–Independent Work Report Spring,2014–

SRON: A Coordinated Overlay Network via Software-Defined Networking

Abstract

In this paper, we consider the design and performance of a Software-Defined Resilient Overlay Network (SRON). While many overlay networks are deployed on end hosts where they are then configured in a distributed, autonomous fashion at the "edge" of the Internet, our overlay network is run on backbone switches that support the OpenFlow protocol, and whose routing tables are coordinated by a centralized controller. Running an overlay on network switches allows our overlay nodes to route packets with much higher bandwidth than overlay nodes that run on end hosts, since the hardware in backbone switches is tailored to routing a large number of packets quickly. Furthermore, while data forwarded over end host overlay nodes must pass through many hops from the Internet backbone to peripheral end hosts, routing through switch allows data to stay within the backbone, cutting out several slow hops at the periphery of the network. The fact that SRON is coordinated by a central OpenFlow controller also allows us to use novel probing techniques that would be impossible in a distributed overlay; for example, SRON is able to compare the performance of unidirectional paths between overlay nodes, whereas in all overlay designs of which the author is aware, only round trip path performance is measured. This centralized controller also means that SRON is able to avoid generating transient routing loops, since when the controller pushes routing updates to switches, it has full knowledge of the routing rules on all switches in the network. As SRON is designed to implemented on SDN switches distributed throughout the Internet, we imagine ISPs may use SRON to make more intelligent routing decisions across AS boundaries. Although the results of our investigation did not yield promising performance results for SRON, it did shed light on some of the unusual concerns overlay designers might face in design Software-Defined Overlays.

1. Introduction

The Internet was designed to be, above all else, scalable. This decision has allowed it to accommodate billions of users across the world, and central to this ability is the fact that the Internet is a memory-less, datagram-switching network. Every router through which a packet passes determines only the next hop for that packet. As a consequence, however, this lack of central coordination may lead to sub-optimal routing decisions, and leaves end hosts without control of the routing of their data. Central to this limitation is the design of the Border Gateway Protocol (BGP), which determines routing between Autonomous Systems (AS's). BGP allows Autonomous Systems (AS's) to advertise or hide available routes to one another so as to conform with complex peering contracts between Internet Service Providers. AS's route via BGP independently of one another, using private route preferences assigned by network administrators in a way that may be financially optimal for the AS but slower for the customers routing data through them. Andersen, Balakrishnan, Kaashoek, and Morris note that "This cost arises because BGP hides many topological details in the interest of scalability and policy enforcement, has little information about traffic conditions, and damps routing updates when potential problems arise to prevent large-scale oscillations"[4].

An overlay network, however, can often make better routing decisions than those of BGP. In an overlay network, a set of network nodes (which may consist of, for example, end hosts running an application-level overlay program) chooses to route data *on top* of the existing Internet by imagining each node in the overlay connected by "virtual links". These virtual links may consist of hops over several physical links, with routing between nodes determined natively by Internet routers. The overlay allows for faster routing than the Internet when the direct path between two nodes in the overlay is slower than a two-virtual-link hop path through nodes in the overlay (i.e. when routing through an intermediary overlay node is faster than routing between the two nodes directly). Andersen et al. at MIT leveraged this fact when they created RON[4], a "Resilient Overlay Network". RON consisted of software running on several end hosts distributed throughout the Internet, working together to decide how packets should be routed through the nodes and links in this overlay. As a result, nodes in RON were able to communicate despite outages must faster than hosts routing through the Internet alone.

While a significant body of work addresses the design of such resilient overlays, none known to this author to date considers an overlay network implemented in a Software-Defined Network (SDN). An SDN overlay presents several advantages over a distributed, end host overlay. For example, the existence of a centralized SDN controller in an overlay network means routing decisions can be made by an entity with a holistic view of the nodes in the overlay, and thus more intelligently than routing decisions made by distributed end hosts. Furthermore, overlays implemented on switches within the Internet backbone are able to route packets much more quickly and in higher quantity than end hosts at the periphery of the Internet.

SRON is such a Software-Defined Overlay. We envision it being used by Internet Service Providers who wish to route more efficiently between their routers and switches that may be scattered widely across the Internet and connected to each other through various other ISP's.

