted; otherwise, the traffic that is using the to-be-deleted rules will be disrupted. Doing so requires some rule capacity to be kept vacant at switches to accommodate the new rules; done simplistically, up to half of the rule capacity must be kept vacant [28], which is wasteful. SWAN sets aside a small amount of scratch space (e.g., 10%) and uses a multi-stage approach to change the set of rules in the network.

4. SWAN DESIGN

Figure 5 shows the architecture of SWAN. A logically centralized *controller* orchestrates all activity. Each non-interactive service has a *broker* that aggregates demands from the hosts and apportions the allocated rate to them. One or more *network agents* intermediate between the controller and the switches. This architecture provides scale—by providing parallelism where needed—and choice—each service can implement a rate allocation strategy that fits.

Service hosts and brokers collectively estimate the service's current demand and limit it to the rate allocated by the controller. Our current implementation draws on distributed rate limiting [5]. A shim in the host OS estimates its demand to each remote DC for the next T_h=10 seconds and asks the broker for an allocation. It uses a token bucket per remote DC to enforce the allocated rate and tags packets with DSCP bits to indicate the service's priority class.

The service broker aggregates demand from hosts and updates the controller every T_s=5 minutes. It apportions its allocation from the controller piecemeal, in time units of T_h, to hosts in a proportionally fair manner. This way, T_h is the maximum time that a newly arriving host has to wait before starting to transmit. It is also the maximum time a service takes to change its sending rate to a new allocation. Brokers that suddenly experience radically larger demands can ask for more any time; the controller does a lightweight computation to determine how much of the additional demand can be carried without altering network configuration.

Network agents track topology and traffic with the aid of switches. They relay news about topology changes to the controller right away and collect and report information about traffic, at the granularity of OpenFlow rules, every T_a=5 minutes. They are also responsible for reliably updating switch rules as requested by the controller. Before returning success, an agent reads the relevant part of the switch rule table to ensure that the changes have been successfully applied.

Controller uses the information on service demands and network topology to do the following every T_c=5 minutes.

1. Compute the service allocations and forwarding plane configuration for the network (§4.1, §4.2).
2. Signal new allocations to services whose allocation has decreased. Wait for T_h seconds for the service to lower its sending rate.
3. Change the forwarding state (§4.3) and then signal the new allocations to services whose allocation has increased.

4.1 Forwarding plane configuration

SWAN uses label-based forwarding. Doing so reduces forwarding complexity; the complex classification that may be required to assign a label to traffic is done just once, at the source switch. Remaining switches simply read the label and forward the packet based on the rules for that label. We use VLAN IDs as labels.

Ingress switches split traffic across multiple tunnels (labels). We propose to implement unequal splitting, which leads to more efficient allocation [13], using *group tables* in the OpenFlow pipeline. The first table maps the packet, based on its destination and other characteristics (e.g., DSCP bits), to a group table. Each group table consists of the set of tunnels available and a weight assignment that reflects the ratio of traffic to be sent to each tunnel. Conversations with switch vendors indicate that most will roll out support for unequal splitting. When such support is unavailable, SWAN uses traffic profiles to pick boundaries in the range of IP addresses belonging to a DC such that splitting traffic to that DC at these boundaries will lead to the desired split. Then, SWAN configures rules at the source switch to map IP destination spaces to tunnels. Our experiments with traffic from a production WAN show that implementing unequal splits in this way leads to a small amount of error (less than 2%).

4.2 Computing service allocations

When computing allocated rate for services, our goal is to maximize network utilization subject to service priorities and approximate max-min fairness among same-priority services. The allocation process must be scalable enough to handle WANs with 100s of switches.

Inputs: The allocation uses as input the service demands d_i between pairs of DCs. While brokers report the demand for non-interactive services, SWAN estimates the demand of interactive services (see below). We also use as input the paths (tunnels) available between a DC pair. Running an unconstrained multi-commodity problem could result in allocations that require many rules at switches. Since a DC pair's traffic could flow through any link, every switch may need rules to split every pair's traffic across its outgoing ports. Constraining usable paths avoids this possibility and also simplifies data plane updates (§4.3). But it may lead to lower overall throughput. For our two production inter-DC WANs, we find that using the 15 shortest paths between each pair of DCs results in negligible loss of throughput.

Allocation LP: Figure 6 shows the LP used in SWAN. At the core is the MCF (multi-commodity flow) function that maximizes the overall throughput while preferring shorter paths; ϵ is a small constant and tunnel weights w_j are proportional to latency. s_{Pri} is the fraction of scratch link capacity that enables congestion-managed network updates; it can be different for different priority classes (§4.3). The SWAN Allocation function allocates rate by invoking MCF separately for classes in priority order. After a class is allocated, its allocation is removed from remaining link capacity.

```
Inputs:
    $d_i$:       flow demands for source destination pair $i$
    $w_j$:       weight of tunnel $j$ (e.g., latency)
    $c_l$:       capacity of link $l$
    $s_{Pri}$:   scratch capacity ($[0, 50\%]$) for class $Pri$
    $I_{j,l}$:   1 if tunnel $j$ uses link $l$ and 0 otherwise
Outputs:
    $b_i = \sum_j b_{i,j}$:   $b_i$ is allocation to flow $i$; $b_{i,j}$ over tunnel $j$
--------------------------------------------------------------------
Func: SWAN Allocation:
$\forall$ links $l$ : $c_l^{\text{remain}} \leftarrow c_l$; // remaining link capacity
for $Pri = \text{Interactive}, \text{Elastic}, \ldots, \text{Background}$ do
    $\{b_i\} \leftarrow$ Throughput Maximization
                       Approx. Max-Min Fairness   $(Pri, \{c_l^{\text{remain}}\})$;
    $c_l^{\text{remain}} \leftarrow c_l^{\text{remain}} - \sum_{i,j} b_{i,j} \cdot I_{j,l}$;
--------------------------------------------------------------------
Func: Throughput Maximization$(Pri, \{c_l^{\text{remain}}\})$:
return MCF$(Pri, \{c_l^{\text{remain}}\}, 0, \infty, \emptyset)$;
--------------------------------------------------------------------
Func: Approx. Max-Min Fairness$(Pri, \{c_l^{\text{remain}}\})$:
// Parameters $\alpha$ and $U$ trade-off unfairness for runtime
// $\alpha > 1$ and $0 < U \leq \min(\text{fairrate}_i)$
$T \leftarrow \lceil \log_\alpha \frac{\max(d_i)}{U} \rceil$; $F \leftarrow \emptyset$;
for $k = 1 \ldots T$ do
    foreach $b_i \in$ MCF$(Pri, \{c_l^{\text{remain}}\}, \alpha^{k-1}U, \alpha^k U, F)$ do
        if $i \notin F$ and $b_i < \min(d_i, \alpha^k U)$ then
            $F \leftarrow F + \{i\}$; $f_i \leftarrow b_i$; // flow saturated
return $\{f_i : i \in F\}$;
--------------------------------------------------------------------
Func: MCF$(Pri, \{c_l^{\text{remain}}\}, b_{Low}, b_{High}, F)$:
//Allocate rate $b_i$ for flows in priority class $Pri$
  maximize     $\sum_i b_i - \epsilon(\sum_{i,j} w_j \cdot b_{i,j})$
  subject to   $\forall i \notin F : b_{Low} \leq b_i \leq \min(d_i, b_{High})$;
               $\forall i \in F : b_i = f_i$;
               $\forall l : \sum_{i,j} b_{i,j} \cdot I_{j,l} \leq \min\{c_l^{\text{remain}}, (1 - s_{Pri})c_l\}$;
               $\forall (i,j) : b_{i,j} \geq 0$.
```

Figure 6: Computing allocations over a set of tunnels.

It is easy to see that our allocation respects traffic priorities. By allocating demands in priority order, SWAN also ensures that higher priority traffic is likelier to use shorter paths. This keeps the computation simple because MCF's time complexity increases manifold with the number of constraints. While, in general, it may reduce overall utilization, in practice, SWAN achieves nearly optimal utilization (§6).

