Your Data Center Switch is Trying Too Hard

Xin Jin
Princeton University

Nathan Farrington
Rockley Photonics

Jennifer Rexford
Princeton University

ABSTRACT

We present Sourcey, a new data center network architecture with extremely simple switches. Sourcey switches have no CPUs, no software, no forwarding tables, no state, and require no switch configuration. Sourcey pushes all control plane functions to servers. A Sourcey switch supports only source-based routing. Each packet contains a path through the network. At each hop, a Sourcey switch pops the top label on the path stack and uses the label value as the switch output port number. The major technical challenge for Sourcey is to discover and monitor the network with server-only mechanisms. We design novel algorithms that use only end-to-end measurements to efficiently discover network topology and detect failures.

Sourcey explores an extreme point in the design space. It advances the concept of software-defined networking by pushing almost all network functionality to servers and making switches much simpler than before, even simpler than OpenFlow switches. It is a thought experiment to show that it is possible to build a *simple* data center network and seeks to raise discussion in the community on whether or not current approaches to building data center networks warrant the complexity.

CCS Concepts

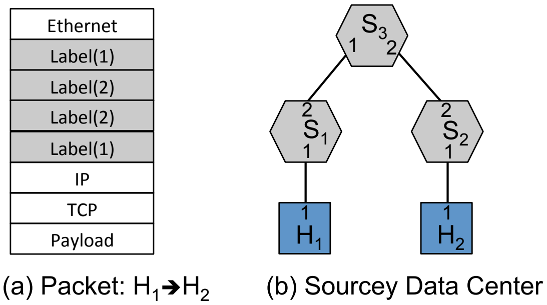
•Networks → Network design principles; Network management; Network monitoring; Data center networks;

Keywords

Software-defined networking; data center networks; network architecture; topology discovery; network monitoring; end hosts

1. INTRODUCTION

Cloud operators invest heavily in their cloud infrastructure. For example, Google, Microsoft and Amazon spent 11.0, 5.3 and 4.9 billion dollars, respectively, on cloud infrastructure in 2014 [1], with an estimated 15% of that investment spent on networking [2]. In our opinion, modern data center switches, and the data center networks created from them, are too expensive, and the reason is because the switches themselves are too complicated. They have too much responsibility and not enough information to make good decisions. They are trying too hard!



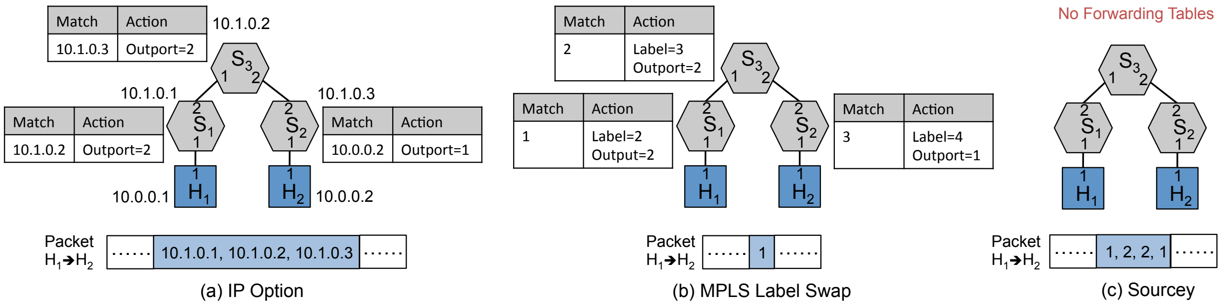
(a) Packet: H_1➔H_2 (b) Sourcey Data Center

Figure 1: Sourcey **architecture. Servers insert labels into each packet to encode an explicit path to the destination. Switches implement only a label pop operation and use the label value as the output port.**

Many cloud data center networks today operate like mini Internets, using IP longest-prefix match (LPM) routing, with distributed routing protocols for detecting link and router failures. For example, Facebook uses BGP as their intra-data center network routing protocol [3]. However, using technologies designed for the Internet introduces unnecessary administrative complexity (increased operating expenditure, or OPEX) and hardware scalability bottlenecks (increased capital expenditure, or CAPEX). BGP is complicated, and this complication leads to bugs between different vendors and product lines, and increased costs for training, planning, and troubleshooting. Performing IP LPM in hardware on the scale of modern cloud data center networks quickly surpasses the number of table entries that can fit on a single chip switch ASIC. IP LPM is a major administrative headache for mega data center operators.

