

Dynamically Creating Custom SDN High-Speed Network Paths for Big Data Science Flows

Sergio Rivera, Mami Hayashida, James Griffioen, Zongming Fei

Laboratory for Advanced Networking

University of Kentucky

Lexington, Kentucky 40506

{sergio,mhaya2,griff,fei}@netlab.uky.edu

ABSTRACT

Existing campus network infrastructure is not designed to effectively handle the transmission of big data sets. Performance degradation in these networks is often caused by middleboxes – appliances that enforce campus-wide policies by deeply inspecting all traffic going through the network (including big data transmissions). We are developing a Software-Defined Networking (SDN) solution for our campus network that grants privilege to science flows by dynamically calculating routes that bypass certain middleboxes to avoid the bottlenecks they create. Using the global network information provided by an SDN controller, we are developing graph databases approaches to compute *custom* paths that not only bypass middleboxes to achieve certain requirements (e.g., latency, bandwidth, hop-count) but also insert rules that modify packets hop-by-hop to create the illusion of standard routing/forward despite the fact that packets are being rerouted. In some cases, additional functionality needs to be added to the path using network function virtualization (NFV) techniques (e.g., NAT). To ensure that path computations are run on an up-to-date snapshot of the topology, we introduce a versioning mechanism that allows for lazy topology updates that occur only when “important” network changes take place and are requested by big data flows.

CCS CONCEPTS

• **Networks** → **Network management**; *Programmable networks*;

KEYWORDS

Software-Defined Networks, Path Calculation, Big Data Flows

ACM Reference format:

Sergio Rivera, Mami Hayashida, James Griffioen, Zongming Fei. 2017. Dynamically Creating Custom SDN High-Speed Network Paths for Big Data Science Flows. In *Proceedings of PEARC17, New Orleans, LA, USA, July 09-13, 2017*, 4 pages.

<https://doi.org/10.1145/3093338.3104155>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PEARC17, July 09-13, 2017, New Orleans, LA, USA

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5272-7/17/07.

<https://doi.org/10.1145/3093338.3104155>

1 BIG DATA TRANSMISSION ISSUES

Methods to process, manipulate, and transform large data sets are evolving at a remarkable rate. However, university researchers often encounter frustratingly slow transmission speeds when trying to move these data sets from one device to another (e.g., sharing it with a collaborator). Oftentimes, this performance degradation is caused by a combination of performance-limiting middleboxes (appliances that enforce policies on all traffic going through the network), and competing non-research traffic (e.g., streaming Netflix) that consumes the university’s limited network resources.

To address this problem, campuses have adopted *Science DMZs* [1] (Figure 1) as the mechanism to avoid performance-limiting middleboxes. In this approach, machines that need high-throughput communications are placed on a separate network (i.e., a Science DMZ) hanging off the router at the junction between the campus network and the Internet. While these privileged machines can send/receive traffic without any performance bottlenecks, they give up the security, protection and policing offered by middleboxes.

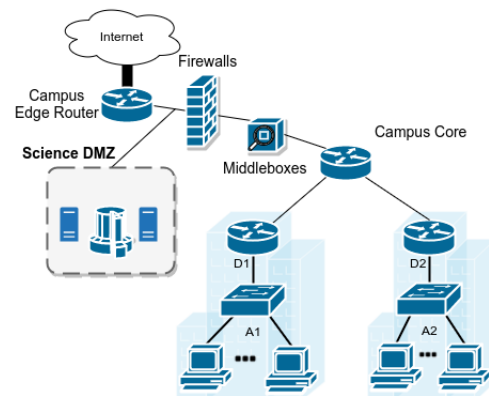


Figure 1: The Science DMZ approach

We are developing an all-campus Science DMZ solution based on Software-Defined Networking (SDN) [4] called *VIP Lanes* [2, 6] that allows researchers to boost transmission speeds for privileged scientific flows while retaining middlebox security and protection for all other flows.