2. Motivations

2.1. SRON: A Software-Defined Approach

Current overlay networks typically have been deployed as application-level programs that run on end hosts. In these overlays, the network nodes are generally bootstrapped into the overlay and participate in routing via a distributed routing protocol (i.e. link state routing)[4]. This implementation makes such an overlay network easy to deploy. That being said, distributed routing protocols have several disadvantages over non-distributed ones. For example, transient routing loops may occur in networks running distributed protocols in the time between when a link goes down and when every node has been informed of the network topology change; this occurs because nodes update their routing tables in response to topology changes in a non-coordinated fashion. Furthermore, routing decisions made at each node can only consider topology at a local scale. In this model, there is nothing to prevent–and indeed it is likely to happen–that two nodes, upon finding a non-congested link, both decide to forward data on that link, and quickly their knowledge that the link was non-congested becomes scale.

SRON, however, is an overlay designed to be deployed by a Software-Defined Network. We imagine SRON being deployed to ISP's who may have SDN-compatible switches and routers scattered throughout the world, some of which may need to route to one another through intermediate switches owned by another ISP. Thus an ISP would deploy SRON on its SDN nodes in order to make more intelligent routing decisions across Autonomous System boundaries, since this is where the suboptimal routing decisions of BGP come into play. ISP's may also be able to use SRON to lease virtual network slices to clients with application-specific routing needs.

2.2. Centralized Control

As is characteristic of SDN, SRON assumes a separation of the control and data planes; a centralized controller (which may still be distributed across the Internet) communicates with all nodes in the SRON overlay and updates the routing tables of these switches/routers based on the results of probing paths between them. Since routing is decided by a single entity, every routing change at a switch is made with full knowledge of the routing rules at every other node in the network (i.e. we no longer get transient loops as described above).

2.3. Deployment on the Internet Backbone

A second advantage SRON has over typical overlays is that routes packets over Internet switches rather than through end hosts. When a packet leaves the Internet backbone to an end host, it must pass through many more hops than if the packet were to be routed through switches directly (see Figure 1). The path between two end hosts in separate networks includes as a subpath the path between the two backbone switches that connect the end hosts to the Internet; hence SRON may save many routing hops at the periphery of the network. Furthermore, switches simply route data much faster than end hosts since their hardware is designed to do so, and indeed in some cases routing is implemented entirely or largely in hardware. Thus we believe SRON should be significantly faster than an end host overlay solution.

Hierarchal Structure of the Internet

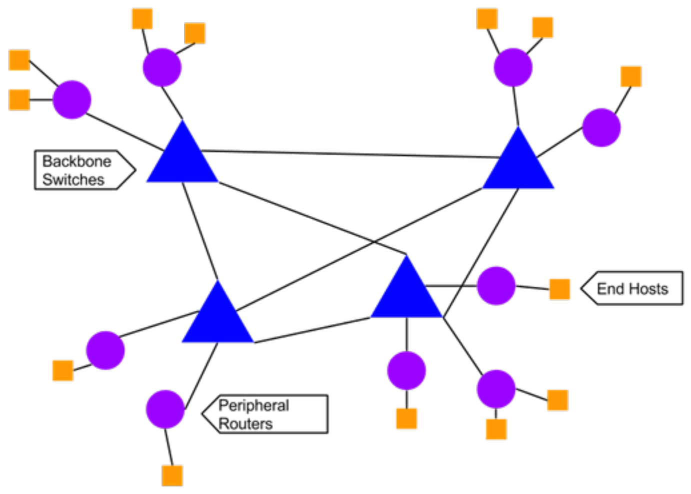

Figure 1: Here, the triangles represent backbone Internet switches, the circles represent peripheral routers (for example, routers/switches for a University), and the squares represent end hosts. By this hierarchal design, there are many fewer hops between two nearby switches than two nearby hosts.

