HyperFlow: A Distributed Control Plane for OpenFlow

Amin Tootoonchian
University of Toronto
amin@cs.toronto.edu

Yashar Ganjali
University of Toronto
yganjali@cs.toronto.edu

Abstract

OpenFlow assumes a logically centralized controller, which ideally can be physically distributed. However, current deployments rely on a single controller which has major drawbacks including lack of scalability. We present HyperFlow, a distributed event-based control plane for OpenFlow. HyperFlow is logically centralized but physically distributed: it provides scalability while keeping the benefits of network control centralization. By passively synchronizing network-wide views of OpenFlow controllers, HyperFlow localizes decision making to individual controllers, thus minimizing the control plane response time to data plane requests. HyperFlow is resilient to network partitioning and component failures. It also enables interconnecting independently managed OpenFlow networks, an essential feature missing in current OpenFlow deployments. We have implemented HyperFlow as an application for NOX. Our implementation requires minimal changes to NOX, and allows reuse of existing NOX applications with minor modifications. Our preliminary evaluation shows that, assuming sufficient control bandwidth, to bound the window of inconsistency among controllers by a factor of the delay between the farthest controllers, the network changes must occur at a rate lower than 1000 events per second across the network.

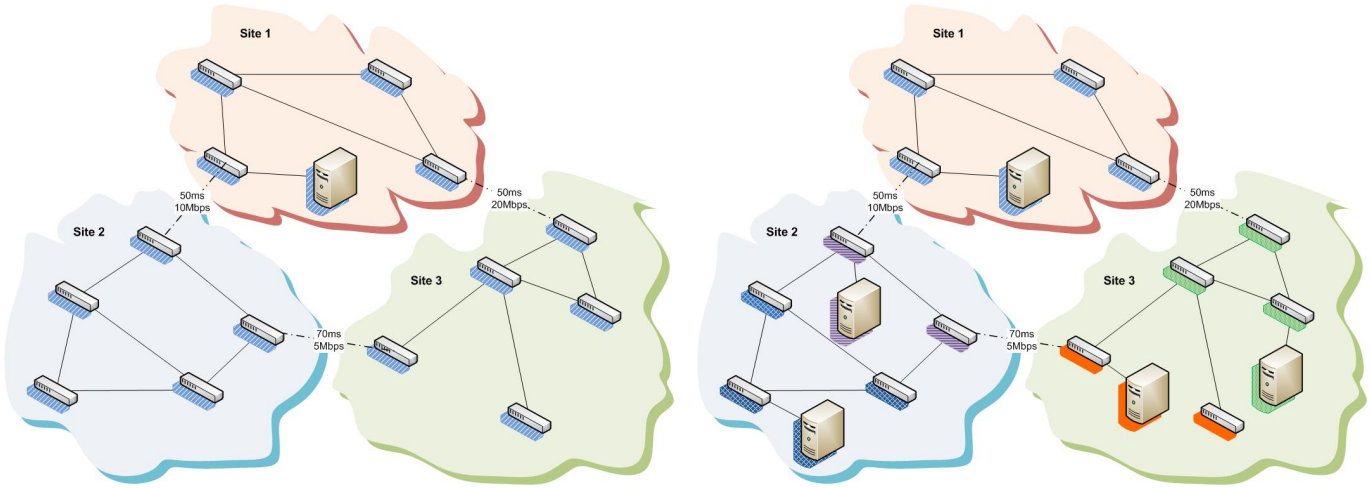
1. INTRODUCTION

The minimalism and simplicity in the Internet's design has led to an enormous growth and innovation atop, yet the network itself remains quite hard to change and surprisingly fragile and hard to manage. The root cause of these problems is the overly complicated control plane running on top of all switches and routers throughout the network [2]. To alleviate this problem, previous works propose to decouple the control (decision making) and data (packet forwarding) planes, and delegate the control functionality to a logically centralized controller [2, 5, 11]. This separation significantly simplifies modifications to the network control logic (as it is centralized), enables the data and control planes to evolve and scale independently, and notably decreases the cost of the data plane elements [6]. In particular, OpenFlow [5] has succeeded in attracting commercial vendors [1].

