ONOS: Towards an Open, Distributed SDN OS

Pankaj Berde†, Matteo Gerola‡, Jonathan Hart†, Yuta Higuchi§,
Masayoshi Kobayashi§, Toshio Koide§, Bob Lantz†,
Brian O'Connor†, Pavlin Radoslavov†, William Snow†, Guru Parulkar†

†*Open Networking Laboratory, USA* §*NEC Corporation of America, USA* ‡*Create-Net, Italy*

ABSTRACT

We present our experiences to date building ONOS (Open Network Operating System), an experimental distributed SDN control platform motivated by the performance, scalability, and availability requirements of large operator networks. We describe and evaluate two ONOS prototypes. The first version implemented core features: a distributed, but logically centralized, global network view; scale-out; and fault tolerance. The second version focused on improving performance. Based on experience with these prototypes, we identify additional steps that will be required for ONOS to support use cases such as core network traffic engineering and scheduling, and to become a usable open source, distributed network OS platform that the SDN community can build upon.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network Operating Systems; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

Keywords

ONOS; Network Operating System; Distributed Controller; Software Defined Networking; SDN; Controller; OpenFlow

1. INTRODUCTION

In just a few years, Software Defined Networking (SDN) has created enormous interest in academia and industry. An open, vendor neutral, control-data plane interface such as OpenFlow [1] allows network hardware and software to evolve independently, and facilitates the replacement of expensive, proprietary hardware and firmware with commodity hardware and a free, open source Network Operating System (NOS). By managing network resources and providing high-level abstractions and APIs for interacting with, managing, monitoring and programming network switches, the NOS provides an open platform that simplifies the creation of innovative and beneficial network applications and services that work across a wide range of hardware.

HotSDN'14, August 22, 2014, Chicago, IL, USA.

ACM 978-1-4503-2989-7/14/08 ...$15.00.
http://dx.doi.org/10.1145/2620728.2620744

To support large production networks, a NOS has to meet demanding requirements for scalability, performance and availability. Based on discussions with network operators, and considering the use case of traffic engineering in service provider networks, we have identified several mutually challenging requirements (see Figure 1):

- **High Throughput:** up to 1M requests/second
- **Low Latency:** 10 - 100 ms event processing
- **Global Network State Size:** up to 1TB of data
- **High Availability:** 99.99% service availability

To address the above challenges, we have begun work on an experimental system – Open Network Operating System (ONOS). ONOS adopts a distributed architecture for high availability and scale-out. It provides a global *network view* to applications, which is logically centralized even though it is physically distributed across multiple servers.

ONOS follows in the footsteps of previous closed source distributed SDN controllers such as Onix [2], but we intend for ONOS to be released as an open source project which the SDN community can examine, evaluate, extend and contribute to as desired.

ONOS capabilities have evolved through the development of two major prototypes:

- Prototype 1 focused on implementing a global network view on a distributed platform for scale-out and fault tolerance.
- Prototype 2 focused on improving performance, notably event latency. We added an event notification framework, made changes to our data store and data model, and introduced a caching layer.

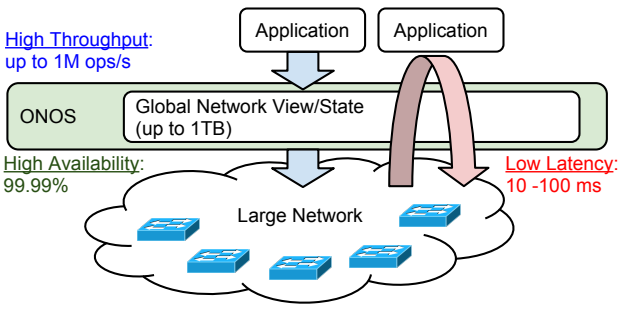

Figure 1: ONOS Requirements

In this paper, we describe our experiences building each ONOS prototype, evaluating what worked and what didn't, and we discuss next steps to evolve ONOS into a system that may come closer to meeting our goals for performance and usability.