3. Past Work

3.1. Wide-Scale Internet Performance

BGP is the path selection algorithm used by backbone Internet routers to make decisions about routing between Autonomous Systems. Interior gateway protocols or IGP's, on the other hand, are routing algorithms used by individual AS's to decide paths within the AS. IGP's like the frequently-used OSPF[6]–a shortest-path link state routing protocol–determine paths by examining link performance and least number of hops. While AS's may be motivated to route packets to customers most efficiently when communication is within their domain, business concerns play a significant role in determining BGP's "policy" routing decisions; that is, BGP allows AS's to hide routes from other AS's if, for example, the two AS's do not have an explicit agreement to share infrastructure. Furthermore, an AS may choose to pass a client's data to another AS so that it has more capacity on its infrastructure, even if such a path is sub-optimal. Varadhan, Govindan, and Estrin proved that BGP policies may not even lead to routing convergence–that is, AS's may choose to route in such a way that leads to persistent oscillations[17]. Labovitz et al.[11] conducted a two year study of Internet routing convergence and found router may take tens of minutes to converge after a fault due to BGP's routing table oscillations.

3.2. Overlay Networks

Overlay networks have been deployed to support several applications that the Internet alone cannot; for example, Content Distribution Networks (CDN's) are overlays that dynamically cache content across the Internet and deliver it to end users to avoid congestion. Today, CDN's make up the bulk of overlay traffic[10].IP multicast–the method of sending a single packet to multiple receivers in a

single transmission–has also been cited as a promising application of overlays. IP multicast makes applications with multiple receivers more efficient, but is difficult to deploy and today is mostly implemented within a single domain rather than spanning multiple domains[10]. MBone[12] was an early overlay that attempted to connect multicast domains by using IP tunneling to transport data across domain boundaries. Unfortunately, MBone experienced unreliability, heavy loss, and low throughput[10]. Narada[5] was proposed as an application-layer alternative to IP multicast, in which overlay nodes created their own routing mesh atop the Internet and serviced multicast in this way.Another important application of overlay networks is end-to-end performance and resiliency;

The Detour[16] overlay used 45 distributed servers to probe paths between them using the traceroute program; they were curious to know whether or not data could be routed from points A to B more efficiently by routing data through one of their overlay nodes C (i.e. could they choose faster routing paths than the Internet natively). For roughly half of the paths they measured, they found an alternative route which was faster. RON[4] was an overlay network that, inspired by the findings of Detour, used frequent probing in combination with link state routing to find low-latency or high-bandwidth paths between overlay nodes. RON was implemented as on end hosts, and these overlay nodes were arranged in a fully connected mesh. The fact that RON was fully connected–each of RON's n nodes had $(n-1)$ neighbors–and used frequent probing meant that it could not (and was not intended to) scale beyond about fifty nodes. However, a 12 node RON overlay was still able to overcome almost all routing outages in under twenty seconds. Later modifications to the basic RON design allowed for overlay networks that could find paths with similar performance to RON nodes with much less probing. Akamai's Sureroute overlay was similarly based on findings by Detour, and selected a best path by holding a race between them, as is implemented in SRON[10]. Another interesting modification to RON was work by Gummadi et al.[8]; in this mesh, routing is done as follows: when a node at A wants to send data to B, it randomly chooses k nodes in the overlay and routes through them; it then uses the intermediary with the first response to route all subsequent packets, and if at any time this path becomes unresponsive, it chooses a second random k nodes and tries again. They found that $k = 4$ gives near maximum benefits.

3.3. Software-Defined Networking

SRON is implemented on a Software-Defined Network–that is, a network on which routers and switches are programmable, and whose routing rules can be updated dynamically from an SDN controller. In SDN, the *control plane*, which makes decisions about how traffic should be handled, is separated from the *data plane*, which itself forwards the traffic[7]. The control plane is embodied by an SDN controller which communicates with network switches and routers in the data plane via an API. In SRON, we use a popular API known as OpenFlow[13], a protocol supported by most major switch vendors. OpenFlow is an API useful for programming several compliant devices, from switches to routers to middleboxes, as it allows devices to forward data based on matching header fields such as source and destination MAC and IP Addresses, TCP ports, VLAN ID, in port, and more. One of the largest SDN deployments has been B4[9], a worldwide private LAN deployed by Google on OpenFlow-compliant, off-the-shelf switches. SDN's are also commonly used to manage network in data centers.