The initial design and implementation of OpenFlow assumed a single controller for the sake of simplicity. However, as the number and size of production networks deploying OpenFlow increases, relying on a single controller for the entire network might not be feasible for several reasons. First, the amount of control traffic destined towards the centralized controller grows with the number of switches. Second, if the network has a large diameter, no matter where the controller is placed, some switches will encounter long flow setup latencies. Finally, since the system is bounded by the processing power of the controller, flow setup times can grow significantly as demand grows with the size of the network. Figure 1(a) illustrates these issues in a sample OpenFlow-based network.

In this paper, we present the design and implementation of HyperFlow, a distributed event-based control plane for OpenFlow, which allows network operators deploy any number of controllers in their networks. HyperFlow provides scalability while keeping network control logically centralized: all the controllers share the same consistent network-wide view and locally serve requests without actively contacting any remote node, thus minimizing the flow setup times. Additionally, HyperFlow does not require any changes to the OpenFlow standard [7] and only needs minor modifications to existing control applications. HyperFlow guarantees loop-free forwarding, and is resilient to network partitioning as well as component failures. Besides, it enables addition of administrative areas to OpenFlow to interconnect independently managed OpenFlow areas. Figure 1(b) shows how HyperFlow addresses the problems associated with a centralized controller in an OpenFlow network.

To the best of our knowledge, HyperFlow is the first distributed control plane for OpenFlow. The only similar design we are aware of is FlowVisor [8] which attacks a slightly different problem. FlowVisor enables multiple controllers in an OpenFlow network by slicing network



(a) An OpenFlow network deploying a single controller

(b) The same network deploying HyperFlow

Figure 1: A multi-site OpenFlow network with single and multiple controllers. Switch and controller association is depicted using colors and shadow pattern. **(a)** Deploying a single controller increases the flow setup time for flows initiated in site 2 and site 3 by 50ms. Also, an increase in flow initiation rates in the remote sites may congest the cross-site links. **(b)** In HyperFlow, all the requests are served by local controllers, and the cross-site control traffic is minimal: controllers mostly get updated by their neighbors.

resources and delegating the control of each slice to a single controller.

An alternative design is to keep the controller state in a distributed data store (*e.g.*, a DHT) and enable local caching on individual controllers. Even though a decision (*e.g.*, flow path setup) can be made for many flows by just consulting the local cache, inevitably some flows require state retrieval from remote controllers, resulting in a spike in the control plane service time. Additionally, this design requires modifications to applications to store state in the distributed data store. In contrast, HyperFlow proactively pushes state to other controllers, thereby enabling individual controllers to locally serve *all* flows. Also, HyperFlow's operation is transparent to the control applications.

We implemented HyperFlow as an application for NOX [3]. The HyperFlow application is in charge of synchronizing controllers' network-wide views (by propagating selected locally generated controller events), redirecting OpenFlow commands targeted to a non-directly-controlled switch to its respective controller, and redirecting replies from switches to the request-originator controllers. To facilitate cross-controller communications, we use publish/subscribe messaging paradigm. The HyperFlow's design and implementation are discussed in the next section. Section 3 discusses when and why a controller application must be modified to become HyperFlow-compatible. To evaluate HyperFlow, Section 4 estimates the maximum level of dynamicity a HyperFlow-based network can have while keeping the window of inconsistency among controllers bounded by a factor of delay between farthest controllers in the network.

2. DESIGN AND IMPLEMENTATION

A HyperFlow-based network is composed of OpenFlow switches as forwarding elements, NOX controllers as decision elements each running an instance of the *HyperFlow controller application*, and an event propagation system for cross-controller communication. All the controllers have a consistent network-wide view and run as if they are controlling the whole network. They all run the exact same controller software and set of applications. Each switch is connected to the best controller in its proximity. Upon controller failure, affected switches must be reconfigured to connect to an active nearby controller.[1] Each controller directly manages the switches connected to it and indirectly programs or queries the rest (through communication with other controllers). Figure 2 illustrates the high-level view of the system.

To achieve a consistent network-wide view among controllers, the HyperFlow controller application instance in each controller selectively publishes the events that change the state of the system through a publish/subscribe system. Other controllers replay all the published events to reconstruct the state. This design choice is based on the following observations: *(a)* Any change to the network-wide view of controllers stems from the occurrence of a network event. A single event may affect the state of several applications, so the control traffic required for direct state synchronization grows with the number of applications, but it is bounded to a small number of events in our solution. *(b)* Only a very small

[1]Currently, in our test environment, we used proprietary hardware vendor configuration interface to reconfigure the controller address.

