Beehive: Simple Distributed Programming in Software-Defined Networks*

Soheil Hassas Yeganeh†
Google Inc., University of Toronto
soheil@{google.com,cs.toronto.edu}

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

Abstract

In this paper, we present the design and implementation of Beehive, a distributed control platform with a simple programming model. In Beehive, control applications are centralized asynchronous message handlers that optionally store their state in dictionaries. Beehive's control platform automatically infers the keys required to process a message, and guarantees that each key is only handled by one light-weight thread of execution (*i.e.*, bee) among all controllers (*i.e.*, hives) in the platform. With that, Beehive transforms a centralized application into a distributed system, while preserving the application's intended behavior. Beehive replicates the dictionaries of control applications consistently through mini-quorums (*i.e.*, colonies), instruments applications at runtime, and dynamically changes the placement of control applications (*i.e.*, live migrates bees) to optimize the control plane. Our implementation of Beehive is open source, high-throughput and capable of fast failovers. We have implemented an SDN controller on top of Beehive that can handle 200K of OpenFlow messages per machine, while persisting and replicating the state of control applications. We also demonstrate that, not only can Beehive tolerate faults, but also it is capable of optimizing control applications after a failure or a change in the workload.

Categories and Subject Descriptors

C.2 [**Computer-communication networks**]: Network Architecture and Design

Keywords

Software-Defined Networking; Distributed Control Platforms; Programming Abstraction; OpenFlow

*A preliminary version published in HotNets'14 [17].

†Contributed to this paper, while he was a PhD student at the University of Toronto

SOSR'16, March 14–15, 2016, Santa Clara, CA.

DOI: http://dx.doi.org/10.1145/2890955.2890958.

1. INTRODUCTION

Distributed control platforms are employed in practice for the obvious reasons of scale and resilience [22, 18]. Existing distributed controllers are designed for scalability and availability, yet they expose the complexities and boilerplates of realizing a control application to network programmers [26]. Designing a distributed control application comprises significant efforts beyond the design and implementation of the application. In such a setting, one needs to address consistency, concurrency, coordination, and other common complications of distributed programming that are pushed into control applications in existing distributed control platforms.

Beehive. Our motivation is to develop a distributed control plane that is scalable, efficient, and yet straightforward to program. Our proposal is composed of a *programming model*[1] and a *control platform*. Using Beehive's programming model, control applications are developed as a collection of message handlers that store their state in dictionaries (*i.e.*, hash-maps), intentionally similar to centralized controllers to preserve simplicity. Beehive's distributed control platform is the runtime of our programming model. Exploiting the proposed programming model, it seamlessly transforms a centralized application into its distributed counterpart. Beehive achieves this by automatically partitioning the application's logic. Application partitioning is a well-studied scalability mechanism [19, 31, 25, 12]. Our goal is to show how Beehive uses partitioning to provide scalability and address issues related to such a design.

Using its programming model, Beehive is able to provide fault-tolerance, instrumentation, and automatic optimization. To isolate local faults, our platform employs application-level transactions with an "all-or-nothing" semantic. Moreover, it employs mini-quorums to replicate application dictionaries and to tolerate failure. This is all seamless and without manual intervention. Beehive also provides efficient runtime instrumentation (including but not limited to resource consumption, and the volume of messages exchanged) for control applications. The instrumentation data can be used to (*i*) automatically optimize the control plane, and (*ii*) to provide feedback to network programmers. In this paper, we showcase a greedy heuristic as a proof of concept for optimizing the placement of control applications, and provide examples of Beehive's runtime analytics for control applications.

[1]By programming model, we mean a set of APIs to develop control applications using a general purpose programming language, not a new domain-specific language (DSL).

Why not an external datastore? Emerging control platforms, most notably ONOS [7], try to resolve controller complexities by delegating their state management to an external system (*e.g.*, replicated or distributed databases). In Beehive, we propose an alternative design, since delegating such functionality has important drawbacks.

It has been shown that, for SDN, using an external datastore incurs a considerable latency overhead [24]. After all, embedding the state inside the controllers (as pioneered by ONIX [22]) has significantly lower overheads than using an external system. Further, creating a correct distributed control function requires coordination and synchronization among control applications. This is essentially beyond state management as demonstrated in [32, 10]. For example, even if two instances of a routing application store their state in a distributed store, they need to coordinate their logics in order to avoid blackholes and loops. Implementing such coordination mechanisms eliminates many of the original motivations of opting for an external datastore.

More importantly, in such a setting, it is the external data store that *dictates* the placement and the topology of control functions. For instance, if the external data store is built as a ring, the control plane cannot form an effective hierarchy, and vice versa. This lack of fine-grained control over the placement of state also limits the capabilities of the control plane to self-optimize, which is one of the goals of Beehive and has shown to be crucial for SDN [14]. For instance, consider a controller with a virtual networking application that is managing a virtual network. If the virtual network migrates, the control plane needs to migrate the state in accordance to the change in the workload. This is difficult to achieve without fine-grained control. Similarly, providing Beehive's runtime instrumentation, analytics, and automatic optimization are quite difficult to realize when using an external datastore.

Overview. Our implementation of Beehive is publicly available on [2]. Using Beehive, we have implemented a full featured SDN controller [1] as well as important sample applications such as Kandoo [16], a distributed key-value store, a message queue, traffic engineering, graph processing, and network virtualization. With systematic optimizations, our implementation can failover in less than 200ms, and our SDN controller can handle more than 200K OpenFlow messages per machine while persisting and replicating the state of control applications. Interestingly, existing controllers that use an external datastore can handle up to a thousand messages with persistence and replication. We also demonstrate that our runtime instrumentation has negligible overheads. Moreover, using this runtime instrumentation data, we show that Beehive is able to automatically reduce the latency of a sample forwarding logic when switches migrate or when there is a failure.

2. PROGRAMMING MODEL

In Beehive, programs are written in a general purpose language (Go in our implementation) using a specific set of APIs called the Beehive programming model. Creating the distributed version of an arbitrary control application that stores and shares its state in free and subtle forms is extremely difficult in general. The Beehive programming model enables us to do that, while imposing minimal limitations on the applications.

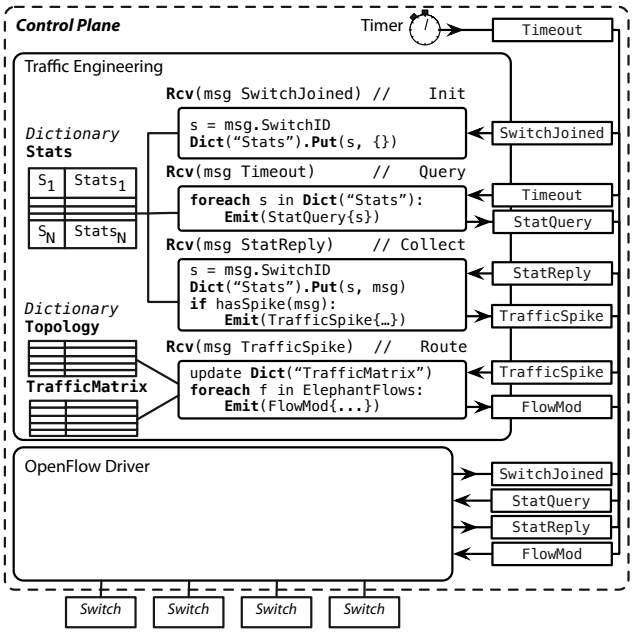

Figure 1: A simple traffic engineering application that periodically collects stats, and accordingly reroutes flows.