OpenFlow-style SDN improves upon the state of the practice by removing the distributed and buggy nature of traditional network protocols. A logically centralized controller with a global view of information can make faster and better decisions than a distributed control plane. And a single implementation leads to fewer bugs. However, OpenFlow-style SDN relying on hardware-based forwarding tables on switches still suffers from the scaling limitations of traditional merchant silicon-based data center networks. One could argue that OpenFlow-style SDN can make better use of on-chip resources than traditional network protocols, but the small forwarding table sizes still remain. At the same time, cloud operators may be hesitant to purchase and deploy OpenFlow-style SDN solutions from a single vendor because they may not be willing to be dependent upon a single vendor for something as critical as

© 2016 ACM. ISBN 978-1-4503-4211-7/16/03 ...$15.00.
DOI: http://dx.doi.org/10.1145/2890955.2890967



(a) IP Option (b) MPLS Label Swap (c) Sourcey

Figure 2: Source routing methodologies.

their data center network. Finally, OpenFlow itself is getting more and more complicated. The number of header fields have increased from 12 (OpenFlow Spec 1.0, December 2009 [4]) to 41 (OpenFlow Spec 1.5, December 2014 [5]), which requires switches to implement complicated packet parsers and flow table pipelines. The number of pages of the OpenFlow Spec has also increased from 42 pages to 277 pages [4, 5]. Accordingly, OpenFlow software agents on switches are also becoming more complicated.

We present Sourcey, a new data center network architecture with extremely simple switches (Figure 1). This paper is a thought experiment to show that it is possible to build reliable, high-performance data center networks using much simpler switching elements than we use today. Sourcey switches are completely stateless. Servers play a pivotal role by using source routing to choose paths through the network. For each packet, a server translates a Layer 2 MAC address or a Layer 3 IP address into a path, represented as a stack of labels. At each hop, a Sourcey switch pops the top label off of the stack, and uses the label value directly as the switch output port number, thus avoiding a stateful table lookup. By the time a packet reaches its final destination, the entire path has been removed from the packet and the destination sees only an ordinary packet.

Sourcey pushes the entire control plane to servers. Switches only perform a simple label pop operation. To enable a packet to reach its destination, the control plane needs to tell the server which labels to put into the packet header. The major technical problem solved in this paper is how to servers can learn the network topology and keep up-to-date with the latest topographical changes, using only server-based mechanisms. This includes two tasks: (i) discover the topology for network bootstrap; and (ii) continuously monitor topology changes to keep update-to-date information. Once the control plane has the topology information, it can implement a wide variety of traffic engineering policies to choose a routing path for each flow or even each packet. The policies range from distributed ones to centralized ones, as discussed extensively in literature [6, 7, 8, 9].

We design new algorithms for Sourcey to perform topology discovery and monitoring. The algorithms run only on servers. The key idea is to send probe packets to the network, and by observing the forwarding behavior of probe packets with different labels (whether they return to the sender or not), to infer the topology and its changes. While it sounds expensive to discover and monitor a entire data center network with server-based probe methods, we show that the carefully-designed algorithms incur low overhead. Especially when compared to high-performance data center networks (10G and 40G are common today), this overhead is negligible. Furthermore, topology discovery only needs to be conducted during the bootstrap phase. Afterwards, only a small stream of packets are required to detect new elements (links, switches, servers) added to the network and existing elements removed from the network (manually or by failure).

Source routing is an old idea [10, 11, 12, 13]. The key novelty of this paper is the design of the architecture with minimal features on switches and the accompanied algorithms. In summary, we make the following two major contributions.

- **Architecture:** We present the Sourcey data center network architecture. Switches in this architecture have no CPUs, no software, no forwarding tables, no state, and require no switch configuration. The entire control plane is pushed to servers.
- **Algorithm:** We present novel server-based algorithms to make Sourcey control plane work. The algorithms leverage end-to-end probe packets to efficiently infer the network topology and detect its changes.