In *VIP Lanes*, our control software computes paths that not only bypass middleboxes to achieve certain requirements – e.g. the fastest path (latency constraint), the widest path (bandwidth constraint), or the shortest path (hop-count constraint) – but also

modify packets as needed (e.g., changing MAC addresses or VLAN ids) to create the illusion of standard routing/forward despite the fact that packets are being rerouted. In some cases, additional functionality needs to be added to the path using network function virtualization (NFV) techniques such as network address translation (NAT). The resulting packet transformations (i.e., OpenFlow [5] rules) are then written to the campus SDN routers/switches to reroute pre-approved flows through bottleneck-free paths.

Rather than implementing the underlying path computation algorithms using imperative programming languages, and possibly introducing bugs, we calculate paths via natural declarative queries in Neo4j¹, a production-ready graph database we use to store versioned fine-grained topology information (e.g., link capacities, vlans, latency, etc) and VIP Lanes flow-related data. We developed an effective topology versioning mechanism that keeps the topology view of the network in Neo4j and the controller’s view in sync. In that way, every path request is guaranteed to be feasible, valid, and dynamically calculated using the most up-to-date snapshot of the network before it is translated into low-level OpenFlow rules installed at every switch along the computed route.

2 GRAPH DB PATH CALCULATION

Figure 2 illustrates the workflow and events involved while requesting a high-throughput VIP Lanes path for a science data transfer. A request can be specified via (1) a web interface or (2) a wrapper library. In the former, users have to manually provide information about the communication that is about to happen, including details like source/destination IP addresses, destination port, and flow duration. In the latter, all the information about the VIP Lanes is determined by a wrapper library that is dynamically loaded along with existing (legacy) applications; the VIP Lanes information is computed when the socket is created and the connection is being established, triggering a request to the path computation service to set up the VIP Lanes. In either case, the request needs to be approved by our permission system (described in [2] and depicted by the Auth N/Z Service in Figure 2).

After being granted permission, the path computation service attempts to find a bottleneck-free path with the provided information. If the resulting path is feasible, a *versioned install request* is sent to the controller and further transformed into OpenFlow rules. Otherwise, an error is reported back to the user/application.

While the path computation could be written in a conventional imperative programming language by calling existing graph libraries, computing the custom paths needed by VIP Lanes would require tailoring the path computation algorithms included in these libraries to handle networks made up of heterogeneous elements, which is an error-prone and time-consuming task for a network programmer/operator. Instead, we opted to leverage the built-in capabilities of the Neo4j graph database to perform the path computation and topology data maintenance within the database. The main reasons for our decision include: (1) the *Cypher* graph DB language – the declarative query language for Neo4j that simplifies the maintenance of topology data and provides an intuitive syntax to construct our own constrained queries, including path computations; (2) a direct mapping from a network topology (devices,

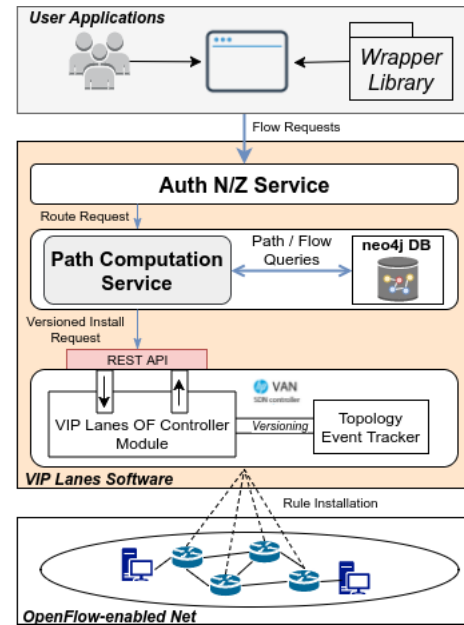


Figure 2: VIP Lanes architecture and workflow

links) into the same representation in the database using nodes and edges. Further, the ability to manipulate sets of labels assigned to the stored elements allows us to represent more complex network abstractions like active flows, IP Pools, topology snapshots, and virtual network functions (e.g., NAT); (3) the ability to store heterogeneous collections of data as properties of elements in the network such as DPIDs for switches, MAC addresses for hosts, and bandwidth capacity for links; and (4) an intuitive GUI (Figure 3) that allows network operators to view current (and past) topology information, and to ask simple questions that are otherwise tedious to implement in imperative programming languages (e.g., “*what active flows go through switch X and avoid middleboxes of type T?*”).