2. PROTOTYPE 1: NETWORK VIEW, SCALE-OUT, FAULT TOLERANCE

The initial challenge for ONOS was to create a useful abstraction, the global network view, on a system that runs across multiple servers for control plane scale-out and fault tolerance. The first prototype was built using several open source building blocks to allow us to quickly validate ideas and explore design possibilities. We based the first prototype on an existing open-source single-instance SDN controller, Floodlight [3]. We made use of several Floodlight modules, including the switch manager, I/O loop, link discovery, module management, and REST APIs. Figure 2 shows the high level system architecture.

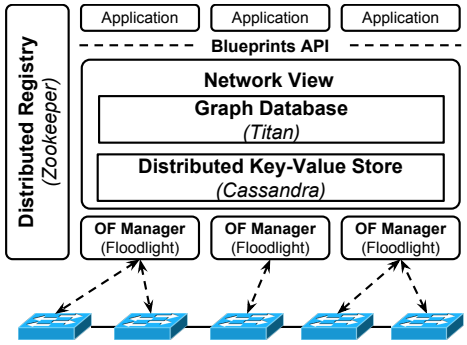

Figure 2: Prototype 1 Architecture

Global Network View. ONOS maintains a global network view to manage and share network state across ONOS servers in a cluster. This abstraction provides a graph model of the network which naturally corresponds to the underlying network structure.

Network topology and state discovered by each ONOS instance such as switch, port, link, and host information is used to construct the global network view. Applications then read from the global network view to make forwarding and policy decisions, which are in turn written to the network view. As applications update and annotate the view, these changes are sent to the corresponding OpenFlow managers and programmed on the appropriate switches.

Our initial network view data model was implemented using the Titan [4] graph database, with the Cassandra [5] key-value store for distribution and persistence, and the Blueprints [6] graph API to expose network state to applications. Since Cassandra is an eventually consistent data store, our network view was eventually consistent.

Scale-out. A key feature of ONOS is its distributed architecture for scale-out performance and fault tolerance. ONOS runs on multiple servers, each of which acts as the exclusive master OpenFlow controller for a subset of switches. An individual ONOS instance is solely responsible for propagating state changes between the switches it controls and the global network view. As the data plane capacity grows or demand on the control plane increases, additional instances can be added to the ONOS cluster to distribute the control-plane workload.

Fault tolerance. The distributed architecture of ONOS allows the system to continue operating when a component or ONOS instance fails by redistributing work to other remaining instances. ONOS's architecture allows components to exist as a single instance at runtime but provides the ability to have multiple redundant instances waiting to take over in case the primary instance fails. This is achieved by electing a leader amongst all the instances at runtime to host the primary instance.

A switch connects to multiple ONOS instances, but only one instance is the master for each switch. The master instance alone is responsible for discovering switch information and programming the switch. When an ONOS instance fails, the remaining instances elect a new master for each of the switches that were previously controlled by the failed instance. A consensus based leader election is performed for each switch to ensure at most one ONOS instance is in charge of each switch.

We used ZooKeeper [7] to manage switch-to-controller mastership, including detecting and reacting to instance failure. An ONOS instance must contact the ZooKeeper ensemble in order to become master for a switch. If an instance loses contact with ZooKeeper, another instance will take over control of its switches. ZooKeeper uses a consensus protocol to maintain strong consistency, and ZooKeeper is fault tolerant as long as a majority of servers are available.

2.1 Evaluation

The first ONOS prototype was developed over the course of four months. Building on existing open source software components allowed us to rapidly prototype the system to demonstrate some of its basic features and evaluate the consistency model. We demonstrated the first ONOS prototype at the Open Networking Summit [8] in April 2013. The demonstration showed ONOS controlling hundreds of virtual switches, programming end-to-end flows using the network view, dynamically adding switches and ONOS instances to the cluster, failover in response to ONOS instance shutdown, and rerouting in response to link failure. Overall, we were able to build and demonstrate the basic features of the system, but several design choices resulted in poor performance and usability, as we explain below.

Consistency and Integrity. Titan's transactional semantics maintained the graph's structural integrity (for example, requiring that edges always connect two nodes) on top of Cassandra's eventually consistent data store. In some of our experiments, we used redundant and identical flow path computation on all instances, and we observed that path computation modules on each instance eventually converged to same path. As with legacy networks that use an embedded control plane and lack a logically centralized view, eventual consistency does not seem to prevent reasonable operation of the control plane, but we still think it may benefit from some degree of sequencing to provide deterministic state transitions.