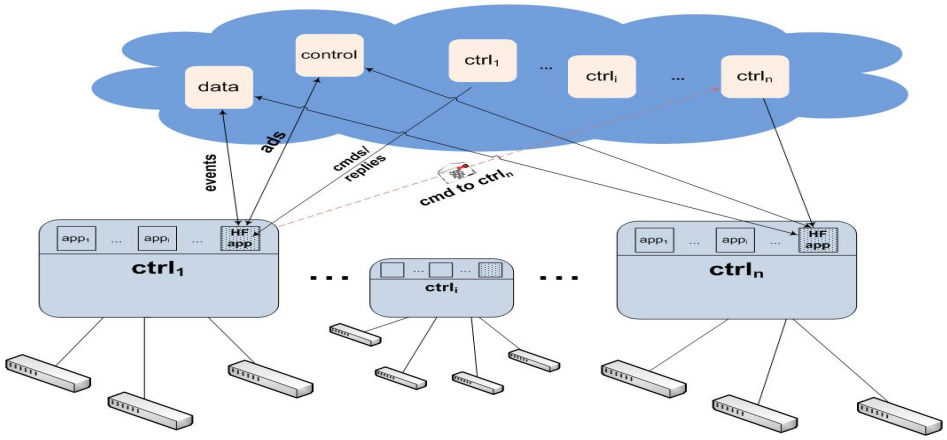


Figure 2: High-level overview of HyperFlow. Each controller runs NOX with the HyperFlow application atop, subscribes to the control, data, and its own channel in the publish/subscribe system (depicted with a cloud). Events are published to the data channel and periodic controller advertisements are sent to the control channel. Controllers directly publish the commands targeted to a controller to its channel. Replies to the commands are published in the source controller.

fraction of network events cause changes to the network-wide view (on the order of tens of events per second for networks of thousands of hosts [3]). The majority of network events (*i.e.*, *packet_in* events) only request service (*e.g.*, routing). *(c)* The temporal ordering of events, except those targeting the same switch, does not affect the network-wide view. *(d)* The applications only need to be minimally modified to dynamically identify the events which affect their state (unlike direct state synchronization which requires each application to directly implement state synchronization and conflict resolution).

2.1 Event Propagation

To propagate controller events to others, HyperFlow uses publish/subscribe messaging paradigm. The publish/subscribe system that HyperFlow uses must provide persistent storage of published events (to provide guaranteed event delivery), keep the ordering of events published by the same controller, and be resilient against network partitioning (*i.e.*, each partition must continue its operation independently and upon reconnection, partitions must synchronize). The publish/subscribe system should also minimize the cross-site[2] traffic required to propagate events, *i.e.*, controllers in a site should get most of the updates of other sites from nearby controllers to avoid congesting the cross-region links. Finally, the system should enforce access control to ensure authorized access.

We implemented a distributed publish/subscribe system satisfying the above requirements using WheelFS [9]. WheelFS is a distributed file system designed to offer flexible wide-area storage for distributed applications. It gives the applications control over consistency, durability, and data placement according to their requirements via *semantic cues*. These cues can be directly embedded in the pathnames to change the behavior of the file system. In WheelFS, we represent channels with directories and messages with files. To implement notification upon message arrival (*i.e.*, new files in the watched directories) HyperFlow controller application periodically polls the watched directories to detect changes.

Each controller subscribes to three channels in the network: the data channel, the control channel, and its own channel. All the controllers in a network are granted permissions to publish to all channels and subscribe to the three channels mentioned. The HyperFlow application publishes selected local network and application events which are of general interest to the data channel. Events and OpenFlow commands targeted to a specific controller are published in the respective controller's channel. Additionally, each controller must periodically advertise itself in the control channel to facilitate controller discovery and failure detection. Access control for these channels are enforced by the publish/subscribe system.

