–Independent Work Report Fall,2012–

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 invernet, our overlay network is run by switches coordinated by a controller that has a complete view of the switches in this overlay.

1. Introduction

The Internet must be scalable, so an overlay can do things the Internet cannot

The Internet was designed to be, above all else, scalable. This decision has allowed it to accommodate billions of users accross 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 the end hosts without a say in 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). 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"[1].

While the design of the Internet may be suited to an ever-expending network of hosts, a small set of network nodes that wish only to communicate amongst themselves may be able to, by sacrificing the ability to scale to an arbitrary number of nodes, make better routing decisions than the Internet alone. Andersen et al. at MIT leveraged this fact when they created RON, a "Resilient Overlay Network". RON was designed to run on a fixed number of end hosts, and allowed these hosts to "route" through one another by envisioning a model wherein the connection between two nodes in the overlay–which is made up of some multi-hop path determined by Internet routers–was thought of as a single, virtual link in an overlay network. The end hosts in RON then worked together to decide how packets should be routed through the ndoes and links in this overlay. As a result, nodes in RON were able to communicate despite outages must faster than hosts in the Internet.

Software-Defined RON

In this paper, we explore how implementing an overlay network like RON on a SDN platform may provide unique performance advantages over a network without an overlay and over a distributed overlay network like RON. We also present such an SRON that is implemented in the networking programming language Pyretic, and explore its performance characteristics.

2. Past Work

2.1. 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[2]–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 significnat 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 persisent oscillations[4]. Labovitz et al.[3] 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. The creators of RON found that an overlay network with a limited number of nodes could overcome outages and find lower-latency paths much more quickly than BGP alone.

3. A Software-Defined Approach

Why is RON, others distributed rather than centrally coordinated?

In the past, overlay networks have been deployed on end hosts and coordinated in a distributed fashion. Indeed this was a natural choice, since users of overlay networks were often individuals scarcely located throughout the Internet– for example, users of the Tor Network (awkward). A more centralized control would require not only coordination amongst network users but also would likely have significant bandwidth overhead. If an overlay need be coordinated by a single entity, then there would have to be some efficient method of communication with that coordinator, but finding efficient routes between overlay nodes is exactly the problem that overlay networks are designed to solve–a chicken-and-egg problem.

SRON runs on a controller + switches

In SRON, we consider a different application of overlay networks. Like a typical overlay, SRON consists of nodes that probe each other to determine link latencies, but these nodes are network switches rather than end hosts. Furthermore, a central SDN controller coordinates the probing and routing rules of the switches in the network.

Motivations

At first blush, one might wonder why an overlay network is necessary on a Software-Defined

Network. After all, generally these networks are deployed by organizations that have entire control over the links in the network (for example, a university network, or Google's private backbone). However, we imainge SRON being deployed on the networks of Internet providers that may span the globe, and who may need to pass data across other ISPs. In this case, we imagine that the network controller is distributed, and the switches is controls are scattered across the globe. The links between our switches are not physical links, but a sequence of hops across the Internet. Some of the nodes in the multihop paths between switches may belong to the ISP deploying the overlay, but others may be managed by other ISPs. Thus the SRON overlay network model assumes that all nodes within it are fully connected, but that the links between them have varying performance characteristics.

Advantages of SRON over RON Why is such a Software-Defined overlay useful? RON[1] showed that overlay networks can perform significantly better than the raw Internet in terms of overcoming routing outages. That being said, routing packets on end hosts has several disadvantages; for one, end hosts are much slower at routing data than network switches, whose hardware is custom-tailored for this task. Second, because routing is done in a hierarchial fashion in the Internet, packets must pass through many more hops to get to an end host (that is at the periphery of the Internet) than a network switch. **This part is really questionable:** For example, suppose a host in the Computer Science Department at Princeton were running an overlay node. While there is likely only a single Cisco switch allocated to all of Princeton, a packet destined to be routed through our host node would have to pass first through this switch, then through Princeton's University switches, then through the Computer Science Department routers, and back again.

Any more reasons to mention?

4. Design of SRON

4.1. Probing Scheme

Rational/all the reasons why probing other ways doesn't work

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 is not so simple; for one, it is almost impossible for two nodes in the Internet to measure one-way trip time, since we cannot assume clocks on these nodes are synchronized (and so packets can't be timestamped). While we can measure path roundtrip 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 performace of a link, we may find that 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 instruct switch A to send a packet or "probe" to switch B, and have switch B pass the packet back 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 again. 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 is somewhat novel in that it 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. Recall that SRON runs on a clique topology, where every SRON node is connected to every other SRON node. 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 $Switch_A$. $Switch_A$ is instructed by the controller to forward these probe packets over all ports connected to other switches in the network. 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 $Switch_B$. We would also like to know about the performance of paths originating at $Switch_A$ that use $Switch_C$ as an intermediate hop. Thus, if $Switch_C$ finds that it received this probe directly from $Switch_A$–it does this by comparing the destination IP address of the probe to the port on which it receieved this probe–it floods the probe on all ports that are not connected to $Switch_A$. Thus eventually $Switch_B$ will receieve 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 it allows us to

1. The way probing is actually done in SRON -> on a timer, ever switch sends a probe to every other probe, forwards once, sort of like multicast tree
2. Controller keeps track of best route, and stores data about how long this has been this best route (i.e. stores history so that it can change if a compelling better route appears)

4.2. Migrating Routes

How does the controller decide when to update flow rules between to switches in the network? On one hand, if $Switch_A$ sends to $Switch_B$ through $link_1$ but we find that $link_2$ is a significantly faster route, we would certainly like to update $Switch_A$'s path to $Switch_B$ in $Switch_A$'s flow table. What do we mean, though, by “significantly” faster? Not only does it take time to push rule updates to the switch flow tables, but switching too frequently may cause routing “oscillations”–that is, flow rules are updated so frequently that they cause routing loops, and so no data can pass through the network! For example, if $Switch_A$ decides to route to $Switch_C$ through $Switch_B$, and then $Switch_A$ chooses a different route to $Switch_C$ but then $Switch_B$ decides to route to $Switch_C$ through $Switch_A$, and so on, a data packet may be caught in a routing loop between $Switch_A$ and $Switch_B$. Finally, Internet protocols such as TCP assume that data travels over more or less the same path betwee hosts in the network (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 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 throuch 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 $Switch_B$:

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

We choose $\alpha = 0.1$ since this is the value used by RON to calculate latency[1], 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 $metric(p',n) \geq \beta metric(p,n)$, where we choose $\beta = 1.3$. By this metric, we migrate from p to p' when p' performs 30% better than p.

4.3. Dealing with Delayed Packets

I do this using a random sequence numbered that is stored in the "protocol" field of each packet (that's 16 bits). The controller does not acknowlegde packets that arrive with an out-of-date sequence number. Something similar to ACK sequence numbers.

4.4. Learning IPs

The lazy way: when a new IP comes in, register it with the switch it came in on.

5. Implementation

5.1. Pyretic

We decided to build SRON in Pyretic[?], 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[?]), 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:

```
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)
```

We use the `if_(`*predicate*`, policy1, policy2)` to indicate that all switches recieving 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 controller 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`.

6. Evaluation

6.1. Experimental Setup

Comparison with the mac learner module. We vary pobe frequency, size of the network etc. Using Harpoon, mininet.

6.2. Discussion

1. Performance evaluation
2. Would have prefered to compare to both real Internet traffic and routing
3. Would have liked to compare to how RON performs
4. Deploy on PlanetLab, too, would have been nice

7. Conclusion

8. Preparation Instructions

References

[1] 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

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

[3] 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

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