The Anatomy of an Application. Using Beehive's programming, a control application is implemented as a set of single-threaded message handlers (denoted by `rcv`) that are triggered by asynchronous messages and can emit further messages to communicate with other applications. In our design, messages are either directly emitted, or generated as a reply to another message. The former is processed by all applications that have a handler for that particular type of message. The reply messages, on the other hand, is processed only at its destination application instance (*i.e.*, in the destination bee as we will explain in Section 3). For example, an OpenFlow driver emits a packet-in message when it receives the event from the switch. That message is processed in all applications, say a learning switch and a discovery application, that have a handler for packet-in messages. The learning switch application, on the other hand, replies to the packet-in message with a flow-mod, and the flow-mod is directly delivered to the OpenFlow driver, not any other application.

To preserve state, message handlers use application-defined dictionaries. As explained in Section 3, these dictionaries are transactional, replicated, and persistent behind the scenes (*i.e.*, hidden from control applications, and without programmer's intervention). Note that message handlers are arbitrary programs written in a general purpose programming language, with the limitation that any data stored outside the dictionaries is ephemeral. Next, we demonstrate this programming model for a simple traffic engineering example.

Example: Traffic Engineering. As shown in Figure 1, a Traffic Engineering (TE) can be modeled as four message handlers at its simplest: (*i*) `Init`, which handles `SwitchJoined` messages emitted by the OpenFlow driver and initializes the flow statistics for switches. (*ii*) `Query`, which handles periodic timeouts and queries switches. (*iii*) `Collect`, which handles `StatReplies` (*i.e.*, replies

to stat queries emitted by the OpenFlow driver), populates the time-series of flow statistics in the `Stats` dictionary, and detects traffic spikes. (*iv*) `Route`, which re-steers flows using `FlowMod` messages upon detecting an elephant flow. `Init`, `Query`, and `Collect` share the `Stats` dictionary to maintain the flow stats, and `Route` stores an estimated traffic matrix in the `TrafficMatrix` dictionary. It also maintains its topology data, which we omitted for brevity.

Inter-Dependencies. Message handlers can depend on one another in two ways: (*i*) they can either exchange messages, or (*ii*) access an application-defined dictionary, only if they belong to the same application. The interplay of these two mechanisms enables the control applications to mix different levels of state synchronization and hence different levels of scalability. As explained shortly, communication using the shared state results in synchronization in the application in contrast to asynchronous messages.

In our sample TE application, `Init`, `Collect`, and `Query` share the `Stats` dictionary, and communicate with `Route` using `TrafficSpike` messages. Moreover, `Init`, `Collect`, `Query`, and `Route` depend on the OpenFlow driver that emits `SwitchJoined` and `StatReply` messages and can process `StatQuery`s and `FlowMod`s.

Message handlers of two different applications can communicate using messages, but cannot access each other's dictionaries. This is not a limitation, but a programming paradigm that fosters scalability (as advocated in Haskell, Erlang, Go, and Scala). Dependencies based on events result in a better decoupling of functions, and hence can give Beehive the freedom to optimize the placement of applications. That said, due to the nature of stateful control applications, functions inside an application can share state. It is important to note that Beehive provides utilities to emulate synchronous and asynchronous cross-application calls on top of asynchronous messages.

Consistent Concurrency. Transforming a "centralized looking" application to a concurrent, distributed system is trivial for stateless applications: one can replicate all message handlers on all cores on all machines. In contrast, this process can be challenging when we deal with *stateful* applications. To realize a valid, concurrent version of a control application, we need to make sure that all message handlers, when distributed on several controllers, have a consistent view of their state and their behavior is identical to when they are run as a single-threaded, centralized application.

To preserve state consistency, we need to ensure that each key in each dictionary is exclusively accessed in only one thread of execution in the distributed control plane. In our TE example (Figure 1), `Init`, `Query` and `Collect` access the `Stat` dictionary on a per switch basis. As such, if we process all the messages of a given switch in the same thread, this thread will have a consistent view of the statistics of that switch. In contrast, `Route` accesses the whole `Topology` and `TrafficMatrix` dictionaries and, to remain consistent, we have to make sure that all the keys in those dictionaries are exclusively owned by a single thread. Such a message handler is essentially centralized.

3. CONTROL PLATFORM

Beehive's control platform acts as the runtime environment for the proposed programming model. This control

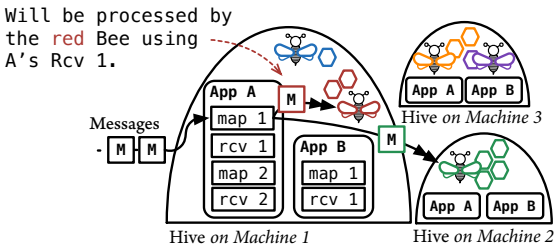

Figure 2: In Beehive, each hive (*i.e.*, controller) maintains the application state in the form of cells (*i.e.*, key-value pairs). Cells that must be colocated are exclusively owned by one bee. Messages mapped to the same cell(s) are processed by the same bee.

platform is built based on the simple primitives of *hive*, *cells*, and *bees* for concurrent and consistent state access in a distributed fashion. Moreover, our control platform provides runtime instrumentation and optimization of distributed applications. We have implemented Beehive in Go. Our implementation is open source and available on [2] as a generic, distributed message passing platform with functionalities such as queuing, parallelism, replication, and synchronization.

3.1 Primitives

Hives and Cells. In Beehive, a controller is denoted as a *hive* that maintains applications' state in the form of *cells*. Each cell is a key-value pair in a specific application dictionary: $(\texttt{dict}, \texttt{key}, \texttt{val})$. For instance, our TE application (Figure 1) has a cell for the flow statistics of switch `S` in the form of $(\texttt{Stats}, \texttt{S}, \texttt{Stats}_\texttt{S})$.

Bees. For each set of cells that must be collocated to preserve consistency (as discussed in Section 2), we create an exclusive, logical and light-weight thread of execution, called a *bee*. As shown in Figure 2, upon receiving a message, a hive finds the particular cells required to process that message in a given application and, consequently, relays the message to the bee that exclusively owns those cells. A bee, in response to a message, invokes the respective message handlers using its cells as the application's state. If there was not such a bee, the hive creates one to handle the message.

In our TE example, once the hive received the first `SwitchJoined` for switch `S`, it creates a bee that exclusively owns the cell $(\texttt{Stats}, \texttt{S}, \texttt{Stats}_\texttt{S})$, since the **`rcv`** function (*i.e.*, `Init`) requires that cell to process the `SwitchJoined` message. Then, the respective bee invokes the **`rcv`** function for that message. Similarly, the same bee handles consequent `StatReply`s for `S` since it exclusively owns $(\texttt{Stats}, \texttt{S}, \texttt{Stats}_\texttt{S})$. This ensures that `Collect` and `Init` share the same consistent view of the `Stats` dictionary and their behavior is identical to when they are deployed on a centralized controller, even though the cells of different switches might be physically distributed over different hives.

Finding the Bee. The precursor to finding the bee to relay a message is to infer the cells required to process that message. In Beehive's control platform, each message handler is accompanied with a **`map`** function that is either (*i*) automatically generated by the platform or (*ii*) explicitly implemented by the programmer.

`map`$(\texttt{A}, \texttt{M})$ is a function of application `A` that maps a message of type `M` to a set of cells (*i.e.*, keys in application

dictionaries $\{(D_1, K_1), \ldots, (D_n, K_n)\}$). We call this set, the *mapped cells* of `M` in `A`. We note that, although we have not used a stateful **`map`** function, a **`map`** function can be *stateful* with the limitation that its state is local to each hive.