Low Performance and Visibility. Although performance hadn't been an explicit goal for the first prototype, latency was much worse than expected; for example, reacting to a link failure could take up to 30 seconds. Off-the-shelf, open source components had helped us to build the system quickly, but diagnosing performance problems required delving into a large, complex, and unfamiliar code base, adding custom

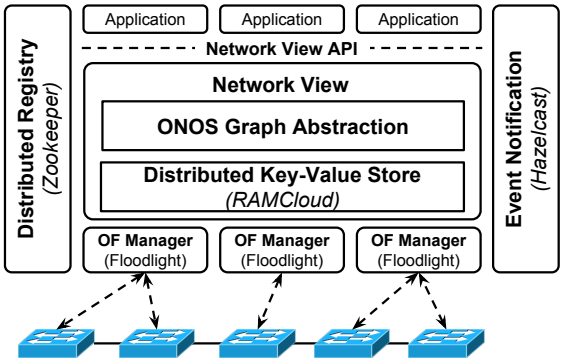

Figure 3: Prototype 2 Architecture

instrumentation to Titan and Cassandra to record the sequence and timing of operations.

Data Model Issues. To implement the network view on top of Titan, we had modeled all data objects (including ports, flow entries, etc.) as vertices. This required indexing vertices by type to enable queries such as enumerating all the switches in the network. Index maintenance became a bottleneck when concurrently adding a large number of objects, including common cases such as creating switches and links during initial topology discovery or installing a large number of flows. Storing and maintaining references between many small objects also meant that dozens of graph database update operations could be required for relatively simple operations such as adding or removing a switch, or clearing a flow table. Additionally, users of the network view were exposed to a data representation that was overly complicated and didn't match the mental model of network devices connected together by network links.

Excessive Data Store Operations. The mapping from Titan's graph data model to Cassandra's key-value data model resulted in an excessive number of data store operations, some of which were remote. Simple ONOS operations such as adding a switch or flow table entry were slowed down due to several network round trips to read or write multiple objects into the data store. In addition, storing vertices, edges, and metadata in a shared table and index introduced unnecessary contention between seemingly independent operations.

Polling. In the initial prototype, we did not have time to implement notifications and messaging across ONOS instances. ONOS modules and applications had to poll the database periodically to detect changes in network state. This had the pernicious effect of increasing the CPU load due to unnecessary polling while simultaneously increasing the delay for reacting to events and communicating information across instances.

Lessons Learned. Our evaluation of our first ONOS prototype indicated that we needed to design a more efficient data model, reduce the number of expensive data store operations, and provide fast notifications and messaging across nodes. Additionally, the API needed to be simplified to represent the network view abstraction more clearly without exposing unnecessary implementation details.

3. PROTOTYPE 2: IMPROVING PERFORMANCE

Our next prototype focused on improving the performance of ONOS. This resulted in changes to our network view architecture and the addition of an event notification framework, as shown in Figure 3.

One of the biggest performance bottlenecks of the first prototype was remote data operations. In the second prototype, we addressed this issue through two different approaches: (1) making remote operations as fast as possible, and (2) reducing the number of remote operations that ONOS has to perform.

RAMCloud Data Store. We began the effort focusing on improving the speed of our remote database operations. To better understand how data was being stored, we replaced the first prototype's graph database stack (Titan and Cassandra) with a Blueprints graph implementation [9] on top of RAMCloud [10]. This allowed us to use our existing code, which used the Blueprints API for data storage, with RAMCloud. RAMCloud is a low latency, distributed key-value store, which offers remote read/write latency in the 15-30 μs range. A simpler software stack combined with extensive instrumentation provided many insights into the bottlenecks introduced by the network view's data model.

Optimized Data Model. To address the inefficiency of the generic graph data model, we designed a new data model optimized for our specific use case. A table for each type of network object (switch, link, flow entry, etc.) was introduced to reduce unnecessary contention between independent updates. The data structures have been further optimized to minimize the number of references between elements, and we no longer maintain the integrity of the data at the data store level. This results in far fewer read/write operations for each update. Indeed, most element types can be written in a single operation because we don't have to update multiple objects to maintain references as generic property graphs do.

