Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking

Soheil Hassas Yeganeh
Department of Computer Science
University of Toronto, Toronto, Canada
soheil@cs.toronto.edu

Yashar Ganjali
Department of Computer Science
University of Toronto, Toronto, Canada
yganjali@cs.toronto.edu

ABSTRACT

Simplicity is a prominent advantage of Software-Defined Networking (SDN), and is often exemplified by implementing a complicated control logic as a simple control application on a centralized controller. In practice, however, SDN controllers turn into distributed systems due to performance and reliability limitations, and the supposedly simple control applications transform into complex logics that demand significant effort to design and optimize.

In this paper, we present Beehive, a distributed control platform aiming at simplifying this process. Our proposal is built around a programming abstraction which is almost identical to a centralized controller yet enables the platform to automatically infer how applications maintain their state and depend on one another. Using this abstraction, the platform automatically generates the distributed version of each control application, while preserving its behavior. With runtime instrumentation, the platform dynamically migrates applications among controllers aiming to optimize the control plane as a whole. Beehive also provides feedback to identify design bottlenecks in control applications, helping developers enhance the performance of the control plane. Our prototype shows that Beehive significantly simplifies the process of realizing distributed control applications.

Categories and Subject Descriptors

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

General Terms

Design

1. INTRODUCTION

In large-scale software-defined networks, distributed control platforms are employed for the reasons of scale and resilience [8]. Albeit instrumental for scalability, existing distributed control platforms do not hide the complexities in realizing a scalable control plane and pass on many design and scalability challenges to network programmers.

To come up with a scalable design, network programmers need to measure control applications, find design bottlenecks, and manually optimize the control plane. They also need to deal with the common complications of distributed systems (such as timing, consistency, synchronization and coordination) that are admittedly pushed into control applications [13]. This directly obviates the promised simplicity of SDN.

In this paper, we aim at realizing a distributed control platform that is straightforward to program and instrument, comparable to centralized controllers in simplicity. To that end:

1. We propose a programming abstraction for the control plane that is similar to centralized controllers yet enables us to automatically infer important characteristics of control applications, including their interdependencies and how they maintain their state.
2. We present the design and implementation of Beehive's distributed control platform that automatically generates and deploys the distributed version of the control applications developed using the proposed abstraction. We ensure that this transformation preserves the intended behavior of the application when distributed over multiple physical machines. Beehive's control platform instruments control applications at runtime, dynamically optimizes their placement, and provides useful feedback to identify design bottlenecks.

We demonstrate that our system is simple and intuitive, yet covers a variety of scenarios ranging from implementing different network applications (*e.g.*, Network Virtualization and Routing) to emulating existing distributed controllers (such as ONIX [11] and Kandoo [7]). We have evaluated our control platform for several applications including network virtualization, routing, and traffic engineering. Using a traffic engineering example, we show how our proposal can dynamically optimize the control plane and provide feedback to improve the application's design.

HotNets-XIII, October 27–28, 2014, Los Angeles, CA, USA.

http://dx.doi.org/10.1145/2670518.2673864

2. ABSTRACTION

One of the most challenging steps in transforming a centralized application into its distributed identical twin is finding a distribution mechanism for the application's state that preserves the application's behavior. This information is quite subtle and difficult to extract from an application written in a general purpose programming language. For that reason, we propose a programming abstraction for developing control applications that enables our framework to automatically infer how the application state is accessed and modified. We intentionally designed this abstraction to be similar to and as simple to use as centralized controllers.

As depicted in Figure 1, we model a control application as a set of functions that are triggered by asynchronous messages and can emit further messages to communicate with other functions. To process a message, a function accesses the application state which is defined in the form of dictionaries (*i.e.*, key-values) with support for transactions. Application functions are arbitrary programs which have to explicitly specify the entries of the state they require for processing a message. Any data stored outside the application's state is ephemeral.

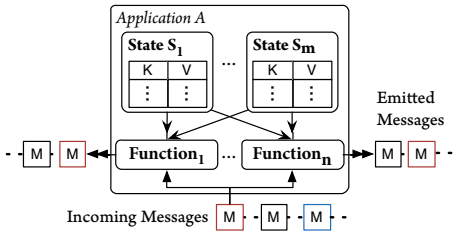

Figure 1: Applications are modeled as stateful functions exchanging asynchronous messages.

