Scalable Programmable Inbound Traffic Engineering

Peng Sun
Princeton University
pengsun@cs.princeton.edu

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

ABSTRACT

With the rise of video streaming and cloud services, enterprise and access networks receive much more traffic than they send, and must rely on the Internet to offer good end-to-end performance. These edge networks often connect to multiple ISPs for better performance and reliability, but have only limited ways to influence which of their ISPs carries the traffic for each service. In this paper, we present Sprite, a software-defined solution for flexible inbound traffic engineering (TE). Sprite offers direct, fine-grained control over inbound traffic, by announcing different public IP prefixes to each ISP, and performing source network address translation (SNAT) on outbound request traffic. Our design achieves scalability in both the data plane (by performing SNAT on edge switches close to the clients) and the control plane (by having local agents install the SNAT rules). The controller translates high-level TE objectives, based on client and server names, as well as performance metrics, to a dynamic network policy based on real-time traffic and performance measurements. We evaluate Sprite with live data from "in the wild" experiments on an EC2-based testbed, and demonstrate how Sprite dynamically adapts the network policy to achieve high-level TE objectives, such as balancing YouTube traffic among ISPs to improve video quality.

Categories and Subject Descriptors:
C.2.3 [**Computer-Communication Networks**]: Network Operations—*network management*;
C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

Keywords:
Traffic engineering; software-defined networking; scalability

1. Introduction

Many edge networks—like enterprises, university campuses, and broadband access networks—connect to multiple Internet Service Providers (ISPs) for higher reliability, better performance, and lower cost. Many research projects [1, 2, 3] and commercial products [4, 5, 6, 7, 8, 9] have shown how multihomed networks can divide their *outbound* traffic over multiple ISPs to optimize peformance, load, and cost. However, relatively few works have explored how to perform *inbound* traffic engineering effectively.

Yet, inbound traffic engineering has never been more important. Modern applications like video streaming and cloud services have highly asymmetric traffic demands; for example, our measurements show that the Princeton University campus receives an average of *eight times* more traffic than it sends. Many applications are performance sensitive; video streaming needs high throughput and low jitter, and many cloud services need low latency to compete with the alternative of running services locally. In addition, end-to-end performance is vulnerable to peering disputes [10], where some ISPs do not devote enough bandwidth for high-quality video streaming, leading video services to decrease the video quality to reduce the bandwidth requirements. Switching the traffic to a different incoming ISP along a better end-to-end path could increase the video quality.

SOSR 2015, June 17–18, 2015, Santa Clara, California, USA.
© 2015 ACM. ISBN 978-1-4503-3451-8/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2774993.2775063.

Unfortunately, inbound traffic engineering (TE) is quite difficult. Since Internet routing is destination-based, the sending Autonomous System (AS) decides where to forward traffic, based on the routes announced by its neighbors. The receiving network has, at best, clumsy and indirect control by manipulating its Border Gateway Protocol (BGP) announcements, through *AS-PATH prepending* (adding fake hops to make the route through one ISP look longer than another) and *selective prefix announcements* (to force all traffic to one group of users to arrive via one ISP rather than another). Both techniques are coarse-grained and do not allow an edge network to (say) receive all Netflix traffic via one ISP, and all Gmail traffic through another.

Edge networks need an TE solution that offers fine-grained, direct control of inbound traffic, without requiring support from the sender or the rest of the Internet. In this paper, we introduce Sprite (Scalable PRogrammable Inbound Traffic Engineering), a software-defined TE solution. Sprite controls inbound traffic by dividing the edge network's public IP address space across the ISPs, and using source network address translation (SNAT) to map each outbound connection to a specific inbound ISP for the return traffic [3], as discussed in more detail in the next section. Given the nature of Internet routing, an edge network cannot fully control the entire end-to-end path—only which entry point receives the traffic. Still, this gives an edge network enough control to balance load and optimize performance by selecting among a small set of end-to-end paths for each connection.

The key contribution of Sprite is a *scalable* solution for realizing *high-level* TE objectives, using software-defined networking (SDN). Sprite achieves scalability in the data plane (by distributing the SNAT functionality over many edge switches, close to the end hosts) and the control plane (by having local agents install the SNAT rules and collecting traffic statistics on behalf of the central controller). The network administrator specifies a high-level traffic-engineering objective based on names (rather than network identifiers) and performance metrics (rather than routing decisions). The controller translates the high-level objective into network policy, and dynamically adjusts the network policy based on traffic and performance measurements. We present the design and implementation of our Sprite architecture, and evaluate our system "in the wild" using an EC2-based testbed and the PEERING testbed [11].