Max-min fairness can be achieved iteratively: maximize the minimal flow rate allocation, freeze the minimal flows and repeat with just the other flows [25]. However, solving such a long sequence of LPs is rather costly in practice, so we devised an approximate solution instead. SWAN provides approximated max-min fairness for services in the same class by invoking MCF in T steps, with the constraint that at step k, flows are allocated rates in the range $\left[\alpha^{k-1}U, \alpha^k U\right]$, but no more than their demand. See Fig. 6, function APPROX. MAX-MIN FAIR. A flow's allocation is *frozen* at step k when it is allocated its full demand d_i at that step or it receives a rate smaller than $\alpha^k U$ due to capacity constraints. If r_i and b_i are the max-min fair rate and the rate allocated to flow i, we can prove that this is an α-approximation algorithm, i.e., $b_i \in \left[\frac{r_i}{\alpha}, \alpha r_i\right]$ (Theorem 1 in Appendix).

Many proposals exist to combine network-wide max-min fairness with high throughput. A recent one offers a search function that is shown to empirically reduce the number of

LPs that need to be solved [9]. Our contribution is showing that one can trade-off the number of LP calls and the degree of unfairness. The number of LPs we solve per priority is T; with $\max d_i$=10Gbps, U=10Mbps and α=2, we get T=10. We find that SWAN's allocations are highly fair and take less than a second combined for all priorities (§6). In contrast, Danna et al. report running times of over a minute [9].

Finally, our approach can be easily extended to other policy goals such as virtually dedicating capacity to a flow over certain paths and weighted max-min fairness.

Interactive service demands: SWAN estimates an interactive service's demand based on its average usage in the last five minutes. To account for prediction errors, we inflate the demand based on the error in past estimates (mean plus two standard deviations). This ensures that enough capacity is set aside for interactive traffic. So that inflated estimates do not let capacity go unused, when allocating rates to background traffic, SWAN adjusts available link capacities as if there was no inflation. If resource contention does occur, priority queueing at switches protects interactive traffic.

Post-processing: The solution produced by the LP may not be feasible to implement; while it obeys link capacity concerns, it disregards rule count limits on switches. Directly including these limits in the LP would turn the LP into an Integer LP making it intractably complex. Hence, SWAN post-processes the output of the LP to fit into the number of rules available.

Finding the set of tunnels with a given size that carries the most traffic is NP-complete [13]. SWAN uses the following heuristic: first pick at least the smallest latency tunnel for each DC pair, prefer tunnels that carry more traffic (as per the LP's solution) and repeat as long as more tunnels can be added without violating rule count constraint m_j at switch j. If M_j is the number of tunnels that switch j can store and $\lambda \in [0, 50\%]$ is the scratch space needed for rule updates (§4.3.2), m_j=$(1-\lambda)M_j$. In practice, we found that $\{m_j\}$ is large enough to ensure at least two tunnels per DC pair (§6.5). However, the original allocation of the LP is no longer valid since only a subset of the tunnels are selected due to rule limit constraints. We thus re-run the LP with only the chosen tunnels as input. The output of this run has both high utilization and is implementable in the network.

To further speed-up allocation computation to work with large WANs, SWAN uses two strategies. First, it runs the LP at the granularity of DCs instead of switches. DCs have at least 2 WAN switches, so a DC-level LP has at least 4x fewer variables and constraints (and the complexity of an LP is at least quadratic in this number). To map DC-level allocations to switches, we leverage the symmetry of inter-DC WANs. Each WAN switch in a DC gets equal traffic from inside the DC as border routers use ECMP for outgoing traffic. Similarly, equal traffic arrives from neighboring DCs because switches in a DC have similar fan-out patterns to neighboring DCs. This symmetry allows traffic on each DC-level link (computed by the LP) to be spread equally among the switch-level links between two DCs. However, symmetry may be lost during failures; we describe how SWAN handles failures in §4.4.

Second, during allocation computation, SWAN aggregates the demands from all services in the same priority class between a pair of DCs. This reduces the number of flows that the LP has to allocate by a factor that equals the number of services, which can run into 100s. Given the per DC-pair allocation, we divide it among individual services in a max-min fair manner.

$$\textbf{Inputs:}\ \begin{cases} \quad q, & \text{sequence length}\\ b^0_{i,j} = b_{i,j}, & \text{initial configuration}\\ b^q_{i,j} = b'_{i,j}, & \text{final configuration}\\ \quad c_l, & \text{capacity of link } l\\ \quad I_{jl}, & \text{indicates if tunnel } j \text{ using link } l \end{cases}$$

$$\textbf{Outputs:}\ \{b^a_{i,j}\}\ \ \forall a \in \{1, \dots q\} \text{ if feasible}$$

$$\begin{array}{ll} \text{maximize} & c_{\text{margin}}\ \text{// remaining capacity margin}\\ \text{subject to} & \forall i, a : \sum_j b^a_{i,j} = b_i;\\ & \forall l, a : c_l \geq \sum_{i,j} \max(b^a_{i,j}, b^{a+1}_{i,j}) \cdot I_{j,l} + c_{\text{margin}};\\ & \forall (i,j,a) : b^a_{i,j} \geq 0;\ c_{\text{margin}} \geq 0; \end{array}$$

Figure 7: LP to find if a congestion-free update sequence of length q exists.

4.3 Updating forwarding state

To keep the network highly utilized, its forwarding state must be updated as traffic demand and network topology change. Our goal is to enable forwarding state updates that are not only congestion-free but also quick; the more agile the updates the better one can utilize the network. One can meet these goals trivially, by simply pausing all data movement on the network during a configuration change. Hence, an added goal is that the network continue to carry significant traffic during updates.[3]

Forwarding state updates are of two types: changing the distribution of traffic across available tunnels and changing the set of tunnels available in the network. We describe below how we make each type of change.

4.3.1 Updating traffic distribution across tunnels

Given two congestion-free configurations with different traffic distributions, we want to update the network from the first configuration to the second in a *congestion-free* manner. More precisely, let the current network configuration be C=$\{b_{ij} : \forall(i,j)\}$, where b_{ij} is the traffic of flow i over tunnel j. We want to update the network's configuration to C'=$\{b'_{ij} : \forall(i,j)\}$. This update can involve moving many flows, and when an update is applied, the individual switches may apply the changes in any order. Hence, many transient configurations emerge, and in some, a link's load may be much higher than its capacity. We want to find a sequence of configurations (C=$C_0, \dots, C_k$=C') such that no link is overloaded in any configuration. Further, no link should be overloaded when moving from C_i to C_{i+1} regardless of the order in which individual switches apply their updates.

In arbitrary cases congestion-free update sequences do not exist; when all links are full, any first move will congest at least one link. However, given the scratch capacity that we engineered on each link (s_{Pri}; §4.2), we show that there exists a congestion-free sequence of updates of length no more than $\lceil 1/s \rceil$−1 steps (Theorem 2 in Appendix). The constructive proof of this theorem yields an update sequence with exactly $\lceil 1/s \rceil$−1 steps. But shorter sequences may exist and are desirable because they will lead to faster updates.

We use an LP-based algorithm to find the sequence with the minimal number of steps. Figure 7 shows how to examine whether a feasible sequence of q steps exists. We vary

[3] Network updates can cause packet re-ordering. In this work, we assume that switch-level (e.g., FLARE [17]) or host-level mechanisms (e.g., reordering robust TCP [35]) are in place to ensure that applications are not hurt.

q from 1 to $\lceil 1/s \rceil - 1$ in increments of 1. The key part in the LP is the constraint that limits the worst case load on a link during an update to be below link capacity. This load is $\sum_{i,j} \max(b^a_{i,j}, b^{a+1}_{i,j}) I_{j,l}$ at step a; it happens when none of the flows that will decrease their contribution have done so, but all flows that will increase their contribution have already done so. If q is feasible, the LP outputs $C_a = \{b^a_{i,j}\}$, for $a = (1, \ldots, q-1)$, which represent the intermediate configurations that form a congestion-free update sequence.