We envision two special cases for the mapped cells: (*i*) if `nil`, the message is dropped for application `A`. (*ii*) if empty, the message is broadcasted to all bees on the local hive. The latter is particularly useful for simple iterations over all keys in a dictionary (such as `Query` in our TE example).

Automatically Generated `map` Functions. Our control platform is capable of generating the **`map`** function at both compile-time and runtime. Our compiler automatically generates the missing **`map`** function based on the keys used by application functions to process the message. The compiler parses the application functions and generates the code to retrieve respective keys to build the intended mapped cells. For example, as shown in Figure 3, the compiler generates the **`map`** functions based on the code in `Collect`, `Query` and `Init`. Note that these cells are application-specific and logical.

```
  // Generated based on Init
1 map(msg SwitchJoined):
2 └ return {("Stats", msg.SwitchID)}
  // Generated based on Query
3 map(msg Timeout):
4 └ return {} // Local broadcast.
  // Generated based on Collect
5 map(msg StatReply):
6 └ return {("Stats", msg.SwitchID)}
7 ...
```

Figure 3: Automatically generated `map` functions based on the code in Figure 1.

Beehive also provides a generic runtime **`map`** function that invokes the **`rcv`** function for the message, and records all state accesses in that invocation without actually applying the modifications.

Life of a Message. In Beehive, a message is basically an envelope wrapping a piece of data, that is emitted by a bee and optionally has a specific destination bee. As shown in Figure 4, a message has the following life-cycle in our platform:

1. A bee emits the message upon receiving an event (such as an IO event, or a timeout) or replies to a message that it is handling in a **`rcv`** function. In the later case, the message will have a particular destination bee.
2. If the emitted message has a destination bee, it is directly relayed to that bee. Otherwise, on the same hive, we pass the message to the **`map`** functions of all applications that are triggered by that type of message. This is done asynchronously via *queen bees*. For each application on each hive, we allocate a single queen bee that is responsible to map messages using the **`map`** function of its application. Queen bees are in essence Beehive's message routers.
3. For each application, the **`map`** function maps the message to application cells. Note that hives are homogeneous in the sense that they all host the same set of applications, even though they process different messages and their bees and cells are not identical at runtime.
4. After finding the mapped cell of a message, the queen bee tries to find the bee that exclusively owns those cells. If there is such a bee (either on the same hive or on a remote hive), the message is automatically relayed. Otherwise, the queen bee creates a new bee[2], assigns the cells to it, and relays the message. To assign cells to a bee, we use the Raft consensus algorithm [29] among hives. Paying one Raft consensus overhead to create a bee can be quite costly. To avoid that, queen bees batch messages which results in paying one Raft overhead for each batch. As shown in Figure 5, the average latency of creating a new bee can be as low as 140μs even when we use a cluster of 11 hives. Clearly, the larger the batch size, the better the throughput.
5. For each application, the respective bee processes the message by invoking the **`rcv`** function.
6. If the **`rcv`** function function replies to the message, the reply message is directly sent to the bee that has emitted the received message (*i.e.*, emitted in Step 1), and if it emits a message, the message will go through Step 1.

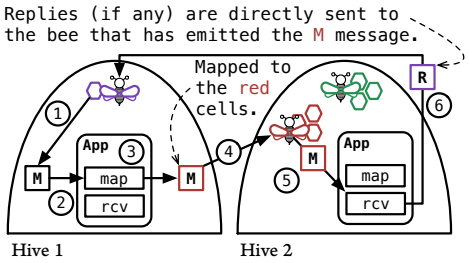

Figure 4: Life of a message in Beehive.

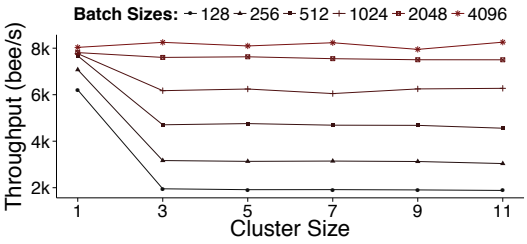

Figure 5: Throughput of a queen bee for creating new bees, using different batch sizes.

Preserving Consistency Using `map`. To preserve consistency, Beehive guarantees that all messages with *intersecting* mapped cells for application `A` are processed by the same bee using `A`'s message handlers. For instance, consider two messages that are mapped to $\{(\texttt{Switch}, 1), (\texttt{MAC}, \texttt{A1}{:}\ldots)\}$ and $\{(\texttt{Switch}, 1), (\texttt{Port}, 2)\}$ respectively by an application. Since these two messages share the cell $(\texttt{Switch}, 1)$, the platform guarantees that both messages are handled by the same bee that owns the 3 cells of $\{(\texttt{Switch}, 1), (\texttt{MAC}, \texttt{A1}{:}\ldots), (\texttt{Port}, 2)\}$. This way, the platform prevents two different bees from modifying or reading the same part of the state. As a result, each bee is identical to a centralized, single-threaded application for its own cells.

Significance of `map`. The exact nature of the mapping process (*i.e.*, how the state is accessed) can impact Beehive's

[2]Although by default the new bee is created on the local hive and moved to an optimal hive later, the application can provide a custom placement strategy when the default local placement is not a good fit for that application's design.

performance. Even though Beehive cannot automatically redesign functions (say, by breaking a specific function in two) or change `map` patterns for a given implementation, it can help developers in that area by providing feedback that includes various performance metrics, as we will explain in Section 3.4.

Auxiliaries. In addition to the core functions presented here, Beehive provides further auxiliary utilities for proactively locking a cell in a handler, for postponing a message, for composing applications, and for asynchronous and synchronous cross-application calls. We skip these auxiliaries for brevity.

3.2 Fault-Tolerance

For some simple applications, having a consistent replication of the state is not a requirement nor is transactional behavior. For example, a hub or learning switch can simply restart functioning with an empty state. For most stateful applications, however, we need a proper replication mechanism for cells.

Transactions. In Beehive, `rcv` functions are transactional. Beehive transactions include the messages emitted in a function in addition to the operations on the dictionaries. Upon calling the `rcv` function for a message, we automatically start a transaction. This transaction buffers all the emitted messages along with all the updates on application dictionaries. If there was an error in the message handler, the transaction will be automatically rolled back. Otherwise, the transaction will be committed and all the messages will be emitted in the platform. We will shortly explain how Beehive batches transactions to achieve high throughput, and how it replicates these transactions for fault-tolerance.

It is important to note that Beehive, by default, hides all the aspects of transaction management. With that, control applications are implemented as if there is no transaction. Having said that, network programmers have full control over transactions, and can begin, commit, or abort a new transaction if they choose to.

Colonies. As shown in Figure 6, to replicate the transactions of a bee, the platform automatically forms a *colony*. Each colony has one leader bee and a few follower bees in accordance to the application's *replication factor*.[3] In a colony, only the leader bee receives and processes messages, and followers act as purely passive replicas.

Replicating Transactions. Bees inside the colony form a Raft [29] quorum to replicate the transactions. Note that the platform enforces that the leader of the Raft quorum is always the leader bee. Before committing a transaction, the leader first replicates the transaction on the majority of its followers. Consequently, the colony has a consistent state and can tolerate faults as long as the majority of its bees ($\frac{1}{2}$ `replication_factor` $+ 1$) are alive.