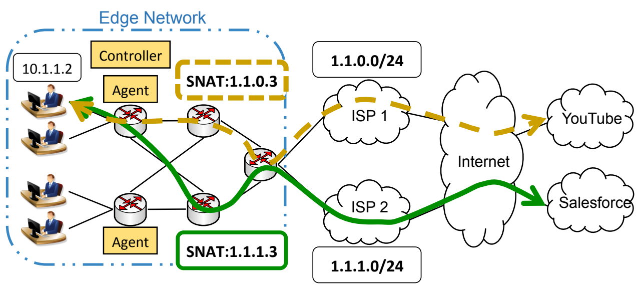

Figure 1: Scalable distributed SNAT in Sprite

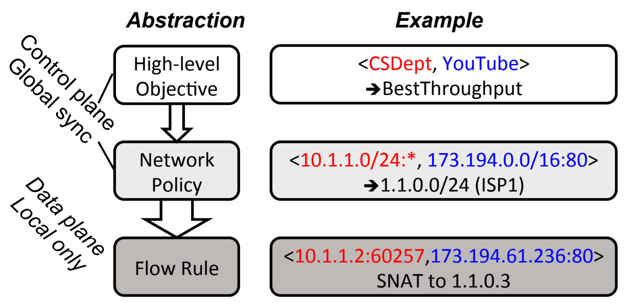
Abstraction
Example
Control plane
Global sync
High-level Objective
<CSDept, YouTube> ➔BestThroughput
Network Policy
<10.1.1.0/24:*, 173.194.0.0/16:80> ➔1.1.0.0/24 (ISP1)
Data plane
Local only
Flow Rule
<10.1.1.2:60257,173.194.61.236:80> SNAT to 1.1.0.3

Figure 2: Three levels of abstraction in Sprite

2. Inbound TE Using Source NAT

An edge network can have direct control over inbound traffic by combining two mechanisms:

Split the IP address block across ISPs: To control the flow of inbound traffic, the edge network's public address space is divided into separate prefixes, and Sprite assigns each prefix to one upstream ISP, similar to the common practice of selective prefix announcements. For example, an edge network with the 1.1.0.0/23 address and two ISPs could announce 1.1.0.0/24 via ISP 1 and 1.1.1.0/24 via ISP 2; for fault tolerance in the case of ISP disconnection, the edge network also announces the supernet 1.1.0.0/23 via both ISPs.

Perform SNAT on outbound traffic: To associate a new connection with a particular inbound ISP, the edge network maps the source IP address of outgoing request traffic to an address in the appropriate prefix. For example, the edge network numbers its client machines using private addresses (e.g., in the 10.0.0.0/8 address block), and maps outgoing connections to a source IP address in either 1.1.0.0/24 (if the destination corresponds to a YouTube server) or 1.1.1.0/24 (if the destination corresponds to a Salesforce server).

While the outbound traffic might leave the edge network through either upstream ISP, these two mechanisms ensure that the response traffic arrives via the selected ISP. If the edge network runs any public services, those hosts should have public IP addresses routable via either ISP.

Using SNAT for inbound traffic engineering is not a new idea. Previous work [3, Sec IV.C] briefly discusses how to realize NAT-based inbound route control at a Web proxy using `iptables`. The main challenges we address in this paper are (i) automatically translating fine-grained TE objectives to low-level rules and (ii) distributing both the control-plane and data-plane functionality for better scalability.

3. Scalable Sprite Architecture

Sprite achieves scalability by distributing the data-plane rules (across switches near the end hosts) and control-plane operations (across local agents on or near the switches), as shown in Figure 1. The network administrator conveys a high-level traffic-engineering objective to the controller, then the controller generates a network policy to distribute to the local agents, and finally the local agents install SNAT rules in the switch data plane, as summarized in Figure 2.

3.1 Data Plane: Edge Switches Near Hosts

SNAT gives edge networks direct, fine-grained control over inbound traffic. Yet performing SNAT at a single proxy or border router poses a scalability problem. SNAT requires dynamically establishing a data-plane rule for each connection. The data-plane state would be large as it is in proportion to the large number of active connections, and control-plane operations are required on every connection set-up. The presence of multiple border routers introduces further complexity, since traffic for the same connection may enter and leave via different locations.

Instead, Sprite performs SNAT on a distributed collection of switches, near the end hosts. These switches can also collect passive traffic measurements (e.g., byte and packet counts per rule) that can help drive traffic-engineering decisions. These edge switches could be virtual switches running on the end hosts, access switches connected directly to a group of end hosts, or a gateway switch connecting a department to the rest of the enterprise network. Compared to the border routers, each switch handles much fewer active connections and a much lower arrival rate of new connections, enabling the use of cheap commodity switches (with small rule tables) or software switches.