We view Sourcey as an extreme point in the design space. It advances the concept of software-defined networking by pushing almost all network functionality to servers and making switches much simpler than before, even simpler than OpenFlow switches (which has sophisticated packet parsers, table pipelines, and software agents). With Sourcey, we seek to raise discussion in the community on whether or not current approaches to building data center networks warrant the complexity.

2. SOURCEY ARCHITECTURE

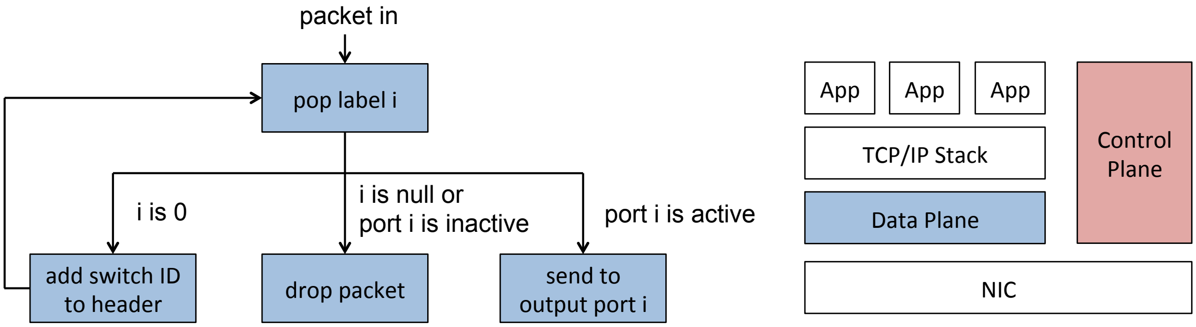
This section gives an overview of Sourcey. We first describe the source routing in Sourcey and compare it against other source routing methodologies. Then we describe the switch and server design in Sourcey.

2.1 Source Routing

In source routing, servers completely or partially specify the path for each packet, and put the routing information into packet headers. Switches forward packets based on header information. We illustrate how Sourcey differs from existing source routing solutions in Figure 2.

IP source routing: In IP source routing, servers put IP addresses into IP option field in each packet header (Figure 2(a)). These IP addresses either specify the entire path (strict source and record route, or SSRR) or specify some hops that the packet must go through (loose source and record route, or LSRR). In the example, the packet header contains the IP addresses of switches at each hop in the IP option field for the packet from server H_1 to server H_2. It requires servers to know the IP addresses of switches at each hop, and switches to keep an IP forwarding table.

MPLS label swap routing: In MPLS label swap routing, servers or ingress switches put an MPLS label on each packet header, and at each hop, the switch forwards the packet based on the label and



(a) Sourcey switch data plane flowchart

(b) Sourcey server software stack for control plane and data plane

Figure 3: Sourcey **switch and server design.**

swaps the label to another one (Figure 2(b)). In the example, server H_1 puts MPLS label 1 into the header and the packet goes through the network to reach H_2. It requires a centralized controller to properly compute and configure the flow tables for each switch.

Sourcey **source routing:** Sourcey completely eliminates the flow table in switches and requires only a label pop function. Servers put labels for the entire path into each packet header; the label values indicates the switch output ports at each hop (Figure 2(c)). Since data center networks have small diameters, the overhead of putting a path into packet headers is small. In the example, server H_1 puts $[1, 2, 2, 1]$ into the packet header. The first value denotes the output NIC port of server H_1, the second value denotes the output port of switch S_1, etc. At each hop, switches simply pop off a label and uses the label value as the output port number. It does not require any configuration of switches.

We have to note that only using the label pop function and specifying the entire path at ingress is not a new idea [10, 11, 12, 13]. But to make this work, it requires the ingress to know what labels to put into a packet header. Existing works either assume there is some sort of an oracle, use distributed protocols, or interacting with the switch software agents. Differently, Sourcey control plane is entirely on servers and uses server-based mechanisms to learn the topology.

2.2 Sourcey Switch

A Sourcey switch has no CPUs, no software, no forwarding tables, no state, and requires no switch configuration. It only implements the simple logic described in Figure 3(a). Switch ports start from 1. We reserve port 0 for switch identification. For each arriving packet, the switch pops the first label from the label stack and performs one of the following actions.