Batched Transactions. With a naive implementation, replicating each transaction incurs the overhead of a quorum for each message. This hinders performance and can result in subpar throughput. To overcome this issue, we batch transactions in Beehive and replicate each batch once. With that, the overheads of replicating a transaction are amortized over all the messages processed in that batch.

[3]Applications can choose to skip replication and persistence.

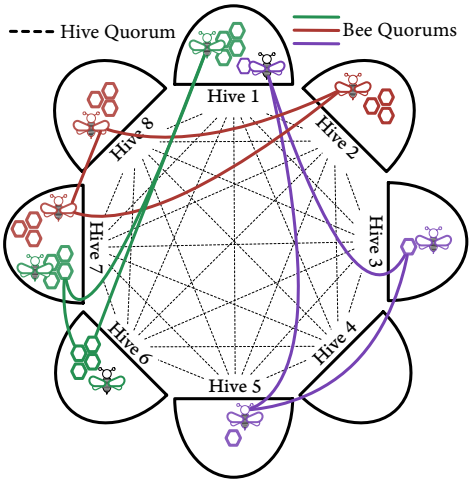

Figure 6: Hives form a quorum on the assignment of cells to bees. Bees, on the other hand, form quorums (*i.e.*, colonies) to replicate their respective cells in a consistent manner. The size of a bee colony depends on the replication factor of its application. Here, all applications have a replication factor of 3.

To evaluate the impact of batching on Beehive's throughput, we have implemented a simple key-value store conceptually similar to memcached [15] in less than 150 lines of code. To measure its throughput, we deployed our sample key-value store application on a 5-node cluster on Google Compute Engine. Each VM has 4 cores, 16GB of RAM, 10GB of SSD storage.

We configured the application to use 1024 buckets for the keys with replication factors of 1, 3 and 5, and measured the throughput for batch sizes from 128 to 8192 messages. For each run, we emit 10M PUT (*i.e.*, write) messages and measure the throughput. As shown in Figure 7a, using larger batch sizes significantly improves the throughput of Beehive applications. This improvement is more pronounced (3 to 4 orders of magnitude) when we use replication factors of 3 and 5, clearly because the overheads are amortized. We note that here, for measurement purposes, all the master bees are on the same hive. In real settings, one observes a higher aggregate throughput from all active hives.

Batching vs Latency. In our throughput micro-benchmarks, we send requests to our application in batches. Although that is the case in practice for most applications, one might ask how the platform performs when the requests are sent once the previous request is retrieved. To answer this question, we used another 20-node cluster to query the key-value store sample application. Each node has 4 clients that send PUT or GET requests over HTTP and then measure the response time (including the HTTP overheads).

As shown in Figure 7b, despite the lack of effective batching for requests, Beehive handles most GETs in less than 1ms if the key is on the local hive, and less than 7ms if the key is on a remote hive. The reason for such a low read latency is that there is no need to replicate read-only transactions. Moreover, it handles most PUTs in less than 20ms. This indicates that, even in such a worst-case scenario, Beehive handles requests with a reasonable latency. As demonstrated above, in the presence of a batch of messages, Beehive achieves a significantly better latency on average.

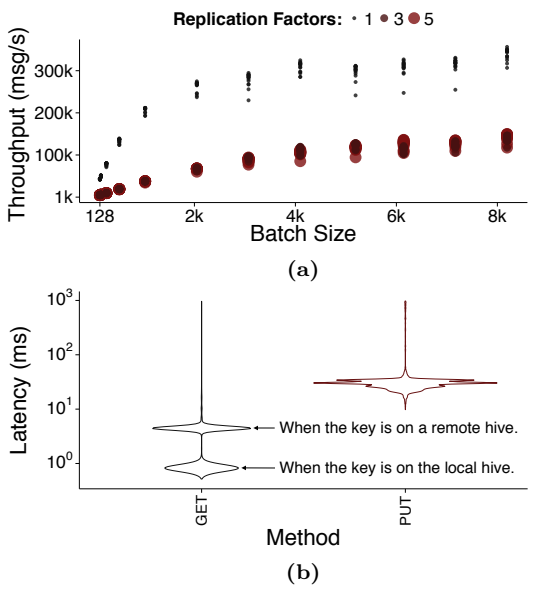

Figure 7: Latency and throughput of the sample key-value store application: (a) Throughput using replication factors of 1 to 5. (b) The PDF of the end-to-end latency for GET and PUT requests in a cluster of 20 hives and 80 clients.

Elasticity. Beehive's colonies are elastic. When a follower fails or when a hive leaves the cluster, the leader of the colony creates new followers on live hives if possible. Moreover, if a new hive joins the cluster and the colony is not fully formed, the leader automatically creates new followers. Note that forming a consistent colony is only possible when the majority of hives are alive, since any update to a colony requires a consensus among hives.

Failover. When the leader bee fails, one of the follower bees will become the leader of the Raft quorum. Once the new leader is elected, the quorum updates its transaction logs and any uncommitted (but sufficiently replicated) transaction will be committed (*i.e.*, the state updates are applied and buffered messages are emitted). Note that the failover timeout (*i.e.*, the timeout to start a new election) is configurable by the user.

To demonstrate the failover properties of Beehive, we ran the key-value store sample application with a replication factor of 3 on a 3-node cluster. In this experiment, we allocate the master bee on Hive 1, and send the benchmark queries to Hive 2. After ~5 seconds, we kill Hive 1, and measure the time the system takes to failover. As shown in Figure 8, it takes 198ms, 451ms, and 753ms for the sample key value store application to failover respectively for Raft election timeouts of 100ms, 300ms, and 500ms. Clearly, the smaller the election timeout, the shorter the failover. Having said that, users should choose a Raft election timeout that does not result in spurious timeouts based on the characteristics of their network.

Network Partitions. In the event of a network partition, by default, the platform will be fully operational in the partition with the majority of nodes to prevent split-brain. For colonies, however, they will be operational as long as the majority of bees are in the same partition. Moreover, if a leader bee falls into a minority partition, it can

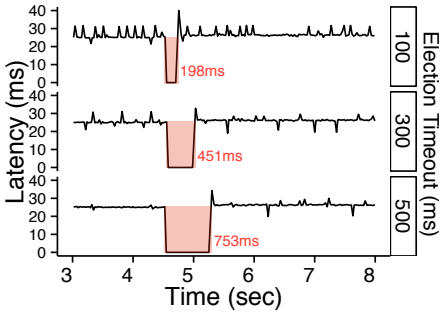

Figure 8: Beehive failover characteristics using different Raft election timeouts. Smaller election timeouts result in faster the failover, but they must be chosen to avoid spurious timeouts based on the network characteristics.

serve messages with a read-only access on its state. For example, when a LAN is disconnected from other parts of the network, the bees managing switches in that LAN will be able to keep the existing switches operational until the LAN reconnects to the network. To achieve partition tolerance, application dependencies and their placement should be properly designed. Headless dependencies can result in subpar partition tolerance, similar to any multi-application platform. To mitigate partitions, one can deploy one cluster of Beehive in each availability zone and connect them through easy-to-use proxy functionalities that Beehive provides. For brevity, we omit the discussion of those functionalities and consider them beyond the scope of this paper.

3.3 Live Migration of Bees

We provide the functionalities to migrate a bee from one hive to another along with its cells. This is instrumental in optimizing the placement of bees.

Migration without replication. For bees that are not replicated, Beehive first stops the bee, buffers all incoming messages, and then replicates its cells to the destination hive. Then a new bee is created on the destination hive to own the migrated cells. At the end, the cells are assigned to the new bee and all buffered messages are accordingly drained.