From congestion-free to bounded-congestion: We showed above that leaving scratch capacity on each link facilitates congestion-free updates. If there exists a class of traffic that is tolerant to moderate congestion (e.g., background traffic), then scratch capacity need not be left idle; we can fully use link capacities with the caveat that transient congestion will only be experienced by traffic in this class. To realize this, when computing flow allocations (§4.2), we use $s_{Pri} = s > 0$ for interactive and elastic traffic, but set $s_{Pri} = 0$ for background traffic (which is allocated last). Thus, link capacity can be fully used, but no more than $(1-s)$ fraction is used by non-background traffic. Just this, however, is not enough: since links are no longer guaranteed to have slack there may not be a congestion-free solution within $\lceil \frac{1}{s} \rceil - 1$ steps. To remedy this, we replace the per-link capacity constraint in Figure 7 with two constraints, one to ensure that the worst-case traffic on a link from all classes is no more than $(1+\eta)$ of link capacity ($\eta \in [0, 50\%]$) and another to ensure that the worst-case traffic due to the non-background traffic is below link capacity. In this case, we prove that *i*) there is a feasible solution within $\max(\lceil 1/s \rceil - 1, \lceil 1/\eta \rceil)$ steps (Theorem 3 in Appendix) such that *ii*) the non-background traffic never encounters loss and *iii*) the background traffic experiences no more than an η fraction loss. Based on this result, we set $\eta = \frac{s}{1-s}$ in SWAN, which ensures the same $\lceil \frac{1}{s} \rceil - 1$ bound on steps as before.

4.3.2 *Updating tunnels*

To update the set of tunnels in the network from P to P', SWAN first computes a sequence of tunnel-sets ($P = P_0, \ldots, P_k = P'$) that each fit within rule limits of switches. Second, for each set, it computes how much traffic from each service can be carried (§4.2). Third, it signals services to send at a rate that is minimum across all tunnel-sets. Fourth, after T_h=10 seconds when services have changed their sending rate, it starts executing tunnel changes as follows. To go from set P_i to P_{i+1}: *i*) add tunnels that are in P_{i+1} but not in P_i—the computation of tunnel-sets (described below) guarantees that this will not violate rule count limits; *ii*) change traffic distribution, using bounded-congestion updates, to what is supported by P_{i+1}, which frees up the tunnels that are in P_i but not in P_{i+1}; *iii*) delete these tunnels. Finally, SWAN signals to services to start sending at the rate that corresponds to P'.

We compute the interim tunnel-sets as follows. Let P^{add}_i and P^{rem}_i be the set of tunnels that remain be added and removed, respectively, at step i. Initially, $P^{add}_0 = P' - P$ and $P^{rem}_0 = P - P'$. At each step i, we first pick a subset $p^a_i \subseteq P^{add}_i$ to add and a subset $p^r_i \subseteq P^{rem}_i$ to remove. We then update the tunnel sets as: $P_{i+i} = (P_i \cup p^a_i) - p^r_i$, $P^{add}_{i+1} = P^{add}_i - p^a_i$, and $P^{rem}_{i+1} = P^{rem}_i - p^r_i$. The process ends when P^{add}_i and P^{rem}_i are empty (at which point P_i will be P').

At each step, we also maintain the invariant that P_{i+1}, which is the next set of tunnels that will be installed in the network, leaves λM_j rule space free at every switch j. We achieve this by picking the maximal set p^a_i such that the tunnels in $p^a_0 \cup \cdots \cup p^a_i$ fit within t^{add}_i rules and the minimal set p^r_i such that the tunnels that remain to be removed ($P^{rem}_i - p^r_i$) fit within t^{rem}_i rules. The value of t^{add}_i increases with i and that of t^{rem}_i decreases with i; they are defined more precisely in Theorem 4 in Appendix. Within the size constraint, when selecting p^a_i, SWAN prefers tunnels that will carry more traffic in the final configuration (P') and those that transit through fewer switches. When selecting p^r_i, it prefers tunnels that carry less traffic in P_i and those that transit through more switches. This biases SWAN towards finding interim tunnel-sets that carry more traffic and use fewer rules.

We show that the algorithm above requires at most $\lceil 1/\lambda \rceil - 1$ steps and satisfies the rule count constraints (Theorem 4 in Appendix). At interim steps, some services may get an allocation that is lower than that in P or P'. The problem of finding interim tunnel-sets in which no service's allocation is lower than the initial and final set, given link capacity constraints, is NP-hard. (Even much simpler problems related to rule-limits are NP-hard [13]). In practice, however, services rarely experience short-term reductions (§6.6). Also, since both P and P' contain a common core in which there is at least one common tunnel between each DC-pair (per our tunnel selection algorithm; §4.2), basic connectivity is always maintained during transitions, which in practice suffices to carry at least all of the interactive traffic.

4.4 Handling failures

Gracefully handling failures is an important part of a global resource controller. We outline how SWAN handles failures. Link and switch failures are detected and communicated to the controller by network agents, in response to which the controller immediately computes new allocations. Some failures can break the symmetry in topology that SWAN leverages for scalable computation of allocation. When computing allocations over an asymmetric topology, the controller expands the topology of impacted DCs and computes allocations at the switch level directly.

Network agents, service brokers, and the controller have backup instances that take over when the primary fails. For simplicity, the backups do not maintain state but acquire what is needed upon taking over. Network agents query the switches for topology, traffic, and current rules. Service brokers wait for T_h (10 seconds), by which time all hosts would have contacted them. The controller queries the network agents for topology, traffic, and current rule set, and service brokers for current demand. Further, hosts stop sending traffic when they are unable to contact the (primary and secondary) service broker. Service brokers retain their current allocation when they cannot contact the controller. In the period between the primary controller failing and the backup taking over, the network continues to forward traffic as last configured.

4.5 Prototype implementation

We have developed a SWAN prototype that implements all the elements described above. The controller, service brokers and hosts, and network agents communicate with each other using RESTful APIs. We implemented network agents using the Floodlight OpenFlow controller [11], which allows SWAN to work with commodity OpenFlow switches.

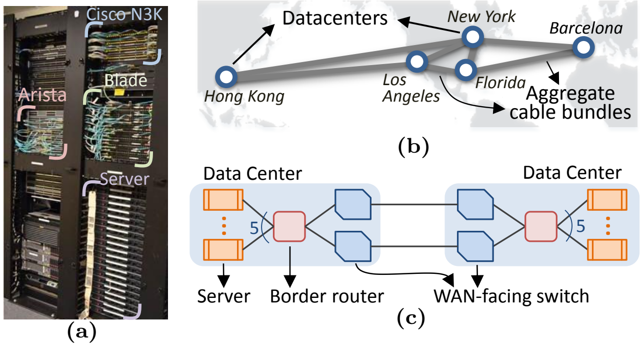


Figure 8: Our testbed. (a) Partial view of the equipment. (b) Emulated DC-level topology. (c) Closer look at physical connectivity for a pair of DC.

We use the QoS features in Windows Server 2012 to mark DSCP bits in outgoing packets and rate limit traffic using token buckets. We configure priority queues per class in switches. Based on our experiments (§6), we set s=10% and λ=10% in our prototype.

5. TESTBED-BASED EVALUATION

We evaluate SWAN on a modest-sized testbed. We examine the efficiency and the value of congestion-controlled updates using today's OpenFlow switches and under TCP dynamics. We will extend our evaluation to the scale of today's inter-DC WANs in §6.