Let us demonstrate this abstraction for a simple traffic engineering example.

Example: Traffic Engineering. As shown in Figure 2, a naive Traffic Engineering (TE) can be modeled as four main functions at its simplest: (*i*) `Init` that initializes the flow statistics of a switch. (*ii*) `Query` that periodically queries switches. (*iii*) `Collect` that collects replies to flow queries and populates the time-series of flow statistics in the `S` dictionary. (*iv*) `Route` that re-steers traffic using OpenFlow `FlowMod` [14] messages if there is a significant change in the traffic. These functions share the dictionary `S` that stores flow statistics of each switch as an entry.

These four functions are invoked in response to asynchronous messages (denoted by `on`). For example, `Init` is invoked when an OpenFlow driver emits a `SwitchJoined` message, `Collect` upon receiving `StatReply`s, and `Route` every 1 second. To invoke a function, one explicitly requests all or some entries in state dictionaries (denoted by `with`), or invokes the function for each entry (using `foreach`).

```
app TrafficEngineering:
    state:
        S /* Flow Stats */, T /* Topology */
    func Init(switch, sEntry):
        sEntry.set(FlowStat(switch))
    func Query(switch):
        emit(FlowStatQuery(switch))
    func Collect(reply, sEntry):
        sEntry.Append(All flow stats in reply)
    func Route(S, T):
        // δ is a user defined threshold.
        if Change in S > δ then
            Use T to reroute flows.
    on SwitchJoined(joined):
        with S[joined.switch] as entry:
            Init(joined.switch, entry)
    on TimeOut(1sec):
        for each switch in S:
            Query(switch)
    on StatReply(reply):
        with S[reply.switch] as mEntry:
            Collect(reply, mEntry)
    on TimeOut(1sec):
        with S and T:
            Route(S, T)
    // Details on handling switch and link
       discovery is omitted for brevity.
```

Figure 2: This simple Traffic Engineering application stores flow stats and the network topology respectively in the `S` and `T` dictionaries, periodically collects stats, and accordingly reroutes traffic.

In our example, to query each switch every second, we simply invoke `Query` for each key in `S`. Upon receiving the flow statistics of a switch, we invoke `Collect` using the single entry representing the traffic data of that switch (*i.e.*, `S[e.switch]`).

In addition to these functions, TE builds its own view of the network topology whenever a switch joins the network or when a link is detected by a discovery application. The topology data is stored in the `T` dictionary and is only used as a whole by `Route`. This application installs default routes to ensure reachability.

Inter-Dependencies. Application functions can depend on one another in two ways: (*i*) they can either exchange messages, or (*ii*) access a shared state (only if they belong to the same application). In our TE example, the functions depend on each other by sharing `S`. `Init`, `Collect`, `Query`, and `Route` depend on an OpenFlow driver that emits `SwitchJoined`s and `StatReply`s and can process `Query`s and `FlowMod`s.

Functions of two different applications communicate using messages and cannot share state. 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 the platform the freedom to optimize the control plane placement. That said, to support stateful control applications, functions inside an application can share state.

State & Distributed Applications. We inten-

tionally designed this abstraction almost identical to the usual centralized controllers. Our ultimate goal, however, is to use this abstraction in transforming applications to their distributed counterparts. More specifically, consider a network in which we have multiple physical machines each running a controller of our distributed control platform. We want to utilize the resources of all these controllers to run the functions of our control applications. For example, we want to run `Query` and `Collect` on as many controllers as we can, hopefully querying a switch on its master controller to lower latency and to scale.

That process is trivial for stateless applications: we can merely replicate all functions on all controllers. In contrast, this process is challenging when we deal with *stateful* applications. To realize a valid, distributed version of a control application, we need to make sure that all control functions, when distributed on several controllers, have a consistent view of their state.

