SDN Controller Comparison: Beehive versus ONOS

Christodoulos Karavasilis and Jeremy Ko

April 9, 2017

Contents

1 Introduction

In this report, we provide a technical comparison between two SDN control platforms, ONOS and Beehive. While both controllers argue they provide high performance and scalability, the design and architecture of both platforms varies greatly. We first provide a high-level overview of the architecture of both Beehive and ONOS. Next, we provide a technical comparison between the two systems, and make predictions as to which system should perform better in different networking scenarios. Finally, we test these predictions by conducting experiments using Mininet and cbench to provide a virtual network. From these results, we make conclusions as to which platform better meets modern networking requirements.

2 Background Information

2.1 Beehive

Beehive is a distributed control platform and a programming model that simplifies application development by automatically transforming centralized applications into distributed ones. Control applications consist of a set of asynchronous message handlers that store their state in application-defined dictionaries. Beehive partitions the application's logic and preserves state consistency by using message handlers that access the same data (entries in a dictionary) that are executed by the same process. Controllers are denoted as hives and the state of applications is stored in cells, which are owned by processes called bees. More specific, a cell corresponds to a key-value pair in a specific application dictionary. The "map" function map(A,M) maps a message of type M to a set of cells in application's A dictionaries. This way we can find the bee that is responsible for processing a specific message.

Beehive achieves fault-tolerance by having a message handler's operations being transactional and replicating these transactions among different hives. This is done by forming a colony with bees of different hives, depending on replicating factor, and having the bees inside the colony form a Raft quorum to replicate the transaction. This is done more efficiently using batched transactions. By providing the ability to migrate bees in real-time between different hives, the platform can use techniques to optimize the placement of bees to maximize switch-to-bee locality.

Beehive is a flexible platform that can support well-known abstractions of existing SDN controllers, as well as accommodate applications of different designs in the same place. The existing Beehive SDN controller maintains a complete graph of the network where each node is a network object model (NOM). Node controllers are the active components that manage NOM objects. They interact with drivers that map physical entities to NOM objects, which makes the controllers protocol agnostic. The node controller is distributed and placed close to the switches. The Path Manager is a centralized application requiring a global view of the network, that takes an end-to-end forwarding rule and compiles it into individual flows across switches. The Consolidator is a poller that makes sure the state of switches and the corresponding node controllers are consistent. The Monitor sends flow stat queries to find out if any triggers should fire. Triggers are active thresholds on bandwidth consumption and duration of flow entries and constitute scalable alternatives to different applications directly polling switches for flow statistics.

2.2 ONOS

The main features of ONOS is to provide a scalable, fault-tolerant, and high-performance networking control platform. ONOS was originally developed as a proof of concept for providing a distributed network control platform using readily available open source distribution libraries. Since then, ONOS has made several adjustments to its core design to remain competitive with other SDN controllers.

The design goals of ONOS were to provide code modularity, configurability, separation of concern, and protocol agnosticism. ONOS follows the layered architecture pattern. Code is modularized into three tiers separating the Northbound Core, Distributed Core, and Southbound Core. As applications only interact with the Northbound Core interface, much of the difficulty of programming distributed systems is abstracted away from the developer. As with all layered architectures, these benefits come at the cost of increased communication latency between the top layer, namely the applications, and the bottom layer, namely the switches.

High performance is achieved by local reads to the Global Network View (GNV), at the cost of consistency. Inconsistencies are minimized by periodically verifying the GNV against neighbouring instances in a process called "anti-entropy". Since all updates to state are timestamped, unsynchronized cluster nodes can reapply updates following the total order across all updates to become synchronized. Additionally, rather than offloading all state management to a centralized data store, the datastore key space is partitioned into smaller units to allow for distributed state management. This is similar to the cell sharding capabilities of Beehive.

Components requiring strong consistency are maintained via the Raft consensus algorithm, provided by Atomix [3]. Transactional updates to network state are batched and applied to all cluster instances using Raft. Cluster membership and leader election is also provided by the Atomix framework to handle cluster coordination and provide fault-tolerance.