5.1 Testbed and workload

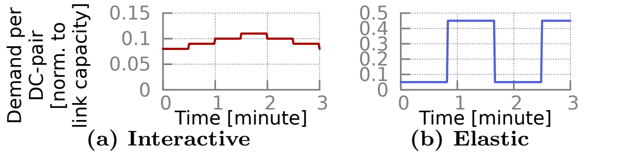
Our testbed emulates an inter-DC WAN with 5 DCs spread across three continents (Figure 8). Each DC has: *i*) two WAN-facing switches; *ii*) 5 servers per DC, where each server has a 1G Ethernet NIC and acts as 25 virtual hosts; and *iii*) an internal router that splits traffic from the hosts over the WAN switches. A logical link between DCs is two physical links between their WAN switches. WAN switches are a mix of Arista 7050Ts and IBM Blade G8264s, and routers are a mix of Cisco N3Ks and Juniper MX960s. The SWAN controller is in New York, and we emulate control message delays based on geographic distances.

In our experiment, every DC pair has a demand in each priority class. The demand of the Background class is infinite, whereas Interactive and Elastic demands vary with a period of 3-minutes as per the patterns shown in Figure 9. Each DC pair has a different phase, i.e., their demands are not synchronized. We picked these demands because they have sudden changes in quantity and spatial characteristics to stress SWAN. The actual traffic per {DC-pair, class} consists of 100s of TCP flows. Our switches do not support unequal splitting, so we insert appropriate rules into the switches to split traffic as needed based on IP headers.

We set T_s and T_c, the service demand and network update frequencies, to one minute, instead of five, to stress-test SWAN's dynamic behavior.

5.2 Experimental results

Efficiency: Figure 10 shows that SWAN closely approximates the throughput of an optimal method. For each 1-min interval, this method computes service rates using a multi-class, multi-commodity flow problem that is not constrained by the set of available tunnels or rule count limits. It's prediction of interactive traffic is perfect, it has no overhead due to network updates, and it can modify service rates instantaneously.



(a) Interactive (b) Elastic

Figure 9: Demand patterns for testbed experiments.

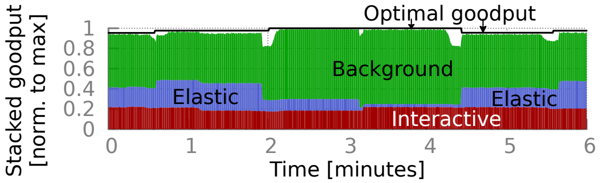
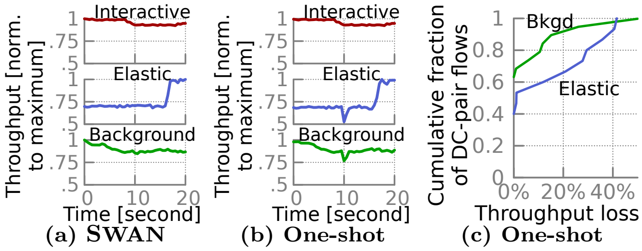


Figure 10: SWAN achieves near-optimal throughput.



(a) SWAN (b) One-shot (c) One-shot

Figure 11: Updates in SWAN do not cause congestion.

Overall, we see that SWAN closely approximates the optimal method. The dips in traffic occur during updates because we ask services whose new allocations are lower to reduce their rates, wait for T_h=10 seconds, and then ask services with higher allocations to increase their rate. The impact of these dips is low in practice when there are more flows and the update frequency is 5 minutes (§6.6).

Congestion-controlled updates: Figure 11a zooms in on an example update. A new epoch starts at zero and the throughput of each class is shown relative to its maximal allocation before and after the update. We see that with SWAN there is no adverse impact on the throughput in any class when the forwarding plane update is executed at t=10s.

To contrast, Figure 11b shows what happens without congestion-controlled updates. Here, as in SWAN, 10% of scratch capacity is kept with respect to non-background traffic, but all update commands are issued to switches in one step. We see that Elastic and Background classes suffer transient throughput degradation due to congestion induced losses followed by TCP backoffs. Interactive traffic is protected due to priority queuing in this example but that does not hold for updates that move a lot of interactive traffic across paths. During updates, the throughput degradation across all traffic in a class is 20%, but as Figure 11c shows, it is as high as 40% for some of the flows.

Failure recovery: Figure 12 shows that SWAN can quickly recover from failures. It plots a zoomed in view of what happens to network throughput when we fail a randomly selected link at 10 seconds and then another at 20 seconds. We see that throughput close to optimal is restored in 1 second. The major sources of the delay include failure detection and error message propagation.

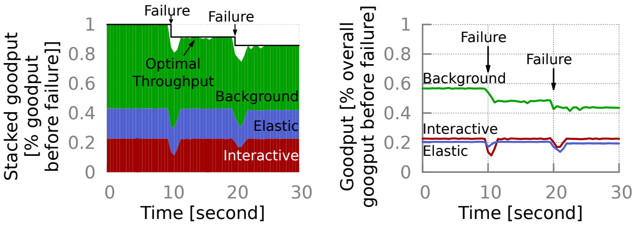


Figure 12: SWAN quickly recovers from failures.

6. DATA-DRIVEN EVALUATION

To evaluate SWAN at scale, we conduct data-driven simulations with topologies and traffic from two production inter-DC WANs of large cloud service providers (§6.1). We show that SWAN can carry 60% more traffic than MPLS TE (§6.2) and is fairer than MPLS TE (§6.3). We also show that SWAN enables congestion-controlled updates (§6.4) using bounded switch state (§6.5).

6.1 Datasets and methodology

We consider two inter-DC WANs:

IDN: A large, well-connected inter-DC WAN with more than 40 DCs. We have accurate topology, capacity, and traffic information for this network. Each DC is connected to 2-16 other DCs, and inter-DC capacities range from tens of Gbps to Tbps. Major DCs have more neighbors and higher capacity connectivity. Each DC has two WAN routers for fault tolerance, and each router connects to both routers in the neighboring DC. We obtain flow-level traffic on this network using sFlow logs collected by routers.

G-Scale: Google's inter-DC WAN with 12 DCs and 19 inter-DC links [14]. We do not have traffic and capacity information for it. We simulate traffic on this network using logs from another production inter-DC WAN (different from IDN) with a similar number of DCs. In particular, we randomly map nodes from this other network to G-Scale. This mapping retains the burstiness and skew of inter-DC traffic, but not any spatial relationships between the nodes. As in IDN, we assume that each DC has two switches, and each switch connects to both switches in adjacent DCs.

We estimate capacity based on the gravity model [29]. Reflecting common provisioning practices, we also round capacity up to the nearest multiple of 80 Gbps. We obtained qualitatively similar results (omitted from the paper) with three other capacity assignment methods: *i*) capacity is based on 5-minute peak usage across a week when the traffic is carried over shortest paths using ECMP (we cannot use MPLS TE as that requires capacity information); *ii*) capacity between each pair of DCs is 320 Gbps; *iii*) capacity between a pair of DCs is 320 or 160 Gbps with equal probability.

With the help of network operators, we classify traffic into individual services and map each service to Interactive, Elastic, or Background class. We assume that the networks were provisioned such that what we measured was the real demand of services which had not been modulated by capacity limitations.

We conduct experiments using a flow-level simulator that implements a complete version of SWAN. The demand of the services is derived based on the traffic information from a week-long network log. If the full demand of a service is not allocated in an interval, it carries over to the next interval. We place the SWAN controller at a central DC and simulate control plane latency between the controller and entities in other DCs (service brokers, network agents). This latency is based on shortest paths, where the latency of each hop is based on speed of light in fiber and great circle distance.

6.2 Network utilization

To evaluate how well SWAN utilizes the network, we compare it to an optimal method that can offer 100% utilization. This method computes how much traffic can be carried in each 5-min interval by solving a multi-class, multi-commodity flow problem. It is restricted only by link capacities, not by rule count limits. The changes to service rates are instantaneous, and rate limiting and interactive traffic prediction is perfect.

We also compare SWAN to the current practice, MPLS TE (§2). Our MPLS TE implementation has the advanced features that IDN uses [4, 23]. Priorities for packets and tunnels protect higher-priority packets and ensure shorter paths for higher-priority services. Per *re-optimization*, CSPF is invoked periodically (5 minutes) to search for better path assignments. Per *auto-bandwidth*, tunnel bandwidth is periodically (5 minutes) adjusted based on the current traffic demand, estimated by the maximum of the average (across 5-minute intervals) demand in the past 15 minutes.