In general, to preserve state consistency, we need to ensure that each key in each application dictionary is accessed on only one controller (*i.e.*, by only one instance of the control application). For example, suppose that we have an application `A` with two functions $\mathtt{f}_1$ and $\mathtt{f}_2$ that respectively handle messages $\mathtt{m}_1$ and $\mathtt{m}_2$. Moreover, assume that, to process $\mathtt{m}_1$ and $\mathtt{m}_2$, $\mathtt{f}_1$ accesses $\mathtt{K}_1 = \{\mathtt{k}_{11} \cdots \mathtt{k}_{1n}\}$ and $\mathtt{f}_2$ accesses $\mathtt{K}_2 = \{\mathtt{k}_{21} \cdots \mathtt{k}_{2m}\}$. If $\mathtt{K}_1 \cap \mathtt{K}_2 \neq \emptyset$, the platform must guarantee that the keys in $\mathtt{K}_1 \cup \mathtt{K}_2$ are always accessed by only one instance of the control application and also $\mathtt{m}_1$ and $\mathtt{m}_2$ are processed by the same instance. Otherwise, we will have an eventually consistent application state at its best (assuming the application detects and resolves potential conflicts in its state), or a chaotic, invalid control logic.

In our naive TE example, `Query` and `Collect` access the flow statistics data (*i.e.*, the `S` dictionary) on a per switch basis, whereas `Route` accesses the whole dictionary. If there was no `Route`, the flow statistics dictionary could be distributed among controllers by assigning the flow statistics of each switch (*i.e.*, an entry in `S`) to one controller. In such a setting, `Query` queries a switch on one controller and `Collect` updates the flow statistics of that switch on the same controller.

Having said that, since `Route` requires the whole dictionary, we have to collocate all keys on the same controller. This essentially means that `Route` and any function that shares state with `Route` would be effectively centralized. This illustrates that our naive TE application cannot scale well, or at least, there is no benefit gained from using a distributed controller based on the current design. We will later illustrate how our control platform provides useful feedback to resolve such a design-level scalability issue by decoupling `Route` from other functions. Moreover, once such a design bottleneck is resolved, our control platform will automatically optimize the placement of control functions.

3. CONTROL PLATFORM

We have designed and implemented Beehive's control platform as the runtime environment for the proposed programming abstraction. This platform is the target onto which Beehive automatically compiles control applications. This distributed platform provides two important functionalities at its core: (*i*) concurrent and consistent state access in a distributed fashion, and (*ii*) runtime instrumentation and optimization of distributed applications.

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 in a specific state dictionary: $(\mathtt{dict}, \mathtt{key}, \mathtt{val})$. For instance, our TE application (Figure 2) has a cell for the flow statistics of each switch $\mathtt{SW}_\mathtt{i}$ in the form of $(\mathtt{S}, \mathtt{SW}_\mathtt{i}, \mathtt{Stat}_{\mathtt{SW}_\mathtt{i}})$.

To preserve consistency, we need to ensure that each cell is accessed on only one hive. For the TE application, as an example, there should be only one hive in the platform that stores the flow statistics of a particular switch, say $\mathtt{SW}_\mathtt{i}$. To preserve consistency, only on that hive, we should invoke `Collect` and `Init` for `StatReply` and `SwitchJoined` messages for $\mathtt{SW}_\mathtt{i}$. In more complex cases, as discussed in Section 2, we may have several cells that must be collocated on the same hive. In our example, due to the subpar design, `Route` uses the whole dictionary, which mandates storing all flow statistic cells on the same hive. In Section 5, we demonstrate how the platform provides feedbacks to detect this issue, and how it can be solved by a simple application redesign.

Bees. For each set of cells that must be collocated, we create an exclusive light-weight thread of execution, called a *bee*. 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 function and provides its cells as the application's state.

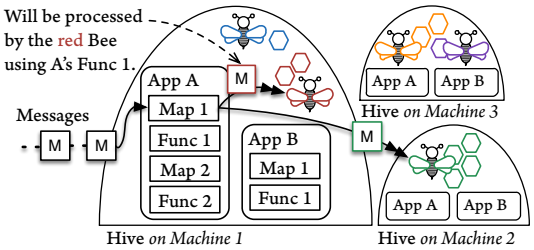

Figure 3: In Beehive, each hive (*i.e.*, controller) maintains the application state in the form of cells (*i.e.*, key-values). Cells that must be collocated are exclusively owned by one bee. Messages mapped to the same cell(s) are processed by the same bee.

For example, once it received the `SwitchJoined` for switch $\mathtt{SW}_\mathtt{i}$, the platform relays the message to the

bee that exclusively owns cell $(\mathtt{S}, \mathtt{SW_i}, \mathtt{Stat_{SW_i}})$. Then, the bee invokes `Init` for that message. Similarly, the same bee handles consequent `StatReply`s for $\mathtt{SW_i}$. This ensures that `Collect` and `Init` share the same consistent view of the state dictionary `S` and their behavior is identical to when they are deployed on a centralized controller, even though they might be physically distributed over different controllers (*i.e.*, hives).