3 Technical Comparison

While both Beehive and ONOS promise performance, availability, and scalability, they achieve these goals using vastly different architectures.

We predict that Beehive's event-driven architecture should react to network events much faster than ONOS's layered architecture. Through Beehive's map function messages to cells, network events are relayed nearly directly to an application's message handlers. On the contrary, network events in ONOS are handled by its Southbound Core layer, and must traverse all layers to reach applications. To test this, we measure the latency required to send a ping between two hosts where new flow entries must be installed via a reactive forwarding application.

It was originally shown that Beehive scales better in large clusters with high replication factor over ONOS. For example, with a cluster of 5 nodes at maximum replication factor, Beehive achieves throughput 1000 times larger than ONOS [6]. ONOS has since moved its centralized external datastore to sharded partitions of stateful resources, which is believed to be the main reason for the difference in throughput. To test this, we rerun the scalability tests done in [6] using ONOS's new design.

For read heavy applications, both ONOS and Beehive rely on reads to a local instance of the network model (GNV for ONOS, NOM for Beehive). Beehive and ONOS should only differ in performance when updating stateful resources. For applications that do not require strong consistency of stateful resources, the eventual consistency model provided by ONOS should allow for better performance over Beehive due to lower synchronization overhead.

Finally, it will be interesting to compare failover times on both systems. Beehive is reliant on a self-implemented Raft consensus algorithm to elect a new master bee in the case of faults. ONOS handles cluster membership and leader election using the Atomix framework, which is also based on the Raft consensus algorithm. We suspect that the distribution framework provided by Atomix may be better tested by the open-source community compared to

Beehive. However, the lightweight solution provided by Beehive may achieve better performance than the heavily abstracted implementation of Atomix meant to be used in general purpose applications. The other performance advantage of Beehive is to optimize the placement of newly elected master bees after a fault has occurred so that messages destined for a particular cell do not have to be relayed between hives. Such optimization is desirable for sharded stateful resources.

To test these hypothesis, we have conducted a variety of networking experiments as described in the following section.

4 Performance Test Results

For the following tests, we ran the latest stable release of ONOS, version 1.8.2 (available at [2]). For Beehive, we cloned the latest master branch from its public git repository (available at [1]). We use the Mininet version 2.3 (available at [4]) to emulate our network topology. Tests were ran on the Google Compute Engine (GCE) using its high CPU virtual machines with 8 virtual cores and 7.2 GB of RAM.

4.1 Latency between Applications and Switch Events

Testing how responsive the controllers are with varying number of nodes and replication factors constitutes a good metric for comparing them. As a first step, we deployed each controller on one node and tested them using cbench (available at [5]). This benchmarking tool emulates a number of switches sending packet_in messages to the controller and waiting for matching flow mods responses. As shown in Figure 1, this low-level test shows ONOS responding to more packet_in requests than Beehive, therefore having lower latency.

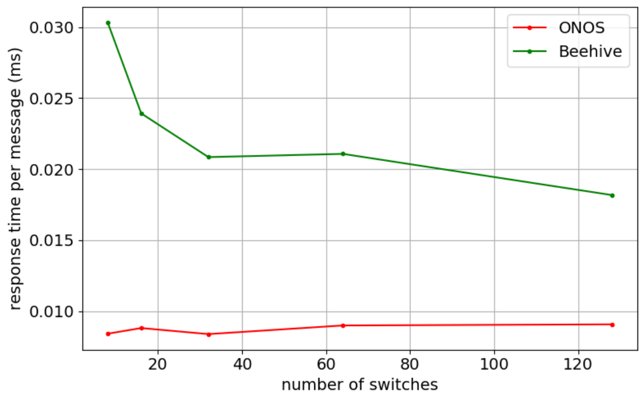

Figure 1: Packet_in to flow mod latency using cbench for varying number of emulated switches.