Figure 13 shows the traffic that different methods can carry compared to the optimal. To quantify the traffic that a method can carry, we scale service demands by the same factor and use binary search to derive the maximum admissible traffic. We define admissibility as carrying at least 99.9% of service demands. Using a threshold less than 100% makes results robust to demand spikes.

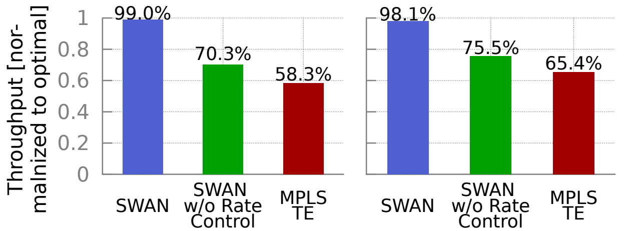
We see that MPLS TE carries only around 60% of the optimal amount of traffic. SWAN, on the other hand, can carry 98% for both WANs. This difference means that SWAN carries over 60% more traffic that MPLS TE, which is a significant gain in the value extracted from the inter-DC WAN.

To decouple gains of SWAN from its two main components—coordination across services and global network configuration—we also simulated a variant of SWAN where the former is absent. Here, instead of getting demand requests from services, we estimate it from their throughput in a manner similar to MPLS TE. We also do not control the rate at which services send. Figure 13 shows that this variant of SWAN improves utilization by 10–12% over MPLS TE, i.e., it carries 15–20% more traffic. Even this level of increase in efficiency translates to savings of millions of dollars in the cost of carrying wide-area traffic. By studying a (hypothetical) version of MPLS that perfectly knows future traffic demand (instead of estimating it based on history), we find that most of SWAN's gain over MPLS stems from its ability to find better path assignments.

We draw two conclusions from this result. First, both components of SWAN are needed to fully achieve its gains. Second, even in networks where incoming traffic cannot be controlled (e.g., ISP network), worthwhile utilization improvements can be obtained through the centralized resource allocation offered by SWAN.

6.3 Fairness

SWAN improves not only efficiency but also fairness. To study fairness, we scale demands such that background traffic is 50% higher than what a mechanism admits; fairness is



(a) **IDN** (b) **G-Scale**

Figure 13: SWAN carries more traffic than MPLS TE.

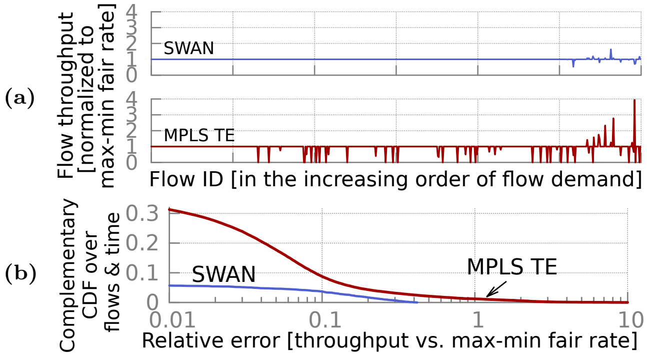


Figure 14: SWAN is fairer than MPLS TE.

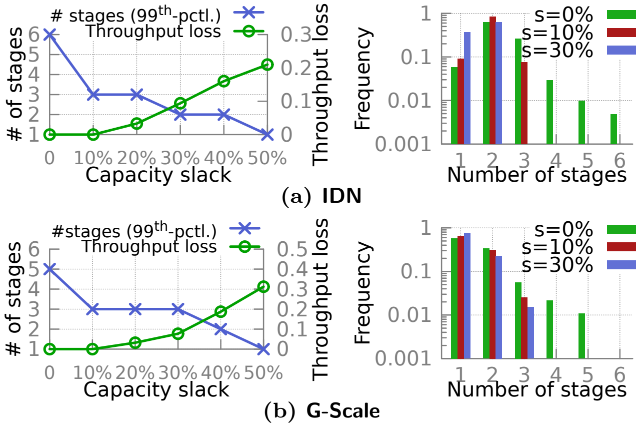
of interest only when traffic demands cannot be fully met. Scaling relative to traffic admitted by a mechanism ensures that oversubscription level is the same. If we used an identical demand for SWAN and MPLS TE, the oversubscription for MPLS TE would be higher as it carries less traffic.

For an exemplary 5-minute window, Figure 14a shows the throughput that individual flows get relative to their max-min fair share. We focus on background traffic as the higher priority for other traffic means that its demands are often met. We compute max-min fair shares using a precise but computationally-complex method (which is unsuitable for online use) [25]. We see that SWAN well approximates max-min fair sharing. In contrast, the greedy, local allocation of MPLS TE is significantly unfair.

Figure 14b shows aggregated results. In SWAN, only 4% of the flows deviate over 5% from their fair share. In MPLS TE, 20% of the flows deviate by that much, and the worst-case deviation is much higher. As Figure 14a shows, the flows that deviate are not necessarily high- or low-demand, but are spread across the board.

6.4 Congestion-controlled updates

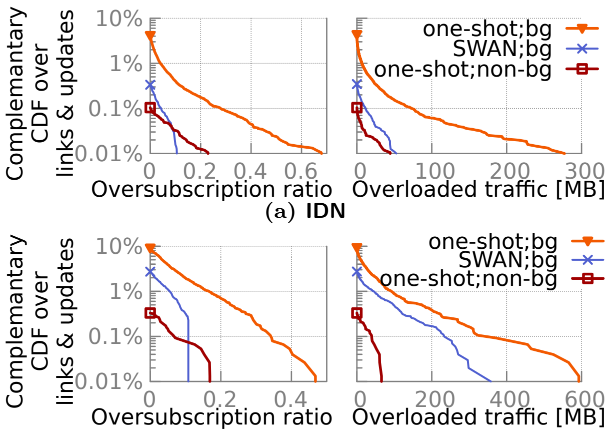
We now study congestion-controlled updates in detail, the tradeoff regarding the amount of scratch capacity and their benefit. Higher levels of scratch capacity lead to fewer stages, and thus faster transitions; but they lower the amount of non-background traffic that the network can carry and can waste capacity if background traffic demand is low. Figure 15 shows this tradeoff in practice. The left graph plots the maximum number of stages and loss in network throughput as a function of scratch capacity. At the s=0% extreme, throughput loss is zero but more stages—infinitely many in the worst case—are needed to transition safely. At the s=50% extreme, only one stage is needed, but the network delivers 25−36% less traffic. The right graph shows the PDF of the number of stages for three values of s. Based on these results, we use s=10%, where the throughput loss is negligible and updates need only 1-3 steps (which is much lower than the theoretical worst case of 9).



(a) **IDN**

(b) **G-Scale**

Figure 15: Number of stages and loss in network throughput as a function of scratch capacity.



(a) **IDN**

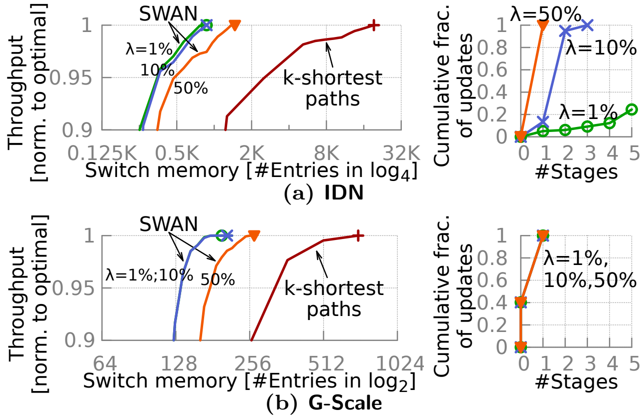
(b) **G-Scale**

Figure 16: Link oversubscription during updates.