Currently, VIP Lanes is capable of calculating three types of paths: the fastest, the widest, and the shortest. The fastest path query chooses the route based on the sum of latencies of all links on the path; the widest path query chooses the route with the maximum (greatest) bandwidth capability of the minimum-bandwidth link in a path; the shortest simply chooses the path with fewest hop counts. While Neo4j provides a built-in function for the shortest path, we wrote queries for the fastest and widest path with relatively simple and straight-forward phrases using Cypher. By default, all three types of paths are middlebox-free. However, it is possible to compute paths that avoid only certain types of middleboxes, e.g., to ensure big data traffic goes through a non-bottleneck monitoring appliance. Relevant middleboxes and non-SDN devices present in the network (red nodes in Figure 3) are identified in the database primarily through JSON-encoded configuration files that contain descriptions of the interfaces present at every middlebox (e.g., MAC and IP addresses, or neighbors). Typically, some of these middleboxes are discovered by the controller as hosts, consequently, the path computation service we developed uses the information

¹The Neo4j[®] Graph Database can be found at <https://neo4j.com>

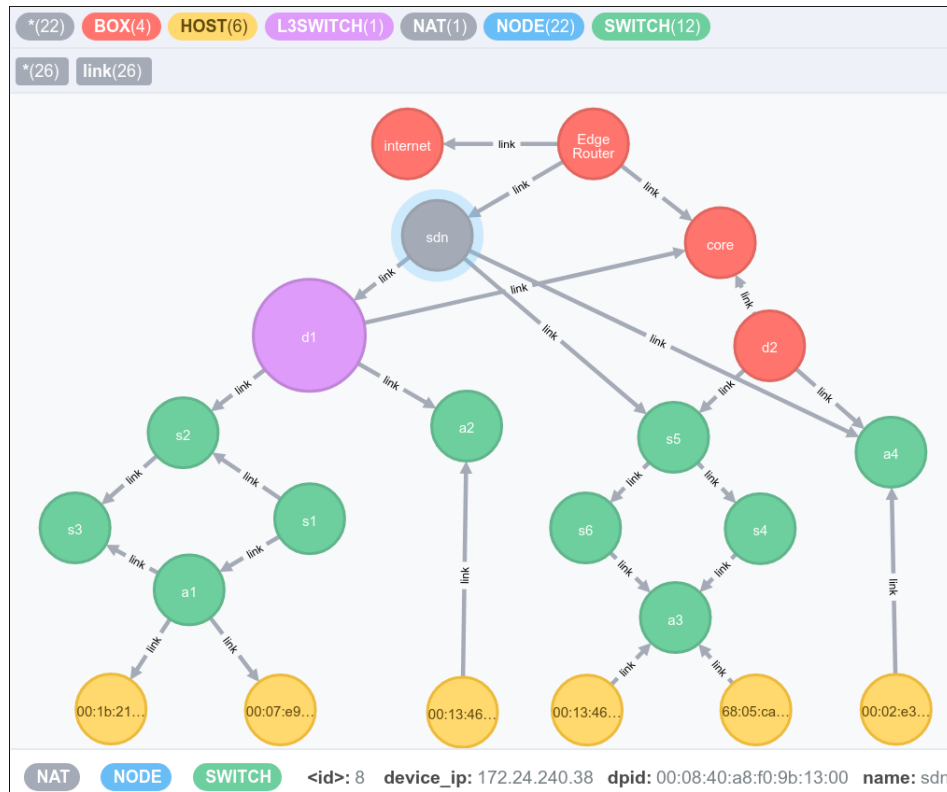


Figure 3: Neo4j GUI displaying the current topology snapshot, list of existing labels in the database, and detailed information assigned to the *sdn* node

stored in the configuration files to override the type of node that needs to be stored in the database.