Next we attempt to see if the same latency results carry over to a more realistic scenario using an emulated network topology. We believe this is a better indication of the response time of controller applications to switch events. Our setup was as follows. Mininet was used to create a linear topology of Open vSwitches of length N with two hosts, h1 and h2, at each end. Additionally, each link was configured in Mininet to have a fixed 0.1 ms delay and 10 Mb/s bandwidth. For both ONOS and Beehive, we run a reactive forwarding application, where upon receiving packet_in messages from each switch, the controller installs a flow entry into the switch to forward the packet to the next switch in the chain. Therefore, for a chain of N switches, each controller must install 2N flow entries (N flows from h1 to h2, and N flows from h2 back to h1) one after the other before h1 is able to successfully ping h2. The round-trip time of the first ping by the hosts is an indication as to how quickly the controller was able to respond to switch events and provide routing instructions.

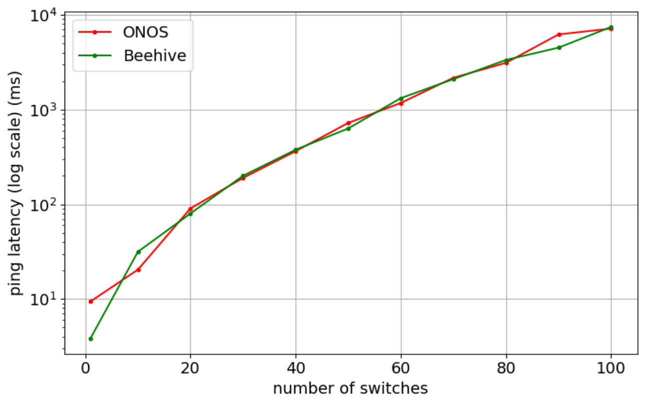

Figure 2: Log latency of the first ping emulated on a linear topology of varying switch length.

Results in Figure 2 show that Beehive has slightly lower latency at low switch counts, but are relatively indistinguishable at higher switch counts. By subtracting link delays from the ping latency, the average time it takes for the first ping to be forwarded from one switch to the next can be estimated. For small N, the time is approximately 1.4 ms for a packet_in request to be sent and the corresponding flow entry to be installed. However, as the number of switches reaches above 50, the overhead of emulating 50 switches begins to stress the CPU capacity of our GCE virtual machine. This most likely explains the exponential increase in ping latency, and makes it difficult to judge the accuracy of our results for large switch counts.

4.2 Cluster Throughput and Scalability

A very important aspect of performance is the ability to maintain high throughput as the size of a cluster increases. As nodes are added to a cluster, the overhead of replication and state persistence increases. However, workload can also be spread across the nodes of a cluster if tasks can be completed in parallel using only local regions of network topology. We measured throughput for a range of cluster sizes and replication factors.

Tests were first conducted to determine the throughput of a single node cluster without replication overhead. We launch cbench in throughput mode to queue packet_in requests to the controllers, and count flow mod requests as the controller responds. We ran cbench for a varying number of emulated switches sending packet_in requests in parallel. Results are shown in Figure 3.

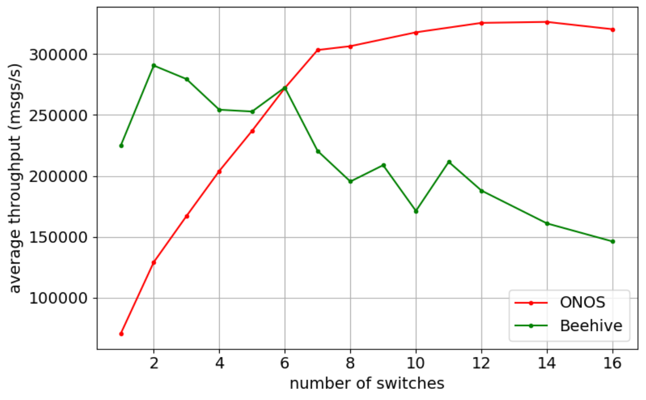

Figure 3: Packet_out throughput using cbench for varying numbers of emulated switches on single-node Beehive and ONOS.