HyperFlow is resilient to network partitioning because WheelFS is. Once a network is partitioned, WheelFS on each partition continues to operate independently. Controllers on each partition no longer receive the advertisements for the controllers on the other partitions and assume they have failed. Upon reconnection of partitions, the WheelFS nodes in both partitions resynchronize. Consequently, the controllers get notified of all the events occurred in the other partition while they

[2]A site is a highly-connected component of the network with a large bisection bandwidth. However, the bandwidth and connectivity between regions is limited. Typically network devices in a single site are geographically co-located.

were disconnected, and the network-wide view of all the controllers converges.[3]

Finally, we note that WheelFS can be replaced by any publish/subscribe system satisfying the above mentioned requirements. We chose WheelFS, primarily because not only it satisfies HyperFlow's requirements, but also enables us to rapidly build a prototype. However, as we show in Section 4, there is room for significant improvements to the existing publish/subscribe system.

2.2 HyperFlow Controller Application

HyperFlow application is a C++ NOX application we developed to ensure all the controllers have a consistent network-wide view. Each controller runs an instance of the HyperFlow application. Our implementation requires minor changes to the core controller code, mainly, to provide appropriate hooks to intercepts commands and serialize events. Below, we describe the functions of the HyperFlow controller application.

Initialization: Upon NOX startup, the HyperFlow application starts the WheelFS client and storage services, subscribes to the network's data and control channels, and starts to periodically advertise itself in the control channel. The advertisement interval must be larger than the highest round-trip time among controllers in a network. The advertisement message contains information about the controller including the identifiers of the switches it directly controls.

Publishing events: The HyperFlow application captures all the NOX built-in events (OpenFlow message events) as well as the events that applications register with HyperFlow. Then, it selectively serializes (using the Boost serialization library) and publishes the ones which are *locally* generated and *affect* the controller state. For that, applications must be instrumented to tag the events which affect their state. Furthermore, applications should identify the parent event of any non-built-in event they fire. This way, HyperFlow can trace each high-level event back to the underlying lower-level event and propagate it instead. Using this method we ensure that the number of events propagated is bounded by the number of the OpenFlow message events generated by the local controller.

The name of the published messages contains the source controller identifier and an event identifier local to the publisher (see Table 1. This scheme effectively partitions the message namespace among controllers and avoids the possibility of any write conflicts. Moreover, a cached copy of a message (file) in our system never becomes stale. Therefore, using semantic cues, we instruct WheelFS to relax consistency requirements and fetch cached copies of files from neighboring controllers as fast as possible.

[3]We note that this requires the network operator define a replication policy appropriate for the network setup.

Message Type	Message Name Pattern
Event	e : *ctrl_id* : *event_id*
Command	c : *ctrl_id* : *switch_id* : *event_id*
Advertisement	*ctrl_id*

Table 1: HyperFlow's message naming convetion. All message types contain the publisher controller id ($ctrl_id$). Events and commands also contain an event identifier ($event_id$) locally generated by the publisher. Commands also contain the identifier of the switch to which the command is targeted.

Replaying events: The HyperFlow application replays all the published events, because source controllers – with the aid of applications – selectively filter out and only publish the events necessary to reconstruct the application state on other controllers. Upon receiving a new message on the network data channel or the controller's own channel, the HyperFlow application deserializes and fires it.

Redirecting commands targeted to a non-local switch: A controller can only program the switches connected directly to it. To program a switch not under direct control of the controller, the HyperFlow application intercepts when an OpenFlow message is about to be sent to such switches and publishes the command to the network control channel. The name of the published message shows that it is a command and also contains the source controller identifier, the target switch identifier, and the local command identifier (similar to the event message identifier).

Proxying OpenFlow messages and replies: The HyperFlow application picks up command messages targeted to a switch under its control (identified in the message name) and sends them to the target switch. To route the replies back to the source controller, the HyperFlow application keeps a mapping between the message transaction identifiers (*xid*) and the source controller identifiers. The HyperFlow application examines the *xid* OpenFlow message events locally generated by the controller. If the *xid* of an event is found in the *xid*-controller map, the event is stopped from being further processed and is published to the network data channel. The name of the message contains both controller identifiers. The original source controller picks up and replays the event upon receipt.