The edge switches also need to receive the return traffic destined to its associated end hosts. Each edge switch maps outbound traffic from internal private addresses to a small set of public IP addresses assigned by the controller to the local agent. As such, the return traffic is easily routed to the right edge switch based on the destination IP prefix. If the edge network runs a legacy routing protocol (e.g., OSPF or IS-IS), the controller configures the injection of these prefixes into the routing protocol. If the edge network consists of SDN switches, the controller can install coarse-grained rules in the border router and interior switches to forward traffic to the right edge switch. By carefully assigning contiguous prefixes to nearby edge switches, the controller could further aggregate these prefixes in the rest of the edge network.

3.2 Control Plane: Local Agents Near Switches

Rather than involving the controller in installing each SNAT rule, a local agent on (or near) each switch performs this simple task. Based on a high-level traffic-engineering objective, the controller computes a network policy that maps network identifiers (e.g., IP addresses and port ranges) to the appropriate inbound ISP. Then, the controller subdivides the unique set of source IP addresses and port ranges across the local agents, so each local agent can generate flow rules on its own as new connections arrive. As a result, Sprite does not send *any* data packets to the controller. The local agent can also collect and aggregate measurement data from the switch's rule counter and via active probing, and share the results with the controller to inform future decisions.

The network policy generated by the Sprite controller only specifies source and destination IP prefixes, TCP/UDP port ranges, and which inbound ISP to use. Each local agent uses the network policy to automatically create SNAT rules for each new flow. Each edge switch has a default rule that sends all outbound traffic (i.e., packets with an internal source IP address) to the associated local agent. Upon receiving the packet, the local agent consults the current network policy to identify the suitable set of public addresses and port numbers, selects a single unused address/port pair, and installs the NAT rules for both directions of the traffic.

The controller needs to assign each local agent a sufficiently large pool of addresses and port numbers, without wasting the limited public address space. The controller can use measurement data collected from the local agent to track statistics on the number of simultaneously active connections. When running low on available address/port pairs, the agent contacts the controller to request additional identifiers; similarly, the controller can reclaim unused ranges of addresses from one local agent and assign them to another as needed. With reasonable "headroom" to over-allocate address/-port pairs to each agent, the controller can limit the churn in the assignments.

4. Dynamic Policy Adaptation

Sprite enables network administrators to express a wide range of TE objectives using high-level names of remote services and groups of users, as well as performance metrics. The controller automatically translates the TE objective into a low-level network policy, and adapts in real time to traffic and performance measurements.

4.1 High-level TE Objectives

Rather than specifying TE objectives on IP addresses and port numbers, Sprite allows administrators to use high-level names to identify services (e.g., YouTube) and groups of users (e.g., CS-Dept). The administrators can let Sprite dynamically map the connections of the users/services onto the ISPs by providing an evaluation function. The function takes many metrics (e.g., the ISP capacity, the connections' performance, etc) as inputs, and returns a score for how the ISP behaves. For example,

USER(*BioDept*) AND SER(*SalesForce*) → BEST(*LatencyCalculationFunction*)

specifies that traffic from SalesForce to the Biology department should use the ISP that offers the lowest latency. Alternatively, the administrator can associate connections with particular named clients and services with a specific set of ISPs (with weights that specify the portion of inbound traffic by ISP). For example,

SER(*YouTube*) → [<*ISP1,1.0*>,<*ISP2,4.0*>,<*ISP3,9.0*>]

specified that YouTube traffic should enter via ISPs 1, 2, and 3 in a 1:4:9 ratio by traffic volume. We summarize the syntax of the language in Table 1.

OBJECTIVE	:=	PREDICATE → ISP_CHOICE
PREDICATE	:=	USER(*User identifier*)
		| SER(*Remote service name*)
		| PREDICATE AND/OR PREDICATE
ISP_CHOICE	:=	DYNAMIC(*eval_func*)
		| BALANCE
BALANCE	:=	[<*ISP identifier, weight*>, ...]
eval_func	:=	*User-defined function over ISP capacity, flow statistics, and other metrics*

Table 1: Syntax of High-level Objective

4.2 Computing the Network Policy

Sprite collects performance metrics of the network policy and uses inputs from the edge network itself to automatically adapt the set of network policies for a high-level objective.