- **Normal case:** It forwards the packet to the output port denoted by the label.
- **Error handling:** If the label stack is empty or if the label value maps to a nonexistent or failed port, the packet is dropped.
- **Switch identification:** If the label is 0, it appends its switch ID to the header and uses the next label to decide which output port to forward the packet.

Because no software agents run on the switches, the last case is necessary for servers to determine the identity of a switch, which is an important primitive in the topology discovery algorithms presented later.

MPLS Compatibility: Sourcey can be made compatible with existing MPLS label switch routers (LSRs). Sourcey labels can use the MPLS header format. Servers insert MPLS headers between the Ethernet header and IP header (Figure 1). For the special case of switch identification, MPLS LSRs must be configured to forward such packets to LSR control plane, and let software agents handle such packets.

2.3 Sourcey Server

Sourcey servers are responsible for putting labels into each packet header. To implement this, it requires a control plane that decides what labels to push for a packet and a data plane that performs label push at line speed, as shown Figure 3(b).

Data plane: The data plane is an independent piece of software at each server. It receives routing decisions from the control plane and pushes labels to each packet based on the routing. The data plane has to handle every packet at line speed. One implementation is as a shim layer below the TCP/IP stack. Existing applications then need not be modified. Implementation choices include in kernel space, integration with the NIC as firmware or hardware, and integration with the hypervisor in virtualized environments.

Control plane: The control plane is a distributed system that runs on all servers. It discovers the network topology, monitors the network status, and chooses routes for each flow or packet.

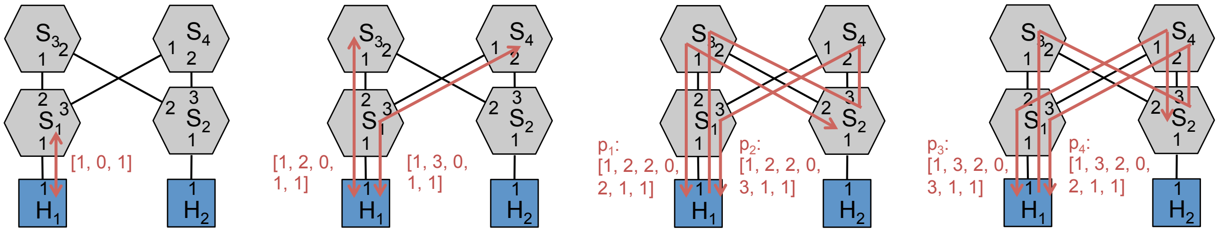
The major technical problem is topology discovery and monitoring. Once the control plane has an updated view of the topology, it is possible to choose routes for each flow or packet with different traffic engineering policies, as discussed in [6, 7, 8, 9]. In the remainder of this paper, we focus on how to implement topology discovery and topology monitoring using server-based mechanisms.

3. SOURCEY CONTROL PLANE

There are two major problems to be solved by the control plane. First, the control plane needs to discover the network topology during bootstrap, so that servers know what labels to use to implement a routing path. Second, the control plane needs to monitor the network and have an up-to-date view of the topology, so that traffic engineering can quickly switch to different paths in face of topology changes. This section describes server-based mechanisms to solve them.

3.1 Topology Discovery

Basic idea: Since the control plane exists entirely on servers, we can not run any distributed protocols on switches to discover the topology. We can only rely on servers. The basic idea is to send probe packets to the network and infer the network topology by


S3
S4
S1
S2
H1
H2
[1, 0, 1]
[1, 2, 0, 1, 1]
[1, 3, 0, 1, 1]
p1: [1, 2, 2, 0, 2, 1, 1]
p2: [1, 2, 2, 0, 3, 1, 1]
p3: [1, 3, 2, 0, 3, 1, 1]
p4: [1, 3, 2, 0, 2, 1, 1]

(a) Discover S_1 (b) Discover S_3 and S_4 (c) Discover S_2 (d) Discover S_2

Figure 4: Example of topology discovery.

observing the behavior of these packets (whether they return to the sender or not). A naive way is to send packets with all combinations of labels to the network and build the topology based on their behavior. The overhead of this solution increases exponentially with the maximum number of hops of the network. To make the problem tractable, we use breadth-first search (BFS). A server gradually explores the network and learns the topology, rather than exploring the entire topology in one shot. This prunes many branches from the search space.