Health checking: The HyperFlow application listens for the controller advertisements in the network control channel. If a controller does not re-advertise itself for three advertisement intervals, it is assumed to have failed. The HyperFlow application fires a *switch leave* event for every switch that was connected to the failed controller. Upon controller failure, HyperFlow configures the switches associated with the failed controller to connect to another controller. Alternatively, either

nearby controllers can serve as a hot standby for each other to take over the IP address.

3. REQUIREMENTS ON CONTROLLER APPLICATIONS

For most controller applications, HyperFlow only requires minor modifications: they must dynamically tag events which affect their state. However, some of them must be further modified to ensure correct operation under temporal event reordering and transiently conflicting controller views, and guarantee scalability. Besides, we discuss how the controller applications must be modified to enable interconnection of independently-managed OpenFlow networks.

Event reordering: In HyperFlow, correct operation of control applications must not depend on temporal ordering of events except those targeting the same entity (*e.g.*, the same switch or link), because different controllers perceive events in different orders. Besides, resilience to network partitioning requires control applications to tolerate out-of-order event delivery (even lagging several hours) without sacrificing correctness, because each partition is notified of the state of the other partitions upon reconnection.

Correctness: Transient inconsistencies among controllers may lead to conflicting decisions. To ensure correct operation in all cases, control applications must forward requests to the *authoritative* controller. The authoritative controller for a given flow is the one managing the flow's source switch. Consider the switching/routing applications as an example: To ensure loop-free forwarding, flow paths must be set up by the controller managing the flow's source switch. Other controllers must redirect the request to the authoritative controller in case they receive a flow initiation event. As another example, consider a network with a policy which requires both the forward and reverse paths of all flows to match. To guarantee this, the source controller must simultaneously set up both paths upon flow initiation. This modification ensures that the policy is always correctly enforced from the source controller's perspective.

Bounded number of possibly effective events: The number of events which possibly affect the state of a HyperFlow-compliant application must be bounded by $O(h + l + s)$, where h is the number of hosts, l is the number of links, and s is the number of switches in the network. In other words, applications whose state may be affected by $O(f(n))$ events, where $f(n)$ is any function of the number of flows in the network, incur a prohibitively large overhead and must be modified.

Measurement applications: Applications which actively query the switches perform poorly under HyperFlow, because the number of queries grows linearly with the number of controllers. Such applications must be modified to partition queries among controllers in a distributed fashion and exchange the results (encapsulated in self-defined events) using HyperFlow. Consider the discovery application as an example. The discovery application must only sends link layer discovery protocol (LLDP) probes out of the switches under its direct control. For each link between a directly and a non-directly controlled switch pair, the discovery application receives an LLDP packet generated by another controller signaling the connectivity. Finally, the discovery application must propagate its link events using HyperFlow.

Interconnecting HyperFlow-based OpenFlow networks: To interconnect two independently managed HyperFlow-based OpenFlow networks (areas), controller applications need to be modified to be made area-aware. They must listen for area discovery events from HyperFlow, enforce the area policies declared using a policy language (*e.g.*, Flow-based Management Language [4]), and exchange updates with the neighboring area through a secure channel providing publish/subscribe service. Applications should encapsulate updates in self-defined events, and have HyperFlow propagate them to the neighboring areas. HyperFlow removes the need for individual control applications to discover their neighbors and communicate directly; instead, control applications just fire events locally and HyperFlow delivers them to neighbors. We note that the implementation of this part is not completed yet.

4. EVALUATION

We performed a preliminary evaluation to estimate the maximum level of network dynamicity it can support while guaranteeing a bounded inconsistency window among controllers. Throughout our experiments we used ten servers each equipped with a gigabit NIC and running as a WheelFS client and storage node. In the near future, we plan to deploy HyperFlow on a large testbed and characterize its performance, robustness, and scalability with a realistic network topology and traffic.

Each NOX instance can handle about 30k flow installs per second [10]. However, it can typically process a far larger number of events which do not trigger an interaction with a switch. Events published through HyperFlow only affect controller state and should not trigger any interaction with controllers. Therefore, using HyperFlow, network operators can easily add more controllers to handle more flow initiation events while keeping the flow setup latency minimal. We note that, since in HyperFlow controllers' operations do not depend on other controllers, they continue to operate even under heavy synchronization load. However, as the load increases, the window of inconsistency among controllers grows (*i.e.*, the time it takes to have the views converge).