Mapping names to identifiers: Sprite maintains two data sources to map the high-level name of a service of group of users to the corresponding IP addresses. For users, Sprite combines the data from the device registry database of the edge network (linking device MAC addresses to users), and the DHCP records. The <user group, list of users> records are provided by the network administrators manually. For external services, Sprite tracks the set of IP addresses hosting them. Like NetAssay [12], Sprite combines three sources of data to automatically map a service's name to the prefixes it uses: *1)* the DNS records obtained from the edge network; *2)* the BGP announcements at the border routers; and *3)* traces coming from a few probe machines that emulate user traffic to popular services. Although Sprite cannot guarantee 100% accuracy, it can discover the prefixes for a majority of the service's inbound volume in practice[1].

Satisfying the TE objective: The Sprite controller translates the high-level objective into a set of clauses for the network policy, expressed as <*user prefix: port range, service prefix: port range*> → *inbound ISP*. For each clause, the Sprite agent collects the performance metrics of each matching connection from the counters of SNAT rules in data plane (e.g., throughput), to richer transport-layer statistics (e.g., round-trip time) if available [15]. The controller collects these metrics periodically, and calculates the aggregate performance. Then the data are fed to the evaluation function provided by the administrators to score how each ISP behaves. If the scores of the ISPs are different enough, the controller adapts the network policy by swapping some users from one inbound ISP to another. Sprite always keeps at least one active user on an ISP so that it can always know the actual performance of inbound traffic via an ISP through passive measurement of real traffic.

We now illustrate the process through an example. Suppose the objective is to achieve the maximum average throughput for YouTube clients. Users in the edge network are in the 10.1.0.0/22 address block. The Sprite controller initially splits the users into two groups (10.1.0.0/23, 10.1.2.0/23), and allocates their traffic with YouTube to use one of the two ISPs. Figure 3 shows the network policy generated in the iteration *T*. Carrying out the network policy, Sprite measures the throughput of each SNATed connection with YouTube, and calculates the average per-user throughput. The average inbound throughput via ISP2 is 1Mbps due to high congestion, while that of ISP1 is 2Mbps. Thus the controller decides to adapt the network policy to move some users from ISP2 to ISP1. In the iteration *T+1*, the users in 10.1.2.0/23 are further split into two smaller groups: 10.1.2.0/24 and 10.1.3.0/24. While users in 10.1.2.0/24 stay with ISP2, users in 10.1.3.0/24 have their new connections use ISP1 for their traffic from YouTube. The new net-

[1]One reason is that the major contributors of inbound traffic (e.g., Netflix and YouTube) are increasingly using their own content delivery networks (CDNs) [13, 14], rather than commercial CDNs. These services' own CDNs usually sit in their own ASes.

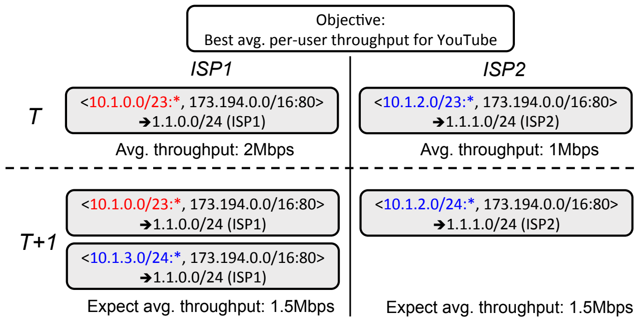
Objective:
Best avg. per-user throughput for YouTube
ISP1
ISP2
T
<10.1.0.0/23:*, 173.194.0.0/16:80> →1.1.0.0/24 (ISP1)
Avg. throughput: 2Mbps
<10.1.2.0/23:*, 173.194.0.0/16:80> →1.1.1.0/24 (ISP2)
Avg. throughput: 1Mbps
T+1
<10.1.0.0/23:*, 173.194.0.0/16:80> →1.1.0.0/24 (ISP1)
<10.1.3.0/24:*, 173.194.0.0/16:80> →1.1.0.0/24 (ISP1)
Expect avg. throughput: 1.5Mbps
<10.1.2.0/24:*, 173.194.0.0/16:80> →1.1.1.0/24 (ISP2)
Expect avg. throughput: 1.5Mbps

Figure 3: Example of Network Policy Adaptation for Dynamic Performance-Driven Balancing

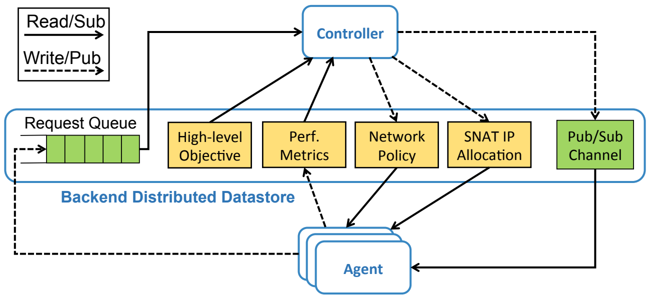

Figure 4: System Architecture of Sprite Implementation

work policy should alleviate congestion on ISP2 and might increase congestion on ISP1, leading to further adjustments in the future.