To evaluate the benefit of congestion-controlled updates, we compare with a method that applies updates in one shot. This method is identical in every other way, including the amount of scratch capacity left on links. Both methods send updates in a step to the switches in parallel. Each switch applies its updates sequentially and takes 2 ms per update [8].

For each method, during each reconfiguration, we compute the maximum over-subscription (i.e., load relative to capacity), at each link. Short-lived oversubscription will be absorbed by switch queues. Hence, we also compute the maximal buffering required at each link for it to not drop any packet, i.e., total excess bytes that arrive during oversubscribed periods. If this number is higher than the size of the physical queue, packets will be dropped. Per priority queuing, we compute oversubscription separately for each traffic class; the computation for non-background traffic ignores background traffic but that for background traffic considers all traffic.

Figure 16 shows oversubscription ratios on the left. We see heavy oversubscription with one-shot updates, especially for background traffic. Links can be oversubscribed by up to 60% of their capacity. The right graph plots extra bytes on the links. Today's top-of-line switches, which we use in our testbed, have queue sizes of 9-16 MB. But we see that oversubscription can bring 100s of MB of excess pack-



(a) IDN

(b) G-Scale

Figure 17: SWAN needs fewer rules to fully exploit network capacity (left). The number of stages needed for rule changes is small (right).

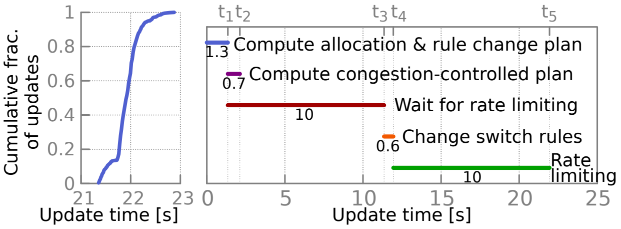
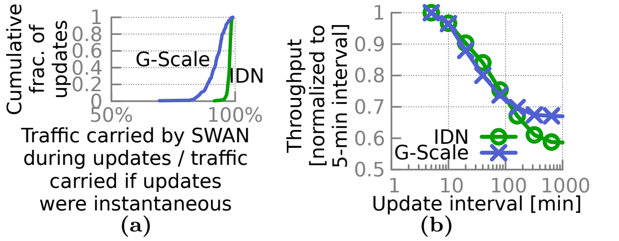


Figure 18: Time for network update.



(a)

(b)

Figure 19: (a) SWAN carries close to optimal traffic even during updates. (b) Frequent updates lead to higher throughput.

ets and hence, most of these will be dropped. Note that we did not model TCP backoffs which would reduce the load on a link after packet loss starts happening, but regardless, those flows would see significant slowdown. With SWAN, the worst-case oversubscription is only 11% $(=\frac{s}{1-s})$ as configured for bounded-congestion updates, which presents a significantly better experience for background traffic.

We also see that despite 10% slack, one-shot updates fail to protect even the non-background traffic which is sensitive to loss and delay. Oversubscription can be up to 20%, which can bring over 50 MB of extra bytes during reconfigurations. SWAN fully protects non-background traffic and hence that curve is omitted.

Since routes are updated very frequently even a small likelihood of severe packet loss due to updates can lead to frequent user-visible network incidents. For e.g., when updates happen every minute, a $\frac{1}{1000}$ likelihood of severe packet loss due to route updates leads to an interruption, on average, once every 7 minutes on the IDN network.

6.5 Rule management

We now study rule management in SWAN. A primary measure of interest here is the amount of network capacity that can be used given a switch rule count limit. Figure 17 (left) shows this measure for SWAN and an alternative that installs rules for the k-shortest paths between DC-pairs; k is chosen such that the rule count limit is not violated for any switch. We see that k-shortest path routing requires 20K rules to fully use network capacity. As mentioned before, this requirement is beyond what will be offered by next-generation switches. The natural progression towards faster link speeds and larger WANs means that future switches may need even more rules. If switches support 1K rules, k-shortest path routing is unable to use 10% of the network capacity. In contrast, SWAN's dynamic tunnels approach enables it to fully use network capacity with an order of magnitude fewer rules. This fits within the capabilities of current-generation switches.

Figure 17 (right) shows the number of stages needed to dynamically change tunnels. It assumes a limit of 750 OpenFlow rules, which is what our testbed switches support. With 10% slack only two stages are needed 95% of the time. This nimbleness stems from 1) the efficiency of dynamic tunnels—a small set of rules are needed per interval, and 2) temporal locality in demand matrices—this set changes slowly across adjacent intervals.

6.6 Other microbenchmarks

We close our evaluation of SWAN by reporting on some key microbenchmarks.

Update time: Figure 18 shows the time to update IDN from the start of a new epoch. Our controller uses a PC with a 2.8GHz CPU and runs unoptimized code. The left graph shows a CDF across all updates. The right graph depicts a timeline of the average time spent in various parts. Most updates finish in 22s; most of this time goes into waiting for service rate limits to take effect, 10s each to wait for services to reduce their rate (t_1 to t_3) and then for those whose rate increases (t_4 to t_5). SWAN computes the congestion-controlled plan in parallel with the first of these. The network's data plane is in flux for only 600 ms on average (t_3 to t_4). This includes communication delay from controller to switches and the time to update rules at switches, multiplied by the number of stages required to bound congestion. If SWAN were used in a network without explicit resource signaling, the average update time would only be this 600 ms.

Traffic carried during updates: During updates, SWAN ensures that the network continues to maintain high utilization. That the overall network utilization of SWAN comes close to optimal (§6.2) is an evidence of this behavior. More directly, Figure 19a shows the %-age of traffic that SWAN carries during updates compared to an optimal method with instantaneous updates. The median value is 96%.

Update frequency: Figure 19b shows that frequent updates to the network's data plane lead to higher efficiency. It plots the drop in throughput as the update duration is increased. The service demands still change every 5 minutes but the network data plane updates at the slower rate (x-axis) and the controller allocates as much traffic as the current data plane can carry. We see that an update frequency of 10 (100) minutes reduces throughput by 5% (30%).

Prediction error for interactive traffic: SWAN predicts the amount of interactive traffic in the next epoch. Figure 20 shows the error in this prediction. It plots pre-

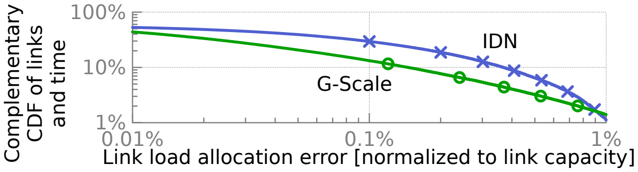


Figure 20: Link allocation error due to imperfect demand prediction for interactive traffic.

dicted versus actual traffic that traverses a link relative to its capacity. We see that the error is low, because interactive traffic is stable at these timescales and tends to be a small fraction of link capacity.

7. DISCUSSION

This section discusses several issues that, for conciseness, were not mentioned in the main body of the paper.

Non-conforming traffic: Sometimes services may (e.g., due to bugs) send more than what is allocated. SWAN can detect these situations using traffic logs that are collected from switches every 5 minutes. It can then notify the owners of the service and protect other traffic by re-marking the DSCP bits of non-confirming traffic to a class that is even lower than background traffic, so that it's carried only if there is any spare capacity.

Truthful declaration: Services may declare their lower-priority traffic as higher priority or ask for more bandwidth than they can consume. SWAN discourages this behavior through appropriate pricing: services pay more for higher priority and pay for all allocated resources. (Even within a single organization, services pay for the infrastructure resources they consume.)

Richer service-network interface: Our current design has a simple interface between the services and network, based on current bandwidth demand. In future work, we will consider a richer interface such as letting services reserve resources ahead of time and letting them express their needs in terms of total bytes and a deadline by which they must be transmitted. Better knowledge of such needs can further boost efficiency, for instance, by enabling store-and-forward transfers through intermediate DCs [20]. The key challenge here is the design of scalable and fair allocation mechanisms that composes the diversity of service needs.

8. RELATED WORK

SWAN builds upon several themes in prior work.