Migration with replication. We use a slightly different method for bees that are replicated. For such leader bees, the platform first checks whether the colony has a follower on the destination hive. If there is such a follower, the platform simply stops the current leader from processing further messages while asking the follower to sync all the replicated transactions. Afterwards, the leader stops, and the follower explicitly campaigns to become the new leader of its colony. After one Raft election timeout, the previous leader restarts. If the colony has a new leader, the leader would become a follower. Otherwise, this is an indication of a migration failure, and the colony will continue as it was. In cases where the colony has no follower on the destination hive, the platform creates a new follower on the destination hive, and performs the same process.

3.4 Runtime Instrumentation

Control functions that access a minimal state would naturally result in a well-balanced load on all controllers in the control platform since such applications handle events

locally in small silos. In practice, however, control functions depend on each other in subtle ways. This makes it difficult for network programmers to detect and revise design bottlenecks.

Sometimes, even with apt designs, suboptimalities incur because of changes in the workload. For example, if a virtual network is migrated from one data center to another, the control functions managing that virtual network should also be placed in the new data center to minimize latency.

There is clearly no effective way to define a concrete offline method to optimize the control plane placement. For that reason, we rely on runtime instrumentation of control applications. This is feasible in our platform, since we have a well-defined abstraction for control functions, their state, and respective messages. Beehive's runtime instrumentation tries to answer the following questions: (*i*) how are the messages balanced among hives and bees? (*ii*) which bees are overloaded? and, (*iii*) which bees exchange the most messages?

Metrics. Our runtime instrumentation system measures the resource consumption of each bee along with the number of messages it exchanges with other bees. For instance, we measure the number of messages that are exchanged between an OpenFlow driver accessing the state of a switch and a virtual networking application accessing the state of a particular virtual network. This metric essentially indicates the correlation of each switch to each virtual network. We also store provenance and causation data for messages. For example, in a learning switch application, we can report that most `packet_out` messages are emitted by the learning switch application upon receiving `packet_in`'s.

Design. We measure runtime metrics on each hive locally. This instrumentation data is further used to find the optimal placement of bees and also utilized for application analytics. We implemented this mechanism as a control application using Beehive's programming model. That is, each bee emits its metrics in the form of a message. Then, our application locally aggregates those messages and sends them to a centralized application that optimizes the placement of the bees. The optimizer is designed to avoid complications such as conflicts in bee migrations.

Overheads of Runtime Instrumentation. One of the common concerns about runtime instrumentation is its performance overheads. To measure the effects of runtime instrumentation, we have run the key-value store sample application for a batch size of 8192 messages and a replication factor of 1, with and without runtime instrumentation. As shown in Figure 9, Beehive's runtime instrumentation imposes a very low overhead on the bees as long as enough resources are provisioned for runtime instrumentation. This is because instrumentation data is processed asynchronously, and does not interfere with the critical path in message processing.

3.5 Automatic Placement

By default, a hive which receives the first message for a specific cell successfully acquires the lock and creates a new bee for that cell. At this point, all subsequent messages for the same mapped cells, will be processed on the new bee on this hive. This default behavior can perform well when most messages have locality (*i.e.*, sources are in close vicinity), which is the case for most networks [6]. Having said that,

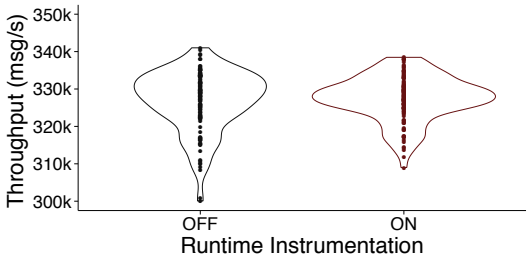

Figure 9: The PDF of a bee's throughput in the sample key-value application, with and without runtime instrumentation.

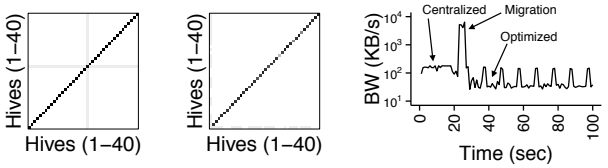

Figure 10: Inter-hive traffic matrix of TE (a) with an ideal placement, and (b) when the placement is automatically optimized. (c) Bandwidth usage of the control plane in (b).

there is a possibility of workload migration. In such cases, Beehive needs to self-optimize via optimizing the placement of bees.

Finding the optimum placement of bees is NP-Hard (by reducing the facility location problem) and we consider it outside the scope of this paper. Here, we employ a greedy heuristic aiming at processing messages close to their source. That is, our greedy optimizer migrates bees among hives aiming at minimizing the volume of inter-hive messages. We stress that optimality is not our goal and this heuristic is presented as a proof of concept.

It is important to note that, in Beehive, bees that perform IO cannot be migrated for obvious reasons. As a result, our greedy heuristic will move other bees close to the IO bees if they exchange a lot of messages. For instance, as we demonstrate in Section 4, bees of a forwarding application will be migrated next to the bee of an OpenFlow driver, which improves latency. We also note that, using our platform, it is straightforward to implement other optimization strategies, which we leave to future work.

Effectiveness of Automatic Placement. To demonstrate the effectiveness of our greedy placement algorithm, we have run an experiment using our TE example shown in Figure 1 and measured the traffic matrix between hives which is readily provided by Beehive. We have simulated a cluster of 40 controllers and 400 switches in a simple tree topology. We initiate 100 fixed-rate flows from each switch, and instrument the TE application. 10% of these flows have spikes (*i.e.*, detected as an elephant flow in `Collect`).

In the ideal scenario shown in Figure 10a, the switches are locally polled and all messages are processed on the same hives except the `TrafficSpike` messages that are centrally processed by `Route`.

To demonstrate how Beehive can dynamically optimize the control plane, we artificially assign the cells of all switches to the bees on the first hive. Once our run-

time instrumentation collects enough data about the futile communications over the control channels, it starts to migrate the bee invoking `Collect` and `Query` to the hive connected to each switch. In particular, it migrates the cell $(\texttt{S}, \texttt{SW}_\texttt{i}, \texttt{Stat}_{\texttt{SW}_\texttt{i}})$ next to the OpenFlow driver that controls $\texttt{SW}_\texttt{i}$. As shown in Figures 10b and 10c, this live migration of bees localizes message processing and results in a placement similar to Figure 10a. Note that this is all done automatically by Beehive at runtime with no manual intervention.

Application-Defined Placement. In Beehive, applications can define their own explicit, customized placement method. An application-defined placement method can be defined as a function that chooses a hive among live hives for a set of mapped cells: `Place(cells, live_hives)`. This method is called whenever the cells are not already assigned to a bee. Using this feature, applications can simply employ placement strategies such as random, consistent hashing, and resource-based placements (*e.g.*, delegate to another hive if the memory usage is more than 70%) to name a few. It is important to note that, if none of these solution works for a particular use-case, clearly, one can implement a centralized load balancer application and a distributed worker application.

4. SDN CONTROL PLATFORM

There are numerous alternative designs to realize a distributed SDN controller. Kandoo [16] and xBar [26] propose hierarchies, ONIX [22] uses an eventually consistent flat model, and Pratyaastha [24] uses partitioning to scale. Beehive is able to support these well-known abstractions: Using logically hierarchical keys in application dictionaries (*e.g.*, x for the parent and xy and xz for its children), one can create hierarchies; using asynchronous messages and local mapped sets, one can create eventual consistency; and partitions are seamlessly inferred by Beehive.