Single-node cbench tests performed by the ONOS team on their latest development branch report throughput of up to 800k messages per second [8]. We believe the difference in this result is due to the CPU of our VM. Our single-node results for Beehive appear to match those reported in [6] at 250k messages per second. For more than 8 emulated switches, CPU usage was capped at its maximum on our GCE virtual machine, which could explain the degradation in Beehive's performance. For ONOS, throughput peaks at 300k messages per second, and performance does not degrade at high switch counts.

Next Beehive was run on a 3-node cluster, with replication factor 1 and 3. As cbench was originally designed as a bench marking tool for centralized controllers, cbench was modified such that the emulated switches are distributed uniformly amongst nodes. Thus for the 12 switch results, each node in a 3-node cluster is the master of 4 switches. While this may not completely describe the characteristics of a real network load, this assumption simplifies our experimental setup.

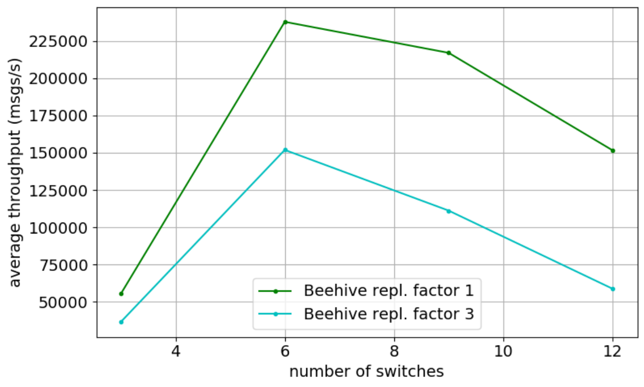

Figure 4: Throughput of 3-node Beehive cluster at replication factor 1 and 3. Cbench switches are divided evenly amongst the nodes

We first observe that there is roughly a 50% decrease in throughput when moving from a replication factor of 1 to 3. This highlights the overhead caused by communication between controllers to maintain cluster membership and state persistence.

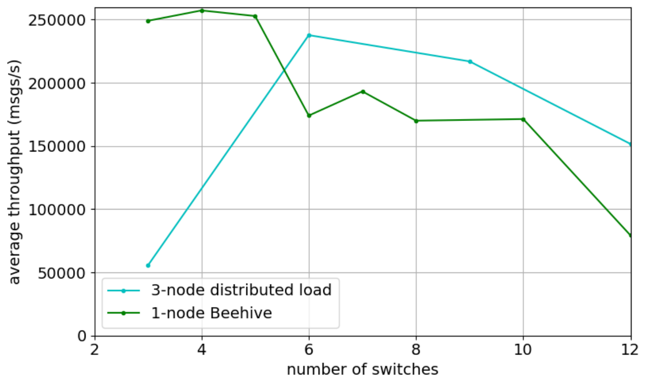

Figure 5: Throughput of 3-node Beehive cluster (with distributed load) compared to a single node instance.

Compared to single-node Beehive, 3-node Beehive performs similarly (Figure 5). Notice that when there are few switches, a single Beehive instance outperforms the 3-node cluster, as the overhead required to maintain the cluster is more than the advantage gained by distributing the network load. The key benefit in the clustered operation is to scale to larger number of switches, and to provide fault-tolerance.

Finally, we analyze the clustered operation of ONOS. Unlike Beehive, ONOS can only be run with replication factor 3 for a 3 node cluster. Additionally, there was difficulty distributing the switches evenly amongst each ONOS cluster instance. Therefore, all cbench switches were targeted on the same controller, as if it were a single centralized controller.