SRON Deployment

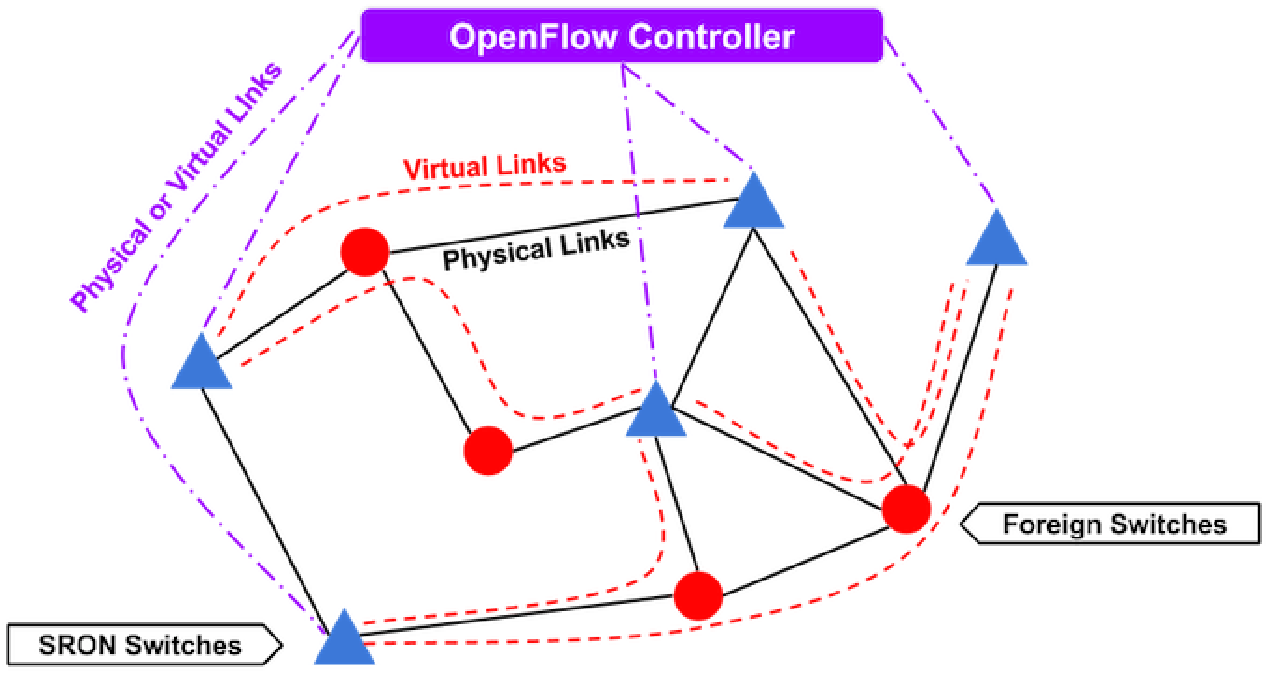

Figure 2: Here, the triangles denote routers and switches in SRON, which we can program via an OpenFlow controller. SRON switches may be distributed throughout the Internet and connect to each other via other switches that are not within the same ISP's domain; these "foreign" switches that are not controlled by the SRON controller are displayed as circles; SRON nodes are connected in a clique topology via virtual links that may be comprised of multiple hops over physical links, as denoted by the dashed line in this figure. The connections between the SRON controller and switches may be direct, physical links or virtual links.

4. Design of SRON

4.1. Probing Scheme

In SRON, our goal is to measure the latency on the virtual links between nodes and make reactionary routing decisions. Here we say "virtual" because these links may consist of a sequence of hops across switches and routers in the Internet. At first, the problem of measuring link latency might seem straightforward–send a packet between two switches in the network and measure the time it takes to get between them. Unfortunately, this problem is not so simply solved; for one, it is almost impossible for two nodes in the Internet to measure one-way trip time, since clocks on these nodes are not necessarily synchronized (and so packets can't be timestamped). While we can measure path round trip time (RTT), there may be a large disparity the latency on either half of the path (indeed, asymmetric paths is a common consequence of policy routing via BGP). Thus if we were to use RTT as an indication of the performance of a link, we may find that we ignore a fast link when RTT is high but only one direction of the connection is slow, or prefer a slow link when RTT is low due to one direction of the connection being fast.