More importantly, Beehive can accommodate applications of different designs in the same place. That is, using Beehive, one can implement a control plane that manages forwarding elements in partitions, gathers statistics in hierarchies, and centrally optimizes traffic. This is very beneficial since network programmers can choose the apt abstraction of each functionality instead of using the same setting for all applications. Exploiting this property we have designed an SDN control platform on top of Beehive to showcase the effectiveness of our proposal. As we explain shortly, our design uses a mixture of sharding, local applications, hierarchies, and centralization to control the network.

SDN Controller Suite. We have designed and implemented an SDN abstraction on top of Beehive. Our implementation is open-source [1]. This abstraction can act as a general-purpose, distributed SDN controller as-is, or can be reused by the community to create other networking systems. Our abstraction is formed based on simple and familiar SDN constructs and is implemented as shown in Figure 11.

Network object model (NOM) is the core of our SDN abstraction that represents the network in the form of a graph (similar to ONIX NIB [22]): nodes, links, ports, and flow entries in a southbound protocol agnostic manner. We skip the definition for nodes, ports, links, and flow entries as they are generic and well-known.

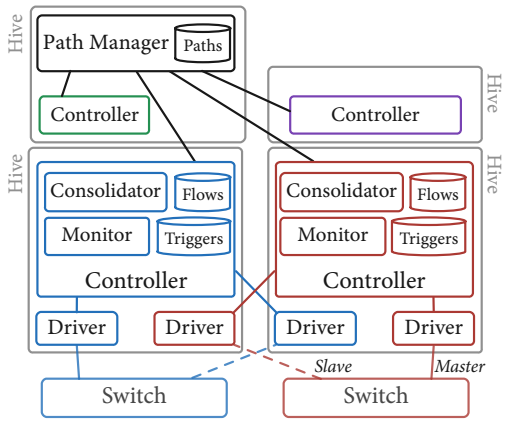

Figure 11: Beehive's SDN controller suite

In addition to these common and well-known graph constructs, Beehive's SDN abstraction supports *triggers* and *paths* that will simplify network programming at scale. Triggers are basically active thresholds on bandwidth consumption and duration of flow entries. They are installed by network applications and pushed close to their respective switches by the node controller (explained shortly). Triggers are scalable alternatives to polling flow statistics from switches directly from different control applications. A path is an atomic, transactional end-to-end forwarding rule that is automatically compiled into individual flows across switches in the network by the Path Manager. Using paths, control applications can rely on the platform to manage the details of routing for them.

Node Controller. In our design, node controllers are the active components that manage the NOM objects. To make them protocol agnostic, we have drivers that act as southbound mediators. That is, drivers are responsible to map physical networking entities into NOM logical entities. The node controller is responsible to manage those drivers (*e.g.*, elect a master). The node controller stores its data on a per switch basis and is automatically distributed and placed close to switches.

Node controller is responsible for settings at node level. At a higher level, we have path managers that install logical paths on network nodes. We have implemented the path manager as a centralized application, since it requires a global view of the network to install proper flow-entries. Note that all the expensive functionalities are delegated to the node controller which is distributed and close to switches.

Consolidator. There is always a possibility of discrepancy between the state of a switch and the state of a node controller: the switch may remove a flow right when the master driver fails, or a switch may restart while the flow is being installed by the driver. The consolidator is a poller removing any flow that is not in the controller's state, and notifies the controller if the switch has removed any previously installed flows.

Monitor. Monitor is another poller in the controller suite that implements triggers. For each node, the monitor periodically sends flow stat queries to find out whether any trigger should fire. The monitor is distributed on a per-node basis and, in most cases, is placed next to the node's controller. This reduces control channels overheads.

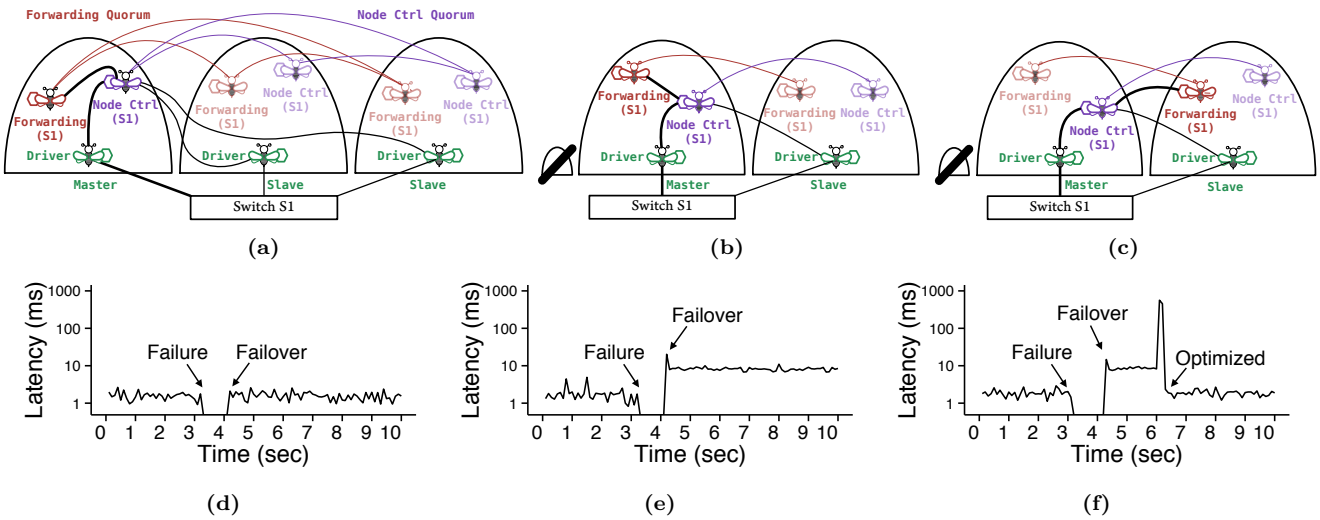

Figure 12: Fault-tolerance characteristics of the control suite using a Raft election of 300ms: (a) The default placement. (b) When the newly elected masters are on the same hive. (c) When the new masters are on different hives. (d) Reactive forwarding latency of (b). (e) Reactive forwarding latency of (c). (f) Reactive forwarding latency of (c) which is improved by automatic placement optimization at second 6.

Beehive vs Centralized Controllers. To compare the simplicity of Beehive with centralized controllers, we have implemented a hub, a learning switch, a host tracker, a firewall (*i.e.*, ACL at the edge) and a layer-3 router with respectively 16, 69, 158, 237 and 310 lines of code using Beehive's SDN abstraction. These applications are similar in functionality and on par in simplicity to what is available on POX and Beacon. The differences in lines of code are because of different verbosity of Python, Java, and Go. The significant difference here is that Beehive SDN applications will automatically benefit from persistent storage, fault-tolerance, runtime instrumentation, and optimized placement, readily available in Beehive.

Fault-Tolerance. To tolerate a failure (*i.e.*, either a controller failure or a lost switch-to-controller channel), our node controller application pings the drivers of each node (by default, every 100ms). If a driver does not respond to a few (3 by default) consecutive pings, the driver is announced dead. If the lost driver is the master driver, one of the slave drivers will be instructed to become the master driver. It is important to note that when the hive hosting the node controller fails, the platform will automatically delegate the node to a follower bee. In such scenarios, once the node controller resumes, it continues to health-check the drivers.