When a path query is run in Neo4j, it returns not only the nodes and edges along the computed path, but also a selection of label and property data for each node and edge. This information is vital to the successful construction of custom paths since it describes how each OpenFlow element must behave in the selected path; thus, it allows us to realize a certain degree of NFV in our SDN network. The control software we developed parses this relevant data and maps it into OpenFlow rules which are further installed at every switch along the path by the Aruba VAN SDN controller [3]. These rules ultimately dictate the behavior of every individual switch for every approved big data science flow. Consequently, it is not rare to have “multi-function” switches (like the *sdn* node in Figure 3) that operate differently based on the location of the end hosts in the computed path. For instance, for on-campus transmissions (e.g., “a transfer from the Computer Science department to the Physics department”) a switch could behave as an L2/L3 switch that rewrites MAC addresses or VLAN tags for every packet header in a flow. Additionally, that **same** switch could serve (simultaneously) as a stateless Network Address Translator (NAT) for big data transfers going off-campus (e.g., “sending data to a national lab”). In this particular case, the flexible nature of graph databases allows us to not only store the *de facto* NAT table, but also the set of public IP addresses (and their availability) to appropriately assign and

produce OpenFlow rules that rewrite the source and destination IP addresses of packet headers for outbound (i.e., from the campus) and inbound (i.e., to the campus) big data flows going through the high-speed SDN network.

3 TOPOLOGY VERSIONING

For a big data science flow to obtain high-throughput, the path computation must have accurate topology data. The challenge, of course, is to determine the frequency of updating the topology data without compromising efficiency. Theoretically, the topology stored on the database T_{db} should always match the actual topology T_c (known by the controller) at any given time. In practice, however, this would add unnecessary overhead: if no science flows are requested for a period of time, it is wasteful to continuously update T_{db} . The opposite approach is to check if $T_{db} = T_c$ before *each* path query and update T_{db} if the condition is not met before computing the path. While this eliminates unnecessary topology updates, it adds a user-noticeable delay to the path calculation process as the topology grows. To tackle this problem, we implemented a topology versioning mechanism. Figure 4 illustrates how the mechanism operates when events take place at different levels of the architecture.

When the controller boots up or a new version of the topology module is deployed (light-gray box), T_c is defined to be the topology learned by the controller. We also associate a random 64-bit number

v_c with T_c indicating the version of the topology stored in T_c . We also define a flag T_{c_req} that is used to indicate whether the topology has been requested by the path computation service or not. Later, when a *topology event* (i.e., change in topology) is detected by the controller (yellow elements), the version number v_c is increased by 1 iff (1) the event is not associated with a host joining or leaving the topology – which happens continually because the controller assumes a host has left when no packets have been seen for some amount of time, but then adds the host to the topology as soon as a packet is seen from the host, and (2) the path computation service has requested a newer version of the topology. This avoids the situation where T_{db} is constantly being updated even though the path computation service does not currently need the latest version.