Cells owned by a bee are basically the keys that must be accessed on the same hive. Those cells are application-defined and can be automatically inferred from the programming abstraction. To find the cells required for processing a message, the control platform automatically generates a `Map` for each application: $\mathtt{Map}(\mathtt{A}, \mathtt{M})$ is a function generated for application `A` that maps a message of type `M` to a set of cells: simply a set of keys in application dictionaries $\{(\mathtt{D_1}, \mathtt{K_1}), \ldots, (\mathtt{D_n}, \mathtt{K_n})\}$. We call this set, the *mapped cells* of message `M` in application `A`. This set is application-specific and is inferred based on the keys in `with` and `foreach` clauses in the programming abstraction.

As shown in Figure 3, to preserve consistency, Beehive guarantees that all messages with *intersecting* mapped cells for application `A` are processed by the same bee using `A`'s functions. For instance, consider two messages that are mapped to $\{(\mathtt{Switch}, 1), (\mathtt{Mac}, \mathtt{FF...})\}$ and $\{(\mathtt{Switch}, 1), (\mathtt{Port}, 12)\}$ respectively by a control application. Since these two messages share the cell $(\mathtt{Switch}, 1)$, the platform guarantees that both messages are handled by the same bee.

This way, the platform ensures that each part of the application state is owned by one single thread of execution on the distributed control plane. In other words, the platform prevents two different bees from modifying or reading the same part of the state.

Life of a Message. On each hive, messages are either generated upon receiving data over IO channels, or emitted by a function. Upon receiving a message, the hive passes the message to the generated `Map` functions of all applications that are triggered by that particular type of message. For each application, using a distributed locking mechanism (*e.g.*, Chubby [4]), the hive finds the bee that owns at least one cell in the application's mapped cells. If there is such a bee (either on the local hive or on a remote hive), the message is accordingly relayed. Otherwise, the local hive creates a new bee, assigns the cells to it, and relays the message.

Migration of Bees. Beehive provides the functionalities to migrate a bee from one hive to another along with its cells. This is instrumental in fault-tolerance and optimization. To migrate a bee, Beehive first stops the bee and buffers all incoming messages. It then moves the cells to the target hive. Then, a new bee is created on the remote host to own the migrated cells. At the end, buffered messages are drained to the new bee.

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 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 to another data center, the functions controlling that virtual network should also be moved with it 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 on our platform since we have a well-defined abstraction for control functions, their state, and respective messages.

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, we store that `packet_out` messages are emitted by the learning switch application upon receiving 80% of `packet_in`'s.

We measure runtime metrics on each hive locally, and periodically aggregate them on a single hive. This merged instrumentation data is further used to find the optimal placement of bees and is also utilized for application analytics. We implemented this mechanism using the proposed abstraction as a control application.

On Optimal Placement. Finding the optimum placement of bees is $\mathcal{NP}$-Hard, as the facility location problem can be reduced to it. In addition to the computational complexities, there are lots of subtle performance side-effects in changing the placement of bees. Thus, to improve the initial placement, we employ a greedy heuristic aiming at processing messages close to their source. Suppose we have two hives $\mathtt{H_1}$ and $\mathtt{H_2}$. We migrate $\mathtt{B_1}$ running on $\mathtt{H_1}$ to $\mathtt{H_2}$, if the majority of messages processed by $\mathtt{B_1}$ are from bees deployed on $\mathtt{H_2}$ and $\mathtt{H_2}$ has enough capacity to host the cells of $\mathtt{B_1}$. We note that, using our platform, it is straightforward to implement other optimization strategies.

Implementation. We have implemented a prototype of Beehive in Go, which is available on [1]. Boilerplates, such as serialization/deserialization, queueing, parallelism, synchronization, and distributed locking are all provided by the platform. We will present a preliminary evaluation of Beehive in Section 5.

4. USE CASES

In this section, we present how important SDN use-cases are implemented using Beehive. We keep our discussion brief and high level as we present a complete example for TE in Section 5.