To evaluate how our SDN controller reacts to faults, we have developed a reactive forwarding application that learns layer-2 and layer-3 addresses, but instead of installing flow entries, it sends `packet_out`s. With that, a failure in the control plane is directly observed in the data plane. In this experiment, we create a simple network emulated in mininet, and measure the round-trip time between two hosts every 100ms for 10 seconds. To emulate a fault, we kill the hive that hosts the OpenFlow driver at second 3. In this set of experiments, we use a Raft election timeout of 300ms.

Let us first clarify what happens for the control applications upon a failure. As shown in Figure 12a, because of the default placement, the master bees of node controller and the forwarding application will be collocated on the hive to which the switch first connects. The driver on the hive consequently will be the master driver. In this setting, the average latency of the forwarding application will be about 1ms.

Failures in any of the hives hosting the slave bees would have no effect on the control application (given that the majority of hives, 2 in this case, are up and running). Now, when the hive that hosts the master bees fail (or goes into a minority partition), one of the follower bees of each application will be elected as a master bee for that application. There are two possibilities here: (*i*) as shown in Figure 12b new masters may be both collocated on the same hive, or (*ii*) as shown in Figure 12c, new masters will be on different hives[4].

As portrayed in Figure 12d, when the new masters are elected from the same hive the latency after the fail-over will be as low as 1ms. In contrast, when the new masters are located on different hives the latency will as high as 8ms (Figure 12e). This is where the automatic placement optimization can be of measurable help. As shown in Figure 12f, our greedy algorithm detects the suboptimality and migrates the bees accordingly. After a short spike in latency because of the buffering of a couple of messages during migration, the forwarding latency becomes 1ms. It is important to note that as demonstrated in Figure 8, the Raft election timeout can be tuned to lower values if a faster failover is required in an environment.

When the link between the switch and the master driver fails (*i.e.*, due to an update in the control network, congestion in channels, or a switch reconfiguration), something similar happens (Figure 13a). In such scenarios, the master driver notifies the node controller that its switch is disconnected. In response, the node controller elects one of the remaining slave drivers as the master. Although the forwarding application and the master driver are not from the same hive anymore, the forwarding application

[4]We cannot prefer a bee over another in a colony at election time, as it will obviate the provable convergence of Raft.

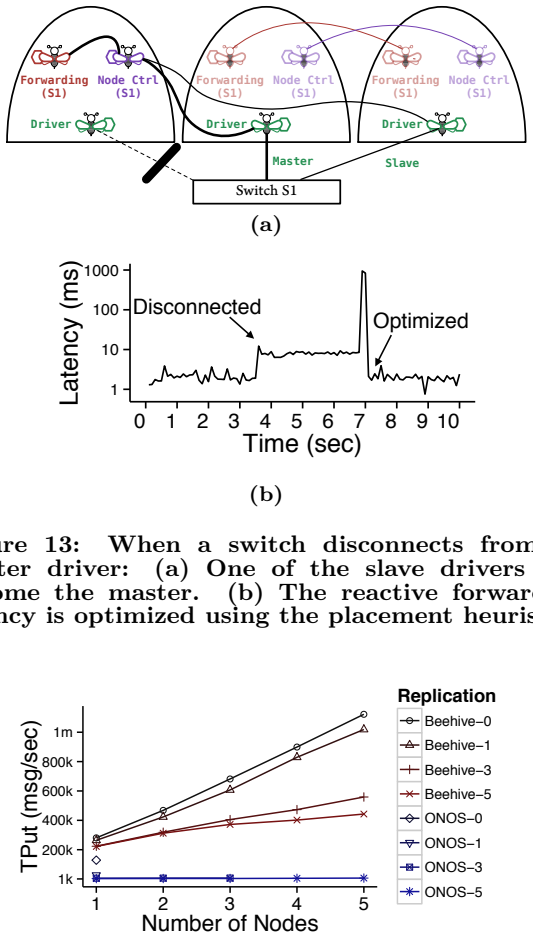

Figure 13: When a switch disconnects from its master driver: (a) One of the slave drivers will become the master. (b) The reactive forwarding latency is optimized using the placement heuristic.

Figure 14: Throughput of Beehive learning switching application with different replication factors. The number of nodes (*i.e.*, the x-axis) in this graph represents the nodes on which we launch `cbench`.

will perform consistently. This is an important property of Beehive. As shown in Figure 13b, the forwarding latency would accordingly be high at first. As soon as the automatic optimizer detects this suboptimality the bees are migrated and the latency will drop. We note that, since the drivers have IO routines they will never be migrated by the optimizer. As a result, it is always the other bees that will be placed close to those IO channels.

Scalability. To characterize the scalability of the Beehive SDN controller, we benchmarked our learning switch application deployed on 1, 3, and 5 hives with no persistence as well as replication factors of 1, 3, and 5, all with a batch size of 8192. We repeat the same tests for ONOS "reactive forwarding" application (*i.e.*, ONOS-0 represents ONOS with no persistence, and ONOS-1, ONOS-3, and ONOS-5 respectively represent 1-node, 3-node, and 5-node clusters). Both Beehive and ONOS applications perform the same task of learning address to port mapping on each switch. Note that, replication size and cluster size are equivalent in ONOS since we do not have fine-grained control over where the state of the application is replicated. As a result, unlike Beehive, we cannot perform a test for a replication factor of R on M VMs, where $R<M$. We stress each controller by one `cbench` process. In these experiments, we use 5 Google Compute Engine (GCE) virtual machines with 16 virtual cores, 14.4GB of RAM, and directly attached SSDs. We launch `cbench` on either 1, 3 or 5 nodes in parallel, each emulating 16 different switches. That is, when we use 1, 3, and 5 nodes to stress the application we have 16, 48, and 80 switches equally spread among 1, 3, and 5 hives, respectively. Moreover, all SDN applications in our controller suite (*e.g.*, the node controller and discovery) are enabled as well as runtime instrumentation. Although disabling all extra applications and using a Hub will result in a throughput of one million messages per second on a single machine, we believe such a benchmark is not a realistic demonstration of the real-world performance of a control platform.

As shown in Figure 14, when we stress only 1 node in the cluster the throughputs of the learning switch application with different replication factors are very close. As we engage more controllers, the throughput of the application with replication factors of 3 and 5 demonstrate increase but at a slower pace. This is because when we use larger replication factors, other hives will consume resources to replicate the transactions in the follower bees.

As indicated by these micro-benchmarks, although the ONOS and Beehive applications perform the same logical task with similar updates in their state, Beehive outperforms ONOS in throughput. This is mainly due to architectural and implementation differences: Beehive handles message passing, replication, and storage internally without relying on an external datastore such as Hazelcast [3]. This allows Beehive to transparently batch messages and transactions to avoid communication overheads. We note that, recently, ONOS has moved towards a design similar to Beehive, as an effort to improve its scalability and performance: *i.e.*, sharded Raft with more granular control over partitioning of data. Although, at the time of this writing, we do not see an improvement in our test results for ONOS's reactive forwarding application when it is configured as a learning switch, we expect to see higher throughput as ONOS's new implementation matures.

5. DISCUSSION AND RELATED WORKS

What are the main limitations of Beehive's programming model? As we mentioned in Section 2, Beehive applications cannot store persistent data outside the dictionaries. Since the value of a dictionary entry can be an arbitrary object, this requirement would not impose a strong limitation on control applications. We note that this requirement is the key enabler for the platform to automatically infer the `map` function and to provide fault-tolerance for control applications.

Another design-level decision is to provide a serialized, consistent view of messages for each partition of cells owned by a bee. That is, each bee is a single thread that processes its messages in sequence. Thus, Beehive does not provide eventual consistency by default and instead processes messages in consistent shards. A Beehive application can implement eventual consistency by breaking the functionalities into local message handlers and using asynchronous messages to share state instead of dictionaries. We note that eventual consistency has many implications and performance drawbacks [9, 13].