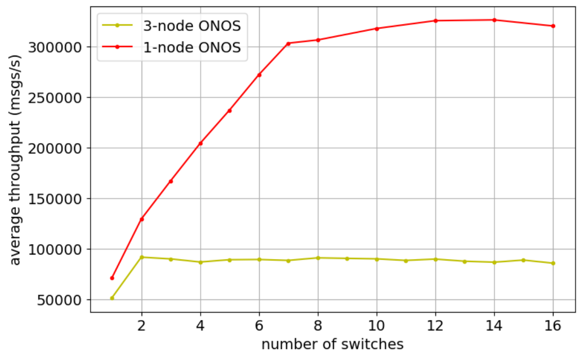

Figure 6: 3-node ONOS cluster versus 1-node ONOS instance with cbench targeting a single node.

As seen in Figure 6, throughput decreases by a factor of 3 compared to the single node ONOS instance. This decrease in throughput is due to the overhead required for state consistency and cluster communication. Additionally, as switching capacity of the controller reaches capacity around 100k messages per second, increasing the number of switches emulated by cbench does not degrade performance. If cbench was distributed uniformly across all controllers in the cluster, we suspect that throughput should increase roughly by a factor of 3. Internal testing by ONOS for loads distributed evenly amongst their cluster report throughput of up to 2.5 million flows per second [9]. However, this is done using a custom testing script and modified southbound providers that bypasses Openflow adapters and device emulation.

Compared to the results described in [6], we see that ONOS has made significant improvements to its clustered throughput. In fact, ONOS appears to have a slight edge over Beehive in terms of single node and clustered throughput. While this is a low-level benchmark, it is still sufficient to showcase the competitiveness between these two controllers and act as motivation to investigate the scalability properties of these controllers even further.

4.3 Fault-tolerance and Failover Time

We have also tested and conducted the first comparison of ONOS and Beehive in terms of fault tolerance. In this section we present our results in measuring the failover time of the two controllers.

For our experiments, we made the controllers only respond to packet_in requests with packet_out messages instead of installing flows. For Beehive, this is achieved by modifying the source code of the learning switch application. For ONOS, the packetOutOnly configuration for its reactive forwarding application was set to be true. These modifications result in communication between a switch and the controller every time the switch has to forward a packet.

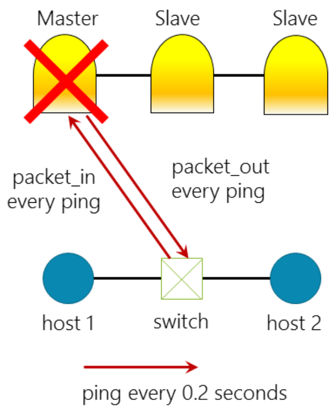

Figure 7: Mininet topology used to test failover times of Beehive and ONOS.

The Mininet topology of our network is two hosts connected to a single switch (see Figure 7). We initiate pinging every 0.2 seconds from one host to the other. After killing the controller's master node (consequently making the pinging fail), we compute the time it takes for the controller to elect a new master node for the switch. The killing of the master node in the case of Beehive is done with the Linux kill command, whereas for ONOS we use its provided “onos-die” script. We note that recovery is done automatically by both Beehive and ONOS without any intervention. For Beehive, Raft was configured to announce a node driver dead after 5 consecutive dropped pings each 100 ms apart. For ONOS, Raft was configured similarly with a 500 ms heartbeat timeout.

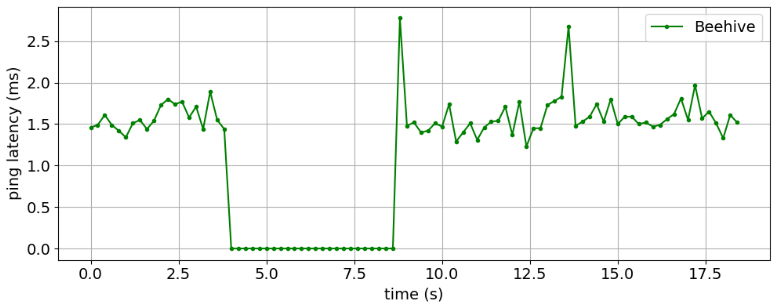

Figure 8: Failover time of 3-node Beehive. At $t = 4.0$ seconds, we crash the master Beehive node. At $t = 9.0$ seconds, Beehive recovers from the failure.