Example: To make the idea more concrete, we describe an example shown in Figure 4. Suppose, server H_1 performs the topology discovery. Initially, it only knows about itself.

Discover S_1: Server H_1 sends probe packets with label stack $[1, 0, j]$ to learn its neighbor (the first label is 1 because server H only has one NIC) where $1 \leq j \leq MAX\_PORT$ and MAX_PORT is the maximum port count of a switch. The label 0 is used to query the switch ID that server H_1 is connected to. Only the packet with label stack $[1, 0, 1]$ returns to server H_1. This tells server H_1 that it is connected to switch S_1 on port 1.

Discover S_3 and S_4: After discovering switch S_1, server H_1 sends probe packets to discover switches two hops away. The probe packets have label stack $[1, i, 0, j, 1]$ where $1 \leq i, j \leq MAX\_PORT$. The first label 1 in the stack is used to reach switch S_1; the last label 1 is used to return to server H_1 from switch S_1; the middle labels $[i, 0, j]$ are used for discovery. Packets with $[1, 2, 0, 1, 1]$ and $[1, 3, 0, 1, 1]$ return to server H_1, and server H_1 learns link S_1-S_3 and link S_1-S_4.

Discover S_2: Now server H_1 sends probe packets to discover switches three hops away. Since there are two switches (S_3 and S_4) that are two hops away, the probe packets need to go one hop beyond each of them. To go beyond S_3, the probe packets use label stacks $[1, 2, i, 0, j, 1, 1]$; to go beyond S_4, the probe packets use label stacks $[1, 3, i, 0, j, 1, 1]$. The following four packets would return to server H_1.

- p_1: $[1, 2, 2, 0, 2, 1, 1]$.
- p_2: $[1, 2, 2, 0, 3, 1, 1]$.
- p_3: $[1, 3, 2, 0, 3, 1, 1]$.
- p_4: $[1, 3, 2, 0, 2, 1, 1]$.

Only looking at these packets, packet p_1 suggests port 2 on switch S_3 is connected to port 2 on switch S_2; packet p_2 suggests port 2 on switch S_3 is connected to port 3 on switch S_2. They are conflicting with each other. If we look at the paths they traverse, we can see that packet p_1 uses H_1-S_1-S_3-S_2-S_3-S_1-H_1 and p_2 uses H_1-S_1-S_3-S_2-S_4-S_1-H_1. The return path of p_2 is not the same as the departure path. To resolve this conflict, we need to send another two probe packets, one with label stack $[1, 2, 2, 2, 0, 1, 1]$ and the other with label stack $[1, 2, 2, 3, 0, 1, 1]$. They would query the switch ID of the first switch on the return path. From them, we know that p_1 uses the same path for the round trip and p_2 does not. Therefore, port 2 on switch S_3 is connected to port 2 on switch S_2.

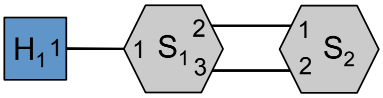


Figure 5: Example for parallel links.

Similarly, for p_3 and p_4 we need to send additional probe packets to determine link S_4-S_2.

Discover H_2: Finally, server H_1 sends probe packets to discover nodes four hops away. Since there is only one switch three hops away, the probe packets use label stacks $[1, 2, 2, i, 0, j, 2, 1, 1]$. The probe packet with label stack $[1, 2, 2, 1, 0, 1, 2, 1, 1]$ would return to server H_1 with the ID of server H_2. This finishes the topology discovery process.

Special case - parallel links: When there are multiple link between two switches, we cannot determine the specific ports for each parallel link. For example, in Figure 5, there are two parallel links between switch S_1 and switch S_2. The packets with label stacks $[1, 2, 0, 1, 1]$, $[1, 2, 0, 2, 1]$, $[1, 3, 0, 1, 1]$, and $[1, 3, 0, 2, 1]$ will all return to server H_1. Even if we use another probe packet to test the switch ID on the other side of the links, it will not make a difference as the switch is always switch S_1. We can not tell whether port 2 on switch S_1 is connected to port 1 or 2 on switch S_2.