Topology Cache. Topology information is updated infrequently but is read-heavy, so we can optimize for read performance by keeping it in memory on each instance. In our second prototype, we implemented a caching layer to reduce the number of remote database reads for commonly read data. Furthermore, the in-memory topology view implements a set of indices on top of the data to allow faster lookups. These indices are not stored in the data store, but can be reconstructed from the data store at any time. This process requires reading an entire snapshot of the topology data from the data store; however, it only occurs when a new ONOS node joins the cluster, so it is not time-critical.

The in-memory topology data is maintained using a notification-based replication scheme. The replication scheme is eventually consistent – updates can be received at different times in different orders on different instances. However, to ensure that the system maintains the *integrity* of the topology schema observed by the applications, updates to the topology are applied atomically using schema integrity constraints on each instance. This means an application cannot read the topology state in the middle of an update or observe, for example, a link that does not have a port on each end.

Event Notifications. We addressed the polling issue by building an inter-instance publish-subscribe event notification and communication system based on Hazelcast [11]. We

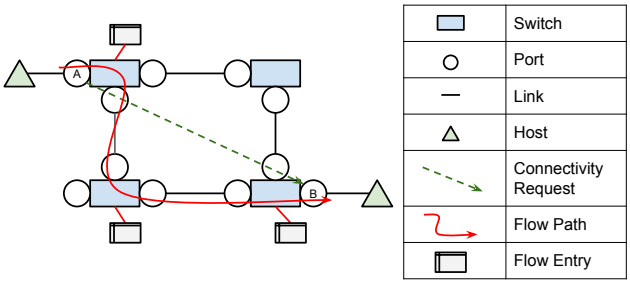

Figure 4: Network View: Connectivity requests cause flow paths to be created using flow entries.

created several event channels among all ONOS instances based on notification type, such as topology change, flow installation and packet-out.

Network View API. We took the opportunity to replace the generic Blueprints graph API with an API designed specifically for network applications. Figure 4 shows the application's view of the network. Our API consisted of three main areas:

- A *Topology* abstraction representing the underlying data plane network, including an intuitive graph representation for graph calculations.
- *Events* occurring in the network or the system that an application may wish to act on.
- A *Path Installation* system to enable applications to set up flows in the network.

3.1 Evaluation

Our second prototype has brought us much closer to meeting our performance goals, in particular our internal system latency requirements. In the following sections, we will show the evaluated performance of three categories: (1) basic network state changes, (2) reaction to network events, and (3) path installation.

3.1.1 Basic Network State Changes

The first performance metrics are latency and throughput for network state changes in the network view. As modifying the network state in a network view is a basic building block of ONOS operations, its performance has a significant impact on the overall system's performance.

For this measurement, we connected 81 OpenFlow switches, arranged in a typical WAN topology, to a 3-node ONOS cluster and measured the latency of adding switches and links to the network view. The switches were Open vSwitches [12] and had an average of 4 active ports.

Table 1 shows the latency for adding a switch, and its breakdown. With RAMCloud using the generic graph data model of our first prototype, it requires 10 read and 8 write RAMCloud operations to add a single switch, which takes 22.2 ms on our ONOS cluster connected with 10 Gb/s Ethernet. With the new data model, adding a switch and a port requires only one write operation each, and the total latency is significantly reduced to 1.19 ms. To improve serialization time, we switched serialization frameworks from Kryo [13] (which is schema-less) to Google Protocol Buffers [14] (which uses a fixed schema); this change reduced the operation's latency to 0.244 ms. We also explored how performance is improved using optimized network I/O (e.g., kernel bypass) and 10 Gb/s Infiniband hardware on a RAMCloud cluster; with these combined optimizations, adding a switch took 0.099 ms.

We also evaluated the latency to add a link, and the results were similar. With the new data model and serialization optimization, we reduced the latency from 0.722 ms (generic data model) to 0.150 ms. Using Infiniband with kernel bypass, it is further reduced to 0.075 ms.