An alternative approach we might consider in a Software-Defined Network is to have the network controller measure the time it takes for a packet to travel from one switch to another. It might do this by instructing $Switch_A$ to send a packet or "probe" to $Switch_B$, and having $Switch_B$ pass the packet back up to the controller. This probe packet may be marked with a distinguished IP address, *PROBEIP*, to allow switches to treat it as a probe rather than normal network traffic. Since the controller is connected to either probe, it can measure the time it takes for a packet to get from the controller to switch A to switch B and back up to the controller. This avoids the problem of measuring round-trip time from A to B, but is only a valid measurement if the path between controller and switches is short compared to the time it takes for a packet to travel from A to B. Since the controller-switch connection is likely to run over the Internet, though, we do not believe this would be a valid measure of the latency between A and B because the time it takes for a packet to get from the controller to either of these switches may well rival that of the path time between switches.

Our measurement of link performance in SRON does not attempt to calculate the time taken along virtual links in the network, but evaluates routes only by their relative performance to other routes. For example, a path p from $Switch_A$ to $Switch_B$ is "better" than path p' from $Switch_A$ to $Switch_B$ if packets sent from A along p reach B faster than those sent from A along p'. In SRON, we do this by sending probes along all single and double hop paths from A to B. SRON runs atop a clique topology, where every SRON node is connected to every other SRON node via a virtual link. To determine the best path from $Switch_A$ to $Switch_B$, we have the controller inject probe packets–packets with designed source IP addresses *PROBEIP*–to $Switch_A$. These packets are also given a special destination IP address that denotes that this probe packet originated from $Switch_A$.

Switch Rules for Handling Probes

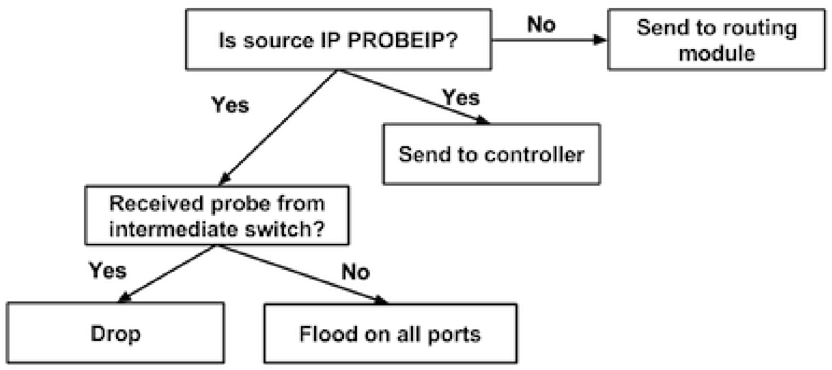

Figure 3: If a switch receives a probe with the source IP address $PROBEIP$, it sends the probe to be logged at the controller; furthermore, if this switch is the first hop in the probe's path–determined by the destination IP of the probe–then the switch forwards it on all ports. Otherwise, if this switch is the second hop, it drops the probe.

$Switch_A$ is instructed by the controller to forward these probe packets to all other switches in the networks. When a second switch, $Switch_C$, receives such a packet, it recognizes it as a probe by checking that the source IP is $PROBEIP$, and sends the packet up to the controller. This probe gives the controller information about the performance of the direct path between $Switch_A$ and $Switch_C$. We would also like to know about the performance of paths originating at $Switch_A$ that use $Switch_C$ as an intermediate hop en route to a third switch. Thus, if $Switch_C$ finds that it received this probe directly from $Switch_A$ rather than an intermediary node–it does this by comparing the destination IP address of the probe to the port on which it received this probe (hence determining if the probing switch directly sent the packet)–it floods the probe on all ports that are not connected to $Switch_A$. Thus eventually $Switch_B$ will receive a probe originating at $Switch_A$ but passing through $Switch_C$ (since the network is a clique), and will send this packet up to the controller. Thus the controller has knowledge about all single and two-hop paths from $Switch_A$ to $Switch_B$, and knows their ordering by comparing the order in which it received probes amongst all paths.

Evaluating routes in this way is convenient because by having the controller inject only $n(n-1)$ packets into the network, where n is the number of nodes, we are able to learn the relative speed of all single and double hop paths in the overlay. We also have the benefit that, when $Switch_A$ sends a probe to $Switch_B$ and $Switch_B$ forwards this packet over all ports, we explore all paths of the form $Switch_A -> Switch_B -> Switch_i$ for $i \neq A, B$ while only sending a single packet over the the link from $Switch_A -> Switch_B$. This reduces the traffic and time it would take to probe all double hop paths individually.