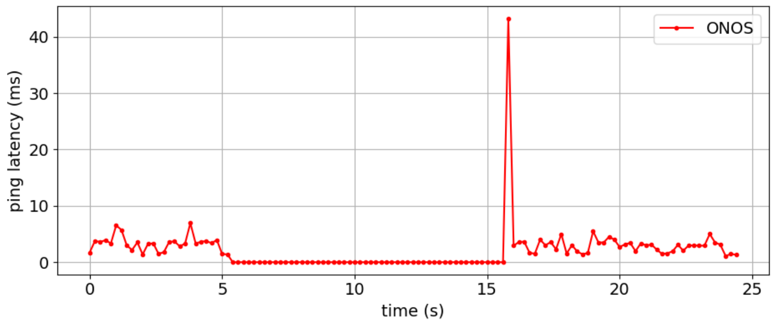

Figure 9: Failover time of 3-node ONOS. At $t = 5.4$ seconds, we crash the master ONOS node. At $t = 16.0$ seconds, ONOS recovers from the failure.

As shown in Figure 8 Beehive has a 5.0 seconds failover time. On the other hand, ONOS 9 has a 10.6 seconds failover time, more than double that of Beehive’s. These results confirm our original hypothesis that even though both controllers are based on the Raft consensus algorithm to recover after a failure, the custom lightweight implementation of Beehive is more efficient than the Atomix framework used by ONOS.

While experiment focuses on controller failure, it doesn’t deal with other aspects of fault tolerance such as failures in the underlying network (such as switch or link failures). Additionally, we have not considered post-failure optimization in distributing processing load among the remaining controllers. While Beehive provides automatic optimization of Bee placement after a failure has occurred, such characteristics are not documented by ONOS. Future work can be done to test this behaviour.

5 Conclusion

We have conducted a comprehensive performance comparison between Beehive and ONOS in terms of latency, throughput, and fault-tolerance. Both controllers had nearly identical latency in our Mininet simulations. Both controllers showed decent throughput during single node tests, although ONOS appears to outperform Beehive for high switch loads. Both controllers demonstrated the ability to scale throughput in 3-node cluster, although differences in experimental setup make it hard to make comparisons. Finally, we saw that Beehive’s failover time was half that of ONOS using its custom implementation of Raft consensus.

These tests show that both SDN controllers are making progress at reaching performance requirements of real-life networks. In particular, we have seen vast improvements made by ONOS since its v1.3 release. The testing methodology we have demonstrated in our report can be used to monitor both ONOS and Beehive as changes are made in the future. Additionally, we hope in future work that our methodology can be extended to other distributed SDN controllers, thus providing a standardized scale for comparison.

References

[1] Beehive Network Controller. Available: `https://github.com/kandoo/beehive-netctrl`.

[2] ONOS. Available: `http://onosproject.org/`.

[3] Atomix. Available: `http://atomix.io/`.

[4] Mininet. Available: `https://github.com/mininet/mininet`.

[5] cbench. Available: `https://github.com/mininet/oflops/tree/master/cbench`.

[6] S. Yeganeh and Y. Ganjali. "Beehive: Simple Distributed Programming in Software-Defined Networks," in *ACM Symposium on SDN Research*, 2016.

[7] P. Berde et al. "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of HotSDN '14*, pages 1-6, 2014.

[8] Master: Experiment G - Single-node ONOS Cbench: `https://wiki.onosproject.org/display/ONOS/Master%3A+Experiment+G+-+Single-node+ONOS+Cbench`

[9] 1.9: Experiment F - Flow Subsystem Burst Throughput: `https://wiki.onosproject.org/display/ONOS/1.9%3A+Experiment+F+-+Flow+Subsystem+Burst+Throughput`