In our current design, network state is written to RAMCloud sequentially, so the throughput is simply the inverse of the latency.

Table 1: Latency for Adding a Switch
(Ser. = Serialization, Des. = Deserialization; Unit: ms)

	Read	Write	Ser.	Des.	Other	Total
1. Generic Graph Data Model	10.1	3.5	7.2	0.93	0.56	22.2
2. New Data Model	–	0.28	0.89	–	0.017	1.19
3. (2)+Proto. Buf.	–	0.23	0.01	–	0.006	0.244
4. (3)+Infiniband	–	0.08	0.01	–	0.006	0.099

3.1.2 Reaction to Network Events

The second performance metric is end-to-end latency of the system for updating network state in response to events; examples include rerouting traffic in response to link failure, moving traffic in response to congestion, and supporting VM migration or host mobility. This metric is relevant because it is most directly related to SLAs guaranteed by network operators.

For this experiment, we connected a 6-node ONOS cluster to an emulated Mininet [15] network of 206 software [12] switches and 416 links. We added 16,000 flows across the network, and then disabled one of the switch's interfaces, causing 1,000 flows to be rerouted. All affected flows had 6-hop path before and a 7-hop path after rerouting.

Table 2 shows the median and 99th percentile of the latencies of rerouting experiment. The latency is presented as two values: (1) the time from the point where the network event is detected by ONOS (via an OpenFlow port status message in this case) to the point where ONOS sends the first FlowMod (`OFPT_FLOW_MOD`) OpenFlow message to reprogram the network, and (2) the total time taken by ONOS to send all FlowMods to reprogram the network, including (1). The latency to the first FlowMod is more representative of the system's performance, while the total latency shows the impact on traffic in the data plane. Note that we do not consider the effects of propagation delay or switch dataplane programming delay in our total latency measurement, as these can be highly variable depending on topology, controller placement, and hardware.

Table 2: Latency for Rerouting 1000 Flows

Latency	Median	99th %ile
Latency to the 1st FlowMod	45.2 ms	75.8 ms
Total Latency	71.2 ms	116 ms

3.1.3 Path Installation

The third performance metric measures how frequently the system can process application requests to update network state, and how quickly they are reflected to the physical

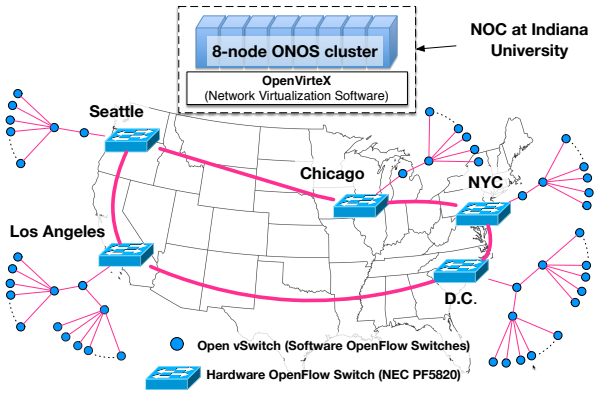

Figure 5: Internet2 Demo Topology and Configuration

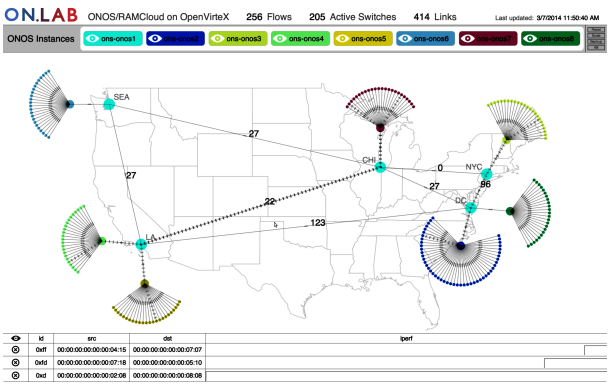

Figure 6: The ONOS GUI shows the correctly discovered topology of 205 switches and 414 links.

network. Application requests include establishing connectivity between hosts, setting up tunnels, updating traffic policy, or scheduling resources. Specifically, we measured the latency and throughput for path installation requests from an application.