4.2. Migrating Routes

How does the controller decide when to update flow rules between two switches in the network? On one hand, we would certainly like to update connections between switches to faster paths if possible, and would like to do so quickly or at least fast enough to provide a significant benefit over Internet routing alone. On the other hand, pushing routing updates too quickly may lead to route oscillation, in which data cannot pass through the network because the routing rules change so quickly that no meaningful path between endpoints exists. Furthermore, Internet protocols such as TCP assume that routing changes are relatively infrequent during a connection (this is a crucial component of its congestion control mechanism), and so frequent routing changes may render this technique ineffective. Thus we need to set a threshold for when we decide to migrate routes, and should also keep a history of the performance of different routes so that our updating mechanism is robust against short term fluctuations in route performance.

We do this as follows: For each pair of switches, we keep a history of the performance of probes that have taken different paths between those two switches. For example, for $Switch_A$ and $Switch_B$, we calculate a metric for each path between them– the direct route between $Switch_A$ and $Switch_B$ as well as the route from A to B through C, A to B through D, and so on. Here is how we calculate the metric of a path p with sequence number n: a probe is sent along every path from A to B and each probe arrives at the controller in some order. Suppose the probe sent along path p is the i'th probe to arrive at the controller amongst all probes sent between A and B with sequence number n. Then the metric $metric(p,n)$ is calculated as an exponentially weighted moving average of the order the probe p arrived amongst all probes between $Switch_A$ and $Switch_B$:

$$metric(p,n) = (1-\alpha)metric(p,n-1) + \alpha i$$

Here, the lower value of i the better, since this means the probe arrived at $Switch_B$ relatively quickly. We choose $\alpha = 0.1$ since this is the value used by RON to calculate latency[4], and similar to the TCP value used to calculate round trip time. Suppose the current route used to route data between A and B is route p. We update this route to p' when $metric(p', n) \leq \beta metric(p, n)$, where we choose $\beta = 0.7$. By this metric, we migrate from p to p' when p' is 30% "faster" than p.

4.3. Delayed Probes

When a switch sends probes to "race" through the network from probing switch to destinations, it is possible that some probe will be so delayed in the network that it will only reach its destination after the next round of probing has begun. In this case, it may be that the delayed probe appears to "win the race" if it arrives at a destination just after a new set of probes has been sent out. In order to avoid this ambiguity, we have each probing switch mark probe packets with a monotonically increasing 16-bit counter that is stored in the packet "protocol" field. When the controller receives a probe from the network with an old sequence number, it logs that route with a metric indicating the probe was the last to arrive on the source to destination path. The first packet that the controller receives with a sequence number which is higher than the highest previously seen sequence number is logged with the best possible route metric, and the counter on the controller is updated to store that value.

4.4. Learning IPs

Because SRON routes at the IP level, it must have a way of dynamically associating IP addresses with switches. We achieve this in a way similar to how switches learn MAC-to-port mappings. When a switch receives a packet with an unrecognized source IP address, it sends the packet to the controller. If, for example, the incoming packet enters the network on $Switch_A$ and has IP 10.0.0.1, the controller logs this association. It then examines all rules it has for routing between other switches in the network and $Switch_A$ (as determined by probing as described above). The controller then updates all switches in the network to use this best path to route all packets with the destination IP address 10.0.0.1. If, on the other hand, a switch receives a packet with an unknown

Switch Rules for IP Learning

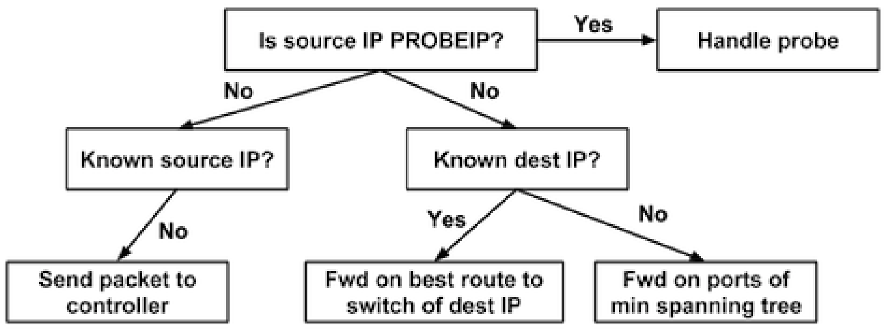
Is source IP PROBEIP?
Yes
Handle probe
No
No
Known source IP?
Known dest IP?
No
Yes
No
Send packet to controller
Fwd on best route to switch of dest IP
Fwd on ports of min spanning tree