To find the number of events that HyperFlow can handle while providing a bounded inconsistency window among controllers, we benchmarked WheelFS independently to find the number of 3-KB sized files (sample serialized *datapath_join* event using the XML archive[4]) we can write (publish) and read. For that, we instrumented the HyperFlow application code to measure the time needed to read and deserialize (with eventual consistency), as well as serialize and write (write locally and don't wait for synchronization with replicas) 1000 such files. We ran each test 10 times and averaged the results. HyperFlow can read and deserialize 987, and serialize and write 233 such events in each second. The limiting factor in this case is the number of reads, because multiple controllers can publish (write) concurrently.

Based on the above analysis, assuming adequate control bandwidth, HyperFlow can guarantee a bounded window of inconsistency among controllers, if the network changes trigger less than around 1000 events per second (*i.e.*, total of 1000 switch and host joins and leaves, and link state changes). We may be able to improve HyperFlow's performance by modifying WheelFS (whose implementation is not mature yet) or designing an alternative publish/subscribe system. Finally, we note that HyperFlow can gracefully handle spikes in network synchronization load without losing any events, however in that period the controller views converge with an added delay.

5. CONCLUSION

This paper presents the design and implementation of HyperFlow which enables OpenFlow deployment in mission-critical networks, including datacenter and enterprise networks. HyperFlow enables network operators deploy any number of controllers to tune the performance of the control plane based on their needs. Besides, it keeps the network control logic *centralized* and *localizes* all decisions to each controller to minimize control plane response time. The HyperFlow application, implemented atop NOX, synchronizes controllers' network-wide views by propagating events affecting the controller state. We choose to build up state by replaying events to minimize the control traffic required to synchronize controller state, avoid the possibility of conflicts in applications' state, and minimize the burden of application modifications. HyperFlow is resilient to network partitions and component failures, minimizes the cross-region control traffic, and enables interconnection of independently-managed OpenFlow networks. We plan to complete our work on cross-area communication and controller bootstrapping, and perform a thorough evaluation of HyperFlow on a large testbed to further characterize its scalability, performance, and robustness.

[4]This is significantly larger than most serialized events. Also, in real deployments we should use binary archives which significantly reduces message sizes.

6. REFERENCES

[1] OpenFlow Consortium. `http://openflowswitch.org/`.

[2] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A clean slate 4D approach to network control and management. *SIGCOMM Computer Communication Review 35*, 5 (2005), 54.

[3] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. NOX: towards an operating system for networks. *SIGCOMM Computer Communication Review 38*, 3 (2008), 105–110.

[4] Hinrichs, T. L., Gude, N. S., Casado, M., Mitchell, J. C., and Shenker, S. Practical declarative network management. In *WREN '09: Proceedings of the 1st ACM Workshop on Research on Enterprise Networking* (New York, NY, USA, 2009), ACM, pp. 1–10.

[5] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: enabling innovation in campus networks. *SIGCOMM Computer Communication Review 38*, 2 (2008), 69–74.

[6] Naous, J., Erickson, D., Covington, G. A., Appenzeller, G., and McKeown, N. Implementing an OpenFlow switch on the NetFPGA platform. In *ANCS* (2008), M. A. Franklin, D. K. Panda, and D. Stiliadis, Eds., ACM, pp. 1–9.

[7] OpenFlow Consortium. OpenFlow switch specification. `http://openflowswitch.org/documents.php`.

[8] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. FlowVisor: A Network Virtualization Layer. Tech. Rep. OPENFLOW-TR-2009-01, OpenFlow Consortium, October 2009.

[9] Stribling, J., Sovran, Y., Zhang, I., Pretzer, X., Li, J., Kaashoek, M. F., and Morris, R. Flexible, wide-area storage for distributed systems with wheelfs. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)* (Boston, MA, April 2009).

[10] Tavakoli, A., Casado, M., Koponen, T., and Shenker, S. Applying nox to the datacenter. In *Proceedings of workshop on Hot Topics in Networks (HotNets-VIII)* (2009).

[11] Yan, H., Maltz, D. A., Ng, T. S. E., Gogineni, H., Zhang, H., and Cai, Z. Tesseract: A 4d network control plane. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)* (2007).