5. Implementation

In this section, we describe the design and implementation of the Sprite system and how we made it efficient and robust.

5.1 Design for Fault Tolerance

Sprite system centers on a distributed datastore (see Figure 4[2]) which keeps all the stateful information related to the high-level objective, the network policy, the performance metrics of SNATed connections, and the status of SNAT IP allocation. The controller and all the agents run independently in a stateless fashion. They never directly communicate with each other, and just read or write data through the distributed datastore.

Making the datastore the single stateful place in Sprite greatly improves the system's robustness. Indeed, device failures are common since Sprite employs a distributed set of agents and commodity switches. In this architecture, any controller or agent failure won't affect the operations of other instances or the stability of the whole system. Recovery from failures also becomes a simple task. We can start a fresh instance of controller or agent to re-fetch states from the datastore to resume the dropped operations.

The architecture also makes the Sprite system very flexible for different deployment environments. For instance, some enterprises may have standard imaging for all machines, and wish to bundle the Sprite agent in the image to run directly on the end host, while others can only place the agent side by side with the gateway routers. The adopters of Sprite can plug in/out or re-implement their own controller or agent to accommodate the deployment constraints, as long as maintaining the read/write interface with the datastore.

The implementation of the distributed datastore depends on our data model. The model of network policy involves the mapping of the four-tuple prefix/port wildcard match and the inbound ISP, and the SNAT IP allocation is the mapping among IP, ISP, allocation state, and agent. Using multiple items as the *keys*, the row-oriented, multi-column-index data structure of Cassandra is the best fit. Thus, we run Cassandra on end hosts as the datastore of Sprite.

5.2 How Components Communicate

[2]Not shown in Figure 4, we use the Floodlight controller as our SDN control module [16], and it only communicates with the Sprite controller for insertion and deletion of routing rules.

The controller and agents of Sprite interact via the datastore in a pull-based fashion. However, the pull-based approach slows Sprite in two places. Firstly, when the controller adapts the network policies, a pull-based agent may take up to its run period to pick up the new set of network policies. This significantly slows down the convergence speed of carrying out the new policy throughout the edge network, thus slowing the convergence of the policy adaptation. A second issue with the pull-based approach arises in the allocation process of SNAT IPs. When agents request the allocation of new source IPs and port ranges, new connections of users may be halted at the agent. A long wait time would trigger the connection to drop, thus affecting the user machine's performance.

We need to add a push-based communication method to balance the robustness and performance of Sprite. Thus, we add two communication points in the datastore for push-based signaling between controller and agents: a publish/subscribe channel and a message queue, as shown in Figure 4. When the controller writes new network policies or new SNAT IP allocations into the datastore, the controller publishes notification via the channel. As all agents subscribe to the channel upon startup, the notification triggers them to refresh the data from the datastore, thus catching up with the new policy or allocation state quickly. The message queue appends the agents' allocation requests to its tail, and the controller only removes the head once it successfully handles the request and updates the datastore. In this way, the message queue guarantees that each request is handled at least once. Thus users' connections are less likely to get stuck at the agents due to lack of source IPs.

5.3 Routing Control for Returning Packets

When we design to scale up the control plane of Sprite, we decide not to synchronize the SNAT states of active connections. These states are kept only locally at each agent/switch. As a result, the returning packets destined for the SNATed IP must arrive at the agent which handles the translation in the first place, in order to reverse the translation correctly.

Assuming an OpenFlow-enabled network in our implementation, Sprite installs routing rules to direct the returning packets to the right agents, i.e., once a source IP/port range is allocated to an agent, the controller installs OpenFlow rules to match the source IP/port range along the switches from the border router to the agent.

Rather than simply installing one rule per allocated source IP in switches, we try to consolidate the routing rules into matching a bigger prefix block. Our current algorithm works in this way: we construct the shorted-path tree rooted at the border router with all agents as the leaves. When allocating a source IP to an agent, we pick the one that is bit-wise closest to the IPs allocated to the agents having the longest shared paths.