Figure 4: When a switch receives a packet with an unknown source IP, it sends it to the controller so that the controller can log the association between IP address and source switch, and update forwarding rules on switches appropriately. If the destination IP address of a packet is known, it is routed according to best path rules determined by probing. Otherwise, the packet is flooded on all ports in the minimum spanning tree.

destination IP, it sends the packet to an underlying mac learning module which forwards the packet to all ports along the network minimum spanning tree.

5. Implementation

5.1. Pyretic

We decided to build SRON in Pyretic[15], a Python platform for programming SDNs. Most major switch vendors support the OpenFlow API for programming switches, but writing OpenFlow messages directly is tedious, akin to programming in assembly. Pyretic sits above an OpenFlow controller platform (in our case, POX[1]), and allows programmers to write SDN programs in a high-level, modular way. For example, consider the following excerpt from SRON that updates the network routing policy:

We use the `if_(`*predicate*`, policy1, policy2)` to indicate that all switches receiving a packet with source ip *PROBEIP* should route that packet using `probingPolicy`, which includes both forwarding that probe over all ports or dropping it (depending on whether the switch receiving this packet is the first or second hop on its path), as well as sending the packet up to the con-

```
def updatePolicy(self):
 print "Updating policy..."
 # probingPolicy explains how probes should be routed. self.query explains how the controller should log
     receieved probes
 probingPolicy = self.metricsPolicy + self.query

 # If a reRoutingPolicy has been created based on probe metrics, use it. Otherwise revert to a mac learner.
 if self.reRoutingPolicy:
   self.policy = if_(match(srcip= PROBEIP), probingPolicy, self.reRoutingPolicy)
 else:
   self.policy = if_(match(srcip= PROBEIP), probingPolicy, self.macLearn)
```

troller to be logged. If the packet is not a probe packet, then it should either be routed via the `reRoutingPolicy` or, if no such policy exists (for example, when the network is first brought up), should be routed via a simple mac learning module. Note that rules in Pyretic are not written for individual switches. Rather, rules for routing packets throughout the entire network are updated when the variable `self.Policy` (where here the parent object is of type `DynamicPolicy`) is updated. Instructing only a particular switch to perform an action can be specified by creating a rule of the form: `match(switch=someSwitch) » somePolicy`. Indeed Pyretic makes SDN programming concise.

6. Evaluation

6.1. Experimental Setup

To test the performance of SRON, we used a network simulation running on Mininet[2]. Mininet is a network emulator that can simulate switches, end hosts, and routers on a Linux kernel. We created a network of switches connected to end hosts, with the switches arranged in a fully-connected mesh. In order to simulate the fact that these links are virtual rather than physical links–that is, that they consist of many sometimes unreliable hops through the Internet rather than direct links–we had pairs of end hosts across these links generate variable amounts of traffic across them. To generate this traffic, we used Harpoon, an open-source flow-level traffic generator that simulates network traffic. For each pair of nodes, we set up a Harpoon client and server on either end of the nodes, that continuously exchanged files of varying sizes. We tagged the IP addresses of these hosts to be in IP range that SRON would not reroute, so that this traffic would represent the routing of the underlying Internet. In theory, SRON should find the pair of paths between switches whose links carry the least Harpoon traffic. In order to measure the bandwidth of these links, we set up a pair of iperf servers and clients on the same switches as the Harpoon clients and servers.

6.2. Results

Unfortunately, we did not find consistent results when testing SRON in this way. For one, we found that SRON took a significantly longer time (on the order of seconds) than a simple Mac Learner module to route packets through a non-congested network. Although we are not probing at an ambitious interval (every 5 seconds), we believe this may be caused by the fact that SRON can only push routing rules to switches when it learns of the existence of an IP address; in other words, if SRON knows the best route between $Switch_A$ and $Switch_B$, it pushes these rules to all switches as soon as it learns of the existence of an IP at either $Switch_A$ or $Switch_B$. As a result, the controller floods the network with routing updates at once.