Centralized Applications. A centralized application is a composition of functions that require the whole application state in one physical location. In our framework, a function is centralized if it accesses the whole dictionaries to handle messages. As discussed earlier in Section 3, for such a function, Beehive guarantees that the whole state, *i.e.*, all cells of that application, are assigned to one bee. It is important to note that, since applications do not share state, the platform may place different centralized applications on different hives to satisfy extensive resource requirements (*e.g.*, a large state).

Kandoo. At the other end of the spectrum, there are local control applications proposed in Kandoo [7] that use the local state of a single switch to process frequent events. The functions of a local control application use switch IDs as the keys in their state dictionaries and, to handle messages, access their state using a single key. As such, Beehive conceives a cell for each switch and allocates one bee for each cell. In practice, this results in local functions being replicated on all controllers to handle switches local to that controller.

An important advantage of Beehive, over Kandoo, is that it automatically pushes control functions as close as possible to the source of messages they process (*e.g.*, switches for local applications). In such a setting, network programmers do not deliberately design for a specific placement. Instead, the platform automatically optimizes the placement of local applications. This is in contrast to proposals like Kandoo where the developer has to decide on function placement (*e.g.*, local controllers close to switches).

ONIX's NIB [11]. NIB is basically an abstract graph that represents networking elements and their interlinking. To process a message in a NIB manager, we only need the state of a particular node. As such, each node would be equivalent to a cell managed by a single bee in Beehive. With that, all queries (*e.g.*, flows of a switch) and update messages (*e.g.*, adding an outgoing link) on a particular node in NIB will be handled by the node's bee in the platform.

Network Virtualization. Typically, network virtualization applications (such as NVP[10]) process messages of each virtual network independently. Such applications can be modeled as a set of functions that, to process messages, access the state using a virtual network identifier as the key. This is basically sharding messages based on virtual networks, with minimal shared state in between the shards. Each shard basically forms a set of collocated cells in Beehive and the platform guarantees that messages of the same virtual network are handled by the same bee.

Routing. A distributed routing application can be easily defined in Beehive by storing the RIBs on a prefix basis or based on source and/or destination. This results in fine-grain cells that can be automatically placed throughout the platform to scale. Furthermore, approaches such as Portland [16] and Seattle [9] can be easily implemented in a distributed fashion.

5. EVALUATION

We have implemented several applications including routing, network virtualization, and Kandoo using our prototype. In this section, for the sake of space, we use our TE example to evaluate 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. Here, 10% of these flows have a rate more than a user-defined re-routing threshold (*i.e.*, δ in Figure 2).

Naive TE. Instrumenting the TE implementation in Figure 2, Beehive provides the network programmer with feedbacks on how the application functions behave as a distributed system. For example, the platform provides the number of messages exchanged between bees (as summarized in Figure 4a) and their bandwidth consumption (depicted in Figure 4d).

From this analytical data, one can easily observe that most messages are sent to/from the bees on only one hive. The detailed instrumentation data (not shown here) also indicate that `Collect` and `Query` are always invoked by the same bee because of sharing cells with `Route`. Accordingly, control channel consumption is relatively high considering the size of the simulated network. This shows that our design is effectively centralized on only one bee, and cannot scale well due to the strong coupling among its functions.

We have two alternatives to address this issue: (*i*) We can redesign `Route` to use a small portion of the state for re-routing, or (*ii*) we can decouple `Route` from `Collect` and `Query` by eliminating the shared state. Here, we discuss the latter as it is simpler to explain.

Decoupling Functions. To decouple TE functions, the programmer needs to create a separate dictionary for `Route`, and send aggregated events from `Collect` to notify `Route` about flow stat updates. That is, `Collect` sends updates to `Route` when the flow's bandwidth consumption passes a threshold. This simply eliminates the need to collocate `Collect`/`Query` with `Route`.

Instrumenting the new design, the network programmer observes that most messages are now processed locally (the diagonal line in Figure 4b) but we still have occasional communications to/from the hive that happens to host the centralized bee for `Route` (the cross in

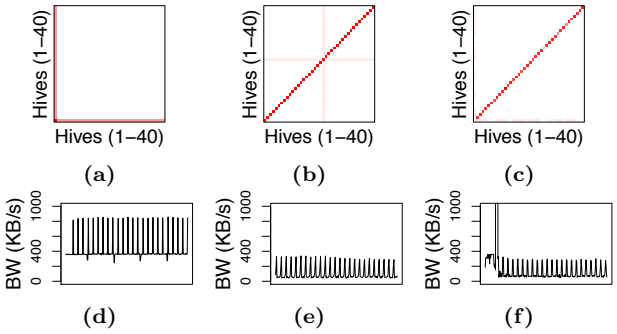

Figure 4: Inter-hive traffic matrix and control channel bandwidth consumption of TE when the functions are centralized (a & d), when decoupled (b & e), and when optimized at runtime (c & f).