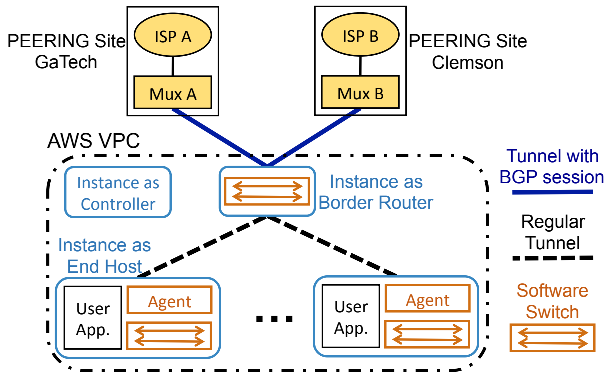

Figure 5: Multihomed Testbed Setup

6. Evaluation

We evaluate Sprite with a pilot deployment on an EC2-based testbed. Experiments in this section will demonstrate how Sprite achieves multiple TE objectives.

6.1 Multi-ISP Deployment Setup

We build a testbed in AWS Virtual Private Cloud (VPC) to emulate an enterprise network with multiple upstream ISPs, with the help of the PEERING testbed [11, 17]. PEERING is a multi-university collaboration platform which allows us to use each participating university as an ISP. Our testbed connects with two PEERING sites to emulate a two-ISP enterprise network, and both sites offer 1Gbps capacity.

Figure 5 shows the testbed setup. In the AWS VPC, we launch one machine (i.e., an AWS EC2 instance) to function as the border router. The border-router instance runs Quagga software router to establish BGP sessions with the PEERING sites in Georgia Tech and Clemson University. For each PEERING site, we have one /24 globally routable block to use.

Behind the border-router instance, we launch many EC2 instances to function as the “user” machines. These user-machine instances connect with the border-router instance via regular VPN tunnels to create a star topology. On each user-machine instance, we run the Sprite agent and OpenVSwitch. The Sprite agents uses `iptables` and OpenVSwitch to monitor and SNAT the connections. We will launch applications (e.g., YouTube) from the user-machine instances to emulate the traffic.

6.2 Inbound-ISP Performance Variance

ISPs perform differently when delivering the same service to the edge networks, e.g., YouTube and Netflix. The performance difference among ISPs can be caused by various reasons [10]. An example is the recent dispute between Netflix/Cogent and Verizon. The video quality of Netflix is bad when delivered by Verizon, due to the limited capacity of the peering links between Verizon and Netflix. In contrast, Cogent does not have the quality issue as its peering links have higher capacity,

Using Sprite, we can show that different ISPs provide different quality towards the same service by specifying an objective of equally splitting the number of users and assigning them to use one of the two ISPs. On all user machines, we launch YouTube for a 2-hour-long movie, and we explicitly set the users to stream the movie from the same YouTube access point. In the process, we

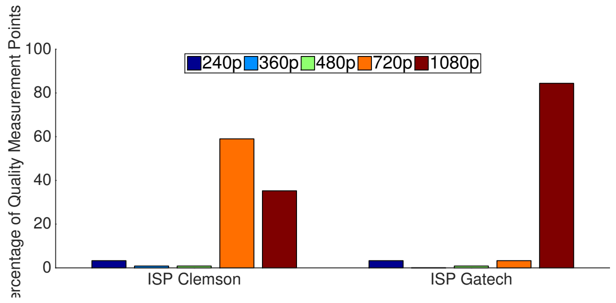

Figure 6: Video Quality Difference via Two ISPs

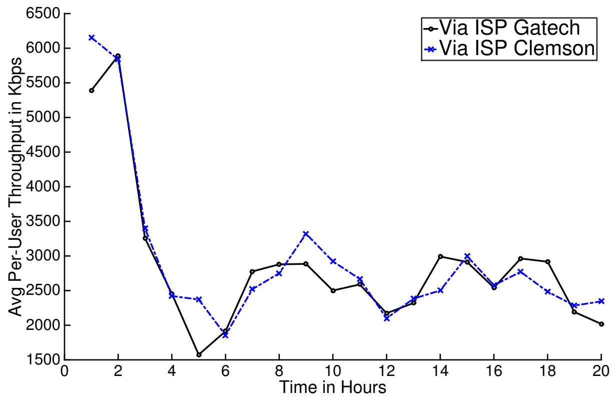

Figure 7: Time Series of Avg. Per-User Throughput of YouTube

measure the video quality of the video every 1 minute on every machine. Figure 6 shows the histogram of all these quality measurement points to examine the characteristics of the two ISPs for streaming YouTube. As Figure 6 shows, the GaTech PEERING site consistently delivers video of higher quality than the Clemson site.