How does Beehive compare with existing distributed programming frameworks? There is a significant body

of research on facilitating distributed programming. Proposals such as Coign [19], J-Orc- hestra [31], Live-Objects [30] and Fabric [25] provide vertical partitioning frameworks to place different modules/objects of a program on different computing resources to scale.

There are also domain-specific programming frameworks that transparently partition a centralized program into different pieces. Some of these approaches use offline static analysis. For example, Pleiades [23] and Wishbone [28] compile a centralized sensor network program into multiple units of work that are offloaded onto nodes across the wireless network. Other approaches mix-in dynamic profiling at runtime to optimize their partitioning. For example, CloneCloud [12] profiles a mobile application at runtime and offloads functionalities onto the cloud to minimize power usage and maximize performance. These proposals are all limited to a specific application domain that does not match SDN controller requirements.

Moreover, special-purpose programming models, such as MapReduce [4], MPI, Dryad [20], Cilk [8] and Spark [33] have been proposed for parallel and distributed processing. Although these proposals are focused on specific problems (*e.g.*, batch workloads) and are not aptly applicable for generic distributed applications, Beehive has been inspired by the way they simplify distributed programming.

Further, There are Domain-Specific Languages (DSLs), such as BOOM [5], that propose new ways of programming for the purpose of provability or verifiability. In contrast, our goal in this paper is to create a distributed programming environment similar to centralized controllers. To that end, control applications in Beehive are developed using a general purpose programming language using a familiar the programming model.

How does Beehive compare with existing distributed controllers? Existing control planes such as ONIX [22] and ONOS [7] are focused on scalability and performance. Beehive, on the other hand, provides a programming paradigm that simplifies the development of scalable control applications. We note that abstractions such as ONIX NIB can be implemented on Beehive, as we demonstrated for Beehive's SDN controller. More importantly, Beehive's automatic placement optimization leads to results similar to what ElasticCon [14] achieves by load-balancing. Our optimization, however, is not limited to balancing the workload of OpenFlow switches, applies to all applications, and can be extended to accommodate alternative optimization methods.

How does Beehive compare with fault-tolerance proposals? LegoSDN [11] is a framework to isolate application failures in a controller. Beehive also isolates application failures using automatic transactions, which is similar to LegoSDN's Absolute Compromise policy for centralized controllers. Other LegoSDN policies can be adopted in Beehive, if needed. Moreover, Ravana [21] is a system that provides transparent slave-master fault-tolerance for centralized controllers. As demonstrated in Figure 13a, Beehive has the capabilities not only to tolerate a controller failure, but also seamlessly and consistently tolerates faults in the control channel that is outside the scope of Ravana.

Do applications interplay well in Beehive? The control plane is an ensemble of control applications managing the network. For the most part, these applications have interdependencies. No matter how scalable an application is on its own, heedless dependency on a poorly designed application may result in subpar performance. For instance, a local application that depends on messages from a centralized application might not scale well. Beehive cannot automatically fix a poor design, but provides analytics to highlight the design bottlenecks of control applications, thus assisting the developers in identifying and resolving such performance issues.

Moreover, as shown in [10] and [27], there can be conflicts in the decisions made by different control applications. Although we do not propose a specific solution for that issue, these proposals can be easily adopted to implement control applications in Beehive. Specifically, one can implement the Corybantic Coordinator [27] or the STN Middleware [10] as a Beehive application that sits in between the SDN controller and the control applications. With Beehive's automatic optimization, control modules are easily distributed on the control platform to make the best use of available resources.

6. CONCLUSION

In this paper, we have presented a distributed control platform that provides an easy-to-use programming model. Using this programming model, Beehive automatically infers cohesive parts of an application's state and uses that to seamlessly transform the application to its distributed version. By instrumenting applications at runtime, we optimize the placement of control applications and provide feedback to developers. We have demonstrated that our proposal is capable of accommodating several different usecases. Implementing a SDN controller on top of Beehive, we have demonstrated that this approach is simple yet effective in designing scalable, distributed control planes.

7. ACKNOWLEDGEMENTS

We would like to thank our shepherd Marco Canini and the anonymous reviewers for their insightful comments and suggestions. This work is partially supported by NSERC Research Network for Smart Applications on Virtual Infrastructures (SAVI) grant, and NSERC Discovery Grant No. 346203.

8. REFERENCES

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

[2] Beehive. `http://github.com/kandoo/beehive`.

[3] Hazelcast. `http://hazelcast.org`.

[4] MapReduce: Simplified Data Processing on Large Clusters.

[5] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *Proceedings of EuroSys'10*, pages 223–236, 2010.

[6] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of IMC'10*, pages 267–280, 2010.

[7] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor,

P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of HotSDN '14*, pages 1–6, 2014.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August.

[9] E. Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer Magazine*, 45(2):23–29, 2012.

[10] M. Canini, P. Kuznetsov, D. Levin, S. Schmid, et al. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *Proceedings of INFOCOM'15*, pages 190–198, 2015.

[11] B. Chandrasekaran and T. Benson. Tolerating SDN Application Failures with LegoSDN. In *Proceedings of HotNets-XIII*, pages 22:1–22:7, 2014.

[12] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of EuroSys '11*, pages 301–314, 2011.

[13] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of OSDI'12*, pages 251–264, Berkeley, CA, USA, 2012.

[14] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. ElastiCon: An Elastic Distributed SDN Controller. In *Proceedings of ANCS '14*, pages 17–28, 2014.

[15] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, August 2004.

[16] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proceedings of HotSDN'12*, pages 19–24, 2012.

[17] S. Hassas Yeganeh and Y. Ganjali. Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking. In *Proceedings of HotNets'14*, pages 13:1–13:7, 2014.

[18] B. Heller, R. Sherwood, and N. McKeown. The Controller Placement Problem. In *Proceedings of HotSDN'12*, pages 7–12, 2012.

[19] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of OSDI '99*, pages 187–200, 1999.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of EuroSys '07*, pages 59–72, 2007.

[21] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller Fault-Tolerance in Software-Defined Networking. In *Proceedings of SOSR '15*, 2015.

[22] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. ONIX: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of OSDI'10*, pages 1–6, 2010.

[23] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and Efficient Programming Abstractions for Wireless Sensor Networks. In *Proceedings of PLDI '07*, pages 200–210, 2007.

[24] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson. Pratyaastha: An Efficient Elastic Distributed SDN Control Plane. In *Proceedings of HotSDN'14*, pages 133–138, 2014.

[25] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of SOSP'09*, pages 321–334, 2009.

[26] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker. Extending SDN to Large-Scale Networks. In *Open Networking Summit*, 2013.

[27] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner. Corybantic: Towards the Modular Composition of SDN Control Programs. In *Proceedings of HotNets-XII*, pages 1:1–1:7, 2013.

[28] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proceedings of NSDI'09*, pages 395–408, 2009.

[29] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of USENIX ATC'14*, pages 305–320, 2014.

[30] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn. Programming with Live Distributed Objects. In *Proceedings of ECOOP'08*, pages 463–489, 2008.

[31] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java Programs with Distribution Capabilities. pages 178–204, 2002.

[32] A. Wang, W. Zhou, B. Godfrey, and M. Caesar. Software-Defined Networks as Databases. In *ONS'14*, 2014.

[33] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proceedings of SIGMOD'13*, pages 13–24, 2013.