However, it does not matter that we do not know the specific ports for each parallel link. When we want to send traffic from switch S_1 to switch S_2, we can use either port 2, port 3, or a combination of them. The important thing is we know what ports belong to these parallel links. Identifying the number of parallel links and their ports is as follows. Let l_1 and l_2 be the labels to and from switch S. When there are parallel links between switch S and another switch S', we will observe all probe packets with $[l_1, i, 0, j, l_2]$ where $i \in P$ and $j \in P'$. P are the ports of these parallel links on switch S ($P = \{2, 3\}$ in Figure 4(b)); P' are the ports of these links on S' ($P' = \{1, 2\}$ in Figure 5).

Optimization - multiple servers: To make the explanation simple, we have assumed the topology discovery runs on one server. It is possible to run it on multiple servers. In this case, each server begins the topology discovery without knowing of other servers. After two servers discover each other, they compute the union of their two partially discovered topologies. The union is simply a union of the node set and link set. Then they divide the remaining topology discovery plan into two parts and each server probes one part.

Optimization - a blueprint: In many cases, operators have a blueprint of their planned data center topology. A blueprint should indicate the ID of each switch, and the port numbers on each side of each link. Servers can use such a blueprint to speed topology discovery. Instead of searching the network using BFS, servers can directly generate packets that take paths on the blueprint. These probe pack-

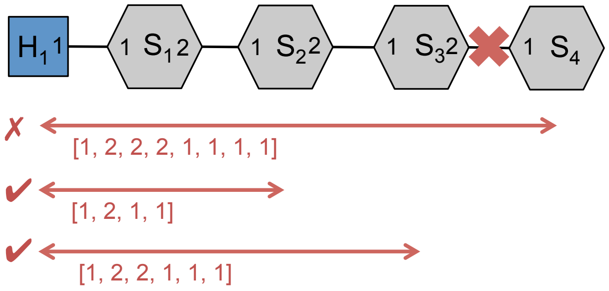


Figure 6: Example for topology monitoring.

ets verify that the physical topology is wired by the operator as expected. For the example in Figure 4, we send the following probe packets to verify all the links, rather than trying different combinations of labels for each link (different combinations of i and j in probe packets).

- Link H_1-S_1: $[1, 0, 1]$
- Link S_1-S_3: $[1, 2, 0, 1, 1]$
- Link S_1-S_4: $[1, 3, 0, 1, 1]$
- Link S_3-S_2: $[1, 2, 2, 0, 2, 1, 1]$ and $[1, 2, 2, 2, 0, 1, 1]$
- Link S_4-S_2: $[1, 3, 2, 0, 3, 1, 1]$ and $[1, 3, 2, 3, 0, 1, 1]$
- Link S_2-H_1: $[1, 2, 2, 1, 0, 1, 2, 1, 1]$

Analysis: Now we analyze the overhead of topology discovery. Since topology discovery is not frequently invoked, we focus on the total traffic instead of the delay. Let the number of switches be n and the maximum port count on a switch is k. Without a blueprint, for each switch, we need to send at most $2k^2$ probe packets. The total probe packets needed to discover a network are $2nk^2$. To make this concrete, let $n = 10000$ and $k = 48$. Then the total probe packets is 46.08 million. Let the packet size be the upper bound 1500 B. Then the total probe traffic is 69.12 GB.

With a blueprint, we need to send at most 2 packets to verify each link. Since there are $nk/2$ links, the total probe packets are nk. When $n = 10000$ and $k = 48$, the total probe packets is 0.48 million. Let the packet size be the upper bound 1500 B. Then the total probe traffic is 0.72 GB.

3.2 Topology Monitoring

Topology monitoring maintains an up-to-date view of the network topology. Its job is to quickly detect topology changes, which includes new links and nodes added to the network, existing links and nodes removed from the network, and failures. The way to detect links and nodes added to the network is similar to topology discovery. Servers send probe packets to explore inactive switch ports, in order to see whether new links or nodes are added to the network and the ports become active. Since the topology has already been discovered and most ports are active, the overhead of detecting additions is much lower than topology discovery. Moreover, since removals (manually by operators) and failures all behave as missing links and nodes in the topology graph, their detection solutions are the same. Finally, we say that a node is *removed* if all of its links are removed. Therefore, we focus on link removals in the rest of this subsection and understand that it also includes node removals.