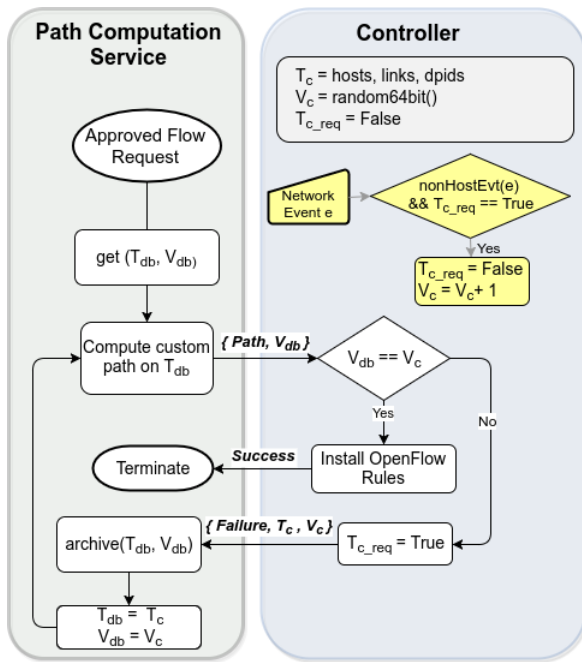


Figure 4: Topology versioning mechanism flow chart

When the database is initialized, T_{db} is set to the current version of T_c , and the database version number is set to match the controller’s version number (i.e., $v_{db} = v_c$). Later, when a user requests a high-speed VIP Lanes path via the path computation service (white elements), the path computation service calculates the VIP Lanes path using T_{db} . It then sends the computed path along with v_{db} to the controller to install the SDN path. When the controller receives the request, it first checks if the v_{db} is equal to the v_c (i.e., the topologies are in sync). If so, OpenFlow rules are generated with the corresponding actions (e.g., MAC, IP or VLAN rewrites) at every hop in the path and a success message is returned to the path computation service. Otherwise, the controller notes that the path computation service needs a new version of the topology (by setting $T_{c_req} = True$), and rejects the current path installation request. As a result, new topology events will cause v_c be increased (yellow elements), and the response message is built

including the most recent values of v_c and T_c . Once the response gets back to the path computation service, current data of T_{db} and v_{db} is archived as an old version in the Neo4j database, and new snapshots are added with the T_c and v_c values provided in the response. After this update, the process starts over again and the path calculation is done on a more recent topology snapshot.

4 CONCLUSION

There is a growing need for high-speed big data transfers both east/west across campus and north/south to the cloud. Optimizing data transfer across and SDN-enabled campus using VIP Lanes requires an intelligent path computation service that understands the details of the campus network topology, including the location of performance-limiting middleboxes that should be avoided. In this paper, we introduced a novel path computation service that leverages the features of the Neo4j graph database to efficiently calculate constrained middlebox-free paths like the shortest, the fastest, and the widest paths, but also determines custom behaviors of individual switches on a per-hop and per-flow basis. We described how our system can implement network function virtualization capabilities such as NAT via OpenFlow rules, and introduced a topology versioning mechanism that guarantees the paths (and the generated OpenFlow rules) are calculated with an accurate snapshot of the current network topology.

ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation under Grants ACI-1541380, ACI-1541426, and ACI-1642134.

REFERENCES

- [1] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski. 2013. The Science DMZ: A network design pattern for data-intensive science. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–10. <https://doi.org/10.1145/2503210.2503245>
- [2] J. Griffioen, K. Calvert, Z. Fei, S. Rivera, J. Chappell, M. Hayashida, C. Carpenter, S. Yongwook, and H. Nasir. 2017. VIP Lanes: High-speed Custom Communication Paths for Authorized Flows. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. [to appear].
- [3] Hewlett Packard Enterprise. 2017. HP Virtual Applications Network SDN Controller. <https://www.hpe.com/us/en/product-catalog/networking/networking-software/pip.hpe-van-sdn-controller-software.5443866.html>. (2017).
- [4] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (Jan 2015), 14–76. <https://doi.org/10.1109/JPROC.2014.2371999>
- [5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communications Review* 38, 2 (March 2008), 69–74. <https://doi.org/10.1145/1355734.1355746>
- [6] S. Rivera, J. Chappell, M. Hayashida, A. Groenewold, P. Oostema, C. Voss, H. Nasir, C. Carpenter, Y. Song, Z. Fei, and J. Griffioen. 2017. Creating Complex Testbed Networks to Explore SDN-based All-Campus Science DMZs. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Atlanta, GA, USA, (May 2017).