Furthermore, we found under any significant amount of Harpoon traffic, SRON was unable to find shorter two-hop paths between hosts and could perform no better than the Mac Learner module. We believe this may be due to the difficulty of pushing routing updates to switches when they are experiencing heavy traffic. Unlike an end host overlay, where the underlying links between nodes may be experiencing failures but the overlay nodes themselves still have the processing power to make routing updates (i.e. the nodes in an end host overlay are not overwhelmed), in our overlay, the time at which it is most advantageous for a switch to push a routing table change is precisely when that switch is most overwhelmed with traffic. We believe this may explain the poor performance of SRON under heavy traffic.

6.3. Future Work

Our finding that our design of SRON did not perform well under heavy network loads was admittedly disheartening, but inspires us that SDN's may have their own unique and interesting requirements for supporting overlay networks!

That being said, it is difficult for any network simulation to capture all of the subtleties of real Internet traffic, and indeed our Mininet simulation of the Internet could not have given us fully accurate representation of how SRON might perform if deployed across the globe . In the future, we would like to test SRON's performance when deployed on a PlanetLab[3] slice. PlanetLab is a network of computers consisting of over 1090 nodes at 507 sites worldwide, which is designed for testing experimental network and distributed systems research projects. With a virtual machine or "slice" of the PlanetLab network, we could test SRON across AS boundaries and with real network traffic, so that the difficult-to-predict decisions of BGP's policy-based routing could be compared with SRON.

As for making SRON ready for deployment, the current software design is closely tied to the original design of RON, which has since been improved upon in work by Nakao et al. [14], Gummadi et al.[8], and more. As mentioned in Section 3, this research has discovered techniques to reduce the amount of probing between nodes as well as ways to more intelligently use knowledge about the real network topology to design the overlay network topology.

7. Conclusion

We consider our work on SRON to have been an investigation into the deployment of overlays on SDN's. Much to our surprise, we discovered that SRON performed poorly. While we would like to further investigate and confirm that it is indeed the design of SRON that makes it unsuccessful, we are interested in any case in seeing SDN overlays be studied more closely in the future.

8. Acknowledgments

This work would have been impossible without the constant guidance and feedback of Jennifer Rexford.

References

[1] [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[2] [Online]. Available: http://mininet.org/

[3] (2012, 09) Planetlab. [Online]. Available: http://www.planet-lab.org/

[4] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, "Resilient overlay networks," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 131–145, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502048

[5] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 205–217, Aug. 2002. [Online]. Available: http://doi.acm.org/10.1145/964725.633045

[6] R. Coltun, "Ospf for ipv6," 7 2008.

[7] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602219

[8] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall, "Improving the reliability of internet paths with one-hop source routing," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251267

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534169.2486019

[10] J. Kurian and K. Sarac, "A survey on the design, applications, and enhancements of application-layer overlay networks," *ACM Comput. Surv.*, vol. 43, no. 1, pp. 5:1–5:34, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1824795.1824800

[11] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed internet routing convergence," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 175–187, Aug. 2000. [Online]. Available: http://doi.acm.org/10.1145/347057.347428

[12] M. R. Macedonia and D. P. Brutzman, "Mbone provides audio and video across the internet," *Computer*, vol. 27, no. 4, pp. 30–36, Apr. 1994. [Online]. Available: http://dx.doi.org/10.1109/2.274996

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[14] A. Nakao, L. Peterson, and A. Bavier, "Scalable routing overlay networks," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, pp. 49–61, Jan. 2006. [Online]. Available: http://doi.acm.org/10.1145/1113361.1113372

[15] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic."

[16] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan, "Detour: Informed internet routing and transport," *IEEE Micro*, vol. 19, no. 1, pp. 50–59, Jan. 1999. [Online]. Available: http://dx.doi.org/10.1109/40.748796

[17] K. Varadhan, R. Govindan, and D. Estrin, "Persistent route oscillations in inter-domain routing," *Computer Network*, vol. 32, pp. 1–16, 2000.