Basic idea: Servers send probe packets to traverse all the links to detect link removals. If all the links are active, these packets would return to the senders; if a packet does not return, it means at least one link on the probe path of this packet has been removed. Upon detecting link removals, we send more probe packets to locate the removed links. There are two goals for the topology monitor: full coverage (monitor all links) and high efficiency (detect removals with low delay). To achieve full coverage, we compute a Euler cycle for the topology and use the Euler cycle as the probe path for a probe packet. A Euler cycle on a graph is a path that traverses each edge exactly once and returns to the source. We treat each link in the network as two directional edges and the entire network as a directed graph. Based on graph theory, we can always find a Euler cycle in a directed graph. With this, a server continuously sends probe packets that traverse the Euler cycle to monitor *all* links in the network. Since the packet traverses each directed edge only once, the probe packets incur low overhead.

If a packet does not return to the server (after retrying a few times), a link is considered to have been removed, and the next step is to determine which link(s) on the path has(have) been removed. We use *binary search* to quickly locate the removed links. Specifically, we divide the network into two parts and send a probe packet for each part. If a probe packet does not return, then the corresponding part has link removals. We do this recursively until we have located the removed link(s).

Example: We use the example in Figure 6 to illustrate how to detect a link removal. In the example, server H_1 periodically sends probe packets with label stack $[1, 2, 2, 2, 1, 1, 1, 1]$ to monitor four links (H_1-S_1, S_1-S_2, S_2-S_3, S_3-S_4). When all links are active, the probe packets would return to server H_1. Now suppose link S_3-S_4 is removed (or fails). The probe packets would be dropped at switch S_3, indicting a link removal. Then server H_1 sends more probe packets to locate the link removal with binary search. The probe packet with label stack $[1, 2, 1, 1]$ returns to server H_1, indicting links H_1-S_1 and S_1-S_2 are active. Then server H_1 sends a probe packet with label stack $[1, 2, 2, 1, 1, 1]$ to test link S_2-S_3. This probe packet also returns. Therefore, link S_3-S_4 is removed.

Optimization: There are many ways to speed topology monitoring; we introduce two of them here. First, large networks can have hundreds of thousands of links. It takes a long time for a probe packet to traverse all of these links and return to the server. Accordingly, we need to set a long timeout to determine a packet loss. To solve this, we can divide the network into multiple parts and transmit one probe packet per part. These probe packets can be generated by a single server or multiple servers (to distribute probe traffic load). In this way, we can reduce the timeout parameter and thus reduce removal detection delay. Second, rather than using binary search, we can use k-ary search that divides the probe path into k parts each time. In the extreme case, for a probe path with l links, we can divide the path into l parts and send a probe packet to test each link in one-shot. This can reduce the delay of localizing the link removal to only *one* timeout.

Analysis: Now we analyze the delay to detect a link removal. Suppose a probe path traverses l links and the delay of traversing one link is t (including switching delay and propagation delay). Normally, a probe packet would take $2lt$ time to return. Suppose we set time out to 3 times of that, i.e., $6lt$. Upon a timeout, the k-ary search uses one probe packet for each link. So it takes one timeout to detect a link removal and one timeout to locate the link removal. The total delay is $12lt$. To make this concrete, let l be 100 and t be 10 μs. The total delay is 12 ms.

4. RELATED WORK

Data center network architectures: There are many papers on data center network architectures [13, 14, 15, 16, 17]. They design new topologies, new addressing schemes, and new traffic engineering algorithms that are tailored for data center networks. Instead of focusing on topology, addressing and routing, Sourcey focus on the division of labor between servers and switches. Sourcey is an extreme design that pushes the entire control plane to servers.

Software-defined networking: SDN decouples the control plane from the data plane, and aims to simplify network management. OpenFlow is the de facto protocol of SDN [18]. Although simpler than existing solutions, OpenFlow requires switches to implement a flow table and keep flow state [5]. The protocol is becoming more and more complicated with each new release. Sourcey takes a fresh look at data center networks and proposes to completely remove unnecessary features from switches. As compared to the Link Layer Discovery Protocol (LLDP), Sourcey completely eliminates the need to implement a topology discovery protocol on switches and thus greatly simplifies the switch design. In terms of the SDN control plane, there are many works that propose techniques to make the control plane scalable and high performance [19, 20, 21, 22, 23]. These techniques are orthogonal to, and can be used by Sourcey.