We started with the same network topology used in Section 3.1.2. With 15,000 pre-installed static flows, we added 1,000 6-hop flows and measured the throughput and latency to add the new flows.

Table 3 shows the latency performance. The latency is computed in the same way as in Section 3.1.2 except an application event (i.e., path setup request), rather than a network event, starts the timer. Throughput is inversely related to latency due to serialization of processing in this prototype. For example, the median of the throughput was 18,832 paths/sec (derived from the median of total latency, 53.1 ms).

Table 3: Path Installation Latency

	Median	99th %ile
Latency to the 1st FlowMod	34.1 ms	68.2 ms
Total Latency	53.1 ms	97.9 ms

With Prototype 2, we approach our target latency for system response to network events (10-100 ms as stated in Section 1), but we still do not meet our target of the path setup throughput (1M path/sec). The current design does not fully utilize parallelism (e.g., all the path computation is done by a single ONOS instance), and we think we can increase the throughput as we distribute the path computation load among multiple ONOS instances.

3.2 Demonstration on Internet2

At the Open Networking Summit in March 2014, we deployed our second prototype on the Internet2 [16] network, demonstrating (1) ONOS' network view, scale-out, and fault tolerance, (2) operation on a real WAN, (3) using virtualized hardware and software switches, and (4) faster ONOS and link failover. Figure 5 illustrates the system configuration: a geographically distributed backbone network of five hardware OpenFlow switches, each connected to an emulated access network of software switches. We used OpenVirteX [17] to create a virtual network of 205 switches and 414 links on this physical infrastructure, and this virtual network was controlled by a single 8-node ONOS cluster located in the NOC at Indiana University. Figure 6 shows the ONOS GUI and correctly discovered topology. Note that the links between Los Angeles and Chicago and between Chicago and Washington D.C. are virtual links added by OpenVirteX.

4. RELATED WORK

The development of ONOS continues to be influenced and informed by earlier research work [18, 19], and by many existing SDN controller platforms. Unlike other efforts, ONOS has focused primarily on use cases outside the data center, such as service provider networks.

Onix [2] was the first distributed SDN controller to implement a global network view, and it influenced the original development of ONOS. Onix originally targeted network virtualization in data centers, but it has remained closed source and further development has not been described in subsequent publications.

Floodlight [3] was the first open source SDN controller to gain traction in research and industry. It is evolving to support high availability in the manner of its proprietary sibling, Big Switch's Big Network Controller [20] via a hot standby system. It does not support a distributed architecture for scale-out performance.

OpenDaylight [21] is an open source SDN controller project, backed by a large consortium of networking companies, that implements a number of vendor-driven features. Similarly to ONOS, OpenDaylight runs on a cluster of servers for high availability, uses a distributed data store, and employs leader election. At this time, the OpenDaylight clustering architecture is evolving, and we do not have sufficient information to provide a more detailed comparison.

5. DISCUSSION: TOWARDS AN OPEN, DISTRIBUTED NETWORK OS

In this paper, we have described some of our experiences and lessons learned while building the first two prototype versions of ONOS, a distributed SDN control platform which we hope to develop into a more complete Network OS that meets the performance and reliability requirements of large production networks, while preserving the convenience of a global network view.

We are currently working with a small set of partner organizations, including carriers and vendors, to create the next version of ONOS. We intend to prototype several use cases

that will help drive improvements to the system's APIs, abstractions, resource isolation, and scheduling. Additionally, we will need to continue work on meeting performance requirements and developing a usable open source release of the system.

Use Cases. The promise of any OS platform, network or otherwise, is to enable applications. To date, we have implemented a limited set of applications on ONOS: simple proactive route maintenance, and BGP interfacing (SDN-IP [22]). Moving forward, we plan on exploring three broad use cases: traffic engineering and scheduling of packet optical core networks; SDN control of next generation service provider central offices and points of presence (PoPs) comprising network, compute, storage, and customer management functionalities; and remote network management, including virtualization of customer networks.

Abstractions. We expect ONOS to allow applications to examine the global network view and create *flow paths* that specify full or partial routes along with traffic that should flow over that route and other actions that should be taken, or use a global *match-action* (or match-instruction) abstraction which provides the full power of OpenFlow to enable an application to program any switch from a single vantage point.