Figure 4b). Accordingly, as shown in Figure 4e, control channel consumption is significantly improved.

Optimization. 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 runtime instrumentation collects enough data about the futile communications over the control channels, it starts to migrate the bees invoking `Collect` and `Query` to hives directly 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 Figure 4c and Figure 4f, this live migration of control plane functions localizes message processing and results in considerable improvement in control channel consumption. The largest spike in Figure 4f correlates to replicating cells to the other hives. Note that this is all done automatically at runtime with no manual intervention and, after optimization, application's behavior is identical to Figures 4e and 4b.

6. DISCUSSION

How does Beehive compare with existing proposals? Existing distributed controllers focus on tools for distributed programming (*e.g.*, an eventually consistent network graph in ONIX [11]) or balancing the dataplane load among controllers [6]. The focus of these proposals is mostly scalability, and simplified network programming is not necessarily an objective there. Moreover, there are domain specific language proposals (*e.g.*, bloom [2]) to simplify distributed programming. Although vaguely similar in goals, Beehive focuses on familiar programming constructs with a simple abstraction for storing the application state.

Do applications interplay well in Beehive? In a large-scale network, the control plane is an ensemble of control applications managing the network as a cohesive whole. 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 helping the developers identify and resolve design issues.

Moreover, as shown in STN [5] and Corybantic [15], there can be conflicts in the decisions made by different control applications. Although we do not propose a solution for that issue, these proposals can be easily adopted in Beehive. For example, one can implement the Corybantic Coordinator as a Beehive application and implement control modules as applications that exchange objective messages. With Beehive's automatic optimization, control modules can easily be distributed on the platform to utilize available resources.

Can't we simply use a distributed database? Recent distributed control platforms delegate the complexity of managing the state to an external system (*e.g.*, Cassandra [12] and RamCloud [17] in ONOS [3]) which leads into a simpler control platform. Delegating such an important responsibility, however, has three drawbacks. First, it is not straightforward to reason about the control plane's latency since the platform has no control over the physical placement of the network state. Second, communicating with an external system incurs communication overheads both on controllers and on control channels (note that the external datastore is deployed in the control plane). Third, using an external store, network administrators need to manage two separate systems which hinders manageability.

7. CONCLUSION

In this paper, we have presented a framework that simplifies the design and implementation process of distributed control applications in SDN. Using a simple programming abstraction, we infer shared state between application functions, and automatically compile applications into their distributed counterparts. By instrumenting applications at runtime, we optimize the placement of functions, and provide feedback to the developer helping with the design process.

We have demonstrated that our abstraction is able to model existing distributed control planes. Moreover, our evaluations confirm that this approach can be effective in designing scalable control applications. Moving forwards, we are enforcing the foundations of our framework specially for fault-tolerance and smarter optimization strategies, and we are applying this framework for other control logics including routing.

8. ACKNOWLEDGEMENTS

We would like to thank Adam Zarek and our anonymous reviewers for their insight and helpful comments. This work was partially funded by the NSERC SAVI strategic network.

9. REFERENCES

[1] Beehive Control Platform. `http://github.com/kandoo/beehive`.

[2] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[3] 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.

[4] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of OSDI'06*, pages 335–350, 2006.

[5] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software Transactional Networking: Concurrent and Consistent Policy Composition. In *Proceedings of HotSDN'13*, pages 1–6, 2013.

[6] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In *Proceedings of HotSDN'13*, pages 7–12, 2013.

[7] 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.

[8] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of HotSDN'12*, pages 7–12, 2012.

[9] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: A scalable ethernet architecture for large enterprises. In *Proceedings of the SIGCOMM'08*, pages 3–14, 2008.

[10] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *Proceedings of NSDI'14*, pages 203–216, 2014.

[11] 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.

[12] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

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

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, Mar. 2008.

[15] 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.

[16] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of SIGCOMM'09*, pages 39–50, 2009.

[17] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011.