Source Routing: Several works have proposed to use source routing in data centers [10, 11, 12, 13]. The novelty of Sourcey is the fact that such a simple stateless building block can be used to build a scalable, high-performance data center network, and the algorithms that perform topology discovery and network monitoring from servers.

Network Tomography: Network tomography uses end-to-end measurements to discovery network topology and detect failures [24, 25, 26]. Although the high-level objective is similar, Sourcey's control plane algorithms are specially tailored for the minimal features of Sourcey switches.

5. CONCLUSION

In this paper, we presented Sourcey, a new data center network architecture with extremely simple switches. We strip out CPUs, software, forwarding tables and state from switches. Sourcey switches implement only a single operation in hardware. We completely eliminate the need to configure switches. The entire control plane is pushed to servers. We presented algorithms to efficiently detect topology changes with server-based probing methods. This paper shows a particular design of a data center network with minimal features. We believe that Sourcey, and reducing the complexity of switches in general, is a promising direction to pursue for future data center networks.

Acknowledgments We thank the SOSR reviewers for their feedback. Xin Jin and Jennifer Rexford were supported by the NSF under grant CNS-1162112.

6. REFERENCES

[1] "Google had its biggest quarter ever for data center spending. Again." http://tinyurl.com/kezfjv5.

[2] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," *SIGCOMM CCR*, vol. 39, no. 1, 2008.

[3] "Introducing data center fabric, the next-generation Facebook data center network." http://tinyurl.com/kezfjv5.

[4] "OpenFlow Switch Specification 1.0.0." http://tinyurl.com/md8gge7.

[5] "OpenFlow Switch Specification 1.5.0." http://tinyurl.com/qcz3bow.

[6] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford, "Network architecture for joint failure recovery and traffic engineering," in *ACM SIGMETRICS*, June 2011.

[7] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," in *ACM SIGCOMM*, August 2014.

[8] M. Caesar, M. Casado, T. Koponen, J. Rexford, and S. Shenker, "Dynamic route recomputation considered harmful," *SIGCOMM CCR*, vol. 40, pp. 66–71, April 2010.

[9] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul, "SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies," in *USENIX NSDI*, April 2010.

[10] S. A. Jyothi, M. Dong, and P. Godfrey, "Towards a flexible data center fabric with source routing," in *ACM SOSR*, June 2015.

[11] R. M. Ramos, M. Martinello, and C. Esteve Rothenberg, "SlickFlow: Resilient source routing in data center networks unlocked by OpenFlow," in *IEEE Conference on Local Computer Networks (LCN)*, October 2013.

[12] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "SecondNet: A data center network virtualization architecture with bandwidth guarantees," in *ACM CoNEXT*, November 2010.

[13] L. Fang, F. Chiussi, D. Bansal, V. Gill, T. Lin, J. Cox, and G. Ratterree, "Hierarchical SDN for the hyper-scale, highly elastic data center and cloud," in *ACM SOSR*, June 2015.

[14] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: A scalable and fault-tolerant network structure for data centers," in *ACM SIGCOMM*, August 2008.

[15] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A scalable fault-tolerant layer 4 data center network fabric," in *ACM SIGCOMM*, August 2009.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," in *ACM SIGCOMM*, August 2009.

[17] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers," in *ACM SIGCOMM*, August 2010.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, April 2008.

[19] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *USENIX OSDI*, October 2010.

[20] "OpenDaylight Platform." http://www.opendaylight.org/.

[21] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, *et al.*, "Network virtualization in multi-tenant datacenters," in *USENIX NSDI*, April 2014.

[22] "Open Network Operating System (ONOS)." http://onosproject.org/.

[23] "Cisco Application Policy Infrastructure Controller (APIC)." http://tinyurl.com/orc2rxe.

[24] M. Coates, R. Castro, R. Nowak, M. Gadhiok, R. King, and Y. Tsang, "Maximum likelihood network topology identification from edge-based unicast measurements," in *ACM SIGMETRICS*, June 2002.

[25] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot, "NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data," in *ACM CoNEXT*, December 2007.

[26] Y. Huang, N. Feamster, and R. Teixeira, "Practical issues with using network tomography for fault diagnosis," *SIGCOMM CCR*, vol. 38, no. 5, pp. 53–58, 2008.