6.3 Effects of Dynamic Balancing

Sprite can dynamically move traffic among ISPs to achieve the TE objective specified by the administrators. We provide an objective to achieve best average per-user throughput for YouTube traffic, and evaluate how Sprite adapts the network policies for such an objective. The objective is expressed as:

$$\mathrm{SER}(\mathit{YouTube}) \rightarrow \mathrm{BEST}(\mathit{AvgIndividualThroughput})$$

The experiment runs on the EC2-based testbed. We launch YouTube on 10 user machines. We want to examine how the traffic of users moves from one ISP to another over the time, and whether Sprite can keep the average per-user throughput roughly the same (within 5% margin) between the two ISPs. To evaluate how Sprite reacts, we manually limit the capacity of the tunnel with the GaTech PEERING site to emulate high congestion on the link. Figure 7 shows the time series of the average per-user throughput of accessing YouTube on these two ISPs. The average throughput of two ISPs are always kept in line.

7. Related Work

Many works have considered aspects of the problem we address, without providing a complete solution for direct, fine-grained, incrementally-deployable inbound TE.

BGP-based approaches Studying the impact of tuning BGP configuration to an AS's incoming traffic has a long and rich history spanning over a decade [18, 19, 20, 21, 22, 23, 24, 25], including numerous proprietary solutions [4, 26, 27]. All these solutions suffer from at least three problems. First, they are *non-deterministic*. They can only indirectly influence remote decisions but cannot control them, forcing operators to rely on trial-and-error. Second, they are too coarse-grained as they only work at the level of a destination IP prefix. Third, they often increase the amount of Internet-wide ressources (routing table size, convergence, churn) required to route traffic, for the benefit of a single AS. In contrast, Sprite provides direct and fine-grained control (at the level of each connection) without using more Internet resources.

Clean-slate approaches Given the inherent problems with BGP, many works have looked at re-architecting the Internet to enable better control over the forwarding paths. Those works can be classified as *network-based* [28, 29, 30], which modify the way the routers select paths, and *host-based* approaches which do the opposite [31, 32, 33, 34]. While these solutions can offer a principled solution to the problem of inbound traffic engineering, they all suffer from incremental deployment challenges. In contrast, any individual AS can deploy Sprite on its own, right now, and reap the benefits of fine-grained inbound traffic engineering.

8. Conclusion

Inbound traffic engineering is an increasingly important problem for the edge networks with multiple upstream ISPs. Sprite is scalable solution that offers a simple, high-level interface to network administrators. In our future work, we plan to investigate new algorithms for adapting the routing decisions based on observed performance, to identify an appropriate timescale and granularity for routing changes to react quickly to performance problems while preventing oscillations. We also plan to conduct more extensive experiments on our testbed and with the Princeton campus. We also consider supporting more features in the high-level interface, such as enforcing fairness of inbound traffic from multiple services.

Acknowledgments: We thank the Office of Information Technology of Princeton University, especially Christopher Tengi, for providing the traffic data of the campus network. We thank Nanxi Kang for initial discussions and feedback on the work. We also thank Nick Feamster, Arpit Gupta, Mojgan Ghasemi, Roya Ensafi, and the SOSR reviewers for feedback on the paper. The work is supported by NSF grants CNS-1162112 and CNS-1409056.

References

[1] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman, "A Measurement-based Analysis of Multihoming," in *ACM SIGCOMM*, 2003.

[2] D. K. Goldenberg, L. Qiu, H. Xie, Y. R. Yang, and Y. Zhang, "Optimizing Cost and Performance for Multihoming," in *ACM SIGCOMM*, 2004.

[3] A. Akella, B. Maggs, S. Seshan, and A. Shaikh, "On the Performance Benefits of Multihoming Route Control," *IEEE/ACM Transactions on Networking*, vol. 16, pp. 91–104, Feb. 2008.

[4] "Cisco Systems. Performance Routing (PfR)." http://www.cisco.com/c/en/us/products/ios-nx-os-software/performance-routing-pfr/index.html.

[5] "Cisco Optimized Edge Routing (OER)." http://www.cisco.com/en/US/tech/tk1335/tsd_technology_support_sub-protocol_home.html.

[6] "Managed Internet Route Optimizer (MIRO)." http://www.internap.com/network-services/ip-services/miro/.

[7] "netVmg's Flow Control Platform (FCP) puts you in the driver's seat." http://www.davidwriter.com/netvmgw/.

[8] "Sockeye's GlobalRoute 2.0 for managed routing services." http://www.networkcomputing.com/networking/sockeyes-globalroute-20-for-managed-routing-services/d/d-id/1204992?