Intra-DC traffic management: Many recent works manage intra-DC traffic to better balance load [1, 7, 8] or share among selfish parties [15, 27, 30]. SWAN is similar to the former in using centralized TE and to the latter in providing fairness. But the *intra*-DC case has constraints and opportunities that do not translate to the WAN. For example, EyeQ [15] assumes that the network has a full bisection bandwidth core and hence only paths to or from the core can be congested; this need not hold for a WAN. Seawall [30] uses TCP-like adaptation to converge to fair share, but high RTTs on the WAN would mean slow convergence. Faircloud [27] identifies strategy-proof sharing mechanisms, i.e., resilient to the choices of individual actors. SWAN uses explicit resource signaling to disallow such greedy actions. Signaling also helps it avoid estimating demands which other centralized TE schemes have to do [1, 8].

WAN TE & SDN: As in SWAN, B4 uses SDNs in the context of inter-DC WANs [14]. Although this parallel work shares a similar high-level architecture, it addresses different challenges. While B4 develops custom switches and mechanisms to integrate existing routing protocols in an SDN environment, SWAN develops mechanisms for congestion-free data plane updates and for effectively using the limited forwarding table capacity of commodity switches.

Optimizing WAN efficiency has rich literature including tuning ECMP weights [12], adapting allocations across pre-established tunnels [10, 16], storing and re-routing bulk data at relay nodes [20], caching at application-layer [31] and leveraging reconfigurable optical networks [21]. While such bandwidth efficiency is one of the design goals, SWAN also addresses performance and bandwidth requirements of different traffic classes. In fact, SWAN can help many of these systems by providing available bandwidth information and by offering routes through the WAN that may not be discovered by application-layer overlays.

Guarantees during network update: Some recent work provides guarantees during network updates either on connectivity, or loop-free paths or that a packet will see a consistent set of SDN rules [18, 22, 28, 33]. SWAN offers a stronger guarantee that the network remains uncongested during forwarding rule changes. Vanbever et. al. [33] suggest finding an ordering of updates to individual switches that is guaranteed to be congestion free; however, we see that such ordering may not exist (§6.4) and is unlikely to exist when the network operates at high utilization.

9. CONCLUSION

SWAN enables a highly efficient and flexible inter-DC WAN by coordinating the sending rates of services and centrally configuring the network data plane. Frequent network updates are needed for high efficiency, and we showed how, by leaving a small amount of scratch capacity on the links and switch rule memory, these updates can be implemented quickly and without congestion or disruption. Testbed experiments and data-driven simulations show that SWAN can carry 60% more traffic than the current practice.

Acknowledgements. We thank Rich Groves, Parantap Lahiri, Dave Maltz, and Lihua Yuan for feedback on the design of SWAN. We also thank Matthew Caesar, Brighten Godfrey, Nikolaos Laoutaris, John Zahorjan, and the SIGCOMM reviewers for feedback on earlier drafts of the paper.

10. REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[2] D. Applegate and M. Thorup. Load optimal MPLS routing with N+M labels. In *INFOCOM*, 2003.

[3] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP tunnels. RFC 3209, 2001.

[4] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for traffic engineering over MPLS. RFC 2702, 1999.

[5] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[6] Y. Chen, S. Jain, V. K. Adhikari, Z.-L. Zhang, and K. Xu. A first look at inter-data center traffic characteristics via Yahoo! datasets. In *INFOCOM*, 2011.

[7] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[8] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

[9] E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *INFOCOM*, 2012.

[10] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS adaptive traffic engineering. In *INFOCOM*, 2001.

[11] Project Floodlight. http://www.projectfloodlight.org/.

[12] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional IP routing protocols. *IEEE Comm. Mag.*, 2002.

[13] T. Hartman, A. Hassidim, H. Kaplan, D. Raz, and M. Segalov. How to split a flow? In *INFOCOM*, 2012.

[14] S. Jain et al. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[15] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, and C. Kim. EyeQ: Practical network performance isolation for the multi-tenant cloud. In *HotCloud*, 2012.

[16] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *SIGCOMM*, 2005.

[17] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *SIGCOMM CCR*, 2007.

[18] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs. R-BGP: Staying connected in a connected world. In *NSDI*, 2007.

[19] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet inter-domain traffic. *SIGCOMM Comput. Commun. Rev.*, 2010.

[20] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with NetStitcher. In *SIGCOMM*, 2011.

[21] A. Mahimkar, A. Chiu, R. Doverspike, M. D. Feuer, P. Magill, E. Mavrogiorgis, J. Pastor, S. L. Woodward, and J. Yates. Bandwidth on demand for inter-data center communication. In *HotNets*, 2011.

[22] R. McGeer. A safe, efficient update protocol for OpenFlow networks. In *HotSDN*, 2012.

[23] M. Meyer and J. Vasseur. MPLS traffic engineering soft preemption. RFC 5712, 2010.

[24] V. S. Mirrokni, M. Thottan, H. Uzunalioglu, and S. Paul. A simple polynomial time framework for reduced-path decomposition in multi-path routing. In *INFOCOM*, 2004.

[25] D. Nace, N.-L. Doan, E. Gourdin, and B. Liau. Computing optimal max-min fair resource allocation for elastic flows. *IEEE/ACM Trans. Netw.*, 2006.

[26] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. Maltz. Latency inflation with MPLS-based traffic engineering. In *IMC*, 2011.

[27] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[28] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[29] M. Roughan, A. Greenberg, C. Kalmanek, M. Rumsewicz, J. Yates, and Y. Zhang. Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning. In *Internet Measurement Workshop*, 2002.

[30] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *NSDI*, 2011.

[31] S. Traverso, K. Huguenin, I. Trestian, V. Erramilli, N. Laoutaris, and K. Papagiannaki. Tailgate: handling long-tail content with a little help from friends. In *WWW*, 2012.

[32] Broadcom Trident II series. http://www.broadcom.com/docs/features/StrataXGS_Trident_II_presentation.pdf, 2012.

[33] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Seamless network-wide IGP migrations. In *SIGCOMM*, 2011.

[34] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[35] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A reordering-robust TCP with DSACK. In *ICNP*, 2003.

APPENDIX

Theorem 1. Let r_i be the max-min fair rate of flow i,and b_i be the rate allocated to flow i by SWAN's Approx Max-Min Fairness algorithm (Figure 6), also let $U < \min r_i$. Then $b_i \in \left[\frac{r_i}{\alpha}, \alpha r_i\right]$.

Proof Sketch: Observe that the linear program (MCF) per se maximizes overall throughput $\sum b_i$ with a bias towards carrying more of the traffic on paths that have less weight (e.g., shorter paths); ϵ is a small constant. However, Approx Max-Min Fairness invokes MCF in T steps with the constraint that at step k, flows are allocated rates in the range $\left[\alpha^{k-1}U, \alpha^k U\right]$ but no more than their demand. A flow's allocation is *frozen* at step k when it is allocated its full demand d_i at that step or it receives a rate smaller than $\alpha^k U$ due to capacity constraints.

The algorithm's allocation proceeds in steps. Our proof proceeds in epochs that consist of one or more steps. One of three things can happen at each step – first, no link is newly saturated at that step; second, some links are saturated but every link has at least one flow using it that is not capacity saturated, i.e, the flow has other links and paths that it can send more traffic on; third, some links are saturated and all flows on those links are capacity saturated. Note that, by definition, at steps of the first type if a flow is frozen it has to be because its demand is saturated $b_i = d_i$. Because otherwise, the throughput maximization objective will cause that flow to get the maximum possible rate at those steps. The same holds at steps of the second type, because capacity can be freed up on a saturated link by moving some of the traffic belonging to its unsaturated flows off that link. Only at steps of the third type could there be flows frozen because they are limited by capacity. We say that the ongoing epoch ends and a new one begins after each step of the third type.

We will prove that the maximal unfairness for flows that are frozen in each epoch is bounded. Note, that at the end of an epoch, every flow using one of the newly saturated links in this epoch will also be frozen by definition. Hence, these flows and links can be removed from the topology since nothing changes for either at subsequent steps.