For applications that do not depend on specific paths through the network, ONOS provides a simple *connectivity* abstraction. In this case, ONOS modules may handle the mechanics of installing a path and maintaining it as the network topology, host location, or usage changes. Our experience building and deploying our SDN-IP peering application [22] showed that a simple connectivity abstraction was enough to implement the majority of the required behavior.

Isolation and Security. We hope to improve the isolation and security of ONOS applications. We would like to detect and resolve conflicting policies, routes, and flow entries to allow applications to coexist without undesired interference. Additionally, we would like to have a mechanism to manage what applications can see, what they are permitted to do, and what resources they can use. An improved module framework may help to enforce isolation while supporting dynamic module loading and reloading for on-line reconfiguration and software upgrades.

Performance. We are close to achieving low-latency end-to-end event processing in ONOS, but have not yet met our throughput goals. We hope to improve the system's throughput by exploring new ways to parallelize and distribute large workloads.

Open Source Release. We are currently working with our partners to improve the system's reliability and robustness, to implement missing features (e.g. OpenFlow 1.3 [1]) required for experimental deployments, and to prepare a usable code base, a development environment, and documentation. We are also working on infrastructure and processes that will help to support a community of ONOS core and application developers. Our goal, by the end of 2014, is the open source release of a usable ONOS system that the SDN community will be able to examine, use, and build upon.

6. ACKNOWLEDGMENTS

We would like to thank the following people for their contributions to the design and implementation of ONOS: Ali Al-Shabibi, Nick Karanatsios, Umesh Krishnaswamy, Pingping Lin, Nick McKeown, Yoshitomo Muroi, Larry Peterson, Scott Shenker, Naoki Shiota, and Terutaka Uchida.

Thanks also to John Ousterhout and Jonathan Ellithorpe for their assistance with RAMCloud, and to our anonymous reviewers for their comments.

7. REFERENCES

[1] Open Networking Foundation. OpenFlow specification. `https://www.opennetworking.org/sdn-resources/onf-specifications/openflow/`.

[2] T. Koponen, M. Casado, N. Gude, J. Stribling, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI '10*, volume 10. USENIX, 2010.

[3] Floodlight Project. `http://www.projectfloodlight.org/`.

[4] Titan Distributed Graph Database. `http://thinkaurelius.github.io/titan/`.

[5] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2), 2010.

[6] Tinkerpop. Blueprints. `http://blueprints.tinkerpop.com/`.

[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX '10 Annual Technical Conference*, 2010.

[8] Open Networking Summit. `http://www.opennetsummit.org/`.

[9] J. Ellithorpe. TinkerPop Blueprints implementation for RAMCloud. `https://github.com/ellitron/blueprints-ramcloud-graph/`.

[10] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, et al. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS Operating Systems Review*, 43(4), Jan. 2010.

[11] Hazelcast Project. `http://www.hazelcast.org/`.

[12] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker. Extending networking into the virtualization layer. In *HotNets '09*. ACM, 2009.

[13] Esoteric Software. Kryo. `https://github.com/EsotericSoftware/kryo/`.

[14] Google. Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

[15] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Hotnets '10*. ACM, 2010.

[16] Internet2. `http://www.internet2.edu/`.

[17] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, W. Snow, and G. Parulkar. OpenVirteX: A network hypervisor. In *ONS '14*, Santa Clara, CA, 2014. USENIX.

[18] S. Schmid and J. Suomela. Exploiting Locality in Distributed SDN Control. In *HotSDN '13*. ACM, 2013.

[19] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN '13*. ACM, 2013.

[20] Big Switch Networks. Big Network Controller. `http://www.bigswitch.com/products/SDN-Controller/`.

[21] Open Daylight Project. `http://www.opendaylight.org/`.

[22] P. Lin, J. Hart, U. Krishnaswamy, T. Murakami, M. Kobayashi, A. Al-Shabibi, K.-C. Wang, and J. Bi. Seamless interworking of SDN and IP. In *SIGCOMM '13*. ACM, 2013.