[9] "Google chooses RouteScience Internet technology," July 2002. http://www.computerweekly.com/news/2240046663/Google-chooses-RouteScience-Internet-technology.

[10] D. Clark, S. Bauer, K. Claffy, A. Dhamdhere, B. Huffaker, W. Lehr, and M. Luckie, "Measurement and Analysis of Internet Interconnection and Congestion," in *Telecommunications Policy Research Conference (TPRC)*, Sep 2014.

[11] V. Valancius, N. Feamster, J. Rexford, and A. Nakao, "Wide-area Route Control for Distributed Services," in *USENIX ATC*, 2010.

[12] S. Donovan and N. Feamster, "Intentional Network Monitoring: Finding the Needle Without Capturing the Haystack," in *ACM HotNets*, 2014.

[13] "Netflix Open Connect." http://openconnect.itp.netflix.com/.

[14] "Google Global Caching." http://peering.google.com/about/ggc.html.

[15] P. Sun, M. Yu, M. Freedman, J. Rexford, and D. Walker, "HONE: Joint Host-Network Traffic Management in Software-Defined Networks," *Journal of Network and Systems Management*, vol. 23, no. 2, pp. 374–399, 2015.

[16] http://floodlight.openflowhub.org/.

[17] B. Schlinker, K. Zarifis, I. Cunha, N. Feamster, and E. Katz-Bassett, "PEERING: An AS for Us," in *ACM HotNets*, 2014.

[18] R. K. Chang and M. Lo, "Inbound traffic engineering for multihomed ASs using AS path prepending," *IEEE Network*, vol. 19, no. 2, pp. 18–25, 2005.

[19] R. Gao, C. Dovrolis, and E. W. Zegura, "Interdomain Ingress Traffic Engineering Through Optimized AS-path Prepending," in *Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems*, pp. 647–658, Springer, 2005.

[20] F. Wang and L. Gao, "On Inferring and Characterizing Internet Routing Policies," in *Internet Measurement Conference*, pp. 15–26, ACM, 2003.

[21] L. Cittadini, W. Muhlbauer, S. Uhlig, R. Bush, P. Francois, and O. Maennel, "Evolution of Internet Address Space Deaggregation: Myths and Reality," *Journal on Selected Areas in Communications*, vol. 28, no. 8, pp. 1238–1249, 2010.

[22] B. Quoitin, C. Pelsser, L. Swinnen, O. Bonaventure, and S. Uhlig, "Interdomain Traffic Engineering with BGP," *IEEE Communications Magazine*, vol. 41, pp. 122–128, May 2003.

[23] N. Feamster, J. Borkenhagen, and J. Rexford, "Guidelines for Interdomain Traffic Engineering," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 5, pp. 19–30, 2003.

[24] B. Quoitin, S. Tandel, S. Uhlig, and O. Bonaventure, "Interdomain Traffic Engineering with Redistribution Communities," *Computer Communications*, vol. 27, no. 4, pp. 355–363,

2004.

[25] B. Quoitin and O. Bonaventure, “A Cooperative Approach to Interdomain Traffic Engineering,” in *Next Generation Internet Networks*, pp. 450–457, IEEE, 2005.

[26] “Internap. Managed Internet Route Optimizer (MIRO).” http://www.internap.com/network-services/ip-services/miro/.

[27] “Noction. Intelligent Routing Platform.” http://www.noction.com/intelligent_routing_platform.

[28] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, “The Locator/ID Separation Protocol (LISP).” IETF Request for Comments 6830, January 2013.

[29] L. Subramanian, M. Caesar, C. T. Ee, M. Handley, M. Mao, S. Shenker, and I. Stoica, “HLP: A Next Generation Interdomain Routing Protocol,” in *ACM SIGCOMM*, Aug. 2005.

[30] A. Feldmann, L. Cittadini, W. Mühlbauer, R. Bush, and O. Maennel, “HAIR: Hierarchical Architecture for Internet Routing,” in *Workshop on Re-architecting the Internet*, pp. 43–48, ACM, 2009.

[31] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson, “Host Identity Protocol,” April 2008. RFC 5201.

[32] E. Nordmark and M. Bagnulo, “Shim6: Level 3 Multihoming Shim Protocol for IPv6.” IETF Request for Comments 5533, June 2009.

[33] R. J. Atkinson and S. N. Bhatti, “Identifier-Locator Network Protocol (ILNP) Architectural Description.” RFC 6740, Nov 2012.

[34] C. De Launois, O. Bonaventure, and M. Lobelle, “The NAROS Approach for IPv6 Multihoming with Traffic Engineering,” in *Quality for All*, pp. 112–121, Springer, 2003.