To prove this, first note that every flow that is frozen because its demand has been met or it is capacity saturated has rate equaling its max-min fair rate; since other flows that remain unfrozen at the time this flow freezes receive at least this much rate. Second, we divide flows that are frozen due to capacity limits into groups such that two flows will be in the same group if they send non-zero traffic on at least one common link. Within a group, the rate could be allocated unfairly. However, the total rate allocated to these flows remains the same; allocating less reduces overall throughput and allocating more is not possible since the group is capacity constrained. Further since a group of flows simultaneously freezes at the same stage (of type three), it means that the ratio of the lowest flow rate to the largest flow rate in the group is α and the fair rate of the flow falls somewhere in between. □

Theorem 2. If all links in the network have a relative slack s, in both the initial flow C and and the final flow C', then there exists a congestion-free sequence of updates of length no more than $\lceil 1/s \rceil - 1$.

Proof. We prove this constructively. Let $b^0_{i,j} = b_{i,j}$ denote the allocated bandwidth in the initial configuration, and let $b^q_{i,j} = b'_{i,j}$ denote the allocated bandwidth in the final configuration, after q steps. The superscript $0, \dots, q$ refer to the update stages. In each stage, we increase the allocated bandwidth by $(b^q_{i,j} - b^0_{i,j})/q$. This algorithm ensures:

- After $q = \lceil 1/s \rceil - 1$ stages, we reach the allocated bandwidth in the final configuration as $b^0_{i,j} + q \cdot (b^q_{i,j} - b^0_{i,j})/q = b^q_{i,j}$.
- After the k^{th} stage, we need to show the network configuration is still valid, i.e., $\sum_j b^k_{i,j} = b_i$, the total bandwidth of flow i. Because each $b^k_{i,j}$ is a linear combination of $b^0_{i,j}$ and $b^q_{i,j}$, we have $\min(b^0_{i,j}, b^q_{i,j}) \leq b^s_{i,j} \leq \max(b^0_{i,j}, b^q_{i,j})$. By adding up all possible tunnels, we have $\min(\sum_j b^0_{i,j}, \sum_j b^q_{i,j}) \leq \sum_j b^k_{i,j} \leq \max(\sum_j b^0_{i,j}, \sum_j b^q_{i,j})$. Because the original and target configurations are valid, we must have $b_i = \sum_j b^0_{i,j} = \sum_j b^q_{i,j}$. Therefore, $\sum_j b^k_{i,j} = b_i$. Likewise, the flow is valid at every node in every step.
- After the k^{th} stage, every link has slack s. Let w^k_l denote the link l's load after k^{th} stage, we have $w^k_l = \sum_{i,j} b^k_{i,j} \cdot I_{j,l}$. Because the network has a slack s in both original and target configurations, we have $w^0_l \leq (1-s)c_l$ and $w^q_l \leq (1-s)c_l$. Because w^k_l is a linear combination of w^0_l and w^q_l, we have $w^k_l \leq (1-s)c_l$.
- During each stage, we need to show any transient configuration will not overload any link. Let $\Delta^k_{i,j}$ denote the increase of flow i's rate at tunnel j during the k^{th} stage, we have $\Delta^k_{i,j} = b^{k+1}_{i,j} - b^k_{i,j}$. Consider the *worst* update sequence for link l where all the tunnels with increased rate ($\Delta^k_{i,j} > 0$) are already updated, while the tunnels with decreased rate are not updated. In the beginning of the stage, link l has a residual capacity sC_l. Now all the paths with increase rate are updated, the increase of load is $\sum_{i,j;\Delta^k_{i,j}>0} I_{j,l} \cdot (b^q_{i,j} - b^0_{i,j})/q \leq \sum_{i,j} I_{j,l} \cdot b^q_{i,j}/q \leq s \cdot \sum_{i,j} I_{j,l} \cdot b^q_{i,j}/(1-s) \leq sC_l$.

□

Theorem 3. If non-background traffic has slack s in both C and C', then there exists an update sequence such that (i) non-background traffic does not have congestive loss, (ii) the maximal congestion for background traffic at any link is bounded by ηC_l, and (iii) the maximum length of the update sequence is $\max(\lceil 1/s \rceil - 1, \lceil 1/\eta \rceil)$.

Proof. We reuse the same algorithm as given in the proof for Theorem 2 but the maximum length of the update sequence is set to $q = \max(\lceil 1/s \rceil - 1, \lceil 1/\eta \rceil)$ here. Because the non-background traffic goes at higher priority on the data plane, and given $q \geq \lceil 1/s \rceil - 1$, the algorithm ensures that non-background traffic does not have congestive loss by Theorem 2. Thus, we only need to show background traffic can be dropped no more than ηC_l at any link l. Let $\Delta^k_{i,j}$ denote the increase of flow i's rate at tunnel j during the k^{th} stage, we have $\Delta^k_{i,j} = b^{k+1}_{i,j} - b^k_{i,j}$. Consider the *worst* update sequence for link l where all the tunnels with increased rate ($\Delta^k_{i,j} > 0$) are already updated, while the tunnels with decreased rate are not updated. The maximal congestion is at most the total increase of load G:

$$\begin{aligned} G &\leq \sum_{i,j;\Delta^k_{i,j}>0} I_{j,l} \cdot (b^q_{i,j} - b^0_{i,j})/q \\ &\leq \sum_{i,j} I_{j,l} \cdot b^q_{i,j}/q \\ &\leq \eta \cdot \sum_{i,j} I_{j,l} \cdot b^q_{i,j} \\ &\leq \eta C_l. \end{aligned}$$

□

Theorem 4. If any switch j has a memory slack $\lambda \cdot M_j$ in P and P', then the rule change algorithm requires at most $z = \lceil 1/\lambda \rceil - 1$ steps and satisfies the memory constraint.

Proof. Recall that at i^{th} step SWAN picks tunnels to add such that the total added tunnels in the first i steps require at most t^{add}_i rules. Also, SWAN picks tunnels to remove such that the tunnels that remain to be removed require at most t^{rem}_i rules after the removal. Let r_j, r'_j and r^*_j be the number of rules used by $P - P'$, $P' - P$ and $P \cap P'$, respectively. We define the per-step rule addition limit, denote by s_j, to be $M_j - \max(r_j, r'_j) + r^*_j$. Then we set $t^{add}_i = i \cdot s_j$ and $t^{rem}_i = \max(0, M_j - r^*_j - (1+i)s_j)$.

We first show the algorithm will terminate after $z = \lceil 1/\lambda \rceil - 1$ steps. This holds if $t^{add}_z = z \cdot s_j \geq r'_j + r^*_j$ because we will have enough capacity to select entire P' at any switch j. Also, because P and P' provide a slack of λM_j, we have $r_j \leq (1-\lambda)M_j - r_j^*$ and $r_j' \leq (1-\lambda)M_j - r_j^*$. Therefore,

$$\begin{aligned} z \cdot s_j &= (\lceil 1/\lambda \rceil - 1) \cdot (M_j - \max(r_j, r_j') - r_j^*) \\ &\geq (1/\lambda - 1) \cdot (\lambda M_j) \\ &= (1-\lambda) M_j \\ &\geq r_j' + r_j^* \end{aligned}$$

Next, we show the algorithm satisfies the memory constraint at any switch. At i^{th} step, the highest memory load happens when new tunnels have already added to switches but old tunnels have not deleted. The total added tunnels in the first i steps contribute at most $t_i^{add} = i \cdot s_j$ rules, and the tunnels that remain to be removed require at most $t_{i-1}^{rem} = \max(0, M_j - r_j^* - i \cdot s_j)$ rules. Also, each switch stores a static set of tunnels $P \cap P'$ that requires at most r_j^* rules. Adding these up, the maximal memory load is at most $(i \cdot s_j) + \max(0, M_j - r_j^* - i \cdot s_j) + (r_j^*